Test Case Prioritization Based on Path Complexity

Tehseen Afzal[1], Aamer Nadeem[1], Muddassar Sindhu[2], Qamar uz Zaman[1]
[1]Department of Computer Science, Capital University of Science and Technology, Islamabad, Pakistan
tehseen_fjwu@yahoo.com, anadeem@cust.edu.pk, qamar.zaman@cust.edu.pk

[2]Department of Computer Science, Quaid-i-Azam University, Islamabad
masindhu@qau.edu.pk

***Abstract*—Software undergoes many modifications after its release. Regression testing is performed to ensure that the modification has not introduced any errors in the software and the software continues to work correctly. Regression testing is an expensive process. Three types of cost reduction techniques are used in regression testing. These techniques i.e., test case selection, test suite minimization and test case prioritization are used to reduce the cost of regression testing and improve the rate of fault detection. The focus of our research is on test case prioritization. Instead of minimizing test suite or selecting fewer test cases, test case prioritization orders test cases in such a way that the test cases detecting more faults are executed earlier. In case of limited resources, only top priority test cases are executed to ensure the reliability of the software. In this research, we have proposed an approach which uses path complexity and branch coverage to prioritize test cases based on assumption that the complex code is more likely to contain faults. Halstead's metric has been used to calculate the path complexity of the test cases. Proposed approach is compared with branch coverage based prioritization technique using some example programs. The results show that proposed approach outperforms existing branch coverage based approach in terms of APFD (Average Percentage of Faults Detected) up to 18% on average.**

Keywords— Regression testing, test case prioritization

I. INTRODUCTION

Software testing can be defined as a group of activities performed to evaluate some aspect of a piece of software [5]. The major objective of software testing is to find out errors in program under test. Moreover, it ensures that the requirements of the customer(s) are fulfilled by the software under test. Software testing is considered an expensive and critically important phase in process of software development [13]. Testing ensures the quality and correctness of software. Testing should utilize minimum resources to reduce the testing cost as design and development phase already consume many resources [7]. According to the report of National Institute for Standards and Technology (NIST)", almost $60 billion per year are utilized on software testing by US economy [3]. In 2016, the cost of testing jumped to $1.1 trillion per annum [28]. Some effective testing approach may decrease the cost of testing to approximately $22 billion. Software may contain different kinds of errors which include design error, input error, hardware error, statement error, specification error etc. Different testing types are used to identify and fix these errors [27]. Errors or faults are required to be discovered and removed timely so that they may not further be transmitted to the next phases of software development [20].

Whenever modifications are made to the software, it needs to be retested to ensure that the previous functionality of the software is not affected by the change. This type of testing is called regression testing (RT) [23]. As the modifications increase the complexity and size of the software; regression testing plays very important role in maintenance phase of SDLC [12]. Regression testing is a costly process which needs to be performed frequently in order to validate the correctness of the modified software after each modification [30]. During the development of voluminous software, testing cost may increase manifolds for executing a very large test suite [14]. For complex projects, test suite becomes very large. The execution of such a large test suite results in longer testing time [19]. To lower the cost of regression testing, software testers choose test suite using certain Regression testing techniques.

Test suite minimization technique reduces the size of a test suite by removing redundant test cases from the test suite [17]. This technique may result in elimination of some useful test cases. Regression Test Selection (RTS) technique chooses the test cases covering the modified sections of the system and discards the remaining test cases [30]. Only the most relevant test cases are selected for execution. In Test case prioritization technique, the test cases are ordered in such a way that the important test cases are placed first in prioritized list. The main objective of prioritization is to improve the rate of fault detection and minimize the regression testing cost [30].

The fault detection rate in regression testing is affected by the order in which test cases are executed. Due to limited time and resources, software testers use a priority list of test cases to decide when to halt the process of testing. If the testing process is halted due to any reason, it ensures that the important test cases with higher priorities have been executed first which increases the reliability of testing process [8]. Different prioritization techniques are available including white box and black box techniques. Prioritization techniques can use:

1. The code coverage factor like, statements, branches, spanning entities or functions for assigning award values to test cases [6].

Some techniques use the information of specifications, requirements or interaction of different events for ordering the test cases [29].

