Distance-Based Weighted Prioritization for GUI Application Testing

Dmitry Nurmuradov, Renée Bryce
University of North Texas, Denton, TX, 76203
dmitrynurmuradov@my.unt.edu
renee.bryce@unt.edu

Abstract

Recording actual user interactions with a system is often useful for testing software applications. Users-session based test suites that contain records of such interactions often finds a complementary set of faults compared to test suites created by testers. This work utilizes such test suites and presents a new prioritization method that extends the existing combinatorial two-way inter-window prioritization by introducing weights on the distance between windows. We examine how a window distance between a pair of the parameter-value tuples influences the fault detection effectiveness. We evaluate several approaches used to calculate weights. Results show improvement over the original two-way inter-window prioritization technique, while the comparison of different weighting approaches reveals that a negative linear weighting calculation generally performs better in our experiments. The study demonstrates that the distance between windows in a pair is an important factor to consider in test suite prioritization, and that distinguishing windows by their order in a test case also improves the fault detection rate compared to using window labels that were utilized in previous methods. This work provides motivation for future work to develop general n-way combinatorial distance-based prioritization methods that take into account space and processing time requirements to address potential issues with large test suites.

JCMCC **102** (2017), pp. 77-98

1 Introduction

Today, more than 3 billion people use the Internet according to the International Telegraph Union [1]. Companies like Facebook have over a billion active users [2]. The large and diverse user base, combined with the increasing complexity of software, requires more rapid and rigorous software testing. Previously collected or generated test suites become an important part of the testing process. One technique to leverage existing test suites is test suite prioritization. The goal of test suite prioritization is to reorder test cases so that testers detect as many faults as possible as early as possible [3]. Bryce et al. [4] show that prioritizing test cases by the number of **parameter-values** from large to small (PV-LtoS) or by the number of interactions between a pair of **parameter-values** (two-way) generally produces a faster rate of fault detection.

In this paper, we further improve the combinatorial two-way inter-window prioritization technique by introducing weights on a pair of **parameter-values** with respect to the distance between windows. The intuition is that **parameter-values** selected in closer consecutive order by windows have stronger relationships. For instance, during the checkout process, a shopping cart page and a checkout page have stronger connection than a product page and a checkout page since a shopping cart page is usually followed by a checkout page. In order to evaluate the efficiency of the improved method, we use the average percentage of faults detected (APFD) metric [5], which measures the rate of fault detection.

In Section 2, of this paper we discuss previous related work. Section 3 contains the algorithm for the two-way weighted prioritization. In Section 4, we describe the experiment and provide the results. Section 5 provides the conclusion and an outline of future work.

2 Background and Related Work

Test suite prioritization has been studied extensively over the years [3, 6, 7]. Recent studies propose a wide range of prioritization methods varying from prioritization methods that use code coverage information [8] to techniques that use genetic algorithms [9].

Zhang et al. [8] propose an extended model that unifies total and additional prioritization strategies. The total prioritization strategy uses the total number of found criteria to assign the rank of a test case. The additional prioritization strategy uses the number of uncovered criteria for ranking test cases. The authors examine 48 prioritization strategies that

include control total and additional strategies along with 46 hybrid strategies using 19 versions of 4 subject programs. The results show that the large number of hybrid strategies outperform total and additional strategies. The authors, however, rely on the code coverage, whereas our proposed method uses user-session based test suites.

Arafeen and Do [10] use a requirements-based clustering approach for test suite prioritization. The authors apply two control prioritization methods: no prioritization and code coverage along with their clustering approach to two Java applications. The results show that their approach could be beneficial, although efficiency varies depending on the application and parameters set in the proposed method. Our proposed method, on the other hand, uses window distance information instead of the software requirements.

Thomas et al. [11] propose the use of topic models for test suite prioritization. The authors use linguistic data of test cases and the topic modeling algorithm in conjunction with the greedy algorithm that maximizes distance between test cases. They compare the proposed method to other static black-box techniques such as random ordering, the call graph-based technique and string-based technique. The results show that the proposed topic-based technique performs better than other compared techniques. Our proposed method differs in the way that it does not rely on linguistic information of the test cases, instead using distance between windows. In our future work, however, we intend to utilize a string-based approach for pair comparison.

User-session based techniques capture and store a user's interactions with a system [12]. Multiple approaches to represent user-session based test suites exist. For instance, a web application user-session based test case is a sequence of HTTP requests that captured when users access a web application [13]. Such a test case contains base requests along with **parameter-values** at each step of a sequence. The user-session based techniques record such user actions and convert recorded data into test cases for testing purposes. Results of their approach show that a different set of faults was found when user-session based test suites were used compared to other test generation strategies. The authors suggest that user-session based test suites could be used in conjunction with other methods to complement the testing process.

Sampath et al. [14, 15] employ a mathematical approach of concept analysis to user-session based testing of web applications. The authors cluster test cases and create a new reduced test suite by a selective incorporation of test cases from the generated concept lattice. The results of the study show that there is a trade-off between aggressive reduction techniques and the

number of faults found by the resulting test suite. Additionally, the authors found that the use of 2-limited heuristic produces a test suite with a decent balance between the test suite size and the number of faults detected.

