MULTIPARTITE BASED TEST CASE PRIORITIZATION USING FAILURE HISTORY

Md. Abdur Rahman^{1*}, Md. Abu Hasan², Khaled Shah³ and Md. Saeed Siddik⁴

¹*Centre for Advanced Research in Sciences, University of Dhaka, Bangladesh*

^{2,4}*Institute of Information Technology, University of Dhaka, Bangladesh*

³*Waterloo Management of Integrated Manufacturing Systems Lab, University of
Waterloo, Canada*

¹mukul.arahman@gmail.com, ²hasandubits@gmail.com,

³khaled.mostafa.shah@uwaterloo.ca, ⁴saeed.siddik@iit.du.ac.bd

Abstract— Test case prioritization reorders test case execution sequence based on their early fault detection capability. In regression testing when new version is released, both current and previous version test cases are executed to cross check the desired functionality. Historical failure data ensures the previous fault detection information, which leads the potential faults in new version. Most of the existing prioritization strategies are developed either similar code or requirement coverage based, where some incorporated historical failure data. Clustering similar test cases may detect similar fault consecutively, which never ensure the maximum variety of fault coverage. This paper presents a prioritization approach based on multipartite graph clustering using historical data analysis to detect faults of different code coverage in minimum test case execution. In this strategy, test cases within each partite are distinct in terms of function coverage, which ensures dissimilarity among them. Proposed scheme is evaluated using well established Defects4j dataset, and it has reported that multipartite algorithm performs better than normal ordering, random, similarity clustering, and dissimilarity based test case prioritization approaches. The proposed method performs 42.85% and 14.28% better than similarity and dissimilarity based approaches respectively in terms of early fault detection.

Keywords— Test Case Prioritization, Dissimilarity, Multipartite, Failure History

1. INTRODUCTION

Test case prioritization determines proper re-orderings of test cases to enhance fault detection rate. In regression testing, both new and previous versions test cases are considered for checking newly added functionalities. Regression testing involves revalidating the software when new components have been included to meet the modified requirements [1]. Gradually test cases are increased with the evolve of each version, and exhaustive testing will become very expensive and time consuming [2], *e.g.*, more than 50 days are required to test 20,000 lines of code [1].

Since, regression testing is one of the major time challenging activities, various methods such as test case selection, reduction, and prioritization have been introduced to improve regression testing performance on the basis of time effectiveness [1], [3], [4]. Reduction techniques remove test cases from a test suite which are redundant with respect to the coverage of some set of program requirements [5]. Where, selection techniques do

Received: January 5, 2019

Reviewed: March 27, 2019

Accepted: April 5, 2019

not remove but select test cases those are related to the changed portion of the source code [5]. Both techniques reduce the reliability of regression testing by omitting some test cases those can detect certain types of faults. On the other hand, test case prioritization reorders test cases by considering all of them, in such a way so that it meets testing objectives like fault detection rate and code coverage or quick feedback [4]. Researchers introduced different test case prioritization techniques to maximize fault detection rate, which are categorized in several areas, such as requirement coverage [6], [7], historical data analysis [8-10], code coverage [4], [11], [12], *etc.* Regression test case prioritization based on requirement priority has been implemented using historical execution data [8]. While, several code coverage based prioritization techniques are described by Rothermel *et al.*, [3] to improve the rate of fault detection. Test cases are also prioritized using heuristics algorithms, e.g. modified genetic algorithm to improve code coverage has been proposed by [11]. Bayesian Network based prioritization method has been presented by Zhao *et al.*, [13] where test cases are clustered using their method level coverage to improve rate of fault detection.

Fig. 1 Similar Cluster Formation

On the contrast, clustering based prioritization approach was experimented by [6], [7] where test cases are clustered using their code coverage in order to improve earlier fault detection. However, clustering test cases based on their coverage information may detect similar faults consecutively as each cluster contains similar type of test cases. This problem of similarity based clustering approach has illustrated in Fig. 1, where nine (9) test cases and eight (8) faults are covered by three (3) similarity based clusters. Interestingly, faults F4 and F5 are covered by all three clusters. However, faults F3 and F10 are uncovered by all clusters. In this scenario, any single cluster cannot cover all or maximum fault, which is one of the limitations of similarity based test case clustering. Dissimilarity based clustering technique has been proposed by [14], [25] to overcome the limitation, where dissimilar clusters are formed using top test cases from each similar cluster. However, it does not ensure that test cases inside a cluster are uniquely independent in terms of execution or coverage.

To overcome this drawback, this paper proposes multipartite test case prioritization approach to ensure distinct test case selection in terms of function coverage. In the beginning of multipartite prioritization, call dependency graph of current version's test suite is generated to find out the inter function connectivity among the test cases. Where, the connectivity defines either similar or dissimilar among the test cases. After that, test cases are assigned inside the partitions using generated call dependency graph, where connectivity value is either connected (1) or disconnected (0). Therefore, test cases within every partite are independent from each other, which ensures test cases' distinctness in

terms of function coverage. Fault history call dependency graph is generated from new and previous test cases to list out the similar test cases which are failed in previous execution. Test cases similar to previously failed have highest possibility to fail in subsequent execution; hence, these test cases get priority in regression testing [17].