All the prioritization criteria generate a prioritized list of test cases. The prioritized list is then evaluated. The basic parameter to evaluate a prioritized list is fault detection rate i.e., Average Percentage of Faults Detected (APFD). It is a measure to check how early a particular test suite detects faults by using the particular prioritized list of test cases in test suite [18]. APFD is calculated using following formula:

$$APFD = 1 - \frac{TF_1 + \cdots \ldots TF_m}{nm} + \frac{1}{2n} \qquad (1)$$

978-1-7281-6625-4/19/$31.00 ©2019 IEEE
DOI 10.1109/FIT47737.2019.00074

Where T is the test suite with n test cases and F is the set of m faults identified by T. For ordering T', TF_i represents the order of the first test case that exposes the ith fault F_i.

Test case prioritization approaches are classified into two broad categories: Black box prioritization and white box prioritization techniques. Black box techniques are based on specifications and requirements. They do not need access to the source code [6]. White box prioritization techniques are based on source code e.g., History Based prioritization, Coverage Based Prioritization techniques which include function coverage, statement coverage, branch coverage etc. Most of the existing prioritization techniques are based on code coverage [26]. Coverage alone cannot improve the effectiveness of fault detection rate of test suite [25]. In such techniques, combinations of different coverage criteria are used to prioritize test cases to improve rate of fault detection. The techniques using structural complexity of the software can detect faults faster as compare to code coverage based approaches [11].

In this research work, we have used path complexity based on computational complexity by Halstead's Metric as a primary and branch coverage as secondary criterion to generate a priority list of test cases to improve rate of fault detection. Rest of the paper is organized as follows: Section II is about related work, Section III is proposed approach, Section IV gives details about results of proposed approach and Section V is conclusion and future work.

II. Related work

Test case prioritization orders test cases on the basis of some prioritization criterion. Each test case is assigned a priority. Software testers make sure that the test cases with higher priorities run earlier during testing process [30]. The order of test cases in which they are executed has great influence on the rate of fault detection of test suite. Prioritized list increases the probability that if the testing process is halted due to some constraints, the most important test cases with higher priorities will be executed. It enables debugging at early stages of testing and increases the rate of fault detection as well [6].

Two main white box prioritization strategies were introduced by Elbaum [6] and Wong, [29]: "Total" and "Additional". Depending on these strategies, prioritization approach can be single criterion based or multi-criteria based. Prioritization techniques using single criterion, order test cases on the basis of a single criterion whereas; multi-criteria based prioritization approaches use more than one criterion to prioritize test cases. The criteria can be white box or black box. There are different software complexity metrics to measure the complexity of code. The measure of the cost of software development, maintenance and usage is known as Software Complexity Metric [10]. Software complexity metrics are closely related to error distribution in subject code [26]. Many testers use code coverage for prioritization. Only few researchers have considered code complexity for test case prioritization. Chances of fault occurrence are higher in complex code as compare to simple statements. Complexity based approaches assume that fault occurrence rate is higher in complex piece of code. Since large software systems can be used more than 15 years, computer scientists and researchers have put great efforts to measure the complexity of software in previous several years. Some estimates show that 40% to 70% expenses are consumed on maintenance of existing software systems [25].

Test case prioritization techniques help in improving the rate of fault detection in regression testing. In most of the existing techniques, it is assumed that all faults have equal severity; however, it is not practically correct. Henard et al. [11] proposed an approach on the basis of program structure analysis. Authors claimed that their proposed approach detects severe faults earlier. The main idea of their approach is to compute the testing importance of each module by using two factors; fault proneness and importance of module from system perspective and user perspective. Test cases are prioritized on the basis of testing importance of module. Authors also introduced a metric APMC (Average Percentage of fault-affected Modules Cleared per test case) which measures the effectiveness of various prioritization techniques. Improved APFDC metric has been proposed, which takes fault severity and test cost in account. APMC is used to compute the effectiveness of proposed approach. This paper is based on module level complexity.

Khan and Khan [16] proposed a modular based test case prioritization technique. The proposed technique performs regression testing in two stages. First stage comprises of the prioritization of test cases based on modules coverage. In second stage, modular based ordered test suites are merged together for further prioritization. The study was based on fault coverage. The algorithm was compared with Greedy Algorithm and Additional Greedy Algorithm. Their approach considers fault coverage only whereas the nature of fault has not been taken in account.

Bryce and Memon [4] proposed an approach of test case prioritization in which Genetic Algorithm with Multi-Criteria fitness Function is used. Fitness function considers weight of test cases, fault severity, fault rates and number of structural coverage items covered by each test case. Their proposed technique used total strategy to prioritize test cases.

