Int J Syst Assur Eng Manag (Apr-June 2011) 2(2):126–134
DOI 10.1007/s13198-011-0067-4

ORIGINAL ARTICLE

Test suite prioritization by cost-based combinatorial interaction coverage

Renée C. Bryce · Sreedevi Sampath ·
Jan B. Pedersen · Schuyler Manchester

Received: 8 December 2010 / Revised: 22 August 2011 / Published online: 3 September 2011

Abstract Test suite prioritization techniques modify the order in which tests within a test suite run. The goal is to order tests such that they detect faults as early as possible in the test execution cycle. Prioritization by combinatorial interaction coverage is a recent criterion that has been useful for prioritizing test suites for GUI and web applications. While studies show that this prioritization criterion can be valuable, previous studies compute the interaction coverage without considering the cost of individual tests. This paper proposes a new *cost-based combinatorial interaction coverage metric*, an algorithm to compute the new metric, and an empirical study with three subject web applications. Two of our studies show that prioritization by the new metric improves the rate at which faults are detected in relation to cost. A third study reveals an interesting result that the success of the cost-based metric is influenced by the distribution of *t*-tuples in the selected test cases.

Keywords Software testing · Test suite prioritization · Combinatorial testing

R. C. Bryce (✉) · S. Manchester
Computer Science, Utah State University, Logan, UT, USA
e-mail: Renee.Bryce@usu.edu

S. Manchester
e-mail: Schuyler.Machester@aggiemail.usu.edu

S. Sampath
Information Systems, University of Maryland, Baltimore County, MD, USA
e-mail: sampath@umbc.edu

J. B. Pedersen
Computer Science, University of Nevada, Las Vegas, NV, USA
e-mail: matt@cs.unlv.edu

1 Introduction

Test suite prioritization is a regression testing technique where test cases are ordered such that faults can be detected early in the test execution cycle. This is useful because tests accumulate over multiple revisions and versions of the system and it is not feasible to execute all the tests in a limited amount of time.

Formally, the test prioritization problem can be defined as follows: Given a test suite T, Π is the set of all possible test suites obtained by permuting the ordering of tests. Each permutation is referred to as $\pi_i \in \Pi$ and an individual permutation contains n tests, $\pi_i = (\pi_{i1}, \pi_{i2}, \ldots, \pi_{in})$. A function then uses some criteria to compute the importance of each test.

Many prioritization criteria exist to improve the rate at which faults are detected. Previous examples of functions for the prioritization problem use code coverage, requirements, MC/DC coverage, cost estimates, relevant slices, (e.g., Srivastava and Thiagarajan 2002; Bryce and Memon 2007; Do et al. 2004; Elbaum et al. 2002, 2004; Jeffrey and Gupta 2006; Jones and Harrold 2003; Rothermel et al. 2001; Sampath et al. 2008; Srikanth et al. 2005), time-aware test prioritization (Walcott et al. 2006), and explore the use of search-based algorithms for test prioritization (Li et al. 2007). Do et al. (2008) study the effect of time constraints on test prioritization methods, however, they do not study the concept of time constraints on interaction coverage-based prioritization criteria. Srikanth et al. (2009) consider combinatorial interactions and cost-based prioritization for configurable software but their work differs from our work as they focus on configuration switching cost. Their algorithm uses weights that change dynamically. Our work focuses on prioritization of user-session-based test suites and elaborate environment setups are outside of the scope of this work. Surveys conducted by Yoo and Harman (2011) and Engstrom et al. (2010) survey test prioritization and regression test selection methods and identify future research directions, one of which, is the area of prioritization for web and GUI testing.

Our recent work examined a comprehensive set of prioritization criteria for two classes of Event Driven Software (EDS), that of GUI and web applications (Bryce et al. 2011). This work examined ten prioritization criteria that exploited features of test cases for seven different applications and test suites and compared the results against three controls. The results revealed that the characteristics of the application and test suites influenced the result, but that combinatorial interaction coverage was one of the best criteria in terms of the improved rate of fault detection. The length-based prioritization criterion that selects the "longest" test cases also worked well for several of the subject applications. (See Bryce et al. 2011 for a detailed summary of the results and characteristics of the applications and test suites.)

Given the short time availability during regression testing, a prioritization criterion that is effective at selecting tests that detect faults quickly while at the same time, selects tests that execute quickly would be highly beneficial. With this goal in mind, in this paper, we extend upon previous work by modifying the combinatorial interaction coverage metric to incorporate costs of test cases. Numerous testing costs exist. For instance, Do and Rothermel (2008) use several good examples of testing cost, including the time to setup for testing activities, time to identify and repair obsolete tests, time to instrument units, time to execute a technique, time to apply tools to check outputs, human time to inspect results, and more. Testers have limited testing budgets and our previous work on prioritization improved the rate of fault detection, but did not consider the cost of tests. Therefore, the main contributions of this paper are: *a new metric for cost-based combinatorial coverage, an algorithm that prioritizes test suites by interaction coverage with respect to the cost of tests, and empirical studies that evaluate the criteria.*

The remainder of this paper is organized as follows. Section 2 describes our prioritization criteria and algorithm. Section 3 provides empirical studies that use three web applications to evaluate our new prioritization criteria. Section 4 includes threats to the validity and Sect. 5 concludes.