Failure history and degree of connectivity are used to order intra-partite test cases. When test cases are similar to failed in previous execution, number of detected faults with normalized degree of connectivity is used for priority assignment. Otherwise, only normalized degree of connectivity is used for intra-partite test case priority assignment. Finally, execution sequence of each partite is ordered to detect divergent faults at the early stage of regression testing. Cumulative priority score of entire test cases inside a partite is used for ordering the final partite execution sequence. Thus, proposed method can detect divergent faults by executing lower input size as they cover different function in source code. The proposed multipartite framework is experimented with four java projects of Defects4j [15] dataset, and results are compared using standard prioritization measurement metric APFD [4]. On average, proposed multipartite approach has higher average percentage of fault detection rate (APFD 83.53%) compare to untreated, random, similar, and dissimilar [14] 50.16%, 50.02%, 66.64%, and 77.05% respectively. Through the research, it has been concluded that, the combination of distinct test case partite and failure history is significant for detecting faulty modules earlier than others.

Rest of the paper is organized as follows: the literature review has been described in section II. Section III presents the proposed methodology, where experimental result analysis and conclusion are discussed in section IV and V respectively.

2. LITERATURE REVIEW

Test case prioritization technique rearranges test case execution sequence to maximize objective functions, such as improving rate of fault detection. In regression testing, several approaches are proposed for prioritizing test cases, which can be categorized by fault based, requirement based, source code based, etc. Details prominent prioritization techniques are classified and reviewed in this section.

2.1. FAULT BASED PRIORITIZATION

Rothermel *et al.*, described code coverage based prioritization approach to detect fault early in regression testing. This strategy introduces different variations of code coverage based methods, using different criteria for coverage named as branch coverage, statement coverage, and fault exposing potential [1]. Empirical result analysis showed code coverage based prioritization performs better than additional branch coverage.

Nayak *et al.*, introduced a novel approach based on fault rate and fault severity to prioritize test cases so that more faults can be identified at the beginning of the testing phase [16]. In this method, the fault rate defines the total number of faults revealed by a test case in execution time, where fault severity can be different such as higher, lower for various projects based on the software unit testing. The comparative result analysis showed that proposed method APFD value is higher than existing random, reverse and no prioritization approaches.

Tanzeem *et al.*, proposed a prioritization approach using failure history in order to rank test cases similar to previously failed [17]. In this research, test cases those are functionally similar to previously failed got ranked. Similarity is measured using function call sequence between old and new version test cases and experimented with defects4j dataset. The investigation stated that the proposed method performs more efficiently as it can effectively prioritize test cases compared to other traditional strategies. However, this technique cannot work efficiently if fault history is unavailable or insufficient.

2.2. REQUIREMENTS BASED PRIORITIZATION

Arafeen *et al.*, proposed prioritization technique using requirement similarity clustering to investigate regression testing efficiency [6]. In this proposed method, clusters are formed using requirements test case mapping traceability matrix, where test cases within each clusters had been prioritized based on code coverage information. Experimental result analysis denoted that the applied strategy improved the test case prioritization effectiveness by incorporating requirement information. However, test case failure history has not been considered for prioritization, which may affect the performance.

Siddik *et al.*, introduced test case prioritization technique named as RDCC, which collaborates different software artifacts such as requirements, design diagrams and source code in order to overcome the limitation of traditional single SDLC phase [18]. In RDCC approach, requirements connectivity, design inter-dependency and code metrics are collected, and multiplied by their weight for measuring final priority of test case. Experimental analysis figured out that use of collaborative information was significant in test case prioritization. However, significant direction was undeclared for assigning weight to SDLC phases, and the result would be more effective by incorporating historical failure data.

Marijan *et al.*, proposed a multi-perspective test case prioritization framework in time constrained environment considering test case execution time, inter dependent functionality, failure impact and frequency factors for faster fault detection [19]. This approach prioritizes test cases to detect fault early and maximizes test cases having maximum inter dependence functional coverage, failure impact and frequency by selecting cost effective test cases when time is limited. Even though, this strategy used multiple factors for prioritizing test cases, dissimilar based test suite selection has not been considered to detect different type of faults.

2.3. COVERAGE BASED PRIORITIZATION

Konsaard *et al.*, implemented a modified genetic algorithm for prioritizing test cases to improve code coverage [11]. In this method, a control flow graph was generated using the selected program, which derived to get decision graph that may vary for different number of test cases. The experimental result showed that modified genetic algorithm performed better than Bee Colony and random approaches using fitness value and total coverage. Code coverage based adaptive random testing technique for test case prioritization has been proposed by Jiang *et al.* [20] to overcome the limitations of random test case selection. In random prioritization, tie test cases are resolved with random selection, which may select important test cases at the last that leads to ineffective prioritization.

Zhang *et al.*, proposed the total and additional strategies to prioritize covering elements and non-covering additional elements [21]. In additional strategy test cases with maximal coverage are chosen that are not covered in earlier prioritization sequence where, the total strategy always takes statements in account considering previously covered statements history. The contribution of this method is to introducing prioritization approach unifying two models named as total and additional strategies. That can also yield a spectrum of generic strategies having smell of both the total and additional technique.