Arafeen and Do [2] proposed an approach for test case prioritization on the basis of three factors, i.e., rate of fault detection, percentage of fault detected and risk detection ability. The results of their approach were compared with those of Qu et al. [23]. The comparison was made on the basis of APFD. Their results show that the proposed technique outperforms other techniques compared with. Authors are simply using the values of three factors mentioned earlier but no method had been discussed to compute the said values.

Kumar et al. [18] used fault severity as another criterion for test case prioritization. On the basis of severity of faults' effect on the software, faults are divided into four types; fatal, serious, general and minor faults. Fault severity ranges from 2^0 to 2^3 (from minor to fatal faults). On the basis of fault severity, importance of test case is calculated and finally test cases are prioritized. Though authors have effectively described their approach; however, random selection of test cases with equal fault severity may cause important test cases to be prioritized at the end of the list rather than early execution.

Kaur and Goyal [15] proposed a Refactoring based approach (RBA) to improve test case prioritization. Authors used five different refactoring faults models (RFMs) i.e.,

rename method, pull up field, move method, pull up method and add parameter. Authors created five versions of programs under test by seeding refactoring faults in programs. Authors gained APFD of 92% by RBA. The limitation of the proposed approach was, it required execution of complete test suite at least once which is very costly.

Maheswari and Mala [21] used multi-criteria decision making technique TOPSIS in combination with fuzzy logic to prioritize test case. Authors used fault detection probability and execution time for prioritization. Authors used fault failure rate as a measure to detect the potential of test cases in test suite to detect faults. The probability of fault detection and execution time of test cases are provided by the testers which may prone to human error, thus this approach may not provide desired outcome.

Ansari et al. [1] proposed a multiple criteria based prioritization approach using three criteria; Statement coverage, Branch coverage, Function coverage. This empirical study uses the PSO (Particle Swarm Optimization) algorithm for test case prioritization. Particle swarm optimization algorithm is a multi-object optimization technique used to set the ordering of objects. The main objective of this approach is to order the test cases to achieve high rate of fault detection. The empirical results of this study show that Particle swarm optimization improves the performance of regression testing.

Mann [22] proposed an approach which uses more than one criterion to generate a prioritized list of test cases. Author used bank application for experimentation of his proposed approach. The main objective of the proposed approach was to find out the effectiveness of prioritized and un-prioritized test cases in terms of APFD (Average Percentage of Faults Detected), APSD (Average Percentage of Faults Detection), APBD (Average Percentage of Branch Detection) and APPD (Average Percentage of Path Detection). The results of this study show that proposed method outperforms the existing method.

After the through study of above mentioned approaches we have identified that in fault based prioritization approaches, the data of faults and test cases are taken from testing of original software. When software goes under some modification, new faults may possibly be introduced during modification. The modification of a particular part of the software may also affect the other parts of software. Therefore; in regression testing, the previous data of faults may not be useful due the following reasons:

- The faults occurred in initial (original) version have already been fixed so possibly, those faults may not occur again.
- Modification may introduce new faults in software. Thus, previous fault data become invalid for the testing of modified software.

In fault based approaches, the test cases are assigned priority on the basis of their potential to detect fault based on fault data available by testing of original software. Those prioritized test cases may not work well for the modified software due to possibility of occurrence of new faults. In coverage based approaches, only the number of entities (such as statements, branches, modules) covered are considered to prioritize test cases. As, there is a possibility that a test case covering less number of entities may be more vulnerable to faults due to some structural complexity and vice versa. Moreover; modification may cause the change of number of statements or branches in some cases. Thus the structure of the software can also affect the rate of fault detection. If the structural complexity is combined with any one of the coverage criteria for test case prioritization, it may prove a useful milestone in regression testing.

III. PROPOSED APPROACH

From state of the art, we have observed that the existing prioritization techniques are either based upon coverage criteria or rate of fault severity. In case of coverage criteria based techniques, there exists a possibility that the test cases providing greater coverage are not structurally complex, but they are given higher priority based on coverage, thus the test cases covering complex part of the program may be given least priority. Another observation is that some existing techniques use fault severity as a criterion for test case prioritization. In these approaches, it is assumed that the data of fault coverage and fault severity already exist. There are only few techniques which take the complexity of program under consideration. Those techniques are based on modular level complexity. Neither of the techniques is based on the complexity of the path covered by each test case. We are going to propose test case prioritization approach based on path complexity.

