Prioritizing Test Suites using Clustering Approach in Software Testing

Arvind Kumar Upadhyay, A. K. Misra

***Abstract* - Prioritizing test cases enables test suites to be scheduled in a manner that optimize the objective of reducing effort to test exhaustively. Of various techniques/methods available, a need is felt to further improve existing schemes. In this paper, we propose clustering based prioritization and support our effort with average percentage of fault detection (APFD) measure. We target the significant test suites to get priority. Our method can considerably help realization of overall clustering approach.**

Keywords: APFD

I. INTRODUCTION

Testing has a very vast application area; it may range from small subroutines to very large system applications having millions of statements. Now a day's software is written and used by the same organization [8]. Many organizations believe that independent software development and operation leads to better security and better testing. The sole objective of testing revolves around bugs' prevention. If test cases are designed suitably then it may help to achieve the ultimate goal of software quality [13]. Designing test cases is a challenging task. Starting from familiar predicates, testing uses predefined procedures and has predictable outcomes; only whether or not the program passes the test is unpredictable. Testing can and should be planned, designed, scheduled and prioritized. Testing shows faults in code, designing or possibly correctness. Testing proves programs failures. Automation can be achieved in execution and design. The central idea of prioritization is minimizing test suites satisfying some rational, non arbitrary criteria [15]. Prioritizing in this manner entails which features of current software are essential and the possibilities if some of features are not taken into account. Test case prioritization schedules test cases for regression testing in an order that attempts to maximize some objective function. For example, testers might wish to schedule test cases in an order that achieves code coverage at the fastest rate possible. Minimization technique can lower cost by reducing a test suite to a minimal subset [16]. For instance following priority categories may be determined for the test cases:

Priority 1. The test cases must be executed before the final product is released to remove the critical bugs.
Priority 2. If time permits, the test cases may be executed.
Priority 3. The test cases are not important prior to the current release. It may be tested shortly after the release of the current software version.
Priority 4. The test case is never important, as its impact is nearly negligible.

Such a priority scheme ensures that low priority test cases do not create problems for software [9]. At times customers demand that some important features of software be tested and presented in the first version of software itself. There important features become criteria. Priority can be advertisement based because the company might have promised about essential features to customers [5]. Fault detection rate of a test suite reveals about the likelihood of faults earlier. Coverage criteria should be met earlier in test process.

Manuscript received September 02, 2012.

Arvind Kumar Upadhyay, Deptt. of Computer Sc. and Engg. EIT Faridabad (HR)-India.
A. K. Misra, Deptt. of Computer Sc. and Engg, MNNIT Allahabad (UP)-India.

II. TEST CASE PRIORITIZATION

It schedules test suites according to some criterion. The objective of this technique is to enhance the possibility that if the test suites are used for regression testing in the given order, they will more closely meet some objective than they would if they were executed in some other order.

Test case prioritization can address a wide variety of objectives, as given below:

1. Software developers/testers intend to increase the rate of fault detection.
2. Detecting the high-risk faults earlier in testing life cycle.
3. To increase the possibility of regression errors related to specific code changes very early in testing process.
4. To enhance the coverage of coverable code at a faster rate.
5. To make a system more reliable.

2.1. Prioritizing Rate of Fault Detection

For an objective, many prioritization techniques may be applied to test suites. For instance, to attempt to meet the first objective stated above, we may prioritize test cases in terms of the failure rates, measured historically; of the modules they exercise [17]. Else, we may prioritize test cases in terms of their increasing cost-per-coverage of code components, or in terms of their increasing cost-per-coverage of requirement features [7]. In nutshell, the intent behind the choice of a prioritization criterion is to increase the likelihood that the prioritized test suite can better meet the objective than would a random ordering of test cases. In our approach, we wish to increase the possibility of revealing faults earlier in the testing process. We illustrate this objective, informally, as one of improving our test suite's *rate of fault detection*: we intend to use APFD measure [18]. The inspiration for meeting this objective is an improved rate of fault detection during regression testing. As it can provide faster feedback on the system under test, or early evidence that quality goals have not been met; it can also let debuggers begin their work earlier than might otherwise be possible [5]. We consider nine different test case prioritization techniques described as following:

T1: No prioritization. Quite ironically, we begin without any prioritisation at all [3]. But yes, the success of an untreated test suite in meeting an objective may depend upon the manner in which it is initially formulated.