Rothermel *et al.*, introduced a modern approach for regression test case selection using control flow graph theory [22]. Here control flow graph of present and previous version program is constructed in order to select test cases those covers new source code. The proposed method can select test cases form the original set suit to test modified as well as deleted source code efficiently. And thus this technique ensures that new version software does not affect existing code adversely.

The experimental analysis indicates that proposed method reduce execution time and manage big data size in regression testing.

2.4. PROBABILITY BASED PRIORITIZATION

A Bayesian Network (BN) based novel prioritization framework was proposed to improve fault detection rate by Mirarab *et al.*, [12]. The framework took program modification, tendency of fault occurrence and test case coverage information as input and produces the probability of fault detection by a test case as output. Quality metric, test coverage, and changes of data analysis were taken for construction of Bayesian model. The experimental result showed that proposed method performed better than other implemented techniques such as normal ordering and random ordering, if the available faults are remarkable.

Zhao *et al.*, proposed an enhanced BN based prioritization technique to overcome the drawback of detecting similar faults by several test cases in BN based approach [13]. This strategy clustered test cases based on method level coverage matrix, and ordered intra cluster elements with fault detection probability by BN. Source code change information, class level coverage matrix was fed in BN model to get failure probability. The result showed this scheme is more effective than normal BN model for prioritization. However, failure history which is effective to detect fault in regression testing [17] has not been considered for prioritization.

2.4. HISTORY BASED PRIORITIZATION

To improve testing efficiency, history based dynamic regression test case prioritization using requirement priority was proposed by Wang *et al.*, [8]. In this method, test cases were prioritized with requirements assigned priority where numbers of faults detected by test cases were recorded to be used for next version requirement priority. The results analysis has showed that the proposed method improves testing effectiveness and fault detection ability compare to existing random and other methods, where the efficiency depends on how accurately requirement priority has been assigned.

History based test case prioritization for failure information has been proposed by Chao *et al.*, [10], where failure history is statistically analyzed to reorder the test cases by utilizing the correlation data from previous test result. Wong *et al.* proposed a mixed up mechanism having the combination of code modification, and code minimization with previous version history [23]. This paper implemented an Automatic Testing Analysis tool in C to compute the test selection and showed improve performance rate by reducing response time.

2.5. SUMMARY

The literature analysis of existing approaches shows different prioritization strategies that have been implemented for regression testing such as requirement, code coverage, historical data analysis, probability based etc. Very few researchers incorporate the historical data and similarity clusters, where similar test cases are pointing the uniform faults together, which are imperfect in different fault detection. However, no direction has been found yet to identify distinct test cases in terms of function coverage to detect dissimilar faults using historical data analysis, which may increase fault detection rate in regression testing.

3. PROPOSED METHODOLOGY

The multipartite clustering based test case prioritization framework is proposed to improve the performance of regression testing. This method selects and executes more

significant test cases earlier with intent to different region code coverage for detecting variance faults. This section presents the details of the underlying ideas behind the proposed framework. The whole process is divided in the following five steps:

Step 1: Generate Current Version Call Dependency Graph (CDG)

Step 2: Partite Formation using CDG

Step 3: Generate Fault History CDG

Step 4: Order Intra Partite Test Cases

Step 5: Final Ordering of Each Partite

Detail description of the above steps is given in the following sub-sections.

3.1. GENERATE CURRENT VERSION CALL DEPENDENCY GRAPH (CDG)

A call dependency graph is generated using the connectivity among the test cases functions. This is a special type of directed control flow graph, which describes exactly those calls made from one entity to another [24]. In a program call graph represents binary relationship over the selected entities; for example, over functions, over modules, over files *etc.* In this research, test case similarity is realized using call dependency graph where test cases are vertices (V) and function dependency denotes the connectivity edge (E) of the graph $G(V, E)$. The call dependency graph is generated with the value of presence or absence of functional connectivity between the test cases. The connectivity value is Boolean, which is either connected (1) or disconnected (0).

Algorithm 1: Generate Call Dependency Graph

Input: Current Version Test Suite (T_c)

Output: Function Call Dependency Graph (D_{ij})

```

1:  Begin
2:   $T_c \leftarrow$  read all current version test cases  $t_i$ 
3:   $F_i \leftarrow$  all function in Test Case  $t_i$ 
4:  for all function  $f_i \in F_i$  do
5:    for all function  $f_{i+1} \in F_{i+1}$  do
6:      if  $f_i$  is connected to  $f_{i+1}$  then
7:         $D_{ij} \leftarrow 1$ 
8:      else
9:         $D_{ij} \leftarrow 0$ 
10:     end if
11:   end for
12: end for
13: End

```

The functional steps of call dependency graph generation from current version test cases are explained in Algorithm 1. The symbols T_c , F_i , D_{ij} in Algorithm 1 are used for list of current version's test cases, all functions of a single test case, connectivity value of each test case in call dependency graph respectively.