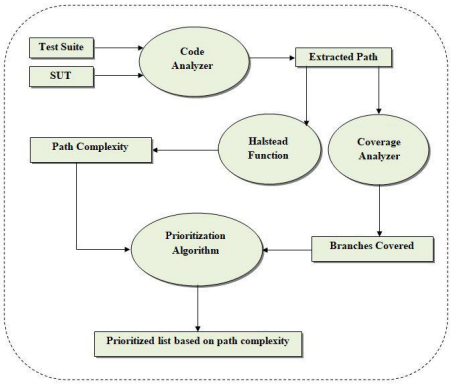

Fig. 1. An illustration of the proposed approach

Due to the use of path complexity of test cases rather than coverage, it is likely that path complexity based prioritization will produce better prioritization in terms of APFD as compared to existing coverage based prioritization techniques.

A. Halstead's Metric

Halstead's metric is structural complexity based metric. It was proposed by Maurice Halstead in his theory of software science [9]. Halstead's metric interprets the source code on

the basis of tokens and classifies each token either as an operand or an operator. Halstead's metric measures following different properties of software:

n1: number of unique operators
n2: number of unique operands
N1: total occurrences of operators
N2: total occurrences of operands

All other measures of Halstead's metric are based on these four quantities with certain fixed formulas. In Halstead complexity metric, all the identifiers, all the TYPENAME, TYPESPEC (Type specifiers), Keywords, all the numeric, string or character constants are considered as operands. Storage class specifiers, all the keywords which are used to reserve storage space are considered as operators in Halstead's metric.

Halstead's metric is program flow control based structure complexity metric, therefore, in this research; we are using Halstead's metric to measure the structure complexity of flow of test cases. Other measures of Halstead's metric are Size of vocabulary, program length N, volume V of program, Program Level, Difficulty D or error proneness, Effort to implement, Number of Bugs delivered, Estimated time etc. We have used Difficulty (error proneness) measure of Halstead in our approach. It means that if same operands are used multiple times in program, the program is more prone to faults. This is very important measure of Halstead's metric. It can be measured by using following formula:

$$\textbf{Difficulty D = (n1 / 2) * (N2 / n2)} \qquad (2)$$

Test cases are generated by using BVT (Boundary Value Testing). Code analyzer takes source code and test case file as input and extracts path executed by each test case. The extracted path is further passed to Halstead's function which calculates path complexity for each test case and stores in a file. Code coverage is also recorded by coverage analyzer. The values of path complexity and code coverage (branch coverage) are passed to prioritization algorithm. The algorithm prioritizes test cases on the basis of path complexity. In case of tie, the branch coverage is used as secondary criterion to break the tie among test cases.

B. Prioritization Algorithm

Algorithm 1 Procedure for prioritizing regression tests

Require: T : Set of Regression tests for P, N : Set of Path Complexity value for each test case in T, Cov : Set of entities covered by executing P against t, X' : A temporary set of regression tests for calculations
Ensure: PrT : A sequence of tests based on path complexity N

```
X' = T
while X' ≠ ∅ and Cov ≠ ∅ do
  for all t ∈ X' do
    if N(t)>N(u) then
      prt = prt + t
      X' = X' − {t}
    else if N(t)=N(u) then
      if COV(t) ≥ COV(u) then
        prt = prt + t
        X' = X' − {t}
      else
        prt = prt + t
      end if
    end if
  end for
end while
```

C. Example of Proposed Approach

The following example (Triangle problem) is used to define the concepts of our proposed path complexity based prioritization algorithm. Triangle program takes three input variable a, b and c which represent the sides of a triangle. There are three types of triangles which are equilateral, Isosceles and Scalene. For illustrating the example; 13 test cases were generated using SBVT (Simple Boundary Value Testing) for 3 inputs.

