Journal of Theoretical and Applied Information Technology
30th September 2018. Vol.96. No 18
© 2005 – ongoing JATIT & LLS

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

TEST CASE PRIORITIZATION TECHNIQUE FOR EVENT SEQUENCE TEST CASES BASED ON REDUNDANCY FACTOR

[1]JOHANNA AHMAD, [2]SALMI BAHAROM, [3]MYZATUL AKMAM SAPAAT
[1,2]Department of Information System and Software Engineering, Faculty of Computer Science and Information Technology, Universiti Putra Malaysia
[3]Malaysian Administrative Modernization and Management Planning Unit (MAMPU), Putrajaya, Malaysia
E-mail: [1]anna_lee207@yahoo.co.uk, [2]salmi@upm.edu.my, [3]angahmyz@gmail.com

ABSTRACT

Software testing is often used to verify and validate the output of the system to confirm that no discrepancy has taken place throughout the development phase. Test case prioritization (TCP) is one of the techniques applied to modify the order of test cases based on best test scenarios and to prioritize them. The main objectives of the TCP are to increase the effectiveness of the testing process, while reducing time and cost, which would increase when the system reaches a certain level of complexity. Numerous TCP techniques have been proposed in the past; however, only a handful of researches were truly focused on TCP techniques for test cases involving the sequence of events. TCP technique for sequence of events is more complex compared to the conventional code-based application due to the properties of the sequence of events. The size of the sequence of events' test cases can be infinite and large sized test cases have considerable degrees of redundancy. This means that there is a possibility for these test cases to have combinations of events with a large input parameter. Redundancy is one of the major issues that have been discussed by previous researchers. This paper proposes a technique that can detect the redundancy within the test suites and produce a unique weight value. This paper will also present how test cases were prioritized based on the obtained unique weight value. The experiment results obtained indicates that the prioritized test suite is effective compared with the original test suite. The effectiveness of the proposed approach is evaluated using Average Percentage of Faults Detected (APFD).

Keywords: *Test Case Prioritization, Software Testing, Unique Weight, Event Sequences.*

1. INTRODUCTION

A large volume of published studies have described the effectiveness of the TCP technique compared to the execution of test cases in a non-prioritized order [1]. Numerous approaches have been developed to prioritize test cases based on code coverage, function coverage, and requirement coverage [1]. Furthermore, different researchers have proposed different combinations of factors, but with the same aims and goals. The primary goal of the TCP is to maximize the rate of fault detections, whereby it must be able to detect faults as early as possible during the testing process [2]. Furthermore, TCP should be able to reveal faults whenever specific codes are changed, and to reveal high risk faults during the early stage of the testing process.

A majority of the previous researches applied TCP technique in single event test cases. Only a few researches applied the TCP technique in event sequence test cases. The flexibility of event sequence test cases enables the combination of interactions [3]. However, it is very tedious and wastes time to test all the possible combinations of events [4]. The size of the test cases can be larger because of this reason. This characteristic makes the application of event sequences even more complex compared to traditional applications due to the possibility of the former having infinite input domain. Furthermore, previous researchers have neglected to address several issues, such as not considering the transitions between states and the value of the internal data states. It is important to consider the close connection between states and events to avoid sensitivity problems and the state-based execution of event sequences test case [5]. A number of researchers have considered redundancy as an important factor that can improve the effectiveness of the TCP technique. Other factors that could influence the effectiveness of the TCP technique are fault matrix, complexity, execution time, and permutation. According to [6], test cases within a suite have been compared to determine the redundancy inside the test suite. However, previous researchers have not dealt with the value of data state.

This study is a part of an on-going research towards enhancing the existing TCP technique for event sequences. This paper will present the application of the redundancy factor in the TCP process by considering the values of internal data states. Two types of redundancies have been defined; redundancy type 1, and redundancy type 2. Redundancy type 1 is linked to the redundancy of same data state, which may reoccur, but in different positions within a test case. Meanwhile, redundancy type 2 is for a scenario where some of date state in a test case is a subset of another test case within the test suite. Executing the same test case more than once would be impractical and would be a waste of time, efforts, and costs during the testing phase. Furthermore, the weighting method will be applied to all test cases to calculate the final priority value. If similar priority values exist during the final weight calculations, Jaccard Distance would be used to break the ties. Meanwhile, the Average Percentage of Faults Detected (APFD) would be used to evaluate the effectiveness between prioritized and original test suites.

The remainder of this paper is organized as follows. Section 2 describes the previous works related to TCP techniques. Section 3 contains the proposed work, and its results are described in Section 4. Conclusions are drawn in Section 5, and finally, the acknowledgement is presented in Section 6.

2. RELATED WORKS

There are currently three common techniques that have been developed to improve the effectiveness of the testing phase, the test suite minimization, the test case selection, and the test case prioritization [7]. The test suite minimization is used to remove redundant test cases, which would reduce the size of the test suite. The test case selection is used to select some of the test cases, and to change some portion of the codes. The TCP technique, as previously mentioned, schedules the test case ordering to maximize the rate of fault detection. The objective of the TCP is to detect problems as early as possible, using its improved fault detection rate and to deliver the application at the shortest period of time [8]. Test cases with the highest priorities will be executed earlier than test cases with lower priorities [9].

However, some researchers have found that test suite minimization and test case selection are capable of reducing the duration of the testing phase, but both can cause the increase of costs [10]. Meanwhile, other researchers claimed that test suite minimization can reduce fault detection capability [11]. Several empirical studies have been conducted to refute the statement that test suite minimization can reduce fault detection capability [12]. This view is supported by Wong, Horgan, Mathur, Lafayette, & Pasquini (1997), who claimed that the impact of the test suite minimization can be calculated based on the percentage of reduction in the fault detection.

