Test Case Prioritization Through Efficient Mutation Analysis Using Water Droplet Algorithm

Prerna Sharma
M. Tech. Scholar
Department of Information Technology
Rajasthan College of Engineering For Women, Jaipur.

Vineet Todwal
Associate Professor
Department of Information Technology
Rajasthan College of Engineering For Women, Jaipur.

Abstract

Software Testing is one of the most critical issues in the Software Development Life Cycle. A significant portion of software budget is spent on testing and for human rated software; this budget is many-fold as compared to overall software development budget. Moreover, testing does not only relate to the Software Development Process before its delivery, rather it is done in various forms even after the software is delivered to the customers. Testing in compact form is done while the development of subsequent versions of the software. This form of testing is called Regression Testing. It is a type of software testing that seeks to uncover new software errors, or regressions, in existing software systems after changes like enhancements, patches or configuration changes, have been made to them, when new versions are released of the same software. One of the most important aspect of regression testing is Test Case Prioritization. This is because it is not feasible to run entire suite of test cases before release of every new version of the program. Test Case Prioritization relates with tabulating the effectiveness of test suites so as to define an order in which test cases are to be applied to the current system. Prioritization techniques schedule test cases for execution in an order that attempts to maximize some objective function. A variety of objective functions are applicable; one such function involves rate of fault detection—a measure of how quickly faults are detected within the testing process. An improved rate of fault detection during regression testing can provide faster feedback on a system under regression test and let debuggers begin their work earlier than might otherwise be possible. A measure of effectiveness of the Test Case for some objective function can be made using mutant testing. Mutants of programs are program versions with known faults. Mutation testing is one of the most powerful approaches for evaluating quality of test cases. However, mutation testing is a heuristic procedure and is also one of the most expensive testing approaches. Thus, mutation testing need to be augmented with suitable optimization techniques to reduces the cost of the testing operation on subsequent versions of the software. In this paper, Intelligent Water Drop Algorithm is used to prioritize test cases based on the heuristic of killing probability of test cases on mutants. Simulation is done on empirical data of Mozilla Firefox through MATLAB and results are analyzed. The proposed system provides an improvement over existing Genetic Algorithm based Test case prioritization as it gives a relatively global optimal solution as compared to near optimal solution given by genetic algorithm based test case prioritization.

Keywords: Regression Testing, Mutation Testing, Beta Testing, Test Case Prioritization, Intelligent Water Drop Algorithm, Genetic Algorithm

I. INTRODUCTION

A. Introduction to Test Case Prioritization:

Software developers usually save the test suites they develop for their software, so that they can reuse those suites later as the software evolves. Such test suite reuse, in the form of regression testing, is pervasive in the software industry and, together with other regression testing activities, can account for as much as one-half of the cost of software maintenance. Running all test cases of an existing test suite can consume a large amount of time. For example, for some of the software of social networking and Multi-Level Marketing, running entire suite of test cases would require about 8 weeks. In such cases, it is recommended to order the sequence of test cases [1], and dropping the not-necessary test cases, known as test case reduction, so that those test cases with the highest priority, according to some criterion, are run first.

Test Case prioritization techniques attempts to schedule test cases for Regression Testing [2] in an order so as to maximize some identified objective function. For example, it might be needed to schedule test cases in an ordering that achieve code coverage at the fastest rate possible, exercises features in order of expected frequency of use, or exercises subsystems in an order that reflects their past history to fail. If, for any software, the time required to execute all test cases in a test suite is small, test case prioritization may not be cost effective. It is sufficient simply to schedule test cases in any order. However, when the time required to run all test cases in the test suite is sufficiently long [3], test case prioritization may be beneficial.
A large number of techniques exists for prioritizing test cases and one can empirically evaluate their ability related with rate of fault detection—a measure of how quickly faults are detected within the testing process. A fast rate of fault detection during regression testing can provide earlier analysis on a system under regression test and let developers start debugging and correcting bugs earlier than might otherwise be possible. As the time to test and release is a critical constraint of modern business strategies, Test Case Prioritization is important as it can significantly improve the rate of fault detection of test suites. However, there exist a tradeoffs between various prioritization techniques.