Similarity among the present version test cases are listed out using their inter function connectivity, for this purpose, function call dependency graph $G(V, E)$ of present version test cases is generated. Degree of connectivity (DoC) for each test case is computed on the basis of total connections in the call dependency graph. The degree of connectivity measuring process is formalized as following equation.

$$TC_{DoC} = \sum_{i=1}^n w_1 + w_2 + w_3 + \dots + w_n \quad (1)$$

Where, TC_{DoC} = Degree of connectivity of a test case, n = number of test cases, and w = connectivity value in the call dependency graph for that test case considering position.

After formation of function call dependency graph of present version test cases according to Algorithm 1, it is required to find out connectivity among test cases in order to partite formation, which is described in the following section.

3.2. PARTITE FORMATION USING CDG

Test cases similarity is used to form partite using present version call dependency graph. Test case similarity is explored using inter function connectivity in call dependency graph. If a test case is similar with any other partite's test case, a new partition is formed, otherwise, assigned in existing partition. First test case of test suite is taken to form the first partite. Next, the second test case is selected and checked similarity with existing partite's test case. If there exist similarity between the test cases, the selected test case forms a new partite. Otherwise, the selected test case is assigned inside dissimilar partite.

Algorithm 2: Partite Formation Using CDG

Input: Call Dependency Graph $G(v, e)$

Output: Graph partites P_i

```

1:  Begin
2:   $T_c \leftarrow$  read all test cases  $t_i$ 
3:  Partite  $P_1 \leftarrow \{\}$ 
4:   $P_1 \leftarrow P_1 \cup \{t_1\}$ 
5:  for all test cases  $t_i \in T_c \setminus t_1$  do
6:    if  $t_i$  is connected to all  $P_i$  then
7:      Make new partite  $P_{i+1} \leftarrow \{\}$ 
8:       $P_{i+1} \leftarrow P_{i+1} \cup \{t_i\}$ 
9:    else
10:     if  $t_i$  is disconnected to single  $P_q$  partite then
11:        $P_q \leftarrow P_q \cup \{t_i\}$ 
12:     else
13:       if  $t_i$  is disconnected to  $(n-1)$  partite then
14:          $r \leftarrow$  random assignment (1 to  $n+1$ )
15:          $P_r \leftarrow P_r \cup \{t_i\}$ 
16:       end if
17:     end if
18:   end if
19: end for
20: End

```

If the selected test case is dissimilar with more than one partite, the test case is assigned randomly under any one of partites. This partite making process is continued until the test suite is empty. After the completion of partite formation, a number of partites are generated. Where each partite contains distinct function coverage test case as there have no possibility to exist similarity. This process is given in Algorithm 2, where the symbols T_c and P_i are used as all test cases in the version, and first partite respectively.

3.3. GENERATE FAULT HISTORY CDG

Fault history call dependency graph is generated to find out the list of previously failed similar test cases. Previously failed test cases have highest possibility to fail in subsequent execution; hence, these test cases get priority in regression testing [17]. However, in new version, test cases get priority which is identical to previously failed test cases. Therefore, it is required to identify test cases in new version those are similar to previously failed test cases. The similarity between old and new test cases is measured using inter function connectivity. An example of inter function connectivity of old and new test cases is shown in Table I.

Table I. Function called example

@Test_Old_TestCase	@Test_New_TestCase
public void testCaseOld()	public void testCaseNew()
{ m1(); m2(); m3(); m4(); m5(); m6(); }	{ m2(); m4(); m5(); m7(); }
Common functions m2(); m4(); and m5();	

According to the Table 1, function *testCaseOld()* called *m1()*, *m2()*, *m3()*, *m4()*, *m5()* and *m6()*, where function *testCaseNew()* of new test case called *m2()*, *m4()*, *m5()* and *m7()*. Even though, the name of old and new test cases is not same. However, *m2()*, *m4()* and *m5()* are called by both functions, thus, both test cases are connected for these common function calls.

In this example, similarity score of *testCaseNew()* is three with respect to *testCaseOld()*, which is calculated by the total number of same function calls. After calculating this similarity score of present version test cases, an execution function call is generated in order to get list of test cases those are similar to previously failed test cases. Then, execution function call graph $G(V, E)$ has been generated, where test cases are the vertices (V) and the similarity scores are the edges (E) between two vertices. Degree of connectivity of each test case is computed on the basis of call dependency graph. The fault history CDG generation steps are shown in Algorithm 3.

Algorithm 3: Fault History CDG Generation

Input: Old Version Test Case T_{old} , Current Version Test Case T_{cur}

Output: Failure History CDG D_{ij}

```

1:  Begin
2:  Dependency Graph  $D_{ij} \leftarrow \{\}$ 
3:   $F_{old} \leftarrow$  extract functions of Test Cases  $T_{old}$ 
4:   $F_{cur} \leftarrow$  extract functions of Test Cases  $T_{cur}$ 
    $F_i$  all functions in Test Case  $t_i$ 
5:  for all  $f_i \in F_{cur}$  do
6:    for  $f_j \in F_{old}$  do
7:      if  $f_i$  is connected to  $f_j$  then
8:         $D_{ij} = 1$ 
9:      else
10:        $D_{ij} = 0$ 
17:     end if
18:   end for
19: end for
20: End

```

The symbols f_i and f_j in Algorithm 3 are used for i^{th} function in new version and j^{th} function in present version test suites respectively. In this step, test cases of two subsequent versions (e.g: v2 and v3) are taken as input to find out inter connected

functions. Call dependency graph is generated using test cases inter connected function list of both versions. Similarity between the test cases is measured using degree of connectivity in call dependency graph. The similar to previously failed tests cases with their corresponding number of fault detection history is profiled as previous execution history, which is used for assign priority to present version test cases.

