How Does Regression Test Prioritization Perform in Real-World Software Evolution?

Yafeng Lu[1], Yiling Lou[2], Shiyang Cheng[1], Lingming Zhang[1], Dan Hao[2]*, Yangfan Zhou[3], Lu Zhang[2]
[1]Department of Computer Science, University of Texas at Dallas, 75080, USA
{yxl131230,sxc145630,lingming.zhang}@utdallas.edu
[2]Key Laboratory of High Confidence Software Technologies (Peking University), MoE, China
Institute of Software, EECS, Peking University, Beijing, 100871, China
{louyiling,haodan,zhanglucs}@pku.edu.cn
[3]School of Computer Science, Fudan University, 201203, China
zyf@fudan.edu.cn

ABSTRACT

In recent years, researchers have intensively investigated various topics in test prioritization, which aims to re-order tests to increase the rate of fault detection during regression testing. While the main research focus in test prioritization is on proposing novel prioritization techniques and evaluating on more and larger subject systems, little effort has been put on investigating the threats to validity in existing work on test prioritization. One main threat to validity is that existing work mainly evaluates prioritization techniques based on simple artificial changes on the source code and tests. For example, the changes in the source code usually include only seeded program faults, whereas the test suite is usually not augmented at all. On the contrary, in real-world software development, software systems usually undergo various changes on the source code and test suite augmentation. Therefore, it is not clear whether the conclusions drawn by existing work in test prioritization from the artificial changes are still valid for real-world software evolution. In this paper, we present the first empirical study to investigate this important threat to validity in test prioritization. We reimplemented 24 variant techniques of both the traditional and time-aware test prioritization, and investigated the impacts of software evolution on those techniques based on the version history of 8 real-world Java programs from GitHub. The results show that for both traditional and time-aware test prioritization, test suite augmentation significantly hampers their effectiveness, whereas source code changes alone do not influence their effectiveness much.

1. INTRODUCTION

During software development and maintenance, a software system continuously evolves due to various reasons, e.g., adding new features, fixing bugs, or improving maintainability and efficiency. As the change on the software may incur bugs, it is necessary to apply regression testing to revalidate the modified system. However, it is costly to perform regression testing. As reported, regression testing consumes 80% of the overall testing budgets [1, 2], and some test suite consumes more than seven weeks [3]. Therefore, it is important to set a balance between the tests that ideally should be run in regression testing and the tests that are affordable to be used in regression testing, which is the intuition of test prioritization [4].

Following this intuition, test prioritization [3–5] is proposed to improve the efficiency of regression testing, especially when the testing resources (e.g., time, developers' efforts) are limited. In particular, test prioritization aims to schedule the execution order of existing tests so as to maximize some testing objective (e.g., the rate of fault-detection). In the literature, a huge body of research work has been dedicated to test prioritization to investigate various prioritization algorithms [6–10], coverage criteria [11, 12], and so on [13–15].

However, little work in the literature studied the threats to validity in test prioritization, especially the threat resulting from software changes. In software evolution, a software is usually modified by changing its source code and adding tests. However, the existing work on test prioritization is usually evaluated without considering test change. In particular, for a modified software, its test suite consists of: (1) existing tests, which are designed to test the software system before modification (denoted as S) and can be used to test the modified software system (denoted as S'), and (2) new tests, which are added to test the modification. Although all these tests are used to test the modified software, none of the existing work on test prioritization is applied or evaluated by considering the influence of the added tests. Furthermore, the existing work on test prioritization is usually evaluated based on the software whose changes on the source code include only seeded program faults. That is, the difference between the original software system S and the modified software system S' is only in the seeded faults. However, the changes from S to S' consists of many edits, which are not necessarily related to program faults. In summary, existing work on test prioritization is usually evaluated based on such simplified artificial changes (without real source code changes and test additions), and thus it is not clear whether

*Corresponding author.

ICSE '16, May 14-22, 2016, Austin, TX, USA
 ISBN 978-1-4503-3900-1/16/05…$15.00
DOI: http://dx.doi.org/10.1145/2884781.2884874

the conclusions drawn by the existing work are still valid for real-world software evolution.