Test Case Prioritization techniques schedule test cases in an order of execution in accordance to some criterion. The purpose of this prioritization is to increase the probability that if the test cases are used for regression testing in the given sequence, they will more closely meet some objective than if they were executed in some other order. Test case prioritization can be used to meet a wide variety of objectives, including the following:

- One may wish to increase the rate of fault detection—i.e., the probability of revealing faults earlier in an execution of regression tests [4].
- One may wish to increase the rate of detection of high-risk faults, locating those faults earlier in the testing process.
- One may wish to increase the likelihood of revealing regression errors related to specific code changes earlier in the process of regression testing [5].
- One may wish to increase the coverage of coverable code in the system under test at a faster rate [6].
- One may wish to increase his/her confidence in the reliability and robustness of the system under test at a faster rate [7].

In practice, and depending upon the choice of objective function, test case prioritization problem may be intractable for certain objectives, an efficient solution to the problem would provide an efficient solution to the Knapsack problem [8]. Thus, test case prioritization techniques are typically heuristics. The goal of this work is to investigate, for a specific objective function, namely, maximum fault exposing potential, Intelligent Water Drops (IWD) technique [9].

Given a particular objective, various prioritization criteria may be applied to a test suite with the aim of meeting that objective. For example, to attempt to meet the first objective stated above, one may prioritize test cases in terms of the failure rates, measured historically, of the modules that needs to be exercised. Alternatively, one may prioritize test cases in terms of increasing cost-per-coverage of code components, or in terms of increasing cost-per-coverage of features listed in a some requirements specification. In any case, the motive behind the choice of a prioritization criterion is to increase the probability that the prioritized test suite can meet the objective in better way than would an ad hoc or random ordering of test cases. In this work, the focus is on the first objective listed above, i.e., increasing the likelihood of revealing faults earlier in the testing process. This objective is informally defined as one of improving the test suite’s rate of fault detection. The motivation for meeting this objective is that an improved rate of fault detection during regression testing can provide faster feedback on the system under test [10], or early evidence that the quality goals have not been met. It can also let developers and debuggers begin their work earlier than might otherwise be possible. There are broadly nine different test case prioritization techniques as mentioned below:

1) **No Prioritization:**
This category refers to a trivial case in which no technique is applied to prioritize test case. It provides a base to measure the effectiveness of other prioritization techniques.

2) **Random Prioritization** [11]:
This prioritization is also used to facilitate empirical study. Random prioritization refers to the prioritization technique in which test cases are executed in random order.

3) **Optimal Prioritization** [12]:
To measure the effects of prioritization techniques on rate of fault detection, there is a need of programs that contain known faults. one can determine, for any test suite, which test cases expose which faults, and thus, can determine an optimal ordering of test cases in a test suite for maximizing that suite’s rate of fault detection. This is not a common practical technique, as it requires knowledge of which test cases will expose which faults; however, by using it one gain insight into the success of other practical heuristics.

4) **Total Branch Coverage Prioritization** [13]:
By analyzing a program, one can determine, for any test case, the number of decisions (branches) in that program that were exercised by that test case. One 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. This prioritization can thus be accomplished in time $O\log(n)$ for programs containing $n$ branches.

5) **Additional Branch Coverage Prioritization:**
Total branch coverage prioritization schedules test cases in the order of total coverage achieved. However, having executed a test case and covered certain branches, more may be gained in subsequent test cases by covering branches that have not yet been covered. Additional branch coverage prioritization iteratively selects a test case that yields the greatest branch coverage, then adjusts the coverage information on subsequent test cases to indicate their coverage of branches not yet covered, and then repeats this process, until all branches covered by at least one test case have been covered. Having scheduled test cases in this fashion, one may be left with additional test cases that cannot add additional branch coverage. It could be beneficial to order the remaining test cases using total branch coverage prioritization. Because additional branch coverage prioritization requires
recalculation of coverage information for each un-prioritized test case following selection of each test case, its cost is $O(n^2)$ for programs containing $n$ branches.

6) *Total Statement Coverage Prioritization:*
Total statement coverage prioritization is the same as total branch coverage prioritization, except that test coverage is measured in terms of program statements rather than decisions.

7) *Additional Statement Coverage Prioritization:*
Additional statement coverage prioritization is the same as additional branch coverage prioritization, except that test coverage is measured in terms of program statements rather than decisions. With this technique too, we require a method for prioritizing the remaining test cases after complete coverage has been achieved, and in this work we do this using total statement coverage prioritization.