Generally, a test case selection would have the same problem as the test suite minimization, which is to choose a subset of test cases in the test suite [11]. Processing a subset of test case is important because the execution of repeated test cases would be ineffective, especially for projects that have limited time, resources, and cost. Instead of having to choose a subset from the test case, the test selection consists of how a specific technique is defined and sought out, and how to identify the modifications in the program being tested [11]. Previous researches have proposed a number of approaches using different techniques and criteria, such as the integer programming, data-flow analysis, symbolic execution, dynamic slicing, and textual difference in source code. Integer programming was the first technique proposed in 1977.

A considerable amount of literature has been published regarding the TCP technique since it was first proposed in 1970. Most TCP researches have provided clear evidence that these prioritization techniques are beneficial for the testing phase because of their capability to detect errors as early as possible, which could reduce time, cost, and resources [7][10][14]. Based on the literature review, most researchers believed that apparently, the rate of fault detection would be increased when various factors and attributes interact with one another [15]. Furthermore, fault detection offers the highest chance to break ties in cases of two or more test cases with the same priority value. One of the main issues is to prioritize test cases that may have the same priority value during the TCP processes. The reviewed literature indicated that cases will be randomly picked if the case of a tie ever occurs.

According to the systematic literature review (SLR) conducted for this study in 2016 [16], one of the research questions was, how far had the previous researches considered the issue of same priority value? Based on the SLR analysis, it was concluded that only two studies had considered this issue. In 2010, [17] reported that researchers have tried to solve the issue of same priority value by proposing four stages of factors, namely, the defect factors, time factors, cost factors, and complex factors. However, if the problem is not solved during the fourth stage, the random technique will be performed. Random technique is known as an ineffective technique since it creates bias [1][18][2]. Meanwhile, [18] had successfully enhanced the existing TCP technique, which uses five coverage criteria; branch coverage, statement coverage, fault coverage, function coverage, and path coverage. By considering all five coverage, [18] had managed to obtain unique values for each of the test case, by delaying other test cases that have the same weight, with the assumption that they cover the same segment code.

Thus, a few studies have proposed that the TCP technique should use the multiple criteria, with the aim of avoiding tie cases from occurring. However, the perfect combinations of criteria still need to be investigated. Thus, throughout the years, a huge number of TCP techniques have been proposed, with a variety of combination factors to improve the effectiveness of the test case generation [1][18];[19];[17];[20];[21]. As mentioned in Section 1, this study was focused on the event sequence test cases and the data state values. Based on the SLR analysis conducted in 2016 [16], only 36 percent of the previous researches have applied event sequence test cases in their TCP technique. The main problem with the event sequence test cases, in terms of the length of the sequences, was that they can be unbounded. The possible permutations of the event sequences can cause the test case to become very large [4]. Thus, the implementation of the TCP technique for event sequence test cases may differ, and more complex compared to single event test cases.

This study had also conducted a second SLR in 2017 to find the perfect combinations of factors that can influence the effectiveness of the TCP technique. Based on 70 primary studies, which were published between 2005 and 2016, this study had managed to compose a list of the most utilized factors. Based on this SLR analysis, the top-ranked factors were fault matrix, redundancy, complexity, frequency, and requirements. This study had applied only one factor; redundancy. The detection of redundancy was in terms of the data state value in the test suite. [22] had proposed a redundancy approach, whereby the calculations involved the similarity degree of the identical transitions between two test cases as paths. Meanwhile, [23] had extended the existing redundancy by excluding any repeated transitions, and had proposed an approach that can calculate average path lengths and set the distinction that occurred in both test cases. [23] presented five more distance functions in their research in order to detect any sequencing, matching, or repetitions in the test suite.

[24] reported that by assigning weight to each of the test case had helped to rank the test cases based on their importance. The categories of the events were grouped based on the sensitivity of the events, as proposed by [25]. However, the proposed approach did not consider data state values in the event sequences. In this study, a weight-based method was proposed, using the redundancy detections. Next, the test cases were ranked based on the final weight gained at the end of the experiment.

3. PROPOSED WORK

This study has proposed a TCP technique based on the redundancy factor. The proposed technique applied the weighted method, whereby it will schedule the test cases based on their priority value. In this study, there is no minimization or elimination once redundancy is detected. Each of the test cases will have its priority value based on previously defined criteria. The highest priority value will be ranked as the first test case to be executed, and followed by other priority values. It is possible for the test case to have combinations of events and inputs parameter, which might affect the size of the test suites. Therefore, there are the possibilities for the test case to have redundant data state. When a test case is marked as redundant type 1, it will be checked for redundancy type 2 processes as well. Fig. 1 shows an overview of the proposed approach.

3.1 Redundancy in Test Case (Type 1)

This paper proposes a 2-phase approach to solve test case prioritization. In the first phase, the number of data state *(No.of ds, tc_j)* and the number of redundant data state *(No. of redundant ds, tc_j)* for each of the test case were calculated. Both values were used to calculate dissimilarity weight in the test case *($DWtc_j$)*. This type of redundancy was defined as redundancy type 1. Dissimilarity in weight in a test case is calculated as shown below:

$$DWtc_j = (\text{No ds } tc_j - \text{No of redundant ds } tc_j)/10 \quad (1)$$

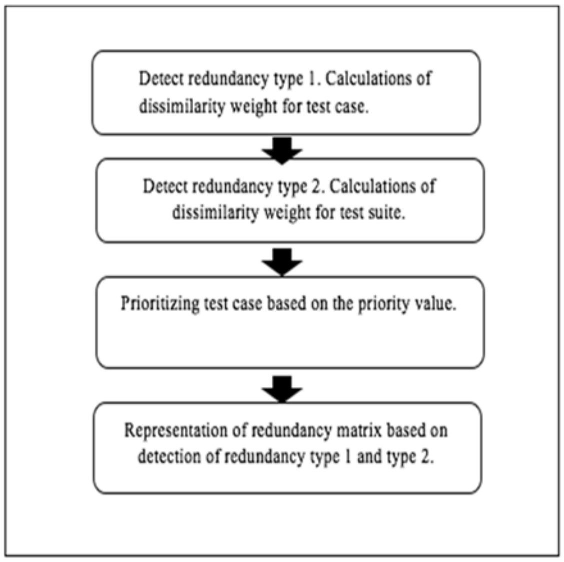
Detect redundancy type 1. Calculations of dissimilarity weight for test case.
Detect redundancy type 2. Calculations of dissimilarity weight for test suite.
Prioritizing test case based on the priority value.
Representation of redundancy matrix based on detection of redundancy type 1 and type 2.