Sampath et al. [13] explore the use of different prioritization methods on user-session based test suites. The authors compare the following criteria for test suite prioritization: test length based on number of base requests, frequency-based prioritization, unique coverage of **parameter-value** tuples, two-way **parameter-value** interaction coverage, test length based on the number of **parameter-value** tuples, and random ordering. Their results show that, depending on the application and the testers goal, different prioritization criteria produce better results. In general, however, two-way prioritization should be considered if a tester wants to execute majority of test cases in a test suite.

Sampath and Bryce [16] further explore prioritization and reduction techniques by examining 40 hybrid reduction/prioritization approaches. They also propose a new evaluation metric *Mod_APFD_C* that incorporates a cost of generation and execution times and allows testers to compare test suites of different sizes. The results of the study demonstrate that some of the proposed methods produce better results than pure prioritization or reduction techniques alone. For example, all accessed sequence (AAS) prioritization criterion along with sequence-based (seq) reduction criterion are recommended for use in hybrid test suite prioritization/reduction methods. While the demonstrated techniques use user-session based test suites, they do not utilize distance-based relationships between windows for test suite prioritization.

Bryce and Colbourn [17] introduce biased covering arrays for combinatorial interaction testing. The authors propose assigning user-specified weights to each of the factors. The proposed method could be used for test case generation. Results show that the proposed algorithm provides a straightforward and practical mechanism for generating prioritized test suites. At the same time, the proposed approach is used for test case generation and does not prioritize already existing test suites.

Huang et al. [18] propose the weight-based GUI test-case prioritization method (WGTCP). While their work demonstrates an improvement over traditional methods, it relies on previously obtained correlation between faults and events and requires manual weight assignment, whereas the two-way weighted prioritization relies only on the information in a test suite and assigns weights automatically.

Bryce et al. [19] propose a cost-based two-way prioritization algorithm, where the authors incorporate the length of the test case as a cost factor.

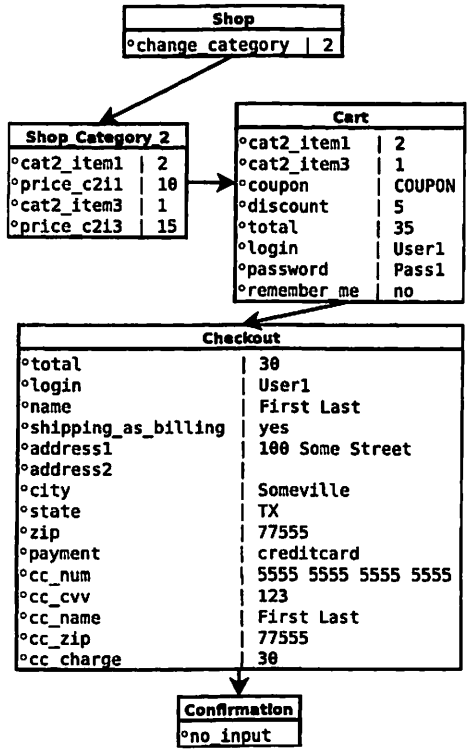
While their results show that the algorithm is generally effective, their cost-based two-way prioritization uses different criteria when compared to the two-way weighted prioritization. Our future work might include a hybrid algorithm that incorporates both techniques.

3 Two-Way Weighted Prioritization

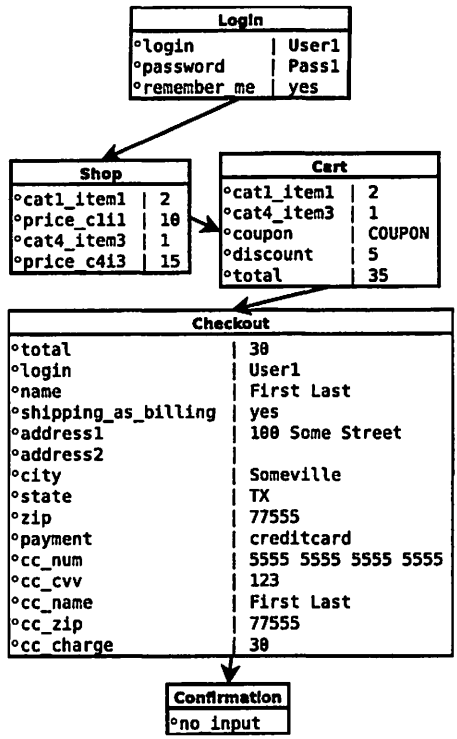
Rothermel et al. [5] define test suite prioritization as following:

> Given a test suite T, a set of all possible permutations of T as P_T, and a function f that maps P_T to a real numbers, find T' in $P_T \mid \forall T''$ in P_T, $T'' \neq T', f(T') \geq f(T'')$

The two-way weighted prioritization method improves upon the standard two-way inter-window prioritization method with the addition of the weighting calculations. The t-way prioritization algorithm, which is a gen-



Test Case 1: A typical scenario where a user does online shopping.



Test Case 2: A user logs in to shop and makes a purchase.

eralized version of two-way prioritization, was explained in great detail by Bryce and Memon [20]. Later in this section, we provide an example of two-way inter-window prioritization.