T2: Random prioritization. For empirical studies, test suites may be prioritised randomly.

T3: Optimal prioritization. To measure the effects of prioritization techniques on rate of fault detection, our empirical study utilize programs that contain known faults. We can determine, for any test suite, which test cases expose which faults, and thus we can determine an optimal ordering of test cases in a test suite for maximizing that suite's rate of fault detection [21]. In practice, of course, this is not a practical technique, as it requires knowledge of which test cases will expose which faults; however, by using it in our study, we gain insight into the success of other practical heuristics.

T4: Total branch coverage prioritization. By instrumenting a program, 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 [14]. This prioritization can thus be accomplished in time for programs containing branches.

T5: 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[21]. 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, we may be left with additional test cases that cannot add additional branch coverage. We could order these next using any prioritization technique; in this work we order the remaining test cases using total branch coverage prioritization[9]. Because additional branch coverage prioritization requires recalculation of coverage information for each unprioritized test case following selection of each test case,

T6: 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.

T7: 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 [23]. 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.

T8: Total fault-exposing-potential (FEP) prioritization. Statement and branch-coverage-based prioritization considers only whether a statement or branch has been exercised by a test case [2]. 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 [19]. 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.

T9: 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, we extend total FEP prioritization to create additional fault-exposing-potential (FEP) prioritization [10]. 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 , we lower the award values for all other test cases that exercise statements exercised by .

The purpose of Test case prioritization lies in ordering test cases based on a particular technique [21]. It takes into account that if such a scheme is followed then it is more likely to meet the objective than it would otherwise. Test case prioritization can address a wide variety of objectives as:

1. To increase the rate of fault detection so that faults may be revealed earlier in regression test.
2. To focus on high-risk faults and detect them earlier in testing process.
3. To speed up the regression errors connected to code changes as early as possible.
4. To cover code coverage in the system under test at a faster rate.
5. To enhance reliability confidence in the system under test at a faster rate.

III. CLUSTERING BASED PRIORITIZATION

3.1 Motivation

The total number of comparisons required for pair-wise comparison is $O(n^2)$ comparisons[20]. Redundancy makes pair-wise comparison very robust but the high cost incurred discourages it from being applied to test case prioritization. The maximum number of comparisons a human can make consistently is approximately 100 [1]; above this threshold, inconsistency grows significantly, leading to reduced effectiveness. But to require less than 100 pair-wise comparisons, the test suite could contain no more than 14 test cases. In real world scenario the issue of scalability is challenging. For example, suppose there are 1,000 test cases to prioritize; the total number of required pair-wise comparisons would be 499,500. Obviously it is unrealistic to expect a human tester to provide reliable responses for such a large number of comparisons [8]. Our approach using K-means cluster based prioritization reduces the number of comparisons and can be very effective. Instead of prioritizing individual test cases, clusters of test cases are prioritized using techniques such as clustering based prioritisation.

3.2 K-means clustering criteria

Broadly speaking, there are two methods of clustering i.e. data can be arranged as a group of individuals or as a hierarchy of groups. It can thereafter be established that whether the data group belong to some preconceived ideas or suggest new ones [4]. Cluster analysis groups data objects into clusters such that objects belonging to the same cluster are similar, while those belonging to different ones are dissimilar. Clustering techniques could be categorized into modes Partitional or Hierarchical:

Partitional: Given a database of objects, a partitional clustering algorithm constructs partitions of the data, where each cluster optimizes a clustering criterion, such as the minimization of the *sum of squared distance from the mean* within each cluster [6]. The complexity of Partitional clustering is large because it enumerates all possible groupings and tries to find the global optimum. Even for a small number of objects, the number of partitions is huge. That's why; common solutions start with an initial, usually random, partition and proceed with its refinement. A better practice would be to run the partitional algorithm for different sets of initial points (considered as representatives) and investigate whether all solutions lead to the same final partition [13]. Partitional Clustering algorithms try to locally improve a certain criterion. First, they compute the values of the similarity or distance, they order the results, and pick the one that optimizes the criterion [11]. Hence, the majority of them could be considered as greedy-like algorithms.

