TEST CASE PRIORITIZATION FOR OPTIMIZING A REGRESSION TEST

Ahlam Ansari¹, Alisha Khan², Anam Khan³ and Konain Mukadam⁴

Department of Computer Engineering, M.H. Saboo Siddik, Mumbai, India

ABSTRACT

Regression testing makes sure that upgradeation of software in terms of adding new features or for bug fixing purposes should not hamper previously working functionalities. Whenever a software is upgraded or modified, a set of test cases are run on each of its functions to assure that the change to that function is not affecting other parts of the software that were previously running flawlessly. For achieving this, all existing test cases need to run as well as new test cases might be required to be created. It is not feasible to re-execute every test case for all the functions of a given software, because if there is a large number of test cases to be run, then a lot of time and effort would be required. This problem can be addressed by prioritizing test cases. Test case prioritization technique reorders the priority in which test cases are implemented, in an attempt to ensure that maximum faults are uncovered early on by the high priority test cases implemented first. In this paper we propose an optimized test case prioritization technique using Ant Colony Optimization (ACO) to reduce the cost, effort and time taken to perform regression testing and also uncover maximum faults. Comparison of different techniques such as Retest All, Test Case Minimization, Test Case Prioritization, Random Test Case Selection and Test Case Prioritization using ACO is also depicted.

KEYWORDS

Regression testing, Test case prioritization, Ant colony optimization.

1. INTRODUCTION

No matter how well a software is designed and tested before being released, it will eventually have to be modified for either adding new features or to fix the bugs in the current version. Regression testing is type of testing, which is performed on the improved build of the software so as to verify that the bug fixes or the added functionalities do not hamper the originally flawless functionalities of the software [1].

Regression testing is the process of verifying the modified software to detect whether new errors have been introduced into previously flawless code and to provide confidence that modifications are correct [2]. But, running all the regression test cases again for every iteration of the software is an expensive and time consuming process. To solve the problem of this “Retest All” approach, we have various techniques such as Test case selection, Test Case Prioritization and the Hybrid approach.

The various regression testing techniques are depicted in Figure 1 [3].
Figure 1. Regression Testing Techniques

a) Retest All: This is simply re-execution of all the test cases in the test suite of a particular function. It is the ideal technique, but however, can consume a lot of time.

b) Regression Test Case Selection: Instead of re-executing the entire test suite, it is logical to select a certain part of the test suite to be run. Test cases selected can be categorized as I) Reusable Test Cases II) Obsolete Test Cases.

Reusable Test cases can be used in succeeding regression cycles.

Obsolete Test cases can't be used in succeeding cycles.

c) Prioritization of Test Cases: In this technique we prioritize the list of test cases, enabling test cases with higher priority, selected on the basis of some predefined criterion, to be executed before the test cases with a lower priority, to meet the required performance goal. So test case prioritization technique does not discard any test case hence avoiding the drawback of test case minimization technique.

d) Hybrid Approach: This technique is a Hybrid Approach of both Regression Test Selection and Test Case Prioritization. So as to achieve the advantages of both.

Various prioritization criteria may be applied to the regression test suite with the objective of meeting the required criteria. Test cases can be prioritized in terms of random, optimal, total statement coverage, failure rates, total branch coverage or total fault exposing potential of the test cases [4]. Many techniques have been implemented for prioritizing the test cases according to one or more chosen criteria to meet the requirement. In this paper we make use of ACO to prioritize the given test cases. ACO is a process based on real life of ants, precisely on their food searching process.

1.1 Concept of Regression Testing

Regression testing is performed whenever any modifications made to the software, to provide confidence that the software behaves correctly and the modifications have not impacted the previously flawless functions and the quality of the software. A regression test is intended to provide a general assurance that enhancement or defect fixes in the software or its environment do not impact the previously working functionalities of the software.
Regression test suites are often simply the test cases that had been previously developed, and that have been saved so that they can be used later to perform regression testing [4]. It is an important and necessary activity as it maintains the quality of modified software systems. Many regression testing techniques have been proposed that are Test Selection, Test Minimization and Test Prioritization. The various regression testing techniques can be seen in Figure 2.

1.2 Activities Performed During Regression Testing

a) Test case selection: Test selection chooses the test cases that are relevant for a specific part of the application or for the performed maintenance operation.
b) Test case minimization: Test minimization reduces the number of test cases to be executed by removing redundant test cases, thus preserving the capability of the suite in discovering faults.
c) Test case prioritization: Test case prioritization determines the execution order of test cases that maximizes the probability of early discovering faults [5]. Minimization of test cases may require discarding of few test cases whereas prioritization of test cases gives an execution order of test cases such that all higher priority test cases are executed based on the selection criteria thereby reducing the probability of degradation in software components quality.