The two-way inter-window interaction is defined by Bryce et al. [19] as interactions of **parameter-values** on different pages or windows. The two-way inter-window prioritization is the two-way prioritization that considers only inter-window pairs, i.e. only pairs of **parameter-values** that have different windows.

In order to demonstrate benefits of the new prioritization technique, consider Test Cases 1 and 2. Test Case 1 shows an example scenario to

1. Select the second category on the front page of a shop
2. Add 2 items from the category to the cart
3. Use a coupon and his login credentials to proceed to checkout
4. Verify that the order details are correct and enter the credit card

information

5. Receive the confirmation of the order

The process in Test Case 2 differs as a user enters their credentials at the beginning of the process and shops items on the front page.

Given the number of inter-window pairs of **parameter-value** tuples, Test Case 1 contains 239 inter-window pair and Test Case 2 contains 227 pairs. Therefore, the standard two-way inter-window prioritization places Test Case 1 above Test Case 2.

The proposed two-way weighted prioritization provides a flexibility that allows testers to define the importance of pairs using the distance relationship between windows in a pair. Consider a fault when an application resets user credentials at the *Cart* window. In this case, testers prefer to run Test Case 2 earlier than Test Case 1. Therefore, a tester may use a weighting formula that increases importance of a pair as the distance between windows increases. The use of such a formula will place Test Case 2 above Test case 1.

The pseudo-code for the proposed algorithm is shown in Algorithm 1 and Algorithm 2.

Table 1 defines different weighting formulas that we use. The variable *dist* is a distance between windows in a given pair, the variable *max_win* is a number of windows of the largest test case in a test suite, and constants c_1 and c_2 are custom factors that have the following values: $c_1 = 1.33$, $c_2 = 2$. The values of the factors are explained in the following paragraph.

```
Data:
    uSet ← set of unordered test cases
Result:
    orderedSet
orderedSet ← ∅
coveredPairs ← ∅
wFunc ← weighting calculation function
while uSet ≠ ∅ do
    bestTestCase = getLargestScoreTestCase (uSet, wFunc,
    coveredPairs)
    append bestTestCase to orderedSet
    remove bestTestCase from uSet
    append all pairs from bestTestCase to coveredPairs
end
```

Algorithm 1: The two-way weighted algorithm.

```
function getLargestScoreTestCase
    Data:
        uSet, wFunc, coveredPairs
    Result:
        bestTestCase
    bestTestCase ← tuple(first test case in uSet, 0)
    for each test case testCase in uSet do
        score ← 0
        for each pair Pair in testCase and not in coveredPairs do
            // wFunc is a weighting calculation function
            score ← score + wFunc (Pair)
        end
        if score > bestTestCase[1] then
            bestTestCase ← tuple(testCase, score)
        end
    end
```

Algorithm 2: Description of the *getLargestScoreTestCase* function from Algorithm 1.

Description	Formula
Linear	$linear = \frac{dist-1}{max\_win}$
Negative Linear	$neg\_linear = 1 - \frac{dist-1}{max\_win}$
Inverse Distance	$inv\_distance = \frac{1}{dist}$
Negative Inverse Distance	$neg\_inv\_distance = 1 - \frac{1}{dist}$
Negative Sigmoid	$neg\_sigmoid = \frac{1}{1+e^{c_1*(dist-c_2-1)}}$

Table 1: Weighting factors that were used for the function *wFunc* in Algorithm 1 and Algorithm 2.

With the introduction of different weighting formulas, we anticipate that window distance in different applications and test suites could have different significance depending on the characteristics of applications and test suites. *Linear* formula places more significance on more distant windows, while *negative linear* formula weights windows with closer distance as more important. Both formulas follow linear distribution. *Inverse distance* formula is similar to *negative linear*, while *negative inverse distance* is similar to *linear*. The difference is that the formulas follow inverse dis-

Test Case	Windows	Tuples
t_1	$w_1 \rightarrow w_2 \rightarrow w_3$	w_1pv_1, w_1pv_2, w_2pv_1, w_3pv_1, w_3pv_2, w_3pv_3
t_2	$w_1 \rightarrow w_3 \rightarrow w_1 \rightarrow w_2$	w_1pv_1, w_1pv_3, w_3pv_2, w_3pv_4, w_3pv_5, w_1pv_4, w_2pv_1, w_2pv_2

Table 2: An example of user-session based test suite.

tribution. *Negative sigmoid* formula uses a negative sigmoid function with constant factors. The intuition for using the specified factor values is that the importance of pairs that have window distances more than 3 drops significantly.

Table 2, Table 3, and Table 4 show an example of the test suite and the results of the execution of 2 prioritization algorithms: the standard two-way inter-window prioritization and two-way weighted prioritization.

Column 1 in Table 2 contains test case identification labels. Column 2 contains the order of windows as they appear in a test case. Column 3 contains **parameter-value** tuples with the prefix of the window.