2 Prioritization criteria

Our experiments prioritize test suites by 2way interaction coverage and cost-based 2way interaction coverage. Referring back to the prioritization problem, an instantiation of the function for prioritization is as follows. The function tCov($\pi_{i,k}$) computes the set of covered t-tuples in a test $\pi_{i,k}$. Then our prioritization function is:

$$f(\pi_i) = \sum_{j=0}^{n} |\bigcup_{k=0}^{j} \text{tCov}(\pi_{i,k})|.$$

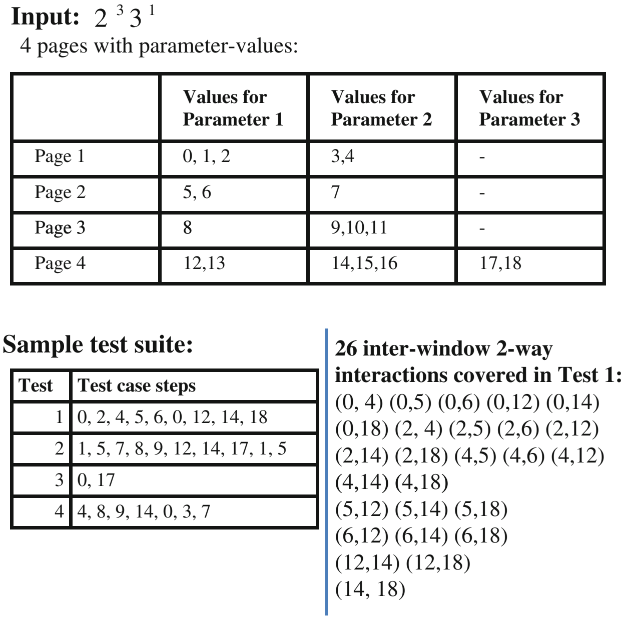
The tCov($\pi_{i,k}$) function that computes the set of covered t-tuples in a test $\pi_{i,k}$. Our experiments compute the previously defined interaction coverage based on covered pairs (Bryce et al. 2011) and the new cost-based interaction coverage which counts the pairs in a test case and divides by the cost of the test case.

2.1 Interaction coverage (2way)

The criterion by *interaction coverage* selects each "*next test*" that covers the largest number of previously uncovered t-way interactions of parameter-values between windows. In this implementation, we break ties at random. For the experiments, we implement for $t = 2$ interaction coverage.

– 2way interaction coverage (2way)-select each "*next test*" that covers the largest number of previously uncovered 2way interactions of parameter-values between windows (i.e., inter-window interaction coverage)

To illustrate the 2way criterion, consider the example shown in Fig. 1 and Table 1. Figure 1 shows the number of pages and parameter-values for a sample web application.


Input: $2^3 3^1$

4 pages with parameter-values:

	Values for Parameter 1	Values for Parameter 2	Values for Parameter 3
Page 1	0, 1, 2	3,4	-
Page 2	5, 6	7	-
Page 3	8	9,10,11	-
Page 4	12,13	14,15,16	17,18

Sample test suite:

Test	Test case steps
1	0, 2, 4, 5, 6, 0, 12, 14, 18
2	1, 5, 7, 8, 9, 12, 14, 17, 1, 5
3	0, 17
4	4, 8, 9, 14, 0, 3, 7

26 inter-window 2-way interactions covered in Test 1:
(0, 4) (0,5) (0,6) (0,12) (0,14)
(0,18) (2, 4) (2,5) (2,6) (2,12)
(2,14) (2,18) (4,5) (4,6) (4,12)
(4,14) (4,18)
(5,12) (5,14) (5,18)
(6,12) (6,14) (6,18)
(12,14) (12,18)
(14, 18)

Fig. 1 Example of interaction coverage for four pages that have parameters and values with a small test suite

There are four web pages in this web application where one page has 5 parameter-values (p-vs), one page has 3 p-vs, one page has 4 p-vs, and one page has 7 p-vs. This is represented as $5^1 3^1 4^1 7^1$. An example of a *page* is a Login screen for an on-line store. An example of a *parameter* is a textfield where a user enters a userid and a *value* is the actual userid that the user enters. In Fig. 1, page one has two parameters where parameter 1 can be set to one of three possible values and parameter 2 has two possible values. Figure 1 also includes a sample test suite that has four tests. The tests are of varying number of steps. Each step includes a parameter-value setting for a web page. For instance, *Test 3* is the shortest test. In the third test, the user specifies the *value 0* for *parameter 1* on *page 1* and then specifies *value 17* for *parameter 3* on *page 4*. Note that not all of the parameters for every page have been assigned values in this test. We assume that values for those parameters are optional. The inter-window interaction coverage considers interactions of parameter-values on different pages, or windows. For instance, the p-v pair of (0,5) is an inter-window interaction because each p-v is on a different page. However, (0,3) is an intra-window interaction and not an inter-window interaction because each p-v is on the same page. The bottom right side of Fig. 1 shows the 26 inter-window 2-way interactions for this example.