In this paper, we point out this important threat to validity in existing test prioritization, and investigate the effectiveness of existing test prioritization in real-world software evolution by conducting an empirical study on 8 open-source projects. To investigate the influence of source code change, we constructed two types of source code changes, seeding faults into the original software like existing test prioritization work does and seeding faults into the modified software (i.e., latter versions of a project). To investigate the influence of test additions, we prioritized tests without considering newly added tests as well as all tests including newly added tests. Furthermore, as time budgets are limited in some cases, we also investigated test prioritization considering the influence of time budgets. That is, in the empirical study, we implemented traditional techniques [3, 7, 8] and time-aware techniques [16], each of which is implemented based on coverage information of various granularities (i.e., statement, method, and branch coverage).

The experimental results show that the test additions have significant influence on the effectiveness of test prioritization. When new tests are added, all the traditional and time-aware test prioritization techniques become much less effective. Another interesting finding is that source code changes alone (without test additions) do not have much influence on the effectiveness of test prioritization. Both traditional and time-aware test prioritization techniques perform quite stable in case of source code changes without test additions, indicating that out-of-date coverage information (due to code changes) can still be effective for test prioritization.

The paper makes the following contributions:

- Pointing out an important threat to validity in the evaluation of existing test prioritization – evolution of source code and tests.
- The first empirical study on investigating the influence of the threat resulting from real-world software evolution, by considering various time-unaware and time-aware prioritization techniques based on various coverage information.
- Interesting findings showing that both traditional and time-aware test prioritization are significantly negatively influenced by test additions, but not influenced much by source code changes alone. Practical guidelines are also learnt from the study to help with test prioritization in practice.

2. STUDIED TECHNIQUES

In this section, we explain the technical details about the techniques investigated in the empirical study.

2.1 Traditional Test Prioritization

Given any test suite T and its set of permutations on tests PT, traditional test prioritization aims to find a permutation $T' \in PT$ that for any permutation $T'' \in PT$ ($T'' \neq T'$), $f(T'') \leq f(T')$, where f is an objective function from a permutation to a real number.

Most existing test prioritization techniques guide their prioritization process based on coverage information, which refers to whether any structural unit is covered by a test. In this section, we explain the prioritization techniques based on statement coverage, but they are applicable to other types of structural coverage (e.g., method coverage or branch coverage), which will be further investigated in Section 3.8.4.

2.1.1 Total&Additional Test Prioritization

The total test prioritization technique schedules the execution order of tests based on the descendent number of statements covered by these tests, whereas the additional test prioritization technique schedules the execution order of tests based on the number of statements that are uncovered by already selected tests but covered by these tests.

The total&additional test prioritization techniques are simple greedy algorithms, but they are recognized as representative test prioritization techniques due to their effectiveness and are taken as the control techniques in the evaluation of existing work [8, 9].

2.1.2 Search-Based Test Prioritization

Search-based test prioritization is proposed by Li et al. [8], and is an application of search-based software engineering [17, 18]. Typically, search-based test prioritization takes all the permutations as candidate solutions and utilizes some heuristics to guide the process of searching for a better execution order of tests. In this empirical study, we use genetic-algorithm-based test prioritization as a representative of search-based test prioritization due to its effectiveness [8].

The genetic-algorithm-based test prioritization technique [8] encodes a permutation of a test suite by an N-size binary array, which represents the position of each test in the prioritized test suite. Initially, a set of permutations are randomly generated, which is taken as the initial population. Each pair of permutations in the population is taken as parent permutations to generate two offspring permutations through crossover on a random position. For each offspring permutation, the mutation operator[1] randomly selects two tests and exchanges their positions. To find an optimal solution, the genetic-algorithm-based test prioritization technique defines a fitness function based on the coverage of tests (e.g., average percentage of statement coverage).

2.1.3 Adaptive Random Test Prioritization

Based on random test prioritization, Jiang et al. [7] presented a set of adaptive random test prioritization techniques by varying the types of coverage information and function f_2 that determines which test to select in prioritization process. In particular, adaptive random test prioritization has three types of f_2: (1) f_2 is defined to select a test that has the largest average distance with the existing selected tests, (2) f_2 is defined to select a test that has the largest maximum distance with the existing selected tests, and (3) f_2 is defined to select a test that has the largest minimum distance with the existing selected tests. Furthermore, the random test prioritization technique of the last type of f_2 is evaluated to be more effective and efficient [7], which is taken as the representative of adaptive random test prioritization in this paper.