3.4. ORDER INTRA PARTITE TEST CASES

Test Cases in each partite are required to re-order according to their calculated weight for executing the most important test cases early. Previous failure history and degree of connectivity are used for internal partite's test cases ordering. The test cases similar to previously failed are assigned priority based on the number of faults detected in previous execution and normalized degree of connectivity; otherwise, only degree of connectivity is used. Test case weight is calculated using Equation 2 and ordered intra partite test cases in descending. When multiple scores were similar, then random assignment would be used for ordering. The partite test cases ordering steps are described in Algorithm 4.

Algorithm 4: Order Intra Partite Test Cases

Input: Partites P , List of detected faults by previous test cases L_f

Output: Prioritize partite Test Cases T_p

```

1:  Begin
2:   $T_c \leftarrow$  read all test cases from partite  $P_i$ 
3:   $DoC \leftarrow$  calculate test cases degree of connectivity
4:   $F_n \leftarrow$  number of faults of each failure test cases
5:   $L_{tw} \leftarrow \{ \}$ 
6:  for all  $t_i \in T_c$  do
7:    if  $t_i \in L_f$  then
8:       $L_{tw} \leftarrow (DoC[t_i]/NoTC) + L_f[t_i]$ 
9:    else
10:      $L_{tw} \leftarrow (DoC[t_i]/NoTC)$ 
11:    end if
12:  end for
13:  for  $i \in [1: L_{tw}]$  do
14:    for  $j \in [i+1: L_{tw}]$  do
15:      if  $L_{tw}[i] < L_{tw}[j]$  then
16:         $temp = L_{tw}[i]$ 
17:         $L_{tw}[i] = L_{tw}[j]$ 
18:         $L_{tw}[j] = temp$ 
19:      end if
20:    end for
21:  end for
22:  End

```

In order to calculate weight of each test case, degree of connectivity is divided by number of test cases and then the outcome is added with number of faults. Degree of connectivity is divided by total test cases to emphasis the weight of fault detection in previous version. The DoC value is also normalized to keep the value within a common range as below.

$$TC_{weight} = (DoC / NoTC) + N \quad (2)$$

Finally, test cases in each partite are ordered internally based on measured weighted score and represented in descending order.

3.5. FINAL ORDERING OF EACH PARTITE

In above step intra partite test cases ordering is measured using failure history and degree of connectivity. Execution sequence of each partite is ordered to cover different function and thus detect divergent faults at early. For this purpose, weighted score of each partite is measured using the cumulative score of every test cases inside a partite. Finally, the partites are ordered in descending based on the measured total weighted score. Random algorithm is used to assign execution order for tie case problem. The overview architecture of the proposed approach is demonstrated in Fig. 2.

Fig. 2 Proposed Framework Overview

3.6. SUMMARY

The proposed multipartite based prioritization strategy uses function call dependency graph to identify similarity between the test cases, which are ranked in descending order based on the failure history and degree of connectivity. The proposed test case orders will be effective for early variance fault detection in regression testing.

4. EXPERIMENTAL RESULTS AND DISCUSSION

This section discusses the whole experimental approach of the proposed prioritization strategy along with its evaluation of divergent function coverage and thus fault detection.

The performance of multipartite prioritization is compared to several most prominent test case prioritization approaches.

4.1. DATASET

Projects from well-known defects4j dataset [15] have been used for experimentation of proposed and compared techniques. The defects4j dataset provides 357 bugs and 20,109 test cases in different java projects which are shown in Table II. All the test cases consider as a single test suite for implementation. Several faults are also mutated inside these projects for experiment. Four selected java projects are used to analysis the prioritization approaches.

Table II. Dataset Details

Identifier	Project	# Bugs	# TCs
Closure	Closure compiler	133	7,927
Chart	JFreechart	26	2,205
JTime	Joda-Time	27	2,245
Lang	Apache commons-lang	65	4,130

4.2. EXPERIMENTAL PRIORITIZATION METHODS

Proposed Multipartite Test case Prioritization (MTP) results are compared with existing four prominent prioritization methods named as Untreated Test case Prioritization (UTP), Random Test case Prioritization (RTP), Similar Test case Prioritization (STP), and Dissimilar Test case Prioritization (DTP). The description of those approaches is listed in Table III.