The example shows how to count interactions in tests. Prioritization by 2way interaction coverage selects earliest tests to maximize interaction coverage on a test-by-test basis. For the example in Fig. 1, the second column of Table 1 shows the number of 2way interactions in each test. The first test selected by 2way in this example would be Test 2. Subsequent tests are selected based on how many uncovered 2way interactions exist in the test.

We have shown in previous work that the 2way prioritization criterion creates effective test orders for event-driven systems (Bryce et al. 2011). The parameter-value interactions in a test targeted by the 2way criterion exercise complex parts of the underlying code base because of which the tests detect a large number of faults.

Table 1 Summary of 2way interaction coverage and cost-based interaction coverage counts for the example in Fig. 1

Test	2-Way interaction coverage	No. of steps in test	Cost-based 2-way interaction coverage
1	26	9	2.88
2	28	10	2.8
3	1	2	0.5
4	20	7	2.86

Prioritization technique	First test selected
2-Way interaction coverage	2
Cost-based 2-way interaction coverage	1

2.2 Cost-based interaction coverage (CB-2way)

The 2way criterion proposed above does not consider the cost of a test case when selecting tests. The 2way criterion works solely with the number of uncovered parameter-value interactions that exist in a test. In our experiments, we use the length of the test to represent cost. Since longer test cases will take more time to execute, the length of the test case could affect the overall effectiveness of the test order. Of course, testers could consider other costs beyond the length of the test cases in future work. The underlying idea behind the CB-2way criterion is to give priority to cheaper tests with the same 2-way interaction coverage, such that the fault detection effectiveness of the test order can be improved.

For the example shown in Fig. 1, we accommodate cost into the interaction coverage metric by dividing the benefit of the test (e.g., number of previously uncovered *t*-way interactions in a test case) by the cost of the test case (e.g., we use the number of parameter-values in a test as a surrogate for cost). For instance, Test Case 1 covers 26 inter-window interactions and that the length of this test is 9 (e.g., 9 parameters are set to values). Therefore, the cost-based interaction coverage is $\left(\frac{26}{9}\right) = 2.88$. This count is also shown in Table 1. The first test selected by CB-2way will be Test 1.

– Cost-based 2way interaction coverage (CB-2way)—select each "*next test*" such that it maximizes the number of previously uncovered 2way interactions divided by the number of parameters set to values in the test case. For instance, if a test covers 10 previously uncovered 2way interactions and the test case sets 5 parameter-values, the cost-based interaction coverage is $\left(\frac{10}{5}\right) = 2$.

2.3 Random ordering

Random ordering permutes the ordering of tests uniformly at random. In this implementation, we use the *rand()* function that is available with the C++ math library.

2.4 Algorithm to select tests for 2way and CB-2way

Figure 2 provides the pseudocode for our algorithm that prioritizes by cost-based interaction coverage. For each iteration, we compute the cost-based interaction coverage for all remaining tests by counting the number of

```
$prioritizationType$ is set by the user to interaction coverage or
  cost-based interaction coverage
$testCount$ = number of tests to prioritize
$bestTest$ = select a test that is has the best interaction coverage for the cost
mark $test_{bestTest}$ as used
selectedTestCount = 1
while(selectedTestCount < $testCount$)
    $tBestCount$ = -1
    for j=1 to ($testCount$-$selectedTestCount$)
      if $test_j$ is not used
        compute $tCount$ as the number of newly covered $t$-tuples in $test_j$
        if($prioritizationType$ = interaction coverage)
          $tScore$ = "tCount"
        else if($prioritizationType$ = cost-based interaction coverage)
          $tScore = \frac{tCount}{test_j}$
        if($tScore > tBestScore$)
          $tBestScore = tScore$
          $bestTest$ = j
        else if($tScore == tBestScore$)
            break the tie at random
      end for
      add $test_{bestTest}$ to $T_{p_i}$
      mark $test_{bestTest}$ as used
      $selectedTestCount$++
end while
```

Fig. 2 Pseudocode to prioritize test suites by 2way and CB-2way interaction coverage

uncovered t-tuples in the test case and dividing by the cost of the test case. Once a test is selected, we mark the t-tuples that were covered in the selected test case. The following iterations repeat this process until all tests are selected, breaking ties by choosing the longer test (e.g., the test that sets more parameters to values) and if there is a tie in the length, we then break the tie at random.

3 Experiments

In these experiments, we examine whether (1) our algorithm for cost-based interaction coverage improves the rate at which 2-tuples are covered in relation to the cost of test cases and (2) if cost-based interaction coverage improves the rate at which faults are detected. Since our algorithm uses random tie-breaking, we run our experiments five times and report the average results.