8) *Total Fault-Exposing-Potential (FEP) Prioritization [14]:*
Statement- and branch-coverage-based prioritization consider only whether a statement or branch has been exercised by a test case. This consideration may mask a fact about test cases and faults: the ability of a fault to be exposed by a test case depends not only on whether the test case reaches (executes) a faulty statement, but also, on the probability that a fault in that statement will cause a failure for that test case. Although any practical determination of this probability must be an approximation, we wished to determine whether the use of such an approximation could yield a prioritization technique superior in terms of rate of fault detection than techniques based on simple code coverage. To obtain an approximation of the fault-exposing potential (FEP) of a test case, we use mutation analysis. Given program $P$ and test suite $T$, for each test case $t \in T$, for each statement $s$ in $P$, one can determine the mutation score $ms(s,t)$ to be the ratio of mutants of $s$ exposed by $t$ to total mutants of $s$. One can, then calculate, for each test case $tk$ in $T$, an award value for $tk$, by summing all $ms(s, tk)$ values. Total fault-exposing-potential prioritization orders the test cases in a test suite in order of these award values. Such a technique could conceivably be much more expensive to implement than a code-coverage-based technique, however if such a technique shows promise, this might motivate a search for cost-effective methods to approximate fault-exposing potential.

9) *Additional Fault-Exposing-Potential (FEP) Prioritization:*
Analogous to the extensions made to total branch (or statement) coverage prioritization to additional branch (or statement) coverage prioritization, one can extend total FEP prioritization to create additional fault-exposing-potential (FEP) prioritization. This lets us account for the fact that additional executions of a statement may be less valuable than initial executions. In additional FEP prioritization, after selecting a test case $0$, The award values for all other test cases that exercise statements exposed by $0$ are lowered.

B. *Problem Statement:*
This work focused on the consideration of test case prioritization techniques with the objective being maximum fault exposing potential. Mutant based testing is primarily used to obtain the effectiveness of test cases in exposing known faults. Based on the version information and historical aspects of effectiveness of test cases against known faults as heuristics, a probabilistic estimate is made for test cases and mutants testing. A mutant is said to be killed if the test case detects a fault. On the other hand, a mutant is said to be survived if the test case fails to find any defect in it. The problem is to find the optimal sequence of application of the test cases so that the average percentage of Faults Detected (APFD) [15] over the lifetime of the test suite is to be maximized.

C. *Motivation:*
Regression Testing is one of the most important aspect related with the software evolution through its various versions over time. However, it is generally not possible to apply entire suite of test case over any new version of the software being developed, adding certain enhancements or patches. Nevertheless, the new version is to be tested for certain features which are modified or code segments which are altered. Thus, an optimum set of test cases from those that are available, needs to be used so as to minimize time and efforts that is to be made if entire test suite is to be used. This selection of optimal subset of test cases, considering the changes in the new version and the ordering of the test case to be executed on the enhanced software is an important aspect of regression testing. These techniques comes under the heading of Test Case Reduction and Test Case Prioritization and forms a increasingly important discipline in software engineering of modern times. Most modern software, spanning from web clients like Mozilla Firefox, Internet Explorer, etc. to specific application based software products like Adobe Photoshop evolves through its various versions released over time [16]. Thus, Test Case Prioritization forms an important aspect and is the need of modern software development techniques.

II. *Research Approach*
The approach for test case prioritization used in this paper is the one based on Mutation testing. A mutant is a version of program with a manually injected fault, and is created with a viewpoint to find an estimate of fault exposing effectiveness of a test case. On the basis of mutant killing probability, which in turn is based on certain heuristics based on historical data, a tabular data can be made regarding the probabilities of mutant killing and test cases. This problem then is similar to the Knapsack problem in which one has to choose the maximum value items, given a limit on the capacity of weight that can be permitted.
Intelligent water Drops algorithm is used to solve this problem in iteration steps. At each iteration step, Intelligent Water Drops follow multiple paths out of which the iteration best path is selected, and added to the total best path. A specified number of iterations are followed and the iteration best path is selected and added to the total best path, thereby gives the optimal sequence of test case execution.

### III. PROPOSED WORK

**A. Mutation Testing:**
Mutation Testing is a fault-based testing technique which provides a testing criterion called the “mutation adequacy score”. The mutation adequacy score can be used to measure the effectiveness of a test set in terms of its ability to detect faults. The general principle underlying Mutation Testing work is that the faults used by Mutation Testing represent the mistakes that programmers often make. By carefully choosing the location and type of mutant, we can also simulate any test adequacy criteria. Such faults are deliberately seeded into the original program, by simple syntactic changes, to create a set of faulty programs called mutants, each containing a different syntactic change. To assess the quality of a given test set, these mutants are executed against the input test set. If the result of running a mutant is different from the result of running the original program for any test cases in the input test set, the seeded fault denoted by the mutant is detected. One outcome of the Mutation Testing process is the mutation score, which indicates the quality of the input test set. The mutation score is the ratio of the number of detected faults over the total number of the seeded faults. The traditional process of mutation analysis is illustrated in Figure 2. In mutation analysis, from a program p, a set of faulty programs p' called mutants, is generated by a few single syntactic changes to the original program p. As an illustration, Table II shows the mutant p', generated by changing the and operator (&&) of the original program p, into the or operator (||), thereby producing the mutant p':