Hierarchical: Hierarchical algorithms create a hierarchical decomposition of the objects. They are either *agglomerative* (*bottom-up*) or *divisive* (*top-down*):

(a) *Agglomerative* algorithms start with each object being a separate cluster itself, and successively merge groups according to a distance measure [14]. The clustering may stop when all objects are in a single group or at any other point the user wants. These methods generally follow a greedy-like bottom-up merging.

(b) *Divisive* algorithms follow the opposite strategy [12]. They start with one group of all objects and successively split groups into smaller ones, until each object falls in one cluster, or as desired [10]. Divisive approaches divide the data objects in disjoint groups at every step, and follow the same pattern until all objects fall into a separate cluster. This is similar to the approach followed by divide-and-conquer algorithms.

K-means clustering method:

K-means *clustering* methods produce clusters from a set of objects based upon the squared-error objective functions:

$$E = \sum_{i=1}^{k} \Sigma_{p \in C_i} |p - m_i|^2$$

Being minimized [2, 3]. In the above expression, c_i are the clusters, p is a point in a cluster c_i and m_i the mean of cluster c_i. The mean of a cluster is given by a vector, which contains, for each attribute, the mean values of the data objects in this cluster, input parameter is the number of clusters, k[22]. As an output the algorithm returns the centers, or means, of every cluster c_i, most of the times excluding the cluster identities of individual points. The distance measure usually employed is the Euclidean distance [4]. Both for the optimization criterion and the proximity index, there are no restrictions, and they can be specified according to the application or the user's preference. The algorithm is as follows:

1. **Select k objects as initial centers;**
2. **Assign each data object to the closest center;**
3. **Recalculate the centers of each cluster;**
4. **Repeat steps 2 and 3 until distribution of data objects in clusters do not change;**

The algorithm is relatively scalable.

IV. THE EXPERIMENT

4.1. Research Questions

We are interested in the following research question.

Q: How can clustering technique facilitate test case prioritization of test suites?

Q: Can test case prioritization improve the rate of fault detection of test suites?

4.2. Efficacy and Clustering based prioritization Measures

We apply our clustering based prioritization technique on the famous quadratic equation problem. This problem reads a, b, c as the three coefficients of a quadratic equation $ax^2 + bx + c = 0$. It determines the nature of the roots of this equation. First, we write its procedure:

```
Proc roots
A.      Int a, b, c;
B.      D= b*b - 4*a*c;
C.      If (D<0)
D.      real = -b/2 * a; // imaginary roots
        D = -D;
        num = pov ((double) D, (double) 0.5);
        image = num/ (2*a);
E.      else if (D==0)
F.      root 1 = -b/(2*a)
        root 2 = root 1;
G.      else if (D>0)
H.      root 1 = ( -b + sqrt (d)/2 *a ;
        root 2 = ( -b  - sqrt (d)/2 *a ;
I.      end
```

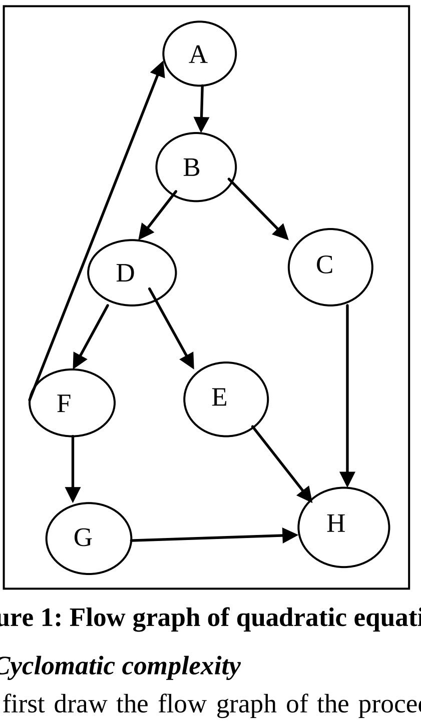

Figure 1: Flow graph of quadratic equation problem

4.2.1 Cyclomatic complexity

We first draw the flow graph of the procedure of finding roots of the quadratic equation. The flow graph has been drawn as shown in figure 1. With the help of flow graph, we can evaluate the cyclomatic complexity as follows:

1. V (G) = P+1, P is number of predicates.
   = 3+1
   = 4
2. V(G) = Number of regions +1
   = 3 + 1
   = 4
3. V(G) = e – n +2
   Where e is number edges, n is number of Nodes.
   = 11- 9 +2 =4

Thus we see that the cyclomatic complexity of above problem is four.