3.1 Research questions

We examine the following research questions:

RQ1. Which algorithm has the fastest rate of t-tuple coverage in relation to test cost?
RQ2. Which technique finds 100% of the faults first?
RQ3. Which technique provides the fastest rate of fault detection in relation to costs?

3.2 Independent and dependent variables

Independent variables in our study are the user-session-based test suites, the seeded faults and the test case prioritization techniques. Dependent variables are rate of fault detection, rate of 2-tuple coverage, and average percent of faults detected - cost ($APFD_C$) (Elbaum et al. 2001). When we determine which prioritization techniques detect faults early in the test execution cycle, we use the $APFD_C$ metric presented by Elbaum et al. (2001).

Rothermel et al. (2001) proposed the *APFD* (average cumulative percent of faults detected) metric, that measures the goodness of a test suite based on the performance goal of rate of fault detection. They consider all test cases to have the same cost (in terms of executing the test case), and all faults to have the same severity. However, traditionally, some test cases are more expensive to execute than others and some faults are more severe than others. Several variations of the *APFD* metric exist, such as the metric $APFD_C$ proposed by Elbaum et al. (2001) and *NAPFD* proposed by Qu et al. (2007). *NAPFD* particularly applies when working with test suites of unequal sizes. Elbaum et. al. raise the practical consideration that not all test cases or faults are of equal cost and propose the $APFD_C$ metric metric that accounts for test case cost and fault severity. Using the notation presented in Elbaum et al. (2001), the $APFD_C$ metric can be defined as follows:

For a test suite, T with n test cases, and costs $t_1, t_2, t_3, \ldots t_n$, if F is a set of m faults detected by T with severities $f_1, f_2, f_3, \ldots f_m$, then let TF_i be the position of the first test case t in T', where T' is an ordering of T, that detects fault i. Then, the $APFD_C$ metric for T' is given as

$$APFD_C = \frac{\sum_{i=1}^{m}(f_i \times (\sum_{j=TF_i}^{n} t_j - \frac{1}{2} t_{TF_i}))}{\sum_{i=1}^{n} t_i \times \sum_{i=1}^{m} f_i} \qquad (1)$$

To graph $APFD_C$, the x-axis denotes the percentage of the total test case costs incurred and the y-axis represents the percentage of the cumulative fault severity that has been detected. The area under the curve plotted in the graph gives the value for $APFD_C$. If the test case costs and all fault severities are equal, the $APFD_C$ reduces to that of APFD. In this work, we consider the number of parameter-values in a test as the cost of the test, and we consider all faults to have equal severity, so f_i in the above formula will be 1 for all the faults.

3.3 Subject applications

We use three web-based applications and their pre-existing test suites, where test suites are the previously recorded user-sessions in experiments by Sampath et al. (2007) and Sprenkle et al. (2005). The subject programs are an open-source, e-commerce bookstore (Book) that allows users to login, browse and shop for books; a Course Project Manager (CPM) that allows instructors and students in a course to create demo slots for projects, sign up for demos, post and view grades; and the web application used for the

Mid-Atlantic Symposium on Programming Languages and Systems (MASPLAS) which allows conference attendees to submit abstracts and papers, register for the conference, and view program schedule and conference details. Table 2 summarizes the characteristics of the subject programs and test suite characteristics. For instance, CPM is our largest application with 9,401 lines of code and MASPLAS is the smallest with 999 lines of code. Our largest test suite is for CPM, with 890 test cases. The test suites for Book and MASPLAS have 125 and 169 test cases respectively.

Figure 3 shows the Login page from our Bookstore example. Table 3 shows the parameter-values on this Login page. For instance, there are four parameters that have values, including the three values that the user entered (i.e., they entered values into the two text fields and clicked the button) and one hidden value (i.e., the FormName = Login). We use these user-specified and hidden parameter-values that are contained within the user-session-based test cases that we use.

3.4 Results

We break our results into two sections: (1) compare the rate at which our algorithm covers the 2way interactions in relation to cost and (2) examine the fault finding effectiveness of our prioritized test suites as measured by the time to locate 100% of the faults and $APFD_C$.

3.4.1 *Comparison of algorithms tuple coverage*

Which algorithm has the fastest rate of t-tuple coverage?

Figures 4, 5 and 6 show the rate of 2way interaction coverage for our user-session-based test cases that are prioritized by (1) interaction coverage and (2) cost-based interaction coverage. The *x*-axis represents cumulative test cost and the *y*-axis is the cumulative count of 2way interactions covered. In all three graphs, it is not surprising that the algorithm that prioritizes by cost-based 2way interaction coverage has the fastest rate of 2way interaction coverage in relation to cost. Covering 2way interactions sooner is good because that indicates that cheaper, more effective tests are being selected early in the test execution cycle.