Table III. Comparison with existing methods

Label	Description
UTP	Normal order of test cases (TCs) of a test suite
RTP	Random order of TCs of a test suite
STP	Similarity based TCs clustering and ordered with fault detection average weight
DTP	Dissimilarity based TCs clustering and ordered with failure history and DoC
MTP	Multipartite based test case partition and ordered with failure history and DoC

4.3. PERFORMANCE MEASUREMENT METRIC

Average Percentage of Faults Detection (APFD) is used to calculate the accuracy of test case prioritization approaches [4]. The limit of APFD result is 0 to 100, where higher number indicates faster fault detection rate. Let a test suite T contains n test cases, and let $F_1 \dots F_m$ be m faults revealed by test suite T ; i is the position number of first test case of test suite T which detects fault F_i . The APFD is calculated using the Equation 3.

$$APFD = 1 - \frac{TF_1 + TF_2 + TF_3 + \dots + TF_m}{n \times m} + \frac{1}{2n} \quad (3)$$

The statistical analysis of prioritization methods' accuracy has been measured by APFD metric, which indicate the effectiveness of prioritization technique [4].

4.4. RESULT ANALYSIS

Comparison results of proposed MTP with UTP, RTP, STP, and DTP methods have been described briefly and shown in Table V. APFD percentage of different test case input size for different techniques is shown in Table IV. And Fig. 3 represents APFD of five prioritization methods for all dataset.

Table IV. APFD Based on Input Size

Data Sets	Methods	APFD (%)				
		20%	40%	60%	80%	100%
Closure V2	UTP	36.53	36.53	42.30	100	100
	RTP	19.61	45.38	57.69	100	100
	STP	2.00	2.00	60.00	100	100
	DTP	100	100	100	100	100
	MTP	100	100	100	100	100
Closure V3	UTP	34.61	40.38	46.15	100	100
	RTP	28.84	56.92	68.07	97.69	100
	STP	0.00	0.00	84.61	84.61	100
	DTP	90	90	90	100	100
	MTP	56	90	100	100	100
Chart V2	UTP	10.52	10.52	42.10	100	100
	RTP	14	32.63	64.21	88.37	100
	STP	26.31	26.31	57.89	100	100
	DTP	89	89	100	100	100
	MTP	89	100	100	100	100
Chart V3	UTP	0	0	29.16	83	100
	RTP	20	36.65	62.50	100	100
	STP	54	88	100	100	100
	DTP	79	79	79	96	100
	MTP	83	83	83	100	100
JTime V2	UTP	0	20	20	100	100
	RTP	10	18	64	94	100
	STP	80	90	100	100	100
	DTP	80	100	100	100	100
	MTP	90	100	100	100	100
JTime V3	UTP	0	28	28	100	100
	RTP	17	45.60	63	91	100
	STP	72	88	92	100	100
	DTP	100	100	100	100	100
	MTP	100	100	100	100	100
Lang V2	UTP	11	27	27	82	100
	RTP	24	43	43	90	100
	STP	36	89	100	100	100
	DTP	82	84	87	100	100
	MTP	80	84.44	97.77	100	100
Lang V3	UTP	13	15	54	80	100
	RTP	24	34	85	92	100
	STP	68	75	84	100	100
	DTP	78	84	94	100	100
	MTP	76.19	90.47	92.06	100	100

According to Fig. 3 average APFD values of UTP, RTP, STP, DTP and MTP are 50.16%, 50.02%, 66.64%, 77.05% and 83.53% respectively. This result shows that proposed strategy performs better than existing four methods for all (Closure, Chart, JTime and Lang) datasets. It also indicates that MTP detects faults faster than other techniques as its APFD value is higher than others. The comparative results of proposed MTP with four prominent mentioned methods are listed below:

UTP vs MTP: In the experiment, for every project of two versions, MTP performs better than UTP which is shown in Table IV. For example, Closure V3 dataset MTP APFD values for different input sizes are 56%, 90%, 100%, 100%, and 100% where UTP APFD values are 34.61%, 40.38%, 46.15%, 100%, 100% respectively. Fig. 3 depicts the area under the curve represents the APFD, and it shows that proposed MTP method APFD is 83.53%, which is higher than UTP APFD 50.16%.

Fig 3 APFD of All Methods

RTP vs MTP: According to Table IV, RTP APFD is lower than proposed MTP method. In the experimental analysis for dataset Chart V3 the MTP APFD values for different input sizes are 83%, 83%, 83%, 100%, 100%, which are higher than RTP APFD values 20%, 36.65%, 62.50%, 100%, and 100% respectively. As stated in Fig. 3, APFD value of RTP is 50.02%, which is lower than MTP APFD 83.53%.

STP vs MTP: The proposed MTP APFD is higher than STP as shown in Table IV. In the investigation for dataset JTime V2 the MTP APFD values for different input sizes are 90%, 100%, 100%, 100%, 100%, which are higher than STP APFD values 80%, 90%, 100%, 100%, and 100% respectively. As shown in Fig. 4, the STP APFD is 66.64, which is lower than MTP strategy APFD 83.53%.

DTP vs MTP: In this paper the proposed method MTP detects faults earlier than DTP in 75% cases which is measure according to the APFD metric. According to the Table 4 for dataset Lang V2, the calculative APFD for proposed MTP method for various input size are 80%, 84.44%, 97.77%, 100%, 100% where DTP APFD are

82%, 84%, 87%, 100% and 100% respectively. According to Fig. 4 the area under the curve represents the APFD, and it shows that MTP method APFD is 83.53%, which is higher than all other APFD value.