2.2 Time-Aware Test Prioritization

Considering the time budget, sometimes it is impossible to run all the tests and thus it is necessary to investigate

[1]The concept "mutation operator" used here refers to an operation in genetic programming, and is different from the concept used in mutation testing.

test prioritization with time constraints. Considering the time limit and various execution time of tests, time-aware test prioritization is proposed in the literature [16, 19, 20], which is usually formalized as follows. Given a test suite T, its set of permutations on the tests PT, and time budget $time_{max}$, time-aware test prioritization aims to find a permutation $T' \in PT$ satisfying that for any element $T'' \in PT$ $(T' \neq T'')$, $f(T') \geq f(T'')$, $time(T') \leq time_{max}$, and $time(T'') \leq time_{max}$, where f and $time$ are the objective and cost functions from permutations to real numbers [16], respectively. Compared with traditional test prioritization, time-aware test prioritization additionally requires the prioritized test suite to satisfy the time constraint.

2.2.1 Total&Additional Test Prioritization

Time-aware total/additional test prioritization is adopted from traditional total/additional test prioritization by considering the time budget [16]. In particular, time-aware total/additional test prioritization first schedules the execution order of tests like traditional total/additional test prioritization, and then keeps the preceding tests whose total execution time does not exceed the time budget by removing the remaining tests.

2.2.2 Total&Additional Test Prioritization via Integer Linear Programming

Based on traditional total&additional test prioritization, Zhang et al. [16] proposed integer linear programming (ILP) based test prioritization, which formalizes test selection in the process of test prioritization using an ILP model. In particular, Zhang et al. [16] presented a total test prioritization technique via ILP, which first selects a set of tests that has the maximum sum of the number of statements covered by **each test** and whose total execution time does not exceed the given time budget and then schedules the execution order of these selected tests using the traditional total strategy. Also, Zhang et al. [16] presented an additional test prioritization technique via ILP, which selects a set of tests that maximizes the number of statements covered by **these tests** and whose total execution time does not exceed the given time budget.

For each technique, we implement its variants based on various coverage criteria (i.e., method, statement, and branch coverage), and thus we have $8 * 3 = 24$ variant techniques.

3. EMPIRICAL STUDY

3.1 Research Questions

This study investigates the following research questions:

- **RQ1:** How does software evolution influence traditional test prioritization techniques in real-world evolving software systems?
- **RQ2:** How does software evolution influence time-aware test prioritization techniques in real-world evolving software systems?
- **RQ3:** How do source code differences alone (i.e., without test additions) influence traditional and time-aware test prioritization techniques?
- **RQ4:** How do different test prioritization techniques compare with random prioritization in software evolution?

In *RQ1* and *RQ2*, we investigate the impact of real-world software evolution (including both code changes and test additions) for both traditional and time-aware test prioritization. Since added tests vary for different projects, we further distinguish the impacts of added tests from the impacts of source code changes (which cause test coverage changes). That is, in *RQ3*, we exclude the tests that are newly added, and only investigate the effectiveness of various prioritization techniques for tests that exist in earlier versions (and thus only have source code changes). Furthermore, in *RQ4*, we investigate when the simple random prioritization can be competitive to the existing techniques.

3.2 Implementation and Supporting Tools

To collect various coverage information, we used on-the-fly bytecode instrumentation which dynamically instruments classes loaded into the JVM through a Java agent without any modification of the target program. We implemented code instrumentation based on the *ASM byte-code manipulation and analysis framework*[2] under our *FaultTracer* tool [21,22]. To implement the ILP based techniques (i.e., total&additional test prioritization via ILP), we used a mathematical programming solver, *GUROBI Optimization*[3], which is used to represent and solve the equations formulated by the ILP-based techniques. All the test prioritization techniques have been implemented as Java and Python programs. Note that all the tools developed by ourselves are available from our project website. Finally, we used the *PIT* mutation testing tool[4] to seed faults into our subject programs.

3.3 Subject Systems, Tests, and Faults

In the empirical study, we selected 8 GitHub[5] Java projects that have been widely used in software testing research [23–25]. Each project has accumulated various commits during real-world software evolution. Following prior work [23], for each project, we first chose the latest commit that can be successfully applied with the used tools, then selected up to 10 versions by counting backwards 30 commits each time. Note that each project version has a corresponding JUnit test suite accumulated during software development. In software testing research, it has been shown by previous studies [26–28] that mutation faults are close to real faults and are suitable for software testing experimentation, since real faults are usually hard to find and small in number, making it hard to perform statistical analysis. Furthermore, mutation faults have also been widely used in test prioritization research [9, 12, 29]. Therefore, in this work, we use mutation faults generated by the *PIT* mutation testing tool to investigate the effectiveness of various test prioritization techniques.