4.2.2 Independent paths

In the above problem, the independent path would be 4,

Path 1: A- B-C-H
Path 2: A-B-D-E-H
Path 3: A-B-D-F-G-H
Path 4: A-B-D-F-A

4.2.3. Test Cases for each path:

Path 1: test case 1
a,b,c : valid input
expected results: D<0, imaginary roots
Path 2: test case 2
a,b,c : valid input
expected result : D=0 , equal roots
Path 3: test case 3
a,b,c : valid input
expected results : D>0, root 1 root 2 are real
Path 4 : test case 4
a,b,c : valid input
expected results : D is not >0, read a,b,c again

IV. RESULT & ANALYSIS

We now apply the k-means clustering method for quadratic equation problem. For this we make use of independent paths of 4.2.2. In this there are four paths in every path testing criteria. Initially we took two clusters as k-value and by using the algorithm we finally calculate that two clusters to have following combination:

C1: path1, path2, path4
C2: path3

In our prioritization, we priorities the clusters according to dendrogram method. So for our example the test cases would be executed in the order: path2, path4, path1, path3, i.e. path 2 gets the highest priority and there after the sequence is followed. In the following figure, we have shown the dendrogram to show the prioritization of test cases.

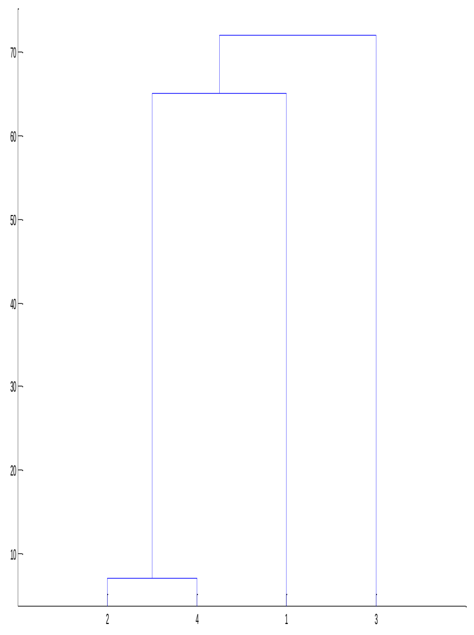

Figure 2: Dendrogram of test cases

We there after use the APFD method to calculate effectiveness of our method by using the formula:

APFD (average percentage of faults detected)
= 1- ((TF1 + TF2 +…….TFM)/nm) + 1/2n

Where

TF_i is the position of the first test suite T that exposes faults i. m is the total number of faults exposed in the system or module under T n is the total number of test cases in T.

Now, when we do not apply any clustering based dendrogram methods for prioritization then the APFD value is 0.5 but when we apply clustering based dendrogram method for prioritization method then there is significant improvement in average percentage of faults detected and the value is 0.625.

V. CONCLUSION & FUTURE WORK

Test case prioritization involves scheduling of test cases in an order that increases their effectiveness in meeting some performance goals. One such goal is APFD(average percentage of faults detected) measure that increases the chances of finding faults earlier in the software testing lifecycle and may facilitate the ultimate goal of software development by improving quality. We want to use many more techniques which help in this direction, particularly the data mining techniques. As it is well known that test suite development is quite expensive and more often , running an entire suite is not possible in its entirety as it takes more time to run and more human resources are required to actually execute them. Our method can address this issue very successfully.