<table>
<thead>
<tr>
<th>Program p</th>
<th>Mutant p'</th>
</tr>
</thead>
<tbody>
<tr>
<td>...........</td>
<td>...........</td>
</tr>
<tr>
<td>if (a &gt;0 &amp;&amp; b&gt;0)</td>
<td>if (a &gt;0</td>
</tr>
</tbody>
</table>

Fig. 3.1: Example of a program and a possible mutant

A transformation rule that generates a mutant from the original program is known as a mutation operator. Fig contains only one example of a mutation operator; there are many others. Typical mutation operators are designed to modify variables and expressions by replacement, insertion or deletion operators.

In the next step, a test set T is supplied to the system. Before starting the mutation analysis, this test set needs to be successfully executed against the original program p to check its correctness for the test case. If p is incorrect, it has to be fixed before running other mutants. If the result of running p' is different from the result of running p for any test case in T, then the mutant p' is said to be ‘killed’, otherwise it is said to have ‘survived’. After all test cases have been executed, there may still be a few ‘surviving’ mutants. To improve the test set T, the program tester can provide additional test inputs to kill these surviving mutants. However, there are some mutants that can never be killed, because they always produce the same output as the original program. These mutants are called Equivalent Mutants. They are syntactically different but functionally equivalent to the original program. Automatically detecting all equivalent mutants is impossible [35], [187], because program equivalence is un-decidable.

The equivalent mutant problem has been a barrier that prevents Mutation Testing from being more widely used. Mutation Testing concludes with an adequacy score, known as the Mutation Score, which indicates the quality of the input test set. The mutation score (MS) is the ratio of the number of killed mutants over the total number of non-equivalent mutants. The goal of mutation analysis is to raise the mutation score to 1, indicating the test set T is sufficient to detect all the faults denoted by the mutants.

Although Mutation Testing is able to effectively assess the quality of a test set, it still suffers from a number of problems. One problem that prevents Mutation Testing from becoming a practical testing technique is the high computational cost of executing the enormous number of mutants against a test set.

**B. Intelligent Water Droplet (IWD) Algorithm:**
The natural systems that have developed for so long are one of the rich sources of inspiration for inventing new intelligent systems. Swarm intelligence is one of the scientific fields that are closely related to natural swarms existing in nature, such as ant colonies, bee colonies, brain and rivers. A natural river often finds good paths among lots of possible paths in its ways from the source to destination. These near optimal or optimal paths are obtained by the actions and reactions that occur among the water drops and the water drops with the riverbeds. The intelligent water drops (IWD) algorithm is a new swarm-based optimization algorithm inspired from observing natural water drops that flow in rivers.
The IWD algorithm is based on the dynamic of river systems, actions and reactions that happen among the water drops in rivers. The natural water drops are used to develop IWD and the IWDs cooperate together to reach a better solution for a given problem. The IWD algorithm may be used for maximization or minimization problems. The solutions are incrementally constructed by the IWD algorithm. Therefore, the IWD algorithm is a population-based constructive optimization algorithm. In the IWD algorithm, IWDs are created with two main properties:

1) velocity
2) soil

Both of the two properties may change during the lifetime of an IWD. An IWD flows from a source to a destination. The IWD begins its trip with an initial velocity and zero soil. During its trip, it travels in the environment from which it removes some soil and it may gain some speed. An IWD is supposed to flow in discrete steps. From its current location to its next location, the IWD velocity is increased by the amount non-linearly proportional to the inverse of the soil between the two locations. Therefore, a path with less soil lets the IWD become faster than a path with more soil. An IWD gathers soil during its trip in the environment. This soil is removed from the path joining the two locations. The amount of soil added to the IWD is non-linearly proportional to the inverse of the time needed for the IWD to pass from its current location to the next location. The time taken is proportional to the velocity of the IWD and inversely proportional to the distance between the two locations. Moreover, those parts of the environment that are used with more IWDs will have less soil. It may be said that soil is the source material of information such that the environment and water drops both have memories for soil.