Table V. APFD Comparison

Project	Version	APFD%				
		UTP	RTP	STP	DTP	MTP
Closure	V2	46.58	47.94	72.10	95.35	95.46
	V3	48.67	48.02	69.249	91.84	90.42
Chart	V2	49.82	52.39	42.87	63.64	62.51
	V3	29.12	50.70	51.25	56.63	70.64
JTime	V2	42.21	49.12	98.76	92.03	96.99
	V3	44.40	51.96	50.53	59.87	76.65
Lang	V2	74.75	49.69	70.21	77.15	88.44
	V3	65.71	50.35	78.17	79.91	87.12
Average		50.16	50.02	66.64	77.05	83.53

4.5. DISCUSSION

Table V and Fig. 4 show the results of experimented Closure, Chart, JTime and Lang projects. The Table V shows that the performance rankings of five different test case prioritization techniques are $UTP < RTP < STP < DTP < MTP$ in terms of APFD metric calculation. According to 5 UTP and RTP test case prioritization techniques APFD values are always lower than proposed MTP technique, where 87.50% cases MTP has higher APFD value than STP and 75% cases MTP has higher APFD value than DTP method. The box-plot of Fig. 4 represents the average APFD value of each test case prioritization scheme of four projects where MTP has higher value 83.53% compare to UTP, RTP, STP and DTP 50.16%, 50.02%, 66.64%, and 77.05% value respectively.

Fig. 4 APFD Comparisons

In addition to result comparison in Table V, Fig. 5 represents the total percent of test cases necessary to detect all faults for each test case prioritization method. The lower value indicates faster fault detection rate that means few numbers of test cases can decrease testing cost by detecting all faults faster. According to Fig. 5 UTP, RTP, STP, DTP and MTP execute 87.5%, 95%, 75%, 60% and 52.5% test cases respectively to detect all faults. From this comparison, it is shown that proposed technique executes only 52.5% test cases to detect all faults. It can

therefore be concluded that the multipartite prioritization framework can reduce the test execution time and maximize the fault detection.

Fig 5 Percent of Test Case to Detect All Faults

4.6. THREATS TO VALIDITY

Internal Validity: The weighted score accuracy of this framework depends on the correctness of failure history profile and *DoC* value. Incorrect failure history profiling or higher numbers of *DoC* value may miss lead the prioritization objectives. Hence, the greatest concern for internal validity involves the accuracy of listing test cases with their corresponding faults detected in previous execution. To control this threat, APFD and weighted score are calculated with four projects average value and normalized *DoC* respectively. In addition, proposed methodology may little bit time consuming for constructing underlying test case execution call graph and combining previous failure history.

External Validity: The proposed approach performs the worst in two cases, (i) when test cases have no inter function connectivity and (ii) all test cases are inter connected. In first case, when all test cases are dissimilar will form a single partite. On other hand, when all test cases are similar, number of partite will be equal to number of test cases.

4.7. SUMMARY

Four different java projects from defects4j have been used as dataset for experimental analysis. Old and current version test suites of each project have been used to identify test cases similar to failed in previous execution. The investigation suggests that it is better to use multipartite based prioritization when volume of test suite is larger.

5. CONCLUSION

Test case prioritization technique reduces software testing time and cost by maximizing fault detection and minimizing the percentage of execution. In the proposed method, call dependency graph has been formed using test case connectivity, and the partites are created from that graph, so test cases in each partite are distinct. To measure intra partite test cases weighted score, failure history and degree of connectivity have been used. Finally, cumulative partite weights are calculated to order final execution sequence. Defects4j dataset and

APFD values have been used for experimentation and result measurement respectively. The average APFD value of each prioritization method shows that proposed scheme has higher value (83.53%) compare to untreated, random, similar, and dissimilar clustering schemes, which get 50.16%, 50.02%, 66.64%, and 77.05% value respectively. The proposed multipartite method performs 14.28% and 42.85% better than dissimilarity and similarity based approaches respectively for early fault detection. Efficiency of failure history based test case prioritization can be improved categorizing the previously detected faults using their severity, which is one of the potential future works to find out highly impactful faults earlier.