TABLE I. TEST CASES FOR THE TRIANGLE PROGRAM

Test case(t)	Path Covered	No. of Br. Cov.	Path Complexity
T1 (1,4,5)	2,4,6,7,9	5	4
T2 (2,4,5)	2,4,6,7,10,12,14	7	4
T3 (3,4,5)	2,4,6,7,10,12,14	7	4
T4 (4,4,5)	1,4,6,8,15,18	6	6
T5 (5,4,5)	2,3,6,8,16,19,22	7	6
T6 (3,2,5)	2,4,6,7,9	5	4
T7 (3,3,5)	1,4,6,8,15,18	6	6
T8 (3,6,5)	2,4,6,7,10,12,14	7	4
T9 (3,7,5)	2,4,6,7,10,12,14	7	4
T10 (3,4,3)	2,3,7,8,16,19,22	7	6
T11 (3,4,4)	2,4,5,8,16,20,23,26	8	9
T12 (3,4,7)	2,4,6,7,9	5	4
T13 (3,4,8)	2,4,6,7,9	5	4

The prioritized list becomes:

PrT: {T11,T5,T10,T4,T2,T8,T3,T1,T7,T12,T9,T13,T6}

IV. IMPLEMENTATION AND RESULTS

We have used three java based case studies to evaluate our proposed approach. Summary of case studies is given below in Table II.

TABLE II. SUMMARY OF SUBJECT PROGRAMS

Program	LOC	No. of Inputs	No. of branches
Quadratic Equation Problem	41	3	04
Simple Calculator Program	223	3	10
Triangle Problem	80	3	40

We have used path complexity and branch coverage data for implementation. Path complexity is our primary criterion and branch coverage is secondary criterion. The existing technique which we have used for comparison is Additional Branch Coverage. For each of these subject programs, priority lists are generated for our proposed path complexity based prioritization technique and existing coverage based prioritization technique. For each of these subject programs, priority lists are generated for our proposed path complexity based prioritization technique and existing coverage based prioritization technique.

The priority lists are then compared using APFD. To calculate the APFD, Faults are seeded in the subject programs by using mutation operator AORB (Arithmetic Operator Replacement Binary). Following Table III shows the APFD of subject programs.

TABLE III. APFD OF SUBJECT PROGRAMS

Program	No. of Test cases	No. of faults seeded	No. of faults detected	APFD for Proposed approach	APFD for existing approach
Quadratic Equation Problem	125	32	30	96.50%	87.40%
Simple Calculator Program	140	24	24	96.10%	52.80%
Triangle Problem	125	48	36	95.30%	92.51%

Following are the graphical representation of results for each test program.

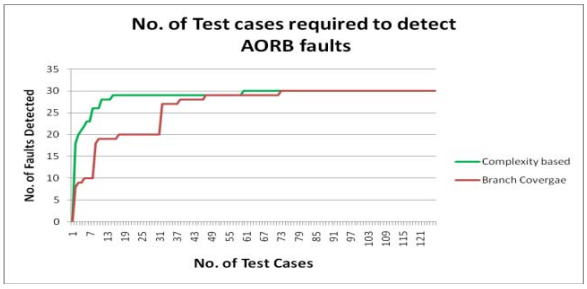

Fig. 2. Graphical representation of fault detection of test cases for Quadratic Equation Problem

On quadratic equation program, the proposed technique performed very well as compared to existing approach. The x-axis shows number of test cases executed and y-axis shows the number of faults detected by each test case. The graph shows that the proposed approach has detected maximum (28 out of 30) faults at earlier as compared to the existing approach. APFD of complexity based approach is higher than single criterion branch coverage based technique (Fig. 2). The difference between APFD of proposed and existing approach is 9.10%.

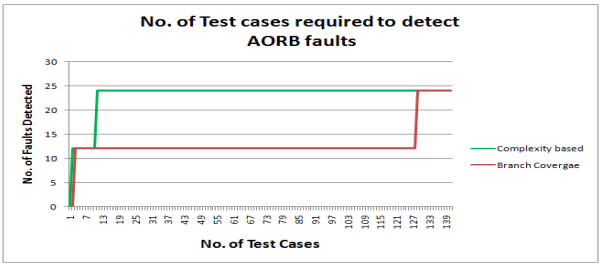

Fig. 3. Graphical representation of fault detection of test cases for Simple Calculator Program

In case of simple calculator program, the graph shows considerable improvement in APFD of proposed approach in comparison with existing approach. It can be seen from the graph that all of the faults has been detected very early by the proposed approach by executing very few test cases (Fig. 3). The proposed approach performs 43.30% better than existing approach in terms of APFD.