Figure 1: Steps in Proposed Approach

3.2 Redundancy within Test Suite (Type 2)

When a test case is marked as redundant type 1, it will be checked for redundancy type 2 processes too. The second phase has been defined as redundancy type 2, whereby the process of redundancy detection is within the test suite. For this phase, the total number of data state within a test suite (No. of ds, ts_j), the number of redundant data state within test suite (No. of redundant ds, ts_j), the number of non-redundant data state within test suite (No.of non-redundant ds, ts_j), and the dissimilarity weight within test suite ($DWts_j$) were calculated. Finally, the values of $DWtc_j$ and $DWts_j$ were used to rank all the test cases. Higher values in $DWtc_j$ and $DWts_j$ were prioritized to be executed earlier than others. Then, the redundancy matrix was developed for a comprehensive report and to improve visibility. With the redundancy matrix, it was easier to detect and find redundancy that existed between test cases. The following formulas were used to calculate the dissimilarity weight in test suites.

$$\text{No non redundant ds } ts_j = \text{No ds } ts_j \,/\, \text{No of redundant ds } ts_j \tag{2}$$

$$DWts_j = \text{No non redundant ds } ts_j \,/\, \text{No of ds } ts_j \tag{3}$$

4. CASE STUDY

A total of 28 test cases were used as part of the case study to evaluate the effectiveness of the proposed technique. Table 1 shows the list of test cases, which were taken from a simple circular queue program, as the case study in this research. As mentioned in Section 3, there were two types of redundancy; type 1 and type 2. During the early stage, five variables must be calculated before moving to the second stage, which is redundancy type 2. First, the data state value for each test case was calculated. In addition to the data state value, a list of test cases that have redundant data state values were also identified. Then, two more variables were calculated, namely, the number of data state, and the number of redundant data state in the test case. Dissimilarity weight in the test case became the final value for redundancy type 1. Details of the calculations are as shown in Section 3.1. Examples of calculations that were involved in stage redundancy type 1 are shown below.

4.1 Data State Value in Test Case

The case study is based on the circular queue concept. The behaviour of the circular queue program can be described by a constant QSIZE, which holds the length of the array size. For this case study, the QSIZE was 10. Meanwhile, other variables became the front, rear, and len. For example, if the QSIZE was set to 10, in case the QSIZE was equalled to 10, the next value will be set to zero. The front and rear would hold the first and final data. The data state value would start with the initial value. The initial and front event would not be considered as redundancy. The front event would be known as the insensitive access program, whereby it would only display the output without causing any changes on the data structure in any condition [26]. When the process of adding data begins in the circular queue, the process starts from the rear and the value would be increased by 1. Nevertheless, the removal process would begin from the front of a queue, and the value of the front event would be increased by 1 [26]. The following Eq. (4) and Eq. (5) were used to calculate values for the front and rear:

$$\text{front} = (\text{front} + 1)\%\ \text{qSize} \tag{4}$$

$$\text{rear} = (\text{rear} + 1)\%\ \text{qSize} \tag{5}$$

Table 2 shows examples of the calculations of data state values for TC1. Based on the table, the redundancy had occurred in the array number of 1, 2, and 3, whereby the front value was 0, the rear value was 9, and the length value was 0. Furthermore, redundancy was found in TC5, TC6, TC8, TC9, TC19 and TC21. Table 2 shows that TC1 was the subset of other test cases in the test suite. Normally, previous researchers would apply test suite minimization, whereby they would eliminate the test case if it was a subset in the test suite. However, as previously mentioned, this paper does not feature any elimination process, based on the concept that all test cases may have the capability to detect errors. However, the execution was ranked based on the priority value.

Table 1: List of Test Cases.