An IWD needs a mechanism to select the path to its next location or step. In this mechanism, the IWD prefers the paths having low soils to the paths having high soils. This behavior of path selection is implemented by imposing a uniform random distribution on the soils of the available paths. Then, the probability of the next path to select is inversely proportional to the soils of the available paths. Therefore, paths with lower soils have higher chance to be selected by the IWD.

The IWD algorithm gets a representation of the problem in the form of a graph \( (N, E) \) with the node set \( N \) and edge set \( E \). Then, each IWD begins constructing its solution gradually by travelling on the nodes of the graph along the edges of the graph until the IWD finally completes its solution. One iteration of the algorithm is complete when all IWDs have completed their solutions. After each iteration, the iteration-best solution \( TIB \) is found and it is used to update the total-best solution \( TTB \). The amount of soil on the edges of the iteration-best solution \( TIB \) is reduced based on the goodness (quality) of the solution. Then, the algorithm begins another iteration with new IWDs but with the same soils on the paths of the graph and the whole process is repeated. The algorithm stops when it reaches the maximum number of iterations \( \text{itermax} \) or the total-best solution \( TTB \) reaches the expected quality. The IWD algorithm has two kinds of parameters. One kind is those that remain constant during the lifetime of the algorithm and they are called ‘static parameters’. The other kind is those parameters of the algorithm, which are dynamic and they are reinitialized after each iteration of the algorithm.

The IWD algorithm is specified in the following steps:

1) Initialization of static parameters. The graph \( (N, E) \) of the problem is given to the algorithm. The quality of the total-best solution \( TTB \) is initially set to the worst value:

\[
g(TTB) = -\infty
\]

The maximum number of iterations \( \text{itermax} \) is specified by the user. The iteration count \( \text{itercount} \) is set to zero.

The number of water drops \( NIWD \) is set to a positive integer value, which is usually set to the number of nodes \( Nc \) of the graph.

For velocity updating, the parameters are \( av = 1 \), \( bv = .01 \) and \( cv = 1 \). For soil updating, \( as = 1 \), \( bs = .01 \) and \( cs = 1 \). The local soil updating parameter is \( \rho_n \), which is a small positive number less than one, is set as \( \rho_n = 0.9 \). The global soil updating parameter \( \rho_{IWD} \), which is chosen from \([0, 1]\), is set as \( \rho_{IWD} = 0.9 \). Moreover, the initial soil on each path (edge) is denoted by the constant InitSoil such that the soil of the path between every two nodes \( i \) and \( j \) is set by soil\((i, j) = \text{InitSoil} \). The initial velocity of each IWD is set to InitVel. Both parameters InitSoil and InitVel are user selected and they should be tuned experimentally for the application.

2) Initialization of dynamic parameters. Every IWD has a visited node list \( Vc \) (IWD), which is initially empty: \( Vc \) (IWD) = \( \{ \} \). Each IWD’s velocity is set to InitVel. All IWDs are set to have zero amount of soil.

3) Spread the IWDs randomly on the nodes of the graph as their first visited nodes.

4) Update the visited node list of each IWD to include the nodes just visited.

5) Repeat Steps 5.1 to 5.4 for those IWDs with partial solutions.

(1) For the IWD residing in node \( i \), choose the next node \( j \), which does not violate any constraints of the problem and is not in the visited node list \( Vc \) (IWD) of the IWD, using the following probability \( p_{IWD}(j) \):

\[
p_{IWD}(j) = \frac{f(\text{soil}(i,j))}{\sum_{k \in Vc(IWD)} f(\text{soil}(i,k))}
\]

Such that;

\[
f(\text{soil}(i,j)) = \frac{1}{\varepsilon_0 + g(\text{soil}(i,j))}
\]
And
\[
g(soil(i,j)) = \begin{cases} 
    soil(i,j) & \text{if } \min_{l \in \text{vc}(IWD)} (soil(i,l)) \geq 0 \\
    soil(i,j) - \min_{l \notin \text{vc}(IWD)} (soil(i,l)) & \text{otherwise}
\end{cases}
\]

Then, add the newly visited node \( j \) to the list \( \text{vc}(IWD) \).

2) For each IWD moving from node \( i \) to node \( j \), update its velocity \( \text{vel}^{IWD}(t) \) by
\[
\text{vel}^{IWD}(t + 1) = \text{vel}^{IWD}(t) + \frac{a_v}{b_v + c_v \cdot \text{soil}^2(i,j)}
\]
where \( \text{vel}^{IWD}(t + 1) \) is the update velocity of IWD.