Table 1 presents the basic information of each subject, including its number of versions (abbreviated as "Ver"). As each project has more than one versions, the columns "MinS" and "MaxS" present the minimum and maximum numbers of lines of code for each project, whereas the columns "MinT" and "MaxT" present the minimum and maximum numbers of tests for each project. As this empirical study is designed to investigate the influences of test addition, the columns "Ft1" and "Ft2" presents the numbers of faults used for the

[2]http://asm.ow2.org/
[3]http://www.gurobi.com/
[4]http://pitest.org/
[5]https://github.com/

Sub	Ver	MinS	MaxS	MinT	MaxT	Ft1	Ft2
jasmine-maven	5	1,640	4,348	7	118	500	10
java-apns	8	1,362	3,839	15	87	410	80
jopt-simple	4	6,636	8,569	394	657	500	500
la4j	9	8,094	12,555	172	625	500	500
scribe-java	8	2,497	5,957	38	99	500	385
vraptor-core	5	31,176	32,997	985	1,124	500	500
assertj-core	8	55,443	67,282	4,055	5,269	500	500
metrics-core	6	11,477	12,536	270	318	500	500

Table 1: Subject statistics

study with test additions and the number of faults used for the study without test addition, respectively[6].

3.4 Independent Variables

We consider the following independent variables (IVs) in the empirical study.

IV1: Prioritization Scenarios. We consider both (1) traditional test prioritization which prioritizes and executes all tests, and (2) time-aware test prioritization which only prioritizes and executes tests within certain time constraints.

IV2: Prioritization Techniques. For each test prioritization scenario, we also consider various representative prioritization techniques. For traditional prioritization, we consider (1) total technique, (2) additional technique, (3) search-based technique, and (4) adaptive random technique. For time-aware prioritization, we consider (1) total technique, (2) additional technique[7],(3) total ILP-based technique, and (4) additional ILP-based technique. Besides these techniques, we also implement random prioritization as the controlled technique for traditional and time-aware prioritization in comparison. In random prioritization, we randomly generate 1000 execution orders and take their average results as suggested by the literature [30].

IV3: Coverage Criteria. Since all the studied techniques rely on code coverage information, we also investigate the influence of coverage criteria. More specifically, we studied three widely used coverage criteria: (1) method coverage, (2) statement coverage, and (3) branch coverage.

IV4: Revision Granularities. Existing work mainly applies test prioritization based the coverage data of software version v_i on the same version with seeded faults. However, in practice, when prioritizing tests for v_i, we only have coverage of an older version, e.g., v_{i-1}. Therefore, besides, applying test prioritization based on coverage of v_i to v_i (denoted as $v_i \rightarrow v_i$), we also study $v_{i-1} \rightarrow v_i$, $v_{i-2} \rightarrow v_i$, ..., $v_1 \rightarrow v_i$, to study the influence of larger revision for test prioritization.

IV5: Time Budgets. In time-aware test prioritization, only tests within certain time budgets are allowed to execute. Following existing work [16], we consider the following time budgets $b \in \{5\%, 25\%, 50\%, 75\%\}$, where b refers to the percentage of execution time of the entire test suite.

3.5 Dependent Variable

To evaluate the effectiveness of the various test prioritization techniques in software evolution, we used the widely used APFD (Average Percentage of Faults Detected) metric [3, 6, 9, 31]. For any given test suite and faulty program, higher APFD values imply higher fault-detection rate. To better evaluate the effectiveness of time-aware prioritization, we followed existing work on time-aware prioritization and adopted a slight variant of the traditional APFD metric [16, 19]. The details of the variant APFD metric can be found in [19].

3.6 Experimental Setup

For each subject system S, we obtain a set of subsequent revisions, e.g., $S_{v_1}, S_{v_2}, ..., S_{v_n}$. Then, for the latest version S_{v_n}, we apply the following process.

First, we construct various faulty versions of S_{v_n}. According to existing work [9,12,29], a specific program version usually does not contain a large number of faults. Therefore, similar to the existing work [9], we construct faulty versions by grouping 5 faults together. That is, for each program version, we randomly produce up to 100 fault groups each of which contains 5 randomly selected faults. Note that we guarantee that the selected faults can be detected by at least one test, and the different fault groups should not have common faults. That said, we have constructed up to 500 faulty versions for each program version.