#	Test Case	Pairs	# of Pairs	Score
1	t_2	<w_1pv_1,w_3pv_2>, <w_1pv_1,w_3pv_4>, <w_1pv_1,w_3pv_5>, <w_1pv_1,w_2pv_1>, <w_1pv_1,w_2pv_2>, <w_1pv_3,w_3pv_2>, <w_1pv_3,w_3pv_4>, <w_1pv_3,w_3pv_5>, <w_1pv_3,w_2pv_1>, <w_1pv_3,w_2pv_2>, <w_3pv_2,w_1pv_4>, <w_3pv_2,w_2pv_1>, <w_3pv_2,w_2pv_2>, <w_3pv_4,w_1pv_4>, <w_3pv_4,w_2pv_1>, <w_3pv_4,w_2pv_2>, <w_3pv_5,w_1pv_4>, <w_3pv_5,w_2pv_1>, <w_3pv_5,w_2pv_2>, <w_1pv_4,w_2pv_1>, <w_1pv_4,w_2pv_2>	21	21
2	t_1	~~<w_1pv_1,w_2pv_1>~~, <w_1pv_1,w_3pv_1>, ~~<w_1pv_1,w_3pv_2>~~, <w_1pv_1,w_3pv_3>, <w_1pv_2,w_2pv_1>, <w_1pv_2,w_3pv_1>, <w_1pv_2,w_3pv_2>, <w_1pv_2,w_3pv_3>, <w_2pv_1,w_3pv_1>, ~~<w_2pv_1,w_3pv_2>~~, <w_2pv_1,w_3pv_3>	11	8

Table 3: Demonstration of two-way inter-window prioritization for the test suite from Table 2.

Column 1 in Table 3 contains ranking numbers of test cases in the

#	Test Case	Pairs	# of Pairs	Score
1	t_2	$\langle w_1pv_1,w_3pv_2\rangle$, $\langle w_1pv_1,w_3pv_4\rangle$, $\langle w_1pv_1,w_3pv_5\rangle$, $\underline{\langle w_1pv_1,w_1pv_4\rangle}$, $\langle w_1pv_1,w_2pv_1\rangle$, $\langle w_1pv_1,w_2pv_2\rangle$, $\langle w_1pv_3,w_3pv_2\rangle$, $\langle w_1pv_3,w_3pv_4\rangle$, $\langle w_1pv_3,w_3pv_5\rangle$, $\underline{\langle w_1pv_3,w_1pv_4\rangle}$, $\langle w_1pv_3,w_2pv_1\rangle$, $\langle w_1pv_3,w_2pv_2\rangle$, $\langle w_3pv_2,w_1pv_4\rangle$, $\langle w_3pv_2,w_2pv_1\rangle$, $\langle w_3pv_2,w_2pv_2\rangle$, $\langle w_3pv_4,w_1pv_4\rangle$, $\langle w_3pv_4,w_2pv_1\rangle$, $\langle w_3pv_4,w_2pv_2\rangle$, $\langle w_3pv_5,w_1pv_4\rangle$, $\langle w_3pv_5,w_2pv_1\rangle$, $\langle w_3pv_5,w_2pv_2\rangle$, $\langle w_1pv_4,w_2pv_1\rangle$, $\langle w_1pv_4,w_2pv_2\rangle$	23	19
2	t_1	~~$\langle w_1pv_1,w_2pv_1\rangle$~~, $\langle w_1pv_1,w_3pv_1\rangle$, ~~$\langle w_1pv_1,w_3pv_2\rangle$~~, $\langle w_1pv_1,w_3pv_3\rangle$, $\langle w_1pv_2,w_2pv_1\rangle$, $\langle w_1pv_2,w_3pv_1\rangle$, $\langle w_1pv_2,w_3pv_2\rangle$, $\langle w_1pv_2,w_3pv_3\rangle$, $\langle w_2pv_1,w_3pv_1\rangle$, $\langle w_2pv_1,w_3pv_2\rangle$, $\langle w_2pv_1,w_3pv_3\rangle$	11	7.33

Table 4: Demonstration of two-way weighted prioritization using the negative linear weighting formula for the test suite from Table 2.

prioritized test suite. Column 2 contains test case identification labels. Column 3 contains the pairs generated from the tuples provided in Table 2. Overstriked tuples are already covered in the previous test cases. The standard two-way prioritization technique does not consider the order of windows. Column 4 contains the total number of pairs in a test case. Column 5 contains the score of a test case that is computed by counting the number of unique inter-window pairs in a test case.

Column 1 in Table 4 contains ranking numbers of test cases in the prioritized test suite. Column 2 contains test case identification labels. Column 3 contains the pairs generated from the tuples provided in Table 2. Overstriked tuples are already covered in the previous test cases. The two-way weighted prioritization takes into consideration not only identification labels of the windows, but also the order of windows. Column 4 contains the total number of pairs in a test case. Column 5 contains the score of a test case that is computed by the negative linear formula shown in Algorithm 3.

$$
\begin{aligned}
linear &= \frac{2-1-1}{4} = 0 \\
neg\_linear &= 1 - \frac{2-1-1}{4} = 1 \\
inv\_distance &= \frac{1}{2-1} = 1 \\
neg\_inv\_distance &= 1 - \frac{1}{2-1} = 0 \\
neg\_sigmoid &= \frac{1}{1+e^{1.33*(2-1-1-2)}} = 0.9346
\end{aligned}
\tag{1}
$$

Equation 1 shows an example of weight calculations for the pair $<w_1pv_1, w_3pv_2>$ that appears in the test case t_2 in Table 4. The order of window w_1 and window w_3 is 1 and 2 correspondingly.