3) For the IWD moving on the path from node \( i \) to \( j \), compute the soil \( \Delta \text{soil}(i,j) \) that the IWD loads from the path by
\[
\Delta \text{soil}(i,j) = \frac{a_s}{b_s + c_s \cdot \text{time}^2(i,j; \text{vel}^{IWD}(t + 1))}
\]
such that
\[
\text{time}(i,j; \text{vel}^{IWD}(t + 1)) = \frac{\text{HUD}(j)}{\text{vel}^{IWD}(t + 1)}
\]

where

Heuristic Undesirability \( \text{HUD}(j) \) is defined appropriately for the given

4) Update the soil, \( \text{soil}(i,j) \) of the path from node \( i \) to \( j \) traversed by that IWD and also update the soil that the IWD carries \( (\text{soil}^{IWD}) \):
\[
\text{soil}(i,j) = (1 - \rho_n) \cdot \text{soil}(i,j) - \rho_n \cdot \Delta \text{soil}(i,j)
\]
\[
\text{soil}^{IWD}(i,j) = \text{soil}^{IWD}(i) + \Delta \text{soil}(i,j)
\]

6) Find the iteration-best solution \( T^{IB} \) from all the solutions \( T^{IWD} \) found by the IWDs using
\[
T^{IB} = \arg \max_{T^{IWD}} q(T^{IWD})
\]
where function \( q(.) \) gives the quality of the solution.

7) Update the soils on the paths that form the current iteration-best solution \( T^{IB} \) by
\[
\text{soil}(i,j) = (1 + \rho^{IWD}) \cdot \text{soil}(i,j) - \rho^{IWD} \cdot \frac{1}{N^{IB} - 1} \cdot \text{soil}^{IWD} \text{ for all } (i,j) \in T^{IB}
\]
where \( N^{IB} \) is the number of nodes in the solution \( T^{IB} \).

8) Update the total best solution \( T^{TB} \) by the current iteration-best solution \( T^{IB} \) using
\[
T^{TB} = \begin{cases} 
    T^{TB} & \text{if } q(T^{TB}) \geq q(T^{IB}) \\
    T^{IB} & \text{otherwise}
\end{cases}
\]

9) Increment the iteration number by \( \text{Iter}^{count} = \text{Iter}^{count} + 1 \). Then, go to Step 2 if

\[ \text{Iter}^{count} < \text{Iter}^{max} \]

10) The algorithm stops here with the total-best solution \( T^{TB} \).

It is reminded that the IWD has been shown to have the property of convergence in value (Shah-Hosseini, 2008). It means that the IWD algorithm is able to find the optimal solution if the number of iterations be sufficiently large.

2) Proposed Mutation Analysis Using Intelligent Water Drops Algorithm:

To illustrate efficient mutation analysis using intelligent water drops algorithms, consider the following C program (p) to print the number pattern.

```c
#include<stdio.h>

int main()
{
    int n, c, d, num = 1, space;
    scanf("%d", &n);
    space = n - 1;
    for (d = 1; d <= n; d++)
    {
        num = d;
    }
```
Test Case Prioritization Through Efficient Mutation Analysis Using Water Droplet Algorithm

(IIJIRST/ Volume 1 / Issue 10 / 020)

for ( c = 1 ; c <= space ; c++ )
printf(" ");
space--;
for ( c = 1 ; c <= d ; c++ )
{
printf("%d", num);
num++;
}
num--;

for ( c = 1 ; c < d ; c++ )
{
printf("%d", num);
num--;
}
num--;
num--;
for ( c = 1 ; c < d ; c++ )
{
printf("%d", num);
num--;
}
num--;
}
printf("\n");
}
return 0;
}

Program Input : 5
Program Output :

1
2 3 2
2 4 5 4 3
4 5 6 7 6 5 4
5 6 7 8 9 8 7 6 5

The mutants that can be constructed of the program are:

<table>
<thead>
<tr>
<th>Mutant p1</th>
<th>Mutant p3</th>
</tr>
</thead>
<tbody>
<tr>
<td>for ( d = 1 ; d &lt; n ; d++ )</td>
<td></td>
</tr>
<tr>
<td>Statement no. 11</td>
<td></td>
</tr>
<tr>
<td>Mutant p3</td>
<td></td>
</tr>
<tr>
<td>---space;</td>
<td></td>
</tr>
<tr>
<td>Statement no. 18</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Mutant</th>
</tr>
</thead>
<tbody>
<tr>
<td>for ( c = 1 ; c &lt;= d ; ++c )</td>
</tr>
<tr>
<td>Statement no. 20</td>
</tr>
<tr>
<td>Mutant</td>
</tr>
<tr>
<td>printf(&quot;/n&quot;);</td>
</tr>
<tr>
<td>Statement no. 32</td>
</tr>
</tbody>
</table>

Fig.3.2: Four Mutants derived from the original program

The above four changes corresponds to the four mutated programs formed from the program given above.