Test Case	Sequence of Events
TC1	_.remove().remove().remove().add(1).front()
TC2	_.add(1).add(1).add(1).add(1).remove().add(1).add(1).add(1).add(1).remove().add(1). add(1). add(1).add(1).front()
TC3	_.add(1).remove().add(1).remove().add(1). remove().add(1).remove().add(1).remove().add(1). remove().add(1).remove().add(1).remove().add(1).remove().add(1).remove().add(1).add(1). front()
TC4	_.add(1).add(1).add(1).add(1).front()
TC5	_.remove().add(1).remove().add(1).remove().add(1). front()
TC6	_.remove().add(1).add(1).add(1).add(1).add(1).add(1).add(1).add(1).add(1).add(1).front()
TC7	_.add(1).add(1).remove().add(1).add(1).remove().front()
TC8	_.remove().add(1).add(1).add(1).front()
TC9	_.remove().remove().add(1).remove().add(1).remove().add(1).front()
TC10	_.add(1).add(1).add(1).add(1).add(1).remove().add(1).add(1).add(1).add(1).add(1).remove(). remove().remove().remove().remove().remove().remove().remove().remove().add(1). add(1).front()
TC11	_.add(1).remove().remove().remove().front()
TC12	_.add(1).add(1).remove().remove().remove().remove().front()
TC13	_.add(1).add(1).add(1).remove().remove().remove(). add(1).add(1).front()
TC14	_.add(1).add(1).add(1).add(1).add(1).add(1).add(1).add(1).remove().remove().add(1).add(1). add(1).add(1).add(1).add(1).front()
TC15	_.add(1).add(1).add(1).add(1).add(1).add(1).add(1).add(1).remove().remove().add(1).add(1). add(1).add(1).add(1).add(1).front()
TC16	_. add(1).remove().remove().remove().remove().add(1) .add(1).front()
TC17	_. add(1).remove().front()
TC18	_. add(1).remove().remove().add(1).add(1). add(1).add(1).remove().front()
TC19	_. remove().remove().add(1).add(1). remove().front()
TC20	_.add(1).add(1).add(1).remove().add(1).add(1).add(1). add(1).add(1).add(1). add(1).remove(). remove() .remove().remove().remove().remove().remove().remove().remove().add(1).add(1). front()
TC21	_.remove().remove().remove().remove().remove().add(1).front()
TC22	_. add(1).remove().remove().add(1).add(1). front()
TC23	_. add(1).add(1).add(1).add(1).add(1).remove().remove().remove().remove().remove().add(1) . add(1).add(1). add(1).add(1).remove().remove().remove().remove() .remove().add(1).add(1). front()
TC24	_. add(1).add(1).add(1).add(1).add(1).add(1).add(1).add(1).add(1).add(1).remove().remove(). remove() .remove().remove().remove().remove().remove().remove().remove().add(1).add(1). remove().front()
TC25	_. add(1).add(1).add(1).add(1).add(1).front()
TC26	_. add(1).remove().add(1).front()
TC27	_. add(1).add(1).front()
TC28	_. add(1).remove().add(1).remove().front()

Table 2: Calculations of Data State in TC1.

Array No	Events	Data State Value			Found in Test Case
		Front	Rear	Length	
0	initial	0	9	0	Not available
1	remove	0	9	0	5, 6, 8, 9, 19, 21
2	remove	0	9	0	5,6, 8, 9, 19, 21
3	remove	0	9	0	5,6, 8, 9, 19, 21
4	add	0	0	1	5,6, 8, 9, 19, 21
5	front	0	0	1	Not available

4.2 Number of Data State in Test Case, Number of redundant data state in Test Case and Dissimilarity Weight in Test Case

Table 3 shows the values for the number of data states in test cases (No. ds tc_j), the number of redundant data state in test cases (No. of redundant ds, tc_j) and the dissimilarity weight in test case ($DWtc_j$). According to the table, dissimilarity weight of test case for TC1 was 0.00 because all the data states were redundant. Meanwhile, for TC21, the value for $DWtc_j$ was negative (-0.20) since the value of the redundant data state in the test case was higher than the number of data state in the test case.

Table 3: Calculations of Data State in Test Suite.

Test Case No	Number of Data State in Test Case (No ds tc_j)	Number of Redundant Data State in Test Case (No of redundant ds tc_j)	Dissimilarity Weight of Test Case ($DWtc_j$)
TC1	2	2	0.00
TC2	14	1	1.30
TC3	22	0	2.20
TC4	4	0	0.40
TC5	6	1	0.50
TC6	11	0	1.10
TC7	6	0	0.60
TC8	4	0	0.40
TC9	6	2	0.40
TC10	22	0	2.20
TC11	2	2	0.00
TC12	3	3	0.00
TC13	8	0	0.80
TC14	14	3	1.10
TC15	7	0	0.70
TC16	4	3	0.10
TC17	2	0	0.20
TC18	5	1	0.40
TC19	4	1	0.30
TC20	21	2	1.90
TC21	2	4	-0.20
TC22	4	1	0.30
TC23	22	3	1.90
TC24	23	3	2.00
TC25	5	0	0.50
TC26	3	0	0.30
TC27	2	0	0.20
TC28	4	0	0.40

4.3 Number of Data State in Test Suite, Number of Redundant Data State in Test Suite and Dissimilarity Weight in Test Suite

Table 4 shows the values of the number of redundant data states in test suite, the number of non-redundant data states in test suite, and dissimilarity weight in test suite. The total number of data states in the test suite was 219. The number of data state in test suite was used to calculate the number of non-redundant data state in test suite. For example, in TC1, the number of redundant data state in test suite showed that there were two redundant data state values within the test suite. In this case, the data state in the array number 1 and 3 were redundant with the TC5, TC6, TC8, TC9, TC19 and TC21.

4.4 Final Weight Table

After the process of detecting redundancy type 1 and type 2, all the values were added to the final weight table, to calculate the priority value for each test case. According to Table 5, the test cases were ranked based on their final weight value. However, four cases have the same final weight.

Table 4: Calculations of Dissimilarity Weight in Test Suite.

Test Case No	Number of Redundant Data State in Test Suite (No of redundant ds ts_j)	Number of Non Redundant Data State in Test Suite (No non redundant ds ts_j)	Dissimilarity Weight Within Test Suite ($DWts_j$)
TC1	4	215	0.98
TC2	4	215	0.98
TC3	8	211	0.96
TC4	4	215	0.98
TC5	6	213	0.97
TC6	11	208	0.95
TC7	3	216	0.99
TC8	4	215	0.98
TC9	7	212	0.97
TC10	8	211	0.96
TC11	4	215	0.98
TC12	3	217	0.99
TC13	3	216	0.99
TC14	8	211	0.96
TC15	7	212	0.97
TC16	4	215	0.98
TC17	2	217	0.99
TC18	5	214	0.98
TC19	0	219	1.00
TC20	9	210	0.96
TC21	6	213	0.97
TC22	5	214	0.98
TC23	9	210	0.96
TC24	14	205	0.94
TC25	5	214	0.98
TC26	3	216	0.99
TC27	2	217	0.99
TC28	4	215	0.98

As previously mentioned, if the same final weight value exists, previous researchers would often randomly pick the cases to break the ties. This study applied the Jaccard Distance to solve the same final weight value. In the first case, TC3 and TC10 have the same final weight value of 3.16. In the second case, TC4, TC8, TC18, and TC28 shared the same final weight of 1.38. The third case has a value of 1.19, which was shared by TC17, and TC27. The final case, TC1, TC11, and TC12 have the final weight value of 0.99.