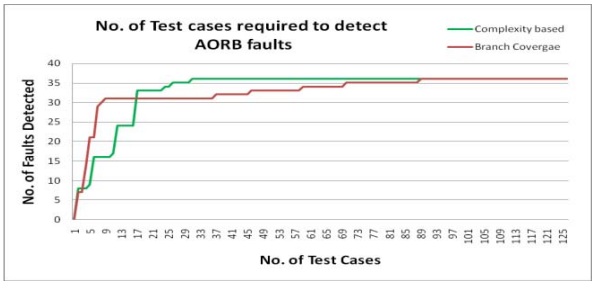

Fig. 4. Graphical representation of fault detection of test cases for Triangle Problem

When the proposed approach is applied on triangle problem, it can be seen in Fig. 4 that there is not a remarkable improvement in fault detection rate which can be due to some test cases which have less path complexity but they cover more entities. Still there is an increase of 2.79% in APFD of proposed approach.

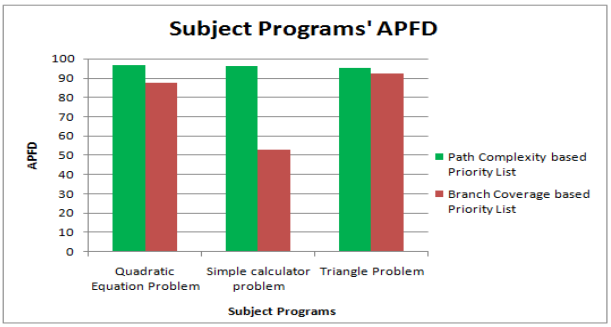

Fig. 5. APFD comparison of proposed and existing approach

An overall comparison of existing and proposed approach is shown in above graph (Fig. 5). The x-axis shows subject programs and y-axis shows APFD of proposed and existing approaches. Graph shows that the proposed path complexity based approach can increase the rate of fault detection as well as it can reduce the cost of regression testing.

V. CONCLUSION AND FUTURE WORK

Though existing white box prioritization approaches perform well, but most of these approaches are based on coverage criteria such as statement or branch coverage. They order the test cases with the assumption that the test cases covering greater number of entities can potentially detect more faults. This assumption does not always hold, e.g., the test cases covering greater number of entities may cover simple I/O statements, whereas test cases with less coverage may cover complex statements and, therefore, may have greater potential of detecting more faults. Very few of the prioritization approaches exist which are fault based or complexity based.

The technique used in this research is based on path complexity. The path complexity has been calculated by using Halstead's complexity metric. The test cases having higher values of path complexity are assigned higher priority. The proposed approach breaks the tie among test cases with same path complexity by using branch coverage as secondary criterion. Thus, the use of path complexity along with branch coverage has provided better rate of fault

detection. The main objective of the proposed technique was to increase the average percentage of fault detection (APFD) of test suite. After generating the complexity based prioritization list and comparing it with the existing approach, we have observed that the APFD of our proposed technique is more than the strongest coverage based prioritization technique which is branch coverage. The difference between the APFDs of branch coverage and complexity based coverage ranges from 2.7% to 42%.

After successful experiments of the proposed technique, we plan to use a combination of complexity metrics for prioritization of test suites in future. Use of more than one complexity metrics may help in increase rate of fault detection. We also plan to perform experiments with larger case studies.

REFERENCES

[1] Ahlam Ansari, Anam Khan, Alisha Khan, and Konain Mukadam. Optimized regression test using test case prioritization. Procedia Computer Science, 79:152-160, 2016.

[2] Md Junaid Arafeen and Hyunsook Do. Test case prioritization using requirements-based clustering. In Software Testing, Verification and Validation (ICST), 2013 IEEE Sixth International Conference on, p. 312-321. IEEE, 2013.

[3] Renée C Bryce and Charles J Colbourn. Prioritized interaction testing for pairwise coverage with seeding and constraints. Information and Software Technology, 48(10):960_970, 2006.

[4] Renée C Bryce and Atif M Memon. Test suite prioritization by interaction coverage. In Workshop on Domain specfic approaches to software test automation: in conjunction with the 6th ESEC/FSE joint meeting, p 1-7. ACM, 2007.

[5] Ilene Burnstein. Practical software testing: a process-oriented approach. Springer Science & Business Media, 2006.

[6] Sebastian Elbaum, Alexey G Malishevsky, and Gregg Rothermel. Test case prioritization: A family of empirical studies. IEEE transactions on software engineering, 28(2):159-182, 2002.

[7] Robert Feldt, Simon Poulding, David Clark, and Shin Yoo. Test set diameter: Quantifying the diversity of sets of test cases, pages 223_233. 2016.

[8] Michael R Garey. A guide to the theory of np-completeness. Computers and intractability, 1979.