Consider the following test suite for this simple program:

Test Suite = {Test Case#1, Test Case#2, Test Case#3, Test Case#4, Test Case#5}

<table>
<thead>
<tr>
<th>Test Case 1</th>
<th>Input</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test Case 2</td>
<td>Input</td>
</tr>
<tr>
<td>Test Case 3</td>
<td>Input</td>
</tr>
</tbody>
</table>

Test Case 1 : Input = 5
Test Case 2 : Input = 4.5
Test Case 3 : Input = 1

One can find the most promising sequence of these 3 test cases based on empirical evaluation of the fault exposing potential by analyzing their outcomes of the four program versions given above, with known faults, generated by inculcating syntactic error in the code segments. This can be done efficiently using intelligent water droplet algorithm as shown below:

| Table - 3.1 |
| Mutant Killing Score of Test Cases |
Therefore, in this very simple example, Test Case 1 and 2 performed equally well killing two mutants in each case. Test Case 3 killed only one mutant. Moreover, mutants 2 and 3 were not killed by any of the test case so these are identical mutants in context of this test suite. Thus, prioritization among these three test case may take the following form:

Test Case Prioritized Order : {Test Case 1},{Test Case 2},{Test Case 3}

or

Test Case Prioritized Order : {Test Case 2},{Test Case 1},{Test Case 3}

Consider N different Mutants with known faults and T different Test Cases (T>>N), and with certain probability distribution of mutant killing by the test cases. This is illustrated as shown:

<table>
<thead>
<tr>
<th>Test Case</th>
<th>Mutant P1 (Equivalent Mutant)</th>
<th>Mutant P2 (Equivalent Mutant)</th>
<th>Mutant P3 (Equivalent Mutant)</th>
<th>Mutant P4 (Equivalent Mutant)</th>
</tr>
</thead>
<tbody>
<tr>
<td>TC1</td>
<td>Killed</td>
<td>Survived</td>
<td>Survived</td>
<td>Killed</td>
</tr>
<tr>
<td>TC2</td>
<td>Killed</td>
<td>Survived</td>
<td>Survived</td>
<td>Killed</td>
</tr>
<tr>
<td>TC3</td>
<td>Survived</td>
<td>Survived</td>
<td>Survived</td>
<td>Killed</td>
</tr>
</tbody>
</table>

where P(tc1,m1) is the probability that mutant m1 is killed by test case TC1.

Mutation Testing concludes with an adequacy score, known as the mutation score, which is a measure of the quality of the input test set. The mutation score (MS) is the ratio of the number of killed mutants over the total number of non-equivalent mutants.

Consider a complete graph with T nodes, in which all nodes are connected to each other through edges. Label all the nodes from 1 to T in any manner. Assign each of the node, a test case using random distribution. The objective is to find an optimal sequence of test case so as to optimize fault exposing potential of the test suite. This problem can be formulated in terms of Intelligent Water Drop Algorithm in the following way:

Initially there are T IWDs and these are one on each of the node of the graph. IWDs may choose the path at which the fault exposing potential is maximum. At each iteration of the graph, certain TCs may get replaced by other TCs forming a new sequence of TCs starting from node 1 to node T. This is done in accordance with the IWD algorithm as stated below.

Let Prob(i,j) denotes the probability that for, i<j, node j replaces node i, so as to be more prior than node j. This probability is computed as:

After every iteration of the graph, each IWD forms a path through the nodes which is the Iteration best solution. Each iteration best solution contributes to the total best solution. Thus, the quality of the total best solution depends upon the number of iterations as specified under the given time cost constraints. Chapter 4 discusses the case study of Mozilla Firefox browser regression testing of beta versions and tabulates the result.

IV. ANALYSIS OF PROPOSED WORK

A. Analysis of Mozilla Firefox Testing:

Consider the following data deduced empirically based on Heuristics on the basis of history of test cases on the Fault Exposing Potentials for Mozilla Firefox version 3.5 beta.