4 Empirical Study

In this paper we examine the following research questions:

RQ1. How does the use of the proposed two-way weighted prioritization method affect the fault detection process?

RQ2. Which weighting formula used in the new method generally produces the fastest rate of fault detection?

RQ3. How do characteristics of applications and test suites influence results produced by the new prioritization method?

4.1 Experimental Setup

In our experimental setup, we use four application test suites: three graphical user interface (GUI) applications and one web application. The GUI applications are a part of TERP Office suite that was developed at the University of Maryland. While there are four applications in the suite, TERP Calc has only two windows, which makes it unsuitable for our experiments. Therefore, only the following applications were used:

1. TERP Office Word
2. TERP Office Spreadsheet

Description	Word	Ssheet	Paint	OJS
Lines of Code	4,893	12,791	18,376	364,290
Number of Classes	104	125	219	1,557
Number of Methods	236	579	644	13,905
Number of test cases	105	268	274	109
Number of faults in a fault matrix	58	34	118	29
Number of test cases with at least 1 fault	87	40	68	106
Maximum number of non-unique windows per test case	13	9	11	74
Average number of non-unique windows per test case	4.03	2.54	2.59	16.75
Maximum number of faults found by a test case	9	7	17	5
Average number of faults found by a test case	3.65	0.24	1.18	2.27

Table 5: Test suite parameters of applications.

3. TERP Office Paint

TERP office development was done in Java. The source files for TERP Office are available at `https://www.cs.umd.edu/users/atif/TerpOffice/`. The detailed description of the GUI applications and their test suites is available in [4].

In addition, we used the following web application test suite:

4. Open Journal Systems (OJS) [21]

Open Journal Systems was developed by the Public Knowledge Project. The test suite for OJS was created by collecting user-session based test cases. That is, we recorded actual user visits in a web log and then translated them to test cases for our tool to replay. The software metrics for Open Journal Systems were retrieved using PHP Depend tool [22]. OJS also contains third-party frameworks such as CodeIgniter [23], Smarty [24], Zend Search Lucene [25], and SimplePie [26], which creates additional challenges as such frameworks increased overall code complexity, compatibility issues, or additional faults. Table 5 demonstrates characteristics of test suites for the subject applications.

4.2 Evaluation Metrics

To evaluate the proposed approach, we have used a widely known fault average percentage faults detected (APFD) metric [5]. The APFD metric is defined as:

$$APFD = 1 - \frac{TF_1 + TF_2 + ...TF_m}{mn} + \frac{1}{2n} \quad (2)$$

In Equation 2, n is a number of test cases in test suite TS that is ordered by some criteria, m is a number of faults found by test suite TS, TF_i is the first test case in TS that finds fault i. APFD metric can be viewed as the rate of fault detection, i.e. the area under the curve on the axis where x is the number of test cases executed and y is the total number of faults found.

4.3 Results

To demonstrate the efficiency of the proposed method, we compared it to random ordering, as well as the traditional two-way inter-window prioritization. The random ordering was performed 50 times, and the average of results was used for comparison. For two-way weighted prioritization, negative linear weighting formula was used for all applications as comparison of different weighting methods revealed that negative linear performs well on all subject applications compared to other methods that performed better in specific instances, but worse in general. The comparison of different weighting methods is also included in the paper in Subsection 4.3.3.

4.3.1 GUI Applications

The results for TERP Office test suites show that two-way weighted prioritization using negative linear weighting performs better than two-way inter-window prioritization for 2 out of 3 test suites on 100% of test cases. At the same time, for the first 5 and 20 test cases two-way weighted prioritization performs equally well or better for all test suites compared to two-way inter-window prioritization.

Random ordering was a baseline for comparison and showed noticeably worse results for all test suites, with an exception of the first 5 test cases in the TERP Word test suite. The reason for such performance could be a high density of bugs per test case: 87 out of all 105 test cases found at least 1 fault with 58 faults total. In fact, the results of random ordering for TERP Word test suite are very close to both two-way prioritization methods with

the exception of the final score. Similar performance of random ordering was found by Sampath et al. [13]. Figure 3 shows the variability of APFD scores for each subject application when random ordering is used. Large variance of results for random ordering indicates that mean values are not indicative of one instance of random ordering. Test suite execution time is often the longest process and executing test suites more than once could be unfeasible. As a result, it is reasonable to conclude that mean values of random ordering do not provide a complete picture for a tester to make a decision regarding the choice of prioritization method.

4.3.2 Web Applications

The results of the Open Journal System test suite shows mixed results with two-way weighted prioritization, finding fewer bugs in the first 5 test cases and having a larger APFD score overall. Random ordering shows close results for the first 20 test cases and on 50% of the test suite, which could be explained by the fact that the test suite contains a subset of good test cases and a larger subset of test cases with an average quality.