3.4.2 *Comparison of the fault finding effectiveness of the prioritization techniques*

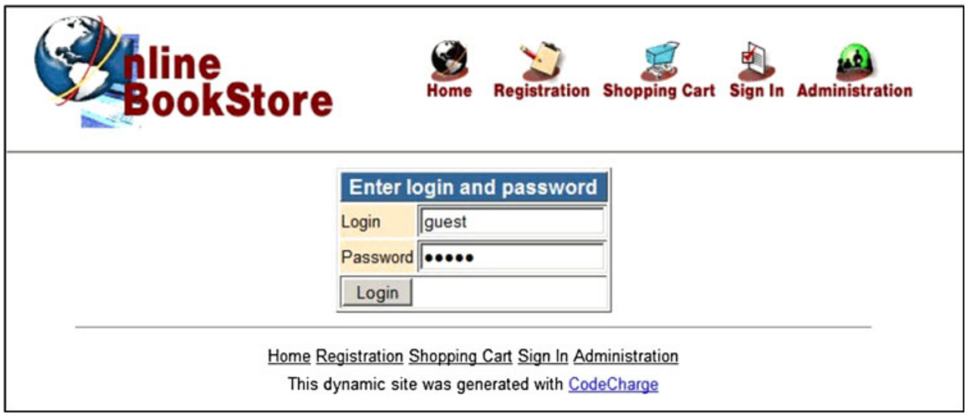
Which technique finds 100% of the faults first?

Table 4 shows the cost to locate 100% of the faults using 2way, cost-based 2way, and random orderings. For the Book, 2way is best, followed by cost-based 2way. For CPM and MASPLAS, cost-based 2way has the smallest cost to find all faults, followed by 2way. Random ordering is the least competitive for all three experiments.

Table 5 shows the number of tests to locate all faults. This data is similar to the previous table that reports the costs to find 100% of faults. We find that the new cost-based 2way technique finds all faults in the smallest number of tests for CPM and MASPLAS. The 2way criterion works best for Book and second best for CPM and MASPLAS. Random ordering is the least competitive for all three applications. In both cases of number of tests and cost to locate 100% faults, we looked at the data and attribute the variation in the results for Book to one particular test case that finds many unique faults and therefore influences the results based on when this one test case is

Table 2 Subject applications and test suite characteristics

	Book	CPM	MASPLAS
Technology	HTML, JSP, MySQL	HTML, Java servlet File-based datastore	HTML, Java servlet, JSP, MySQL
Classes	11	75	9
Methods	319	173	22
Conditions	1720	1260	108
Non-commented lines of code	7615	9401	999
Seeded faults	40	135	29
Total number of user sessions	125	890	169
Total number of requests accessed	3640	12352	1107
Number of unique requests	10	69	24
Largest user session in number of requests	160	585	69
Average user session in number of requests	29	14	7
Number of unique parameter-values	1,415	4,146	645
% of 2way parameter-value interactions covered in pre-existing test suite	92.5	97.8	96.2



Enter login and password

Login	guest
Password	•••••

Home Registration Shopping Cart Sign In Administration

This dynamic site was generated with CodeCharge

Fig. 3 Example: The Login.jsp page for the on-line Bookstore

Table 3 List of parameter-values from the Login.jsp paged

Parameter and value descriptions for Login.jsp
1 < Login text field, guest >
2 < Password text field, guest >
3 < FormAction, Login >
4 < FormName, Login >

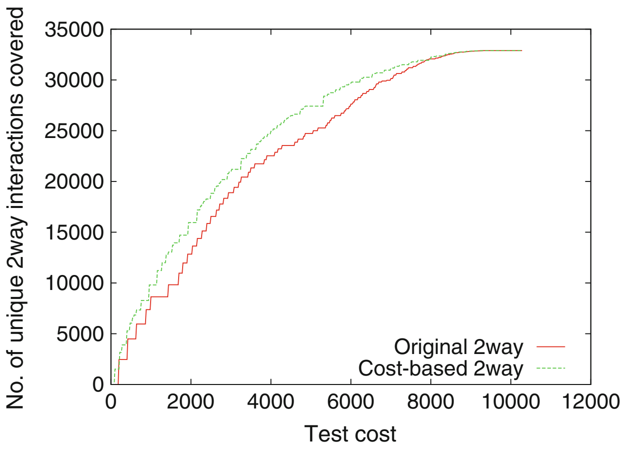


Fig. 4 Book: rate of 2-tuple coverage

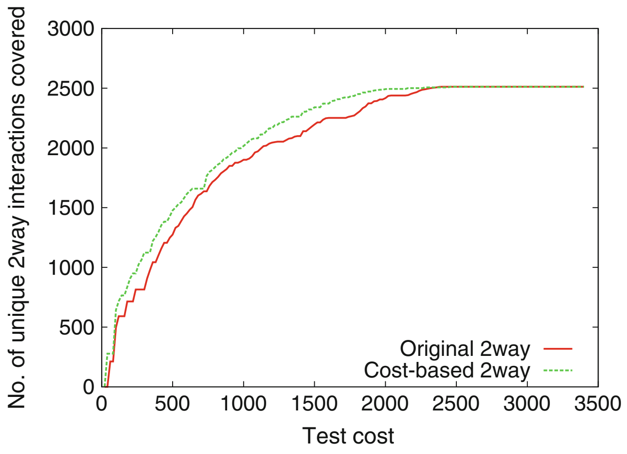


Fig. 5 MASPLAS: rate of 2-tuple coverage

selected. Therefore, the $APFD_C$ data that we present shortly is more informative for Book than the number of tests and cost to find 100% of the faults.