2. Test Case Prioritization

Test case prioritization involves the arrangement of test cases based on some specific criteria. The technique emphasizes on prioritizing the test cases and enhancing the probability of meeting the specified goals within assigned time and cost which was not effectively achievable by non-prioritized test cases.

Test case prioritization is used to achieve various objectives as follows:

a) Software developers/testers intend to increase the rate of fault detection.
b) Early detection of the faults that are more risky in testing life cycle.
c) To increase the reliability of the system [6].
2.1 Test Case Prioritization Techniques

In test case prioritization technique test cases that cover maximum faults in a particular function of the software are selected. Test cases with minimum execution time and maximum usage are assigned high priorities. The objective of this technique is to test the functionalities of a component in minimum time so as to reduce efforts and cost of testing.

The nine different test case prioritization techniques are as follows [7]:

a) No prioritization: One prioritization technique that we consider is simply the application of no technique; that is the test cases are untreated.
b) Random prioritization: Here we apply random prioritization, in which we randomly order the tests in a test suite.
c) Optimal prioritization: We can determine, for any test suite, which test cases expose which faults of a system, and thus we can determine an optimal ordering of test cases for maximizing rate of fault detection.
d) Total branch coverage prioritization: We can determine, for any test case, the number of decisions (branches) in that program that were exercised by that test case. We can prioritize these test cases according to the total number of branches they cover simply by sorting them in order of total branch coverage achieved.
e) Additional branch coverage prioritization: Additional branch coverage prioritization prioritizes test case in order of coverage of branches not yet covered.
f) Total fault-exposing-potential (FEP) prioritization: prioritize in order of total probability of exposing faults.
g) Additional fault-exposing-potential (FEP) prioritization: Prioritize test case in order of total probability of exposing faults, adjusted to consider effects of previous tests.
h) Total statement coverage prioritization: Prioritize in order of coverage of statements.
i) Additional statement coverage prioritization: Prioritize in order of coverage of statements not yet covered.

2.2 Algorithms for Test Case Prioritization

Several researchers have been working on developing algorithms for prioritization of test cases. Some of which are listed below:

a) Greedy Algorithm
b) Additional Greedy Algorithm
c) Optimal Algorithm
d) Hill Climbing Algorithm
e) Genetic Algorithm
f) PORT version1.1
g) Ant Colony optimization [3].

3. PROPOSED SYSTEM

Ant colony optimization technique is used for prioritizing the test cases.

3.1 Ant colony optimization

Ants are blind and small in size and still are able to find the shortest route to their food source. They make the use of antennas and pheromone liquid to be in touch with each other. Ant Colony Optimization (ACO) inspired from the behaviour of live ants on a food hunt is an optimal path
searching technique, capable of synchronization with searching solutions for local problem by maintaining array list to maintaining previous information gathered by each ant [8].

While on food hunt ants follow a certain path. A chemical substance called “pheromone” is left behind by each ant following the path. The ants that follow similar path are able to trace the path by smelling the odour of pheromone that the preceding ants left behind [9].

The optimal or shortest path discovery is done by observing teamwork and pheromone evaluation process. The possible random paths from ant hill to the food particle are shown in Figure 3 [9]. Ants take random paths and the pheromone is deposited on every path taken by the ants. However on their return ants will take the path which has more residual pheromone.

![Figure 3. Random path followed by ants](image)

Pheromone evaporation rate depends upon the path length. The longer the path the more will be pheromone evaporation. Hence the longer path has less residual pheromone.

![Figure 4. Shorter path followed by ants](image)

As seen in Figure 4 [8], remaining succeeding ants will take a shorter path which has more residual pheromone left behind.

3.2 Working of Proposed System

During software testing, testers often encounter time and budget constraints which make it difficult to rerun all test cases. To avoid this situation, we prioritize the given test cases to meet the specified goals within stipulated time and cost. The main focus of this paper is to optimize the list of test cases to make regression testing efficient. To achieve this objective, inputs are taken in the form of matrices and sorted by taking several factors into consideration. Thus the efficiency of the test will be optimized by reducing the time required to perform the test, reducing the cost required to conduct the test, increasing the productivity of the tester and the software and uncovering maximum number of faults.

The basic processing of Test Case Prioritization can be seen in figure 5.
Figure 5 shows the input requirements and the output produced using the proposed technique. The input to the system consists of the following matrices:

a) Fault test case matrix: The no. of faults uncovered by a test case for each requirement of a project.
b) Execution time test case matrix: The time taken by each test case to complete its execution.
c) Usage matrix: The no. of times a test case has been used to uncover faults from each requirement.
d) Pheromone matrix: The pheromone values associated with each test case.