Description	Random	2-way	2-way weighted
TERP Office Paint			
First 5 test cases (# of bugs)	4.54	13	26
First 20 test cases (# of bugs)	20.79	54	57
50% of test cases (# of bugs)	81.86	110	108
100% of test cases (APFD)	0.6407	0.8546	0.8471
TERP Office Word			
First 5 test cases (# of bugs)	14.46	14	19
First 20 test cases (# of bugs)	34.38	38	38
50% of test cases (# of bugs)	47.64	50	49
100% of test cases (APFD)	0.7953	0.8310	0.8468
TERP Office Spreadsheet			
First 5 test cases (# of bugs)	1.26	4	4
First 20 test cases (# of bugs)	3.8	5	13
50% of test cases (# of bugs)	21.26	29	31
100% of test cases (APFD)	0.6123	0.7664	0.8028

Table 6: Results of different prioritization methods for TERP Office applications.

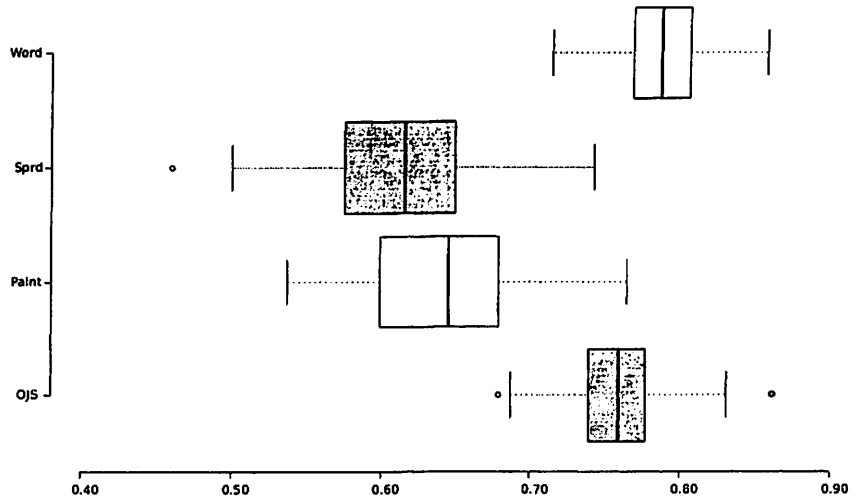


Figure 3: The box plot shows the APFD distribution for random ordering using 50 permutations as samples.

Description	Random	2-way	2-way weighted
First 5 test cases (# of bugs)	7.46	12	9
First 20 test cases (# of bugs)	16.92	17	19
50% of test cases (# of bugs)	24.06	25	25
100% of test cases (APFD)	0.7622	0.7964	0.8132

Table 7: Results of different prioritization methods for OJS application.

The results demonstrated in Tables 6 and 7 allow us to answer RQ1: the proposed two-way weighted prioritization method generally produces better results compared to random ordering or two-way inter-window prioritization. The results also confirm our initial assumption that window distance has an impact on fault detection rate.

4.3.3 Distance Weight Calculations

In order to answer RQ2, we examined different weight calculation approaches compared for test suites of the subject applications. We also introduced a constant weight ($wFunc = 1$ at all times) in order to examine how the order of windows affects effectiveness of the fault detection compared to the use of window labels in the original implementation of two-way inter-window prioritization.

Description	Cons.	Lin.	Neg. Lin.	Inv. Dist.	Neg. Inv. dist.	Neg. Sigm.
TERP Office Paint						
First 5 test cases (number of bugs)	19	20	26	26	30	26
First 20 test cases (number of bugs)	63	51	57	48	51	59
50% of test cases (number of bugs)	108	104	108	108	104	108
100% of test cases (APFD)	0.8484	0.8186	0.8471	0.8459	0.8216	0.8463
TERP Office Word						
First 5 test cases (number of bugs)	13	18	19	22	24	22
First 20 test cases (number of bugs)	37	39	38	38	38	39
50% of test cases (number of bugs)	50	49	49	49	49	49
100% of test cases (APFD)	0.8413	0.8368	0.8468	0.8464	0.8406	0.8466
TERP Office Spreadsheet						
First 5 test cases (number of bugs)	3	2	4	3	5	3
First 20 test cases (number of bugs)	12	12	13	15	12	15
50% of test cases (number of bugs)	31	28	31	31	28	31
100% of test cases (APFD)	0.8010	0.7687	0.8028	0.8045	0.7713	0.8049

Table 8: Results for different distance weighting formulas for TERP Office applications.

Tables 8 and 9 allow us to answer RQ2: the proposed weight calculation approaches have a different effect on different applications. While two-way weighted prioritization using negative linear formula generally outperforms other weighting approaches, a specific application might benefit from a different weighting formula. The results using constant weight demonstrate that the use of the order of the windows instead of window labels has a

Description	Cons.	Lin.	Neg. Lin.	Inv. Dist.	Neg. Inv. dist.	Neg. Sigm.
Open Journal Systems						
First 5 test cases (number of bugs)	9	10	9	7	12	8
First 20 test cases (number of bugs)	19	21	19	19	20	18
50% of test cases (number of bugs)	25	24	25	25	24	24
100% of test cases (APFD)	0.8135	0.8239	0.8132	0.8024	0.8215	0.8037

Table 9: Results for different distance weighting formulas for Open Journal Systems.

positive impact on APFD, thus increasing effectiveness of fault detection.