<table>
<thead>
<tr>
<th>Test Case</th>
<th>Mutation Score Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>tc1</td>
<td>p1 = 0.120364, p2 = 0.551546, p3 = 0.965123, p4 = 0.143911, p5 = 0.943607, p6 = 0.412827, p7 = 0.675559, p8 = 0.331524</td>
</tr>
<tr>
<td>tc2</td>
<td>p1 = 0.141132, p2 = 0.228319, p3 = 0.374584, p4 = 0.121121, p5 = 0.755265, p6 = 0.709601, p7 = 0.841422, p8 = 0.481893</td>
</tr>
<tr>
<td>tc3</td>
<td>p1 = 0.350936, p2 = 0.839121, p3 = 0.614708, p4 = 0.40489, p5 = 0.797502, p6 = 0.635804, p7 = 0.202226, p8 = 0.660692</td>
</tr>
<tr>
<td>tc4</td>
<td>p1 = 0.29847, p2 = 0.665074, p3 = 0.020429, p4 = 0.780166, p5 = 0.853365, p6 = 0.467698, p7 = 0.549148, p8 = 0.829841</td>
</tr>
<tr>
<td>tc5</td>
<td>p1 = 0.183281, p2 = 0.789997, p3 = 0.910998, p4 = 0.046013, p5 = 0.207997, p6 = 0.617524, p7 = 0.782111, p8 = 0.189617</td>
</tr>
<tr>
<td>tc6</td>
<td>p1 = 0.071578, p2 = 0.829321, p3 = 0.961242, p4 = 0.095693, p5 = 0.453571, p6 = 0.430809, p7 = 0.91583, p8 = 0.352996</td>
</tr>
<tr>
<td>tc7</td>
<td>p1 = 0.156444, p2 = 0.071828, p3 = 0.823056, p4 = 0.146381, p5 = 0.194165, p6 = 0.524559, p7 = 0.057524, p8 = 0.2439</td>
</tr>
<tr>
<td>tc8</td>
<td>p1 = 0.63046, p2 = 0.071828, p3 = 0.654877, p4 = 0.108271, p5 = 0.967508, p6 = 0.38501, p7 = 0.285493, p8 = 0.46886</td>
</tr>
</tbody>
</table>
One can choose the optimal sequence of test cases based on IWD algorithm. Consider that initial order of running the test cases is random.

The Probability Distribution of all mutant cases is random.
The probability of all mutants getting survived is given by fig. 4.1.

Fig. 4.1: Probabilities of Mutant Killing

The Probability Distribution of mutant surviving throughout all the test cases is illustrated in figure 4.3.

Fig. 4.3: Probabilities of Mutant Survival throughout the execution of complete test suite.

The Objective Function focused through IWD is to maximize APFD by Prioritizing Test cases in certain order. This ordering is adjusted in every iteration of the algorithm in which the iteration best solution adds to the total best solution. The total best solution is accepted when it reaches certain limit or when the total number of iteration reaches a predefined number.

The solutions obtained by running IWD,s for test cases is depicted as shown:
Test case prioritization is a method to prioritize and schedule test cases. The technique is developed in order to run test cases of higher priority in order to minimize time, cost and effort during software testing phase. Many researchers propose many methods to prioritize and reduce the effort, time and cost in the software testing phase, such as test case prioritization methods, regression selection techniques and test case reduction approaches. This paper concentrates on test case prioritization techniques only. The work shows that there are many prioritization techniques focused on: (a) requirement-based techniques, (b) coverage-based techniques, (c) Fault Exposing Potential Techniques etc. This work reveals that there are many research challenges and gaps in the test case prioritization area. Those challenges and gaps can give the research direction in this field.

Existing test case prioritization techniques assume explicitly that there is only a single test suite. The test suite is a collection of a set of test cases. There are no prioritization techniques to resolve the problem of multiple test suites. Future work will focus on the problem of test case prioritization with multiple test case suites.

V. CONCLUSION AND FUTURE SCOPE

Test case prioritization is a method to prioritize and schedule test cases. The technique is developed in order to run test cases of higher priority in order to minimize time, cost and effort during software testing phase. Many researchers propose many methods to prioritize and reduce the effort, time and cost in the software testing phase, such as test case prioritization methods, regression selection techniques and test case reduction approaches. This paper concentrates on test case prioritization techniques only. The work shows that there are many prioritization techniques focused on: (a) requirement-based techniques, (b) coverage-based techniques, (c) Fault Exposing Potential Techniques etc. This work reveals that there are many research challenges and gaps in the test case prioritization area. Those challenges and gaps can give the research direction in this field.

Existing test case prioritization techniques assume explicitly that there is only a single test suite. The test suite is a collection of a set of test cases. There are no prioritization techniques to resolve the problem of multiple test suites. Future work will focus on the problem of test case prioritization with multiple test case suites.

REFERENCES