The proposed technique uses the above matrices as inputs, applies the algorithm based on fault coverage and produces a pool of prioritized test cases. Prioritization is done so that all faults are detected and cost of execution is minimum.

Output from the system will consist of the following:

a) Updated pheromone matrix.
b) Best path of test case with execution time.
c) Prioritized test cases.

3.3 Proposed Algorithm

The basic steps of the proposed technique applied to test case prioritization are shown in the form of flow chart in Figure 6. A pool of test cases along with their fault covered in each requirement, execution time, probability of usage and pheromone value is taken as input.

Initially a test case is chosen that covers maximum faults. Since the aim is to uncover all faults, thus it is checked if all faults are covered by it or not. If all faults are not covered, then choose the next test case that covers the remaining faults and repeat this until all faults have been covered. Once all faults are covered, calculate the total number of faults covered by each test case which is stored in total fault test case matrix. This procedure will lead to many combinations of test case called paths that cover all faults.
Many such paths are explored while iterating and the best path from all paths explored is selected. The pheromone value is updated on the best path selected. The selection of best path is based on minimum execution time, maximum probability of usage and highest pheromone value.
4. RESULT

The results are divided into two sections viz, output section and comparative analysis. In the output section inputs and outputs screenshots are displayed which are obtained after application of the proposed technique. A comparative analysis is done in second section of results in which various techniques are compared based on different factors such as their execution time, risk, fault detection, representative set, cost etc.

4.1 Output

The output of the system will be generated by taking all the input matrices and prioritizing them based on several criterion. Five test cases along with their faults uncovered, execution time, usage and its corresponding pheromone value is taken as input. The contents of various input files are shown below.

Table 1 shows the Fault test case matrix. The faults that a certain test case uncovers are marked as ‘✓’.

<table>
<thead>
<tr>
<th>Test Case</th>
<th>Fault 1</th>
<th>Fault 2</th>
<th>Fault 3</th>
<th>Fault 4</th>
<th>Fault 5</th>
<th>Fault 6</th>
</tr>
</thead>
<tbody>
<tr>
<td>TC 01</td>
<td>✓</td>
<td></td>
<td></td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>TC 02</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TC 03</td>
<td>✓</td>
<td></td>
<td></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>TC 04</td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TC 05</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
<td>✓</td>
</tr>
</tbody>
</table>

From Table 1 it can be easily concluded that maximum faults are covered by TC 03 and TC 01 covers the faults that are left to be covered.

A test case’s script length determines its execution time. For each Test case, its execution time is stored in the Execution time test case matrix.

Table 2 displays the Execution time test case matrix.

<table>
<thead>
<tr>
<th>Test Case</th>
<th>Execution Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>TC 01</td>
<td>7</td>
</tr>
<tr>
<td>TC 02</td>
<td>5</td>
</tr>
<tr>
<td>TC 03</td>
<td>4</td>
</tr>
<tr>
<td>TC 04</td>
<td>4</td>
</tr>
<tr>
<td>TC 05</td>
<td>5</td>
</tr>
</tbody>
</table>

The number of times a specific test case is used is displayed in the usage matrix shown in Table 3 below.
Table 3. Usage matrix

<table>
<thead>
<tr>
<th>Test Case</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>TC 01</td>
<td>3</td>
</tr>
<tr>
<td>TC 02</td>
<td>2</td>
</tr>
<tr>
<td>TC 03</td>
<td>4</td>
</tr>
<tr>
<td>TC 04</td>
<td>2</td>
</tr>
<tr>
<td>TC 05</td>
<td>3</td>
</tr>
</tbody>
</table>

Table 3 concludes that maximum times used test cases are TC 03, TC 05 and TC 01.

Test cases and their respective pheromone values are displayed in the Pheromone matrix given below.

Table 4 represents the Pheromone matrix.

Table 4. Pheromone matrix

<table>
<thead>
<tr>
<th>Test Case</th>
<th>Pheromone Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>TC 01</td>
<td>0.42</td>
</tr>
<tr>
<td>TC 02</td>
<td>0.4</td>
</tr>
<tr>
<td>TC 03</td>
<td>1</td>
</tr>
<tr>
<td>TC 04</td>
<td>0.5</td>
</tr>
<tr>
<td>TC 05</td>
<td>0.6</td>
</tr>
</tbody>
</table>

Various combinations of test cases called paths are generated and the best path is selected based on which the updation of pheromone values is done. The best path or optimal path is selected by considering several factors such as pheromone value of a test case, probability of its usage and its average execution time.

The output screens are shown below:

1) Registration of Test Case Prioritization Application

Any user must first register in order to use the system. The registration fields are simply the personal details of the user.
2) Login of Test Case Prioritization Application

A registered user can login to the system and upload various files related to system that needs to be tested.