Which technique finds most faults early in the test execution cycle?

Figures 7, 8, and 9 show the rate of fault detection and Table 6 shows the $APFD_C$ for our three experiments. We examine the results for each application next.

Book

Table 6 reveals that cost-based 2way has the best $APFD_C$ for the first 70% of test execution. It is approximately 1% better than 2way throughout the entire test suite execution. The $APFD_C$ of cost-based 2way is considerably better than random ordering during the earliest test execution. For instance, it is 7.15% better after 10% of the test suite is run. However, during the last 30% of test suite execution, random ordering produces a better $APFD_C$.

CPM

Results for CPM in Fig. 8 show that 2way and cost-based 2way are quite comparable for CPM, each outperforming the other at different intervals. Table 6 shows that the two techniques are within 1% of each other in terms of $APFD_C$ with 2way performing slightly better. Finally, our control of random ordering is much less competitive. For instance, in the first 10% of test execution alone, the $APFD_C$ for random is 29.17% lower than that of 2way.

This observation for CPM raises the question of why cost-based 2way does not work as well as 2way. CPM has a larger number of unique parameter values when compared to the other two applications (from Table 2). The distribution of these large number of parameter-values in test cases could result in more complex interactions within the test case, which in turn, could affect the effectiveness of cost-based 2way. In the future, we will conduct further studies to understand the effect of the number and distribution of parameter-values on the cost-based 2way metric.

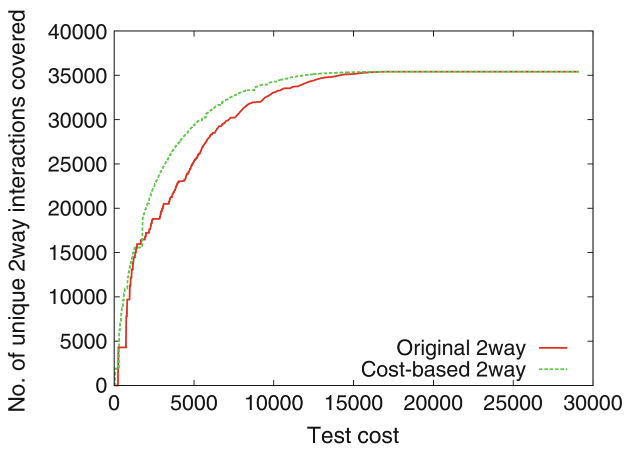


Fig. 6 CPM: rate of 2-tuple coverage

Table 4 Cost to locate 100% of faults

Application	Cost 2way	Cost CB-2way	Cost Random
Book	6,613	8,320	10,471
CPM	21,108	19,839	24,403
MASPLAS	1,882	1,387	2,399

Table 5 Number of tests to locate 100% of faults

Application	No. of tests 2way	No. of tests CB-2way	No. of tests Random
Book	68	76	124
CPM	347	330	815
MASPLAS	59	46	126

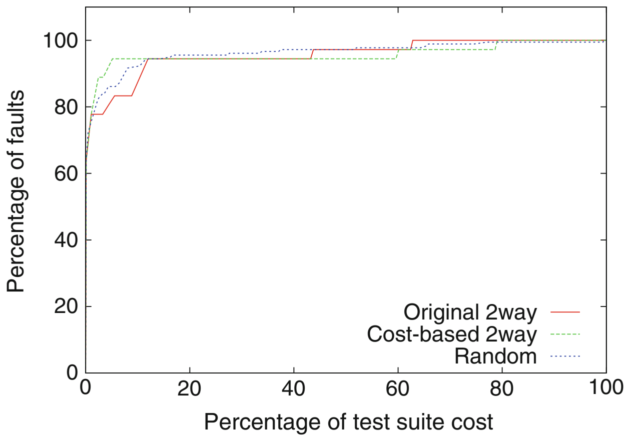


Fig. 7 Book: rate of fault detection

MASPLAS

Finally, Fig. 9 and Table 6 show that cost-based 2way is useful for MASPLAS. The 2way and cost-based 2way orderings are close in $APFD_C$ while random ordering is less effective. The 2way is better early on, but then cost-based 2way has a better $APFD_C$ after approximately 8% of cost is incurred. Table 6 shows that the $APFD_C$ for 2way is 0.24% better than that of cost-based 2way in the first 10% of test execution, but cost-based 2way is better for the latter 90% of the test suite, sometimes achieving as high as 2.8% better $APFD_C$. Similar to our previous experiments, random ordering has the worst $APFD_C$ in this experiment. For instance, in just the first 10% of test execution, the $APFD_C$ for random is 11.82% lower than that of 2way.