REFERENCES

1. Gregg Rothermel, Roland H.Untch,Mary Jean Harrold,"Prioritizing Test Cases For Regression Testing," IEEE Transaction on Software Engineering, Vol.27, No.10 October 2001.
2. G. J. Myers. The Art of Software Testing. Revised and Updated by Tom Badgett and Todd M.Thomas with Corey Sandler, John Wiley & Sons, Inc, Second Edition. 2004, pp. 1-255.
3. Boris Beizer. *Software System Testing and Quality Assurance*. Van Nostrand, New York, 1984.

4. Boris Beizer. *Software Testing Techniques*. Van Nostrand Reinhold, Inc, New York NY, 2nd edition, 1990. ISBN 0-442-20672-0.
5. Richard A. DeMillo, W. Michael McCracken, Rhonda J. Martin, and John F. Passafiume.*Software and Evaluation*.Benjamin/Cummings, Menlo Park CA, 1987.
6. Siripong Roongruangsuwan, Jirapun Daengdej,"Test Case prioritization techniques," Journal of theoretical and applied informational technology,2005
7. Mao ye, boqinFeng, yao Lin 7Li Zhu. "Neural Networks Based Test Case Selection" Proc of IEEEtransactions,2006.
8. T.Y. Chen, Pak-lok poon, t.h. Tse."A choice Relation framework for supporting Category-partition Test Case generation" IEEE transactions on software Engineering, vol.29, No.7, July 2003.
9. Sebastian Elbaum, Alexey G.Malishevsky, Gregg Rothermel."Test Case Prioritization" IEEE transactions on software Engineering, vol.28, No.2, February 2002.
10. Kuo –Chung Tainand Yu Lei. "A Test generation strategy for Pairwisetesting" IEEE transactions on software Engineering, vol.28, No.1, January 2002.
11. Christoph C. Michael, gary McGraw, Michael A. Schatz. "Generating software test data by Evolution". IEEE transactions on software Engineering, vol.27, No.12, December 2001.
12. Shino yahoo & Mark Harman ,Paolo tonella & Angelo susi, "Clustering test cases to Effective & scalable prioritisation incorporating Expert knowledge," ISSTA 09,July 19-23, 2009, Chicago, USA.
13. Gregg rothermel , roland h. untch, chengyun chu, mary jean harrold, " Test case prioritization : An Empirical study," Proceedings of the international conference on software maintenance, oxford ,U.K., September, 1999, IEEE
14. Wei-Tek Tsai and Lian Yu, Feng Zhu, Ray Paul. "Rapid embedded system testing using verification patterns" . IEEE software 2005.
15. S. G. Elbaum, A. G. Malishevsky, and G. Rothermel. Prioritizing test cases for regression testing. In International Symposium on Software Testing and Analysis, pages 102–112. ACM Press, 2000.
16. Martina marre and Antonia Bertolino, "using spanning sets for coverage testing". IEEE transactions on software Engineering, vol.29, No.11, November 2003.
17. H. Do, S. G. Elbaum, and G. Rothermel. Supporting controlled experimentation with testing techniques: an infrastructure and its potential impact. *Empirical Software Engineering*, 10 (4): 405–435, 2005.
18. H. Do, G. Rothermel, and A. Kinneer. Prioritizing JUnit Test cases: an empirical assessment and cost-benefits analysis.*Empirical Software Engineering*, 11: 33–70, 2006.
19. P. M. Duvall, S. Matyas, and A. Glover. *Continuous Integration: Improving Software Quality and Reducing Risk*. Addison Wesley, Upper Saddle River, NJ, 2007.
20. S. G. Elbaum, A. G. Malishevsky, and G. Rothermel.Prioritizing test cases for regression testing. *ACM SIGSOFT Software Engineering Notes*, 25 (5): 102–112, 2000.
21. S. G. Elbaum, A. G. Malishevsky, and G. Rothermel. Test case prioritization: a family of empirical studies. *IEEE Transactions on Software Engineering*, 28 (2): 159–182, 2002.
22. S. G. Elbaum, G. Rothermel, S. Kanduri, and A. G. Malishevsky. Selecting a cost-effective test case Prioritization technique. *Software Quality Control*, 12 (3): 185–210, 2004.
23. David Gustafson,"Theory and Problem of Software Egineering," Computing and Information science Department Kansas state University.