3) Input Data of Test Case Prioritization Application

Input data is taken in the form of various files i.e. Fault test case matrix, Execution time test case matrix, Usage matrix and Pheromone matrix.
4) Output of Test Case Prioritization Application

The output is divided into two parts. The first part shows a table that contains the list of prioritized test cases. The second part of output shows comparative analysis between non prioritized test cases and prioritized test cases on account of factors like Time taken for testing, Size of test cases, Cost of testing, Effort required and the APFD values associated with each test case.
4.2 Comparative Analysis

Table 5 shows comparative analysis between different regression testing techniques based on factors like Representative set, Execution time, Risk, Cost and Fault Detection.

Table 5. Fault Test Case matrix

<table>
<thead>
<tr>
<th>Approach</th>
<th>Representative Set</th>
<th>Execution Time</th>
<th>Risk</th>
<th>Cost</th>
<th>Fault Detection</th>
</tr>
</thead>
<tbody>
<tr>
<td>Retest All Approach</td>
<td>Strong RS</td>
<td>Slow</td>
<td>No</td>
<td>High</td>
<td>High</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>(Yet risk areas are taken care of)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Test Case Minimization</td>
<td>Strong RS</td>
<td>Fast</td>
<td>No</td>
<td>Low</td>
<td>High</td>
</tr>
<tr>
<td></td>
<td>(If based on specifications)</td>
<td></td>
<td></td>
<td></td>
<td>(If based on specifications)</td>
</tr>
<tr>
<td>Test Case Prioritization</td>
<td>Weak RS</td>
<td>Fast</td>
<td>Yes</td>
<td>Low</td>
<td>Lowest</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>(If based on risk exposure)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Table 6 shows the analysis of existing system based on time, cost, size, effort and APFD of untreated test case.

Table 6. Analysis of existing application

<table>
<thead>
<tr>
<th>Sr.No.</th>
<th>Principles</th>
<th>Poor</th>
<th>Average</th>
<th>Good</th>
<th>Very Good</th>
<th>Excellent</th>
</tr>
</thead>
<tbody>
<tr>
<td>i)</td>
<td>Time</td>
<td></td>
<td>*</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ii)</td>
<td>Cost</td>
<td></td>
<td>*</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>iii)</td>
<td>Size</td>
<td></td>
<td>*</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>iv)</td>
<td>Effort</td>
<td></td>
<td>*</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>v)</td>
<td>APFD</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>*</td>
</tr>
</tbody>
</table>

Table 7 shows the analysis of proposed system based on time, cost, size, effort and APFD of prioritized test case.

Table 7. Analysis of proposed system

<table>
<thead>
<tr>
<th>Sr.No.</th>
<th>Principles</th>
<th>Poor</th>
<th>Average</th>
<th>Good</th>
<th>Very Good</th>
<th>Excellent</th>
</tr>
</thead>
<tbody>
<tr>
<td>i)</td>
<td>Time</td>
<td></td>
<td></td>
<td></td>
<td>*</td>
<td></td>
</tr>
<tr>
<td>ii)</td>
<td>Cost</td>
<td></td>
<td>*</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>iii)</td>
<td>Size</td>
<td></td>
<td>*</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>iv)</td>
<td>Effort</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>*</td>
</tr>
<tr>
<td>v)</td>
<td>APFD</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>*</td>
</tr>
</tbody>
</table>

5. CONCLUSION

The proposed technique will optimize test cases for specific components of a software by automatically generating prioritized list of test cases to increase the efficiency of regression testing. Ant colony optimization (ACO) technique is used for prioritization and optimization of test cases. The objective of uncovering maximum faults in the functions of a software will be achieved by prioritization of test cases thereby minimizing effort, time and cost of testing. Application of this algorithm will lead to results that are optimal and reduce testers’ effort and time complexity.

6. FUTURE SCOPE

Currently our system only deals with creating an optimized list of test cases from an already available pool of test cases.
a) Furthermore, the system can be extended to automatically perform the regression test after creating a list of prioritized test cases. So as to partially or to some extent completely eliminate the need of human intervention in the testing process.

b) The accuracy of the system can be increased, if we could somehow estimate that what test cases would prove to be more important in the future, then their priorities could be pushed up. This can also be called as an ideal or the most optimal scenario for testing. But practically, as of now, it is impossible to achieve it, as it is not possible to estimate, in advance, that which test case might prove to be most important in the future.

c) Also the system can be extended to smartly create or add its own test cases to the pool of test cases, if required.

ACKNOWLEDGEMENT

Our thanks to M.H. Saboo Siddik College of Engineering, Department of Computer Engineering, for giving us the initiative to do constructive work. We also thank anonymous reviewers for their constructive suggestions.