Negative inverse distance formula demonstrates an interesting property: it outperforms all other weighting formulas in the first 5 test cases. It appears that a subset of faults in each application is caused by interactions between distant windows. At the same time, gains that were produced in the first 5 test cases vanish as more test cases are evaluated. By the 20th test case, the negative inverse distance formula shows results similar to other weighting formulas.

Linear weighting formula demonstrated the least good performance in general. However, it outperformed approaches that use decreasing window importance in the Open Journal Systems test suite. Given the largest average number of windows per test case and the fact that OJS has a large and complex code, one could see how many fault interactions occur in pairs with longer window distances.

4.3.4 Characteristics of Applications

The summary of the results gives us the answer to the RQ3: while two-way weighted prioritization using the negative linear approach generally produces better results, the choice of weighting formula depends on the application characteristics. A different weight calculation approach may be better suited for a specific application. For instance, the negative linear formula did not perform well on the Open Journal Systems test suite. At the same time, Table 9 shows that linear and negative inverse distance formulas perform better than others. Therefore, interactions between pairs

with further distances are more important than interactions between pairs with close distances. Considering that OJS has a significantly larger average number of non-unique windows per test cases as well as a larger number of classes, methods, and lines of code while maintaining the smallest number of faults in the fault matrix among the tested applications as shown in Table 5, it is reasonable to conclude that interactions that cause faults would be further apart between windows, which is confirmed by the results.

4.3.5 Guidance to Testers

Our results demonstrate that the two-way weighting prioritization using negative linear formula generally outperforms other weighting formulas. It also outperforms two-way inter-window prioritization as well as random ordering. Therefore, it is a good choice when a tester does not have a specific knowledge of the subject application and plans to execute all or a majority of test cases.

When testers possess information regarding the characteristics of the subject application, especially fault matrices for previous versions of the application, which could be the case for regression testing, a tester could explore multiple weighting formulas in order to determine which formula works better for their application. As developers tend to make similar mistakes if they do not adhere to disciplined personal practices such as Personal Software Process (PSP) [27], the chosen weighting formula should have a better fault detection rate for the subsequent versions of the application.

For short preliminary testing, the negative inverse distance weighting formula is a good choice as it consistently produced the best results for the first 5 test cases in our study.

Overall, the choice of weighting formula depends on the prior knowledge that testers possess regarding applications as well as their goals. For instance, testers should examine their use-cases and code to identify whether interactions between windows may trigger faults or if the events on different windows are highly independent.

4.4 Threats to Validity

There are multiple factors that may reduce the applicability of the results to other applications and test suites. One of the factors is the structure of the test suites. While some test suites may have a uniform distribution of the number of windows per test case, others may have a different distribution, which may impact scores for test cases with fewer windows versus test cases

with a larger number of windows. In order to minimize this threat, we used several different applications. Another factor is the distribution of faults in the test suites. Given the fact that faults in the tested applications were seeded manually, test suites with actual faults may demonstrate different results. The small number of test suites in the study is also a factor. Having only four subject applications can potentially produce bias by not covering different styles of software development as well as different types of applications. We minimize this threat by using GUI and Web applications.

5 Conclusions and Future Work

Overall, our experiments show that the proposed two-way weighted prioritization generally performs better than two-way inter-window prioritization or random ordering of test cases. The use of a specific distance weighting formula is application specific, while the negative linear calculation approach could be used in a general case as it consistently demonstrates better results compared to two-way inter-window prioritization.

In future work, we intend to examine the proposed approach on mobile applications. Mobile applications have several distinct characteristics that make them different from traditional GUI or Web applications. They are often simpler, smaller, and mostly rely on a touchscreen as an input source. On the other hand, mobile applications are arguably challenging to test due to context changes that may happen such as screen orientation changes, network connection changes, and other context data. As the number of mobile devices rapidly increases, software testing in the mobile domain becomes more critical. We intend to adapt user-session based test suite data for mobile applications and use different prioritization methods and weighting formulas in the case of two-way weighting prioritization to determine the method that has the largest fault detection rate, code coverage, and element coverage.

Moreover, we will develop higher order n-way distance-based combinatorial prioritization algorithms. There are multiple ways to calculate the distance for n-way method, such as a distance between two farthest windows, an average distance, and more. Such a method will require optimization for space and processing time as these requirements become crucial in the case of large test suites. The algorithm proposed in this paper will also benefit from optimization for very large test suites.

We plan to examine other distance-based weighting formulas and their impact on fault detection rate. We also intend to investigate alternative weighting approaches and hybrid methods, such as similarity-based ap-

proaches and cost-based techniques. In addition, we will examine clustering, genetic, and other search-based and machine learning approaches and their applicability to user-session based test suite prioritization.

References

[1] International Telegraph Union, "Statistics Confirm ICT Revolution of the past 15 Years." [Online]. Available: http://www.itu.int/net/pressoffice/press_releases/2015/17.aspx [Accessed: Jan. 15, 2016].

[2] Facebook, "Company Info | Facebook Newsroom." [Online]. Available: https://newsroom.fb.com/company-info/ [Accessed: Jan. 15, 2016].