Jaccard Distance is also known as the Jaccard Similarity Coefficient. The main objective for applying the Jaccard Distance was to compare the similarities and diversity of sample sets [27]. This study proposed the Jaccard Distance to solve the same priority value, to measure the similarities among test cases. Once a test case has been detected as having the same final weight value with other test cases, they would be grouped together. Therefore, as mentioned in Section 4.4, there were four cases, which meant that there were four different groups. In this context, the highest distance between the two test cases would be executed first compared to others. The Jaccard Distance was calculated using the following Equation 6:

$$\text{Jaccard Distance } (p_a, p_b) = 1 - \frac{|p_a \cap p_b|}{|p_a \cup p_b|} \qquad (6)$$

Table 5: Calculations of Dissimilarity Weight in Test Suite.

Test Case No	Dissimilarity Weight in Test Case ($DWtc_j$)	Dissimilarity Weight Within Test Suite ($DWts_j$)	Final Weight
TC3	2.20	0.96	3.16
TC10	2.20	0.96	3.16
TC24	2.00	0.94	2.94
TC23	1.90	0.96	2.86
TC20	1.90	0.95	2.85
TC2	1.30	0.96	2.28
TC14	1.10	0.96	2.06
TC6	1.10	0.95	2.05
TC13	0.80	0.98	1.78
TC15	0.70	0.97	1.67
TC7	0.60	0.99	1.59
TC25	0.50	0.98	1.48
TC5	0.50	0.97	1.47
TC4	0.40	0.98	1.38
TC8	0.40	0.98	1.38
TC18	0.40	0.98	1.38
TC28	0.40	0.98	1.38
TC9	0.40	0.97	1.37
TC19	0.30	1.00	1.30
TC26	0.30	0.99	1.29
TC22	0.30	0.98	1.28
TC17	0.20	0.99	1.19
TC27	0.20	0.99	1.19
TC16	0.10	0.99	1.09
TC1	0.00	0.99	0.99
TC11	0.00	0.99	0.99
TC12	0.00	0.99	0.99
TC21	-0.20	0.99	0.79

In this study, p_a *and* p_b represent test case number, and it consists of different set of event sequences. According to [27], the distance value varies between range 0 and 1. If the distance value is zero, meaning that p_a *and* p_b are same. However, if the distance value is 1, it indicates that there is no similarity between p_a *and* p_b. The similarity will be based on the data state value in the test case. For group 1, TC3 is the highest but it has same final weight with TC10. Through some observations, the researchers found out that TC3 and TC10 have same properties as follows:

- Number of data state in the test case.
- Number of redundant data state in the test case.
- Dissimilarity weight in test case.
- Dissimilarity weight in test suite.
- Number of events in the test case.

Table 6 shows the distance weight table the four groups. In group 1, TC3 and T10 will be measure with TC 24 since final weight 3.16 which is owned by TC3 and TC10. As can be seen from the Table 6, Jaccard Distance for TC3 and TC24 is 0.98 and Jaccard Distance for TC10 and TC24 is 0.82. As mentioned earlier, if the Jaccard Distance is 1, means there is no similarity between the two test cases. Furthermore, since final weight for TC24 is the third highest, then the highest Jaccard Distance will be the highest priority based on the concept of less similarity with TC24. For group 1, TC3 will be executed first compared with TC10. This condition is different with group 2, whereby based on Table 5; final weight for TC5 is higher than all the test cases in group 2. Therefore, the Jaccard Distance for each of the test cases will be measured with TC5. However, the sorting will be different with group 1, whereby in group 2, test case that have Jaccard Distance value near with 0 will be execute after TC5 with the concept of that test case have similarity with TC5, therefore the capability to detect faults may same with TC5. The final schedules after the Jaccard Distance calculations are TC28, TC18, TC8 and followed by TC4.

Table 6: Distance Weight Table

Group	Test Case No	Final Weight	Jaccard Index	Jaccard Distanc e
1	TC3	3.16	0.02	0.98
	TC10	3.16	0.18	0.82
2	TC4	1.38	0.00	1.00
	TC8	1.38	0.20	0.80
	TC18	1.38	0.36	0.64
	TC28	1.38	0.40	0.60
3	TC17	1.19	0.50	0.50
	TC27	1.19	0.17	0.83
4	TC1	0.99	0.00	1.00
	TC11	0.99	0.67	0.33
	TC12	0.99	0.14	0.86

Group 3 had applied the same concept as in group 2 since the final weight for TC22 was higher than

1.19. With the 0.09 difference, TC22 was used to measure the Jaccard Distance for TC17 and TC27. Then, TC17 was executed first compared to TC27 since the value of Jaccard Distance for TC17 was close to 0. Meanwhile, for group 4, after comparing with TC16, the schedule for this group began with TC11, TC12, and followed by TC1. Table 7 illustrates the final test case prioritization process for all test cases after the Jaccard Distances have been calculated. As mentioned in Section 3, this study has no minimization process. Once the same priority value exists, the test cases will be grouped according to their final weight value. The final schedule was within the group only, for example in group 1. Once TC3 was detected as having the highest Jaccard distance, TC10 was queued after TC3. TC10 was not queued as the last test case because the concept of 'at the beginning of the prioritization process' was implemented; TC10 still held the highest value compared to other test cases.

Table 7: Final Test Case Prioritization Table.