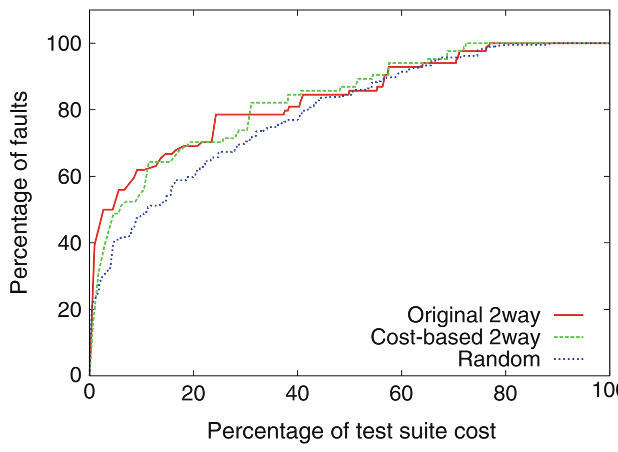


Fig. 8 CPM: rate of fault detection

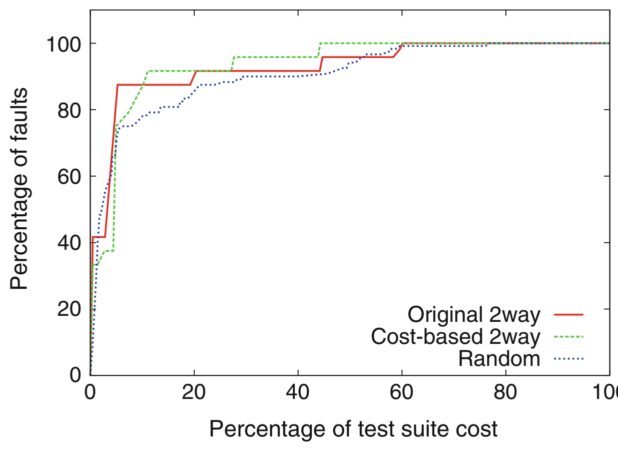


Fig. 9 MASPLAS: rate of fault detection

3.5 Summary of results

Our experiments show that cost-based 2way often improves the $APFD_C$, but also reveals an important implication of using cost-based combinatorial interaction coverage for prioritization. The cost-based 2way interaction coverage metric can achieve more cost-effective coverage of 2-tuples, but one must examine the effect of the distribution of parameter-values that could introduce

Table 6 Book, CPM, and MASPLAS: $APFD_C$ for 2way, cost-based 2way, and random

% of Suite	Book 2way	CB-2way	Rand	CPM 2way	CB-2way	Rand	MASPLAS 2way	CB-2way	Rand
10	92.63	**93.59**	86.44	**77.76**	76.82	48.58	**88.41**	88.17	76.59
20	92.63	**93.59**	93.82	**81.45**	79.84	57.54	88.41	**91.19**	80.93
30	94.19	**93.59**	94.63	**81.88**	81.77	64.59	90.72	**93.52**	86.11
40	94.19	**94.7**	94.98	**83.48**	83.18	69.25	92.41	**93.52**	87.61
50	94.19	**94.7**	94.98	**83.48**	83.18	73.1	92.41	**93.52**	90.53
60	95.23	**95.29**	95.2	**83.48**	83.18	75.47	92.41	**93.52**	91.31
70	95.23	**95.29**	95.2	**83.48**	83.18	77.61	92.41	**93.52**	91.31
80	95.23	95.29	**95.62**	**83.48**	83.18	78.53	92.41	**93.52**	91.51
90	95.23	95.29	**95.62**	**83.48**	83.18	78.76	92.41	**93.52**	91.51
100	95.23	95.29	**95.62**	**83.48**	83.18	78.8	92.41	**93.52**	91.51

The bold font indicates the "best" APFD for each application

complex interactions within the test case. The results for two of our experiments, Book and MASPLAS, generally benefit the most from cost-based 2way in terms of fault detection. The difference in $APFD_C$ between 2way and cost-based 2way for CPM was less than 3%, but nonetheless, we took a closer look to better understand why the cost-based 2way was slightly less effective than 2way. Our analysis revealed that further studies are needed to investigate the effect of number and distribution of parameter-values on the cost-based 2way metric.

4 Threats to validity

The experiments in this paper are based on three web-based applications and pre-existing user-session-based test cases for each application. While we use the number of parameters that are set to values in a test case as a surrogate for the cost of test cases in these experiments, this estimate may vary from the actual time that it takes to execute a test case and for testers to verify the results. Future studies may examine other costs of tests, including scaffolding costs, test execution time, and the time that it takes testers to examine test cases. Second, we use real user-session-based test cases for three applications that are seeded with faults. (See Sampath et al. 2007; Sprenkle et al. 2005) for details on these artifacts that we base on experiments on.) These applications, faults, and test cases may not be representative of all web applications, faults, or test suites. Nonetheless, they did provide a means to evaluate our new cost-based 2way technique.