[3] S. Elbaum, A. G. Malishevsky, and G. Rothermel, "Test case prioritization: A family of empirical studies," *Software Engineering, IEEE Transactions on*, vol. 28, no. 2, pp. 159–182, 2002.

[4] R. C. Bryce, S. Sampath, and A. M. Memon, "Developing a single model and test prioritization strategies for event-driven software," *Software Engineering, IEEE Transactions on*, vol. 37, no. 1, pp. 48–64, 2011.

[5] G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold, "Prioritizing test cases for regression testing," *Software Engineering, IEEE Transactions on*, vol. 27, no. 10, pp. 929–948, 2001.

[6] ——, "Test case prioritization: An empirical study," in *Software Maintenance, 1999.(ICSM'99) Proceedings. IEEE International Conference on.* IEEE, 1999, pp. 179–188.

[7] S. Yoo and M. Harman, "Regression testing minimization, selection and prioritization: a survey," *Software Testing, Verification and Reliability*, vol. 22, no. 2, pp. 67–120, 2012.

[8] L. Zhang, D. Hao, L. Zhang, G. Rothermel, and H. Mei, "Bridging the gap between the total and additional test-case prioritization strategies," in *Software Engineering (ICSE), 2013 35th International Conference on.* IEEE, 2013, pp. 192–201.

[9] W. Jun, Z. Yan, and J. Chen, "Test case prioritization technique based on genetic algorithm," in *Internet Computing & Information Services (ICICIS), 2011 International Conference on.* IEEE, 2011, pp. 173–175.

[10] M. J. Arafeen and H. Do, "Test case prioritization using requirements-based clustering," in *Software Testing, Verification and Validation (ICST), 2013 IEEE Sixth International Conference on.* IEEE, 2013, pp. 312–321.

[11] S. W. Thomas, H. Hemmati, A. E. Hassan, and D. Blostein, "Static test case prioritization using topic models," *Empirical Software Engineering*, vol. 19, no. 1, pp. 182–212, 2014.

[12] S. Elbaum, S. Karre, and G. Rothermel, "Improving web application testing with user session data," in *Proceedings of the 25th International Conference on Software Engineering.* IEEE Computer Society, 2003, pp. 49–59.

[13] S. Sampath, R. C. Bryce, G. Viswanath, V. Kandimalla *et al.*, "Prioritizing user-session-based test cases for web applications testing," in *Software Testing, Verification, and Validation, 2008 1st International Conference on.* IEEE, 2008, pp. 141–150.

[14] S. Sampath, V. Mihaylov, A. Souter, and L. Pollock, "A scalable approach to user-session based testing of web applications through concept analysis," in *Automated Software Engineering, 2004. Proceedings. 19th International Conference on.* IEEE, 2004, pp. 132–141.

[15] S. Sampath, S. Sprenkle, E. Gibson, L. Pollock, and A. S. Greenwald, "Applying concept analysis to user-session-based testing of web applications," *Software Engineering, IEEE Transactions on*, vol. 33, no. 10, pp. 643–658, 2007.

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

[17] R. C. Bryce and C. J. Colbourn, "Test prioritization for pairwise interaction coverage," in *ACM SIGSOFT Software Engineering Notes*, vol. 30, no. 4. ACM, 2005, pp. 1–7.

[18] C.-Y. Huang, J.-R. Chang, and Y.-H. Chang, "Design and analysis of gui test-case prioritization using weight-based methods," *Journal of Systems and Software*, vol. 83, no. 4, pp. 646–659, 2010.

[19] R. C. Bryce, S. Sampath, J. B. Pedersen, and S. Manchester, "Test suite prioritization by cost-based combinatorial interaction coverage," *International Journal of System Assurance Engineering and Management*, vol. 2, no. 2, pp. 126–134, 2011.

[20] R. C. Bryce and A. M. Memon, "Test suite prioritization by interaction coverage," in *Workshop on Domain specific approaches to software test automation: in conjunction with the 6th ESEC/FSE joint meeting.* ACM, 2007, pp. 1–7.

[21] S. P. Muir, M. Leggott, and J. Willinsky, "Open journal systems: An example of open source software for journal management and publishing," *Library hi tech*, vol. 23, no. 4, pp. 504–519, 2005.

[22] M. Pichler, "PHP Depend - software metrics for PHP." [Online]. Available: http://pdepend.org/ [Accessed: Jan. 15, 2016].

[23] A. Andreev, B. Edmunds, J. Parry, and L. Ezell, "Codeigniter web framework." [Online]. Available: https://codeigniter.com/ [Accessed: Jan. 15, 2016].

[24] New Digital Group, Inc, "Smarty - PHP template engine." [Online]. Available: www.smarty.net [Accessed: Jan. 15, 2016].

[25] Zend Technologies, "Zend Search Lucene." [Online]. Available: http://framework.zend.com/manual/1.12/en/zend.search.lucene.overview.html [Accessed: Jan. 15, 2016].

[26] R. Parman, G. Sneddon, and R. McCue, "Simplepie - RSS and Atom feed parsing in PHP." [Online]. Available: http://simplepie.org/ [Accessed: Jan. 15, 2016].

[27] B. Boehm and V. R. Basili, *Foundations of empirical software engineering: the legacy of Victor R. Basili.* Springer Science & Business Media, 2005.