Queue No Before Jaccard Distance Process	Queue No After Jaccard Distance Process	Test Case No	Final Weight	Jaccard Distance
1	1	TC3	3.16	0.98
2	2	TC10	3.16	0.82
3	3	TC24	2.94	Not available
4	4	TC23	2.86	Not available
5	5	TC20	2.85	Not available
6	6	TC2	2.28	Not available
7	7	TC14	2.06	Not available
8	8	TC6	2.05	Not available
9	9	TC13	1.78	Not available
10	10	TC15	1.67	Not available
11	11	TC7	1.59	Not available
12	12	TC25	1.48	Not available
13	13	TC5	1.47	Not available
17	14	TC28	1.38	0.60
16	15	TC18	1.38	0.64
15	16	TC8	1.38	0.80
14	17	TC4	1.38	1.00
18	18	TC9	1.37	Not available
19	19	TC19	1.30	Not available
20	20	TC26	1.29	Not available
21	21	TC22	1.28	Not available
22	22	TC17	1.19	0.50
23	23	TC27	1.19	0.83
24	24	TC16	1.09	Not available
26	25	TC11	0.99	0.33
27	26	TC12	0.99	0.86
25	27	TC1	0.99	1.00
28	28	TC21	0.79	Not available

4.5 Redundancy Matrix

Matrix is widely used in software metrics. [28]had applied weight matrix and joint entropy matrix to represent the structural complexity of a class diagram. Meanwhile, [9] applied the adjacency matrix to show the composite control of flow graph for bank ATM systems. In this study, the redundancy matrix was used as a comprehensive report that shows the occurrence of redundancy type 1 and redundancy type 2 within the test suite. Compared to a structural report, the redundancy matrix would be easier to understand, especially for a large test suite. The proposed redundancy matrix consisted of X-axis (rows) and Y-axis (columns). It can be read starting from the X-axis (rows) and followed by the Y-axis (columns). Both the row and column indicated the relationship between two test cases. Three types of marks can be inserted in each cell, as shown in the following Table 8:

Table 8: Descriptions for the Redundancy Matrix.

Type	Descriptions
0	Test case (X) is compared with Test case (Y) and no redundancy detected
1	Exists redundancy TYPE 1 in the test case
2	Exists redundancy TYPE 2 within the test suite

The redundancy matrix is as shown in the annexure. From the redundancy matrix, for TC1, it can be summarised that there was a redundancy type 1 or there were redundancies of data state in the test case itself. Meanwhile, redundancy type 2 existed between TC1 and TC5, TC6, TC8, TC9, TC19 and TC21. The redundancy matrix can save a lot of time in terms of the process of verifying any redundancy within the test suite.

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

4.6 Measuring Effectiveness

The prioritized test cases were measured to prove their effectiveness to detect faults. In this study, the Average Percentage of Fault Detected (APFD) was applied. APFD represents the weighted average of the percentage of faults detected during the execution of the test suite [29]. The APFD values can range between 0 and 100, and the higher APFD value shows a higher fault detection capability [30]. A large number of researchers had chosen the APFD as a metric to measure the effectiveness of their proposed techniques[29];[21];[20];[9];[31]. Based on the systematic literature review (SLR) conducted for this study in 2016 [16], the most frequently used evaluation metric is the APFD with 58 percent. The SLR was conducted using 50 primary studies, which were selected after a few stages, as proposed by [32]. However, some of the previous researchers had applied more than one metric to achieve the same objective. According to [29],the APFD can be calculated as follows:

$$\text{APFD} = \quad 1 - \frac{\sum_{1}^{m} F}{mn} + \frac{1}{2n} \qquad (7)$$

The variable m refers to the number of faults, while the variable F belongs to the position of the first test case that detects fault m. The variable n refers to the total number of test cases in the test suite. In this study, one subject program, which was implemented in Java Language, namely Circular Queue was selected. Jester was used to create mutants for each of these subject programs. The number of mutants was varied when injected in the subject program. Table 9 shows the details of the mutants that were generated by the Jester using the mutation operators.

Table 9: Jester Mutation Operators.

Type	Descriptions
No	Mutation Operator
1.	Change numerical constants • Mutate 0 to 1 • Mutate 5 to 6 Mutate 9 to 0
2.	Flip Boolean values Mutate true to false and vice versa
3.	Mutate if (condition) to if (true || condition)
4.	Mutate if (condition) to if (false && condition)
5.	Mutate ++ to – and vice versa
6.	Mutate != to == and vice versa

The comparison is drawn between prioritized test suite and original test suite. As mentioned earlier, this study applied weighted method to produce a unique weight for each test case. Most of the previous TCP technique applied random technique once same priority exists during the prioritization process [17], [19]. From the Fig. 2, it is observed that the prioritized test suite managed to identify faults earlier. The results shows the abilities of the 28 test cases detect faults in early stage after the Circular Queue program has been injected by 16 faults. The APFD value for the prioritized test suite was 0.81, while the APFD value for the original test suite was 0.74. The results proved that the prioritized test suite had yielded better fault detections compared to the original test suite. Thus the prioritized test suite will reduce execution time by prioritizing the most important test case. In this study, the main observation is to measure the effectiveness of the proposed method. In future work, the researchers will try to apply the same method with case study from the industry. The APFD values for the prioritized test suite either higher or similar with the original test suite.