5 Conclusion

Test suite prioritization reorders test cases in a test suite by a criterion that identifies which tests to run sooner in a test execution cycle. One recent prioritization criteria is that of interaction coverage. While the criteria has been useful in preliminary studies, such studies do not consider the cost of individual tests when they prioritize test cases. We examine a new criteria to prioritize by interaction coverage that also considers the costs of test cases. Empirical results show that a simple algorithm covers more *t*-way interactions faster in relation to cost. Three empirical studies also examine the fault finding effectiveness of the new metric. For our Book and MASPLAS case studies, cost-based 2way is quite effective. Our CPM case study shows that one must be cautious in the presence of a large number of parameter-values and attention must be paid to the distribution of the parameter-values within a test case that could introduce complex interactions.

Acknowledgments This research is supported by NIST grant 70NANB10H048. Reference to commercial products or trademarks does not imply endorsement by NIST, nor that such products are necessarily best suited to any purpose.

References

Bryce RC, Memon AM (2007) Test suite prioritization by interaction coverage. In: Proceedings of the workshop on domain-specific approaches to software test automation (DoSTA 2007), pp 1–7

Bryce RC, Sampath S, Memon A (2011) Developing a single model and test prioritization strategies for event-driven software. IEEE Trans Softw Eng 37(1):48–64

Do H, Rothermel G (2008) Using sensitivity analysis to create simplified economic models for regression testing. In: Proceedings of the international conference on software testing and analysis (ISSTA). ACM Press, New York, pp 51–62

Do H, Rothermel G, Kinneer A (2004) Empirical studies of test case prioritization in a JUnit testing environment. In: Proceedings of 15th international symposium on software reliability engineering (ISSRE 2004). IEEE Computer Society Press, Washington, pp 113–124

Do H, Mirarab S, Tahvildari L, Rothermel G (2008) An empirical study of the effect of time constraints on the cost-benefits of regression testing. In: Proceedings of the SIGSOFT 2008/FSE-16. ACM Press, New York, pp 71–82

Elbaum S, Malishevsky A, Rothermel G (2001) Incorporating varying test costs and fault severities into test case prioritization. In: Proceedings of the international conference on software engineering, IEEE, pp 329–338

Elbaum S, Malishevsky A, Rothermel G (2002) Test case prioritization: a family of empirical studies. IEEE Trans Softw Eng 18(2):159–182

Elbaum S, Rothermel G, Kanduri S, Malishevsky A (2004) Selecting a cost-effective test case prioritization technique. Softw Qual J 12(3):185–210

Engstrm E, Runeson P, Skoglund M (2010) A systematic review on regression test selection techniques. Inf Softw Technol 52(10):14–30

Jeffrey D, Gupta N (2006) Test case prioritization using relevant slices. In: The international computer software and applications conference, September 2006, pp 18–21

Jones JA, Harrold MJ (2003) Test-suite reduction and prioritization for modified condition/decision coverage. Trans Softw Eng 29(3):195–209

Li Z, Harman M, Hierons RM (2007) Search algorithms for regression test case prioritization. In: IEEE transactions on software engineering. IEEE Computer Society Press, Washington, pp 225–237

Qu X, Cohen MB, Woolf KM (2007) Combinatorial interaction regression testing: a study of test case generation and prioritization. In: Proceedings of the IEEE international conference on software maintenance (ICSM). IEEE Computer Society, Washington, pp 255–264

Rothermel G, Untch RH, Chu C, Harrold MJ (2001) Prioritizing test cases for regression testing. IEEE Trans Softw Eng 27(10): 929–948

Sampath S, Bryce R, Viswanath G, Kandimalla V, Koru AG (2008) Prioritizing user-session-based test cases for web application testing. In: The international conference on software testing, verification and validation, April 2008, pp 141–150

Sampath S, Sprenkle S, Gibson E, Pollock L, Greenwald AS (2007) Applying concept analysis to user-session-based testing of web applications. IEEE Trans Softw Eng 33(10):643–658

Sprenkle S, Gibson E, Sampath S, Pollock L (2005) Automated replay and failure detection for web applications. In: Proceedings of the 20th international conference of automated software engineering. ACM Press, New York, pp 253–262

Srikanth H, Williams L, Osborne J (2005) System test case prioritization of new and regression test cases. In: International symposium on empirical software engineering, pp 64–73

Srikanth H, Cohen MB, Qu X (2009) Reducing field failures in system configurable software: cost-based prioritization. In: Proceedings of the international symposium on software reliability engineering (ISSRE), November 2009, pp 61–70

Srivastava A, Thiagarajan J (2002) Effectively prioritizing tests in development environment. In: Proceedings of the international symposium on software testing and analysis (ISSTA 2002). ACM Press, New York, pp 97–106

Walcott KR, Soffa ML, Kapfhammer GM, Roos RS (2006) Time-aware test suite prioritization. In: Proceedings of the international symposium on software testing and analysis, July 2006, pp 1–12

Yoo S, Harman M (2011) Regression testing minimisation, selection and prioritisation: a survey. J Softw Test Verif Reliab. doi: 10.1002/stvr.430