Second, we collect three types of coverage (i.e., method, statement, and branch coverage) for S_{v_n} and all its earlier versions, i.e., S_{v_1} to $S_{v_{n-1}}$. That said, we have $3n$ coverage files for each project.

Third, we apply all the studied techniques based on the coverage files of S_{v_n} to faulty versions of S_{v_n}, denoted as $S_{v_n} \rightarrow S_{v_n}$. In addition, we also apply the techniques based on the coverage files of earlier versions to faulty versions of S_{v_n}, i.e., $S_{v_i} \rightarrow S_{v_n}$ $(i \in [1, n-1])$. Note that for newly added tests without coverage, we simply put them at the end of the prioritization sequence in an alphabetical order.

Finally, we collect all the prioritization APFD values of S_{v_n} based on the combination of all our independent variables, and apply data analysis.

3.7 Threats to Validity

The threat to internal validity lies in the implementation of the test prioritization techniques used in the empirical study. To reduce this threat, we used existing tools (e.g., GUROBI) to aid the implementation and reviewed all the code of the test prioritization techniques before conducting the empirical study.

The threats to external validity mainly lie in the subjects and faults. Although the subjects may not be sufficiently representative, especially for programs in other programming languages, all the subjects used in the empirical study are real-world Java projects and have been used in previous studies on software testing [23, 24]. This threat can be further reduced by using more programs in various programming languages (e.g., C, C++, Java and C#).

In software development, whenever the developers detect any regression fault, they usually will fix the fault before commiting rather than commiting a version with failed tests. Therefore, it is hard to find real regression faults in open-source code repositories. In addition, according to the experimental study conducted by Do and Rothermel [29], mutation-generated faults are suitable to be used in test prioritization studies. Therefore, we used mutation faults in our evaluation. To further reduce this threat, we performed a preliminary study by using all the 27 real-world faults of joda-time from Defects4J [32] to simulate real regression faults. The study further confirms that test additions impact test pri-

[6]Note that the number of faults used can be different for the two settings because different sets of faults are detected by different sets of tests.

[7]We consider the total and additional techniques here because they have been studied as baseline techniques in time-aware prioritization [16].

oritization results significantly while source code changes do not.

The threat to construct validity mainly lies in the metric used to assess the effectiveness of prioritization techniques. The APFD metric we used has been extensively used in prior work [3, 16, 19], but it does not consider all the costs and benefits related to test prioritization [29]. Further reduction of this threat requires more study using more metrics.

3.8 Result Analysis

In this section, we present our main experimental findings. Note that all our detailed experimental data and results are available online[8].

3.8.1 RQ1: Impacts of Software Evolution on Traditional Test Prioritization

Figure 1 presents the boxplots of the APFD values for all the four traditional test prioritization techniques using the statement coverage criterion on all projects. The results for the other two coverage criteria are quite close to those of statement coverage and can be found on our webpage. Each sub-figure presents the detailed APFD results when using each version's coverage information to prioritize the tests for the latest version for each project using statement coverage. In each sub-figure, the x-axis shows the versions from which the coverage is collected, the y-axis shows the APFD values for the techniques when prioritizing tests for the latest version, and each boxplot presents the APFD distribution (median, upper/lower quartile, and 95th/5th percentile values) based on different fault groups for each technique using each version's coverage information. We use white, blue, green, orange, and red boxes to represent the random, total, additional, ART, and search-based techniques respectively. From the figure, we have the following observations:

First, for all projects, all the techniques except the total technique have a clear growth trend when using more up-to-date coverage for prioritization. For example, for java-apns, the median APFD value of the additional strategy is 0.59 when using v_1's statement coverage to prioritize tests for v_8, and it grows to 0.87 when using v_8's statement coverage to prioritize tests for v_8. The total strategy does not have a clear growth trend for the majority of the subjects. We think the reason to be that the total technique does not use the coverage information effectively, and thus is not sensitive to the coverage changes.

Second, when using the most up-to-date coverage information, for the majority of the subjects, the additional and the search-based techniques are among the best prioritization technique, while the ART technique is also competitive. This finding confirms the previous work on search-based test prioritization [8] and adaptive random test prioritization [7], respectively. In addition, our results further demonstrate that the search-based test prioritization technique is superior to