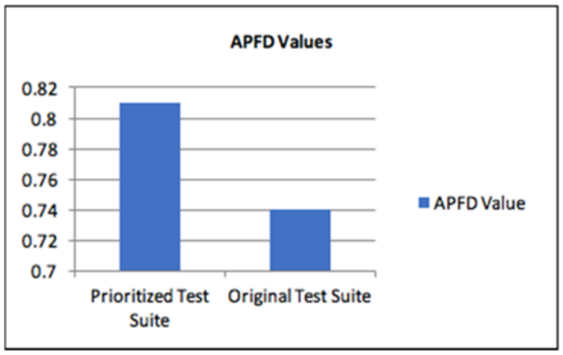

Figure 2: APFD Values for Prioritized Test Suite and Original Test Suite

5. CONCLUSIONS

This study has proposed a weight-based method based on the redundancy factor. The execution of the test case was sorted based on the weight value. The highest weight was executed earlier than the others, with the concept of the highest weight is more important, and more faults could be detected as early as possible. There were two types of redundancy; redundancy that occurred in the test case itself and redundancy that occurred within the test suite. The redundancy matrix was represented in this study as a summary and a comprehensive report, especially for large test suites. Then, when more than one test case shared the same weight, all test cases were grouped together, and would go through to the next stage, using the Jaccard Distance approach. The concept of when the Jaccard Distance value is close to 0 was used. Thus, the test case would be ranked as the first test case in the group, and then, followed by the others. This approach can break ties and solve the same priority value issue. For the experiments, 28 test cases have been chosen. Meanwhile, Jester and Circular Queue program written in Java language was selected for the process of fault detection. The Circular Queue program was injected by mutants that have already been identified by the Jester. Finally, the effectiveness of the proposed technique was measured using the APFD. This evaluation was conducted between the prioritized test suite and the original test suite. The final results were positive, and suggestions for future work would be to include more combinations of factors, which were listed as the top-ranked factors in the SLR of this study. However, one limitation has been found during the experiment; the program consists of one class only. Thus, we believed that there would be more steps involved during the prioritization process if the program consists of more than one class.

6. ACKNOWLEDGEMENTS

The authors would like to acknowledge the Universiti Putra Malaysia for the financial support under the Journal Publication Fund, Research Management Centre, Universiti Putra Malaysia.

REFRENCES:

[1] S. Gupta, H. Raperia, E. Kapur, H. Singh, and A. Kumar, "A NOVEL APPROACH FOR TEST CASE," *Int. J. Comput. Sci. Eng. Appl.*, vol. 2, no. 3, pp. 53–60, 2012.

[2] P. Tonella, P. Avesani, and A. Susi, "Using the Case-Based Ranking Methodology for Test Case Prioritization," *22nd IEEE Int. Conf. Softw. Maint.*, pp. 123–132, 2006.

[3] C. Klammer, R. Ramler, and H. Stummer, "Harnessing Automated Test Case Generators for GUI Testing in Industry," *2016 42th Euromicro Conf. Softw. Eng. Adv. Appl.*, pp. 227–234, 2016.

[4] S. Chaudhury, A. Singhal, and O. P. Sangwan, "Event- Driven Software Testing – An Overview," *Int. J. Adv. Res. Comput. Eng. Technol.*, vol. 5, no. 4, pp. 1189–1193, 2016.

[5] X. Yuan, M. B. Cohen, and A. M. Memon, "GUI interaction testing: Incorporating event context," *IEEE Trans. Softw. Eng.*, vol. 37, no. 4, pp. 559–574, 2011.

[6] P. A. Brooks and A. M. Memon, "Introducing a test suite similarity metric for event sequence-based test cases," *IEEE Int. Conf. Softw. Maintenance, ICSM*, pp. 243–252, 2009.

[7] C. Catal and D. Mishra, "Test case prioritization: a systematic mapping study," *Softw. Qual. J.*, vol. 21, no. 3, pp. 445–478, 2013.

[8] C. P. Indumathi and K. Selvamani, "Test Cases Prioritization Using Open Dependency Structure Algorithm," *Procedia Comput. Sci.*, vol. 48, no. Iccc, pp. 250–255, 2015.

[9] V. Panthi and D. P. Mohapatra, "Generating and evaluating effectiveness of test sequences using state machine," *Int. J. Syst. Assur. Eng. Manag.*, no. Jorgensen 2008, 2016.

[10] H. Do, S. Mirarab, L. Tahvildari, and G. Rothermel, "The Effects of Time Constraints on Test Case Prioritization: A Series of Controlled Experiments Software Engineering," *IEEE Trans. Softw. Eng.*, vol. 36, no. 5, pp. 593–617, 2010.

[11] J. Frolin S. Ocariza, G. Li, K. Pattabiraman, and A. Mesbah, "Automatic fault localization for client-side JavaScript," *Softw. Testing, Verif. Reliab.*, vol. Volume 21, no. Issue 3, pp. 195–214, 2015.

[12] W. Lafayette, "Effect of Test Set Minimization on Fault Detection Effectiveness," *J. Softw. Pract. Exp.*, vol. 28, no. July 1996, pp. 347–369, 1998.

[13] W. E. Wong, J. R. Horgan, A. P. Mathur, W. Lafayette, and A. Pasquini, "Test Set Size Minimization and Fault Detection Effectiveness : A Case Study in a Space Application *," *IEEE Comput. Soc. Int. Comput. Softw. Appl. Conf.*, no. 1, pp. 522–528, 1997.

[14] H. Do, G. Rothermel, and A. Kinneer, "Prioritizing JUnit Test Cases : An Empirical Assessment and Cost-Benefits Analysis," pp. 33–70, 2006.

[15] Y. T. Yu and M. F. Lau, "Fault-based test suite prioritization for specification-based testing," *Inf. Softw. Technol.*, vol. 54, no. 2, pp. 179–202, Feb. 2012.

[16] J. Ahmad and S. Baharom, "A Systematic Literature Review of the Test Case Prioritization Technique for Sequence of Events," *Int. J. Appl. Eng. Res.*, vol. 12, no. 7, pp. 1389–1395, 2017.

[17] S. Roongruangsuwan and J. Daengdej, "Test case prioritization techniques," *J. Theor.*

Appl. Inf. Technol., pp. 45–60, 2010.

[18] A. Ammar, S. Baharom, A. A. A. Ghani, and J. Din, "Enhanced Weighted Method for Test Case Prioritization in Regression Testing Using Unique Priority Value," 2016.