[9] Maurice Howard Halstead. Elements of software science, volume 7. Elsevier New York, 1977.

[10] Mary Jean Harrold. Testing evolving software, Journal of Systems and Software, 47(2-3):173-181, 1999.

[11] Christopher Henard, Mike Papadakis, Mark Harman, Yue Jia, and Yves Le Traon. Comparing white-box and black-box test prioritization. In Software Engineering (ICSE), 2016.

[12] Getachew Mekuria Habtemariam and Sudhir Kumar Mohapatra. A geneticalgorithm-based approach for test case prioritization. In International Conference on Information and Communication Technology for Development for Africa, pages 24-37. Springer, 2019.

[13] Dennis Jeffrey and Neelam Gupta. Test case prioritization using relevant slices. In Computer Software and Applications Conference, 2006. COMPSAC'06. 30th Annual International, volume 1, p. 411-420. IEEE, 2006.

[14] Arun Prakash Agrawal, Ankur Choudhary, Arvinder Kaur, and Hari Mohan Pandey. Fault coverage-based test suite optimization method for regression testing: learning from mistakes-based approach. Neural Computing and Applications, pages 1-16, 2019.

[15] Arvinder Kaur and Shubhra Goyal. A genetic algorithm for regression test case prioritization using code coverage. International journal on computer science and engineering, 3(5):1839-1847, 2011.

[16] Mohd Ehmer Khan, Farmeena Khan. A comparative study of white box, black box and grey box testing techniques. Int. J. Adv. Comput. Sci. Appl, 3(6), 2012.

[17] Jung-Min Kim and Adam Porter. A history-based test prioritization technique for regression testing in resource constrained environments. In Proceedings of the 24th international conference on software engineering, p.119-129. ACM, 2002.

[18] Harish Kumar, Ved Pal, and Naresh Chauhan. A hierarchical system test case prioritization technique based on requirements. In 13th Annual International Software Testing Conference, p. 4-5, 2013.

[19] Qi Luo, Kevin Moran, Denys Poshyvanyk, and Massimiliano Di Penta. Assessing test case prioritization on real faults and mutants. In 2018 IEEE International Conference on Software Maintenance and Evolution (ICSME), pages 240-251. IEEE, 2018.

[20] Baresi L. Pezze M. An introduction to software testing. Electronic Notes in Theoretical Computer Science, 148:89-111, 2006.

[21] R Uma Maheswari and D Jeya Mala. Combined genetic and simulated annealing approach for test case prioritization. Indian Journal of Science and Technology, 8(35):1-5, 2015.

[22] Mukesh Mann. Generating and prioritizing optimal paths using ant colony optimization. Computational Ecology and Software, 5(1):1-15, 2015.

[23] R Pradeepa and K VimalDevi. Effectiveness of testcase prioritization using apfd metric: Survey. In IJCA Proceedings on International Conference on Research Trends in Computer Technologies, p. 1-4, 2013.

[24] Bo Qu, Changhai Nie, Baowen Xu, and Xiaofang Zhang. Test case prioritization for black box testing. In Computer Software and Applications Conference, 2007. COMPSAC 2007. 31st Annual International, volume 1, p. 465-474. Ieee, 2007.

[25] Roland H. Chu Chengyun Harrold Mary Jean Rothermel, Gregg U. Prioritizing test cases for regression testing. IEEE Transactions on software engineering, 27(10):929-948, 2001.

[26] Chu Chengyun Harrold Mary Jean Rothermel Gregg, Untch Roland H. Test case prioritization: An empirical study. In Software Maintenance, 1999.(ICSM'99) Proceedings. IEEE International Conference on, p.179-188. IEEE, 1999.

[27] Singh G. Singh S and Singh S. Software testing. International Journal of Advanced Research in Computer Science, (1(3)), 2010.

[28] Freyja Spaven. Cost of software errors: How much could software errors be costing your company? raygun.com/blog/cost-of-software-errors/, March 2017.

[29] W Eric Wong, Joseph R Horgan, Saul London, and Hiralal Agrawal. A study of effective regression testing in practice. In Software Reliability Engineering, 1997. Proceedings., The Eighth International Symposium on, p. 264-274. IEEE, 1997.

[30] Shin Yoo and Mark Harman. Regression testing minimization, selection and prioritization: a survey. Software Testing, Verification and Reliability, 22(2):67-120, 2012.