REFERENCES

- [1] Rothermel G, Untch RH, Chu C, Harrold MJ. Prioritizing test cases for regression testing. *IEEE Transactions on software engineering*. 2001 Oct;27(10):929-48.
- [2] Nguyen CD, Marchetto A, Tonella P. Test case prioritization for audit testing of evolving web services using information retrieval techniques. In *Web Services (ICWS)*, 2011 IEEE International Conference on 2011 Jul 4 (pp. 636-643). IEEE.
- [3] Catal C. The ten best practices for test case prioritization. In *International Conference on Information and Software Technologies 2012 Sep 13* (pp. 452-459). Springer, Berlin, Heidelberg.
- [4] Rothermel G, Untch RH, Chu C, Harrold MJ. Test case prioritization: An empirical study. In *Software Maintenance. (ICSM'99) Proceedings. IEEE International Conference on 1999* (pp. 179-188). IEEE.
- [5] Catal C, Mishra D. Test case prioritization: a systematic mapping study. *Software Quality Journal*. 2013 Sep 1;21(3):445-78.
- [6] Arafeen MJ, Do H. Test case prioritization using requirements-based clustering. In *Software Testing, Verification and Validation (ICST)*, 2013 IEEE Sixth International Conference on 2013 Mar 18 (pp. 312-321). IEEE.
- [7] Siddik MS, Sakib K. RDCC: An effective test case prioritization framework using software requirements, design and source code collaboration. In *Computer and Information Technology (ICCIT)*, 2014 17th International Conference on 2014 Dec 22 (pp. 75-80). IEEE.
- [8] Wang X, Zeng H. History-based dynamic test case prioritization for requirement properties in regression testing. In *Proceedings of the International Workshop on Continuous Software Evolution and Delivery 2016 May 14* (pp. 41-47). ACM.
- [9] Kim JM, Porter A. A history-based test prioritization technique for regression testing in resource constrained environments. In *Proceedings of the 24th international conference on software engineering 2002 May 19* (pp. 119-129). ACM.
- [10] Cho Y, Kim J, Lee E. History-based Test Case Prioritization for Failure Information. In *2016 23rd Asia-Pacific Software Engineering Conference (APSEC) 2016 Dec 6* (pp. 385-388). IEEE.
- [11] Konsaard P, Ramingwong L. Total coverage based regression test case prioritization using genetic algorithm. In *Electrical engineering/electronics, computer, telecommunications and information technology (ECTI-CON)*, 2015 12th international conference on 2015 Jun 24 (pp. 1-6). IEEE.
- [12] Mirarab S, Tahvildari L. A prioritization approach for software test cases based on bayesian networks. In *International Conference on Fundamental Approaches to Software Engineering 2007 Mar 24* (pp. 276-290). Springer, Berlin, Heidelberg.
- [13] Zhao X, Wang Z, Fan X, Wang Z. A Clustering-Bayesian network based approach for test case prioritization. In *Computer Software and Applications Conference (COMPSAC)*, 2015 IEEE 39th Annual 2015 Jul 1 (Vol. 3, pp. 542-547). IEEE.
- [14] Hasan MA, Rahman MA, Siddik MS. Test Case Prioritization Based on Dissimilarity Clustering Using Historical Data Analysis. In *International Conference on Information, Communication and Computing Technology 2017 May 13* (pp. 269-281). Springer, Singapore.
- [15] Just R, Jalali D, Ernst MD. Defects4J: A database of existing faults to enable controlled testing studies for Java programs. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis 2014 Jul 21* (pp. 437-440). ACM.
- [16] Nayak S, Kumar C, Tripathi S. Effectiveness of prioritization of test cases based on Faults. In *Recent Advances in Information Technology, 3rd International Conference on 2016 Mar 3* (pp. 657-662). IEEE.
- [17] Noor TB, Hemmati H. A similarity-based approach for test case prioritization using historical failure data. In *2015 IEEE 26th International Symposium on Software Reliability Engineering (ISSRE) 2015 Nov 1* (pp. 58-68). IEEE.
- [18] Siddik S, Gias AU, Khaled SM. Optimizing software design migration from structured programming to object oriented paradigm. In *Computer and Information Technology (ICCIT)*, 2013 16th International Conference on 2014 Mar 8 (pp. 1-6). IEEE.
- [19] Marijan D. Multi-perspective regression test prioritization for time-constrained environments. In *Software Quality, Reliability and Security (QRS)*, 2015 IEEE International Conference on 2015 Aug 3 (pp. 157-162). IEEE.

- [20] Jiang B, Zhang Z, Chan WK, Tse TH. Adaptive random test case prioritization. In Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering 2009 Nov 16 (pp. 233-244). IEEE Computer Society.
- [21] Zhang L, Hao D, Zhang L, Rothermel G, Mei H. Bridging the gap between the total and additional test-case prioritization strategies. In Proceedings of the 2013 International Conference on Software Engineering 2013 May 18 (pp. 192-201). IEEE Press.
- [22] Rothermel G, Harrold MJ. A safe, efficient regression test selection technique. ACM Transactions on Software Engineering and Methodology (TOSEM). 1997 Apr 1;6(2):173-210.
- [23] Wong WE, Horgan JR, London S, Agrawal H. A study of effective regression testing in practice. In Software Reliability Engineering, 1997. Proceedings., The Eighth International Symposium on 1997 Nov 2 (pp. 264-274). IEEE.
- [24] Murphy GC, Notkin D, Griswold WG, Lan ES. An empirical study of static call graph extractors. ACM Transactions on Software Engineering and Methodology (TOSEM). 1998 Apr 1;7(2):158-91.
- [25] Md. Abdur Rahman, Md. Abu Hasan and Md. Saeed Siddik. Prioritizing Dissimilar Test Cases in Regression Testing using Historical Failure Data. International Journal of Computer Applications 180(14):1-8, January 2018.