[19] C.-Y. Huang, C.-S. Chen, and C.-E. Lai, "Evaluation and analysis of incorporating Fuzzy Expert System approach into test suite reduction," *Inf. Softw. Technol.*, vol. 79, pp. 79–105, 2016.

[20] R. Pradeepa and K.VimalaDevi, "Effectiveness of Testcase Prioritization using APFD Metric : Survey," *Int. J. Comput. Appl.*, vol. 0975–8887, pp. 1–4, 2013.

[21] S. Sampath and R. C. Bryce, "Improving the effectiveness of test suite reduction for user-session-based testing of web applications," *Inf. Softw. Technol.*, vol. 54, no. 7, pp. 724–738, Jul. 2012.

[22] E. G. Cartaxo, D. L. Machado, and F. G. O. Neto, "On the use of a similarity function for test case selection in the context of model-based testing," no. July 2009, pp. 75–100, 2011.

[23] B. Coutinho, E. Gadelha, and A. Emı, *Analysis of distance functions for similarity-based test suite reduction in the context of model-based testing*. 2014.

[24] C. Y. Huang, J. R. Chang, and Y. H. Chang, "Design and analysis of GUI test-case prioritization using weight-based methods," *J. Syst. Softw.*, vol. 83, no. 4, pp. 646–659, 2010.

[25] M. Memon, M. E. Pollack, and L. Soffa, "Using a Goal-driven Approach to Generate Test Cases for GUIs," *Proc. 1999 Int. Conf. Softw. Eng.*, pp. 257–266, 1999.

[26] M. A. Sapaat and S. Baharom, "A Preliminary Investigation Towards Test Suite Optimization Approach for Enhanced State-Sensitivity Partitioning," no. November, pp. 40–45, 2011.

[27] A. B. Sanchez, S. Segura, and A. Ruiz-Cortes, "A Comparison of Test Case Prioritization Criteria for Software Product Lines," *Softw. Testing, Verif. Valid. (ICST), 2014 IEEE Seventh Int. Conf.*, pp. 41–50, 2014.

[28] Y. Zhou and B. Xu, "Measuring structural complexity for Class Diagrams : An Information Theory Approach," *Proc. ACM Symp. Appl. Comput.*, vol. 2, pp. 1679–1683, 2005.

[29] S. Elbaum, A. Malishevsky, and G. Rothermel, "Incorporating varying test costs and fault severities into test case prioritization," *Proc. 23rd Int. Conf. Softw. Eng. ICSE 2001*, pp. 329–338, 2001.

[30] H. Srikanth, C. Hettiarachchi, and H. Do, "Requirements based test prioritization using risk factors: An industrial study," *Inf. Softw. Technol.*, vol. 69, pp. 71–83, 2016.

[31] H. Srikanth, M. Cashman, and M. B. Cohen, "Test case prioritization of build acceptance tests for an enterprise cloud application: An industrial case study," *J. Syst. Softw.*, vol. 119, pp. 122–135, 2016.

[32] B. Kitchenham and S. Charters, "Guidelines for performing Systematic Literature Reviews in Software Engineering," 2007.

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

ANNEXURE 1:

	TC 1	TC 2	TC 3	TC 4	TC 5	TC 6	TC 7	TC 8	TC 9	TC 10	TC 11	TC 12	TC 13	TC 14	TC 15	TC 16	TC 17	TC 18	TC 19	TC 20	TC 21	TC 22	TC 23	TC 24	TC 25	TC 26	TC 27	TC 28
TC 1	1																											
TC 2	0	0																										
TC 3	0	0	0																									
TC 4	0	2	0	0																								
TC 5	2	0	0	0	0																							
TC 6	2	0	0	2	0	0																						
TC 7	0	0	0	0	0	0	0																					
TC 8	2	0	0	0	0	0	2	0																				
TC 9	2	0	0	0	2	0	0	0	1																			
TC 10	0	0	0	2	0	0	0	0	0	0																		
TC 11	0	0	2	0	2	0	0	0	2	2	1																	
TC 12	0	0	0	0	0	0	0	0	0	0	0	1																
TC 13	0	0	0	0	0	0	0	0	0	0	0	0	0															
TC 14	0	0	0	2	0	0	0	0	0	0	0	0	0	1														
TC 15	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0													
TC 16	0	0	0	0	0	0	0	0	0	0	2	0	0	0	0	1												
TC 17	0	0	2	0	2	0	0	0	2	2	2	0	0	0	0	2	0											
TC 18	0	0	2	2	0	0	0	0	0	0	2	0	0	0	0	2	2	0										
TC 19	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0									
TC 20	2	0	0	0	0	0	0	0	0	0	2	0	0	0	2	0	2	0	0	1								
TC 21	2	0	0	0	0	2	0	2	2	0	0	0	0	0	0	0	0	2	0	0	1							
TC 22	0	0	2	0	0	0	0	0	0	0	2	0	0	0	0	2	2	2	0	0	0	0						
TC 23	0	0	0	2	0	0	0	0	0	0	2	0	0	0	0	0	2	0	0	0	0	0	0					
TC 24	0	0	0	2	0	0	0	0	0	0	2	0	0	0	0	0	2	0	0	0	0	2	2	0				
TC 25	0	0	0	2	0	2	0	0	0	2	0	0	0	2	0	0	0	0	0	0	0	0	0	2	0			
TC 26	0	0	2	0	2	0	0	0	2	0	2	0	0	0	0	2	0	2	0	0	2	0	0	0	0	0		
TC 27	0	2	0	2	0	2	2	2	0	2	0	2	2	2	2	2	0	2	2	0	0	0	0	0	0	0	0	
TC 28	0	0	2	0	2	0	0	0	2	0	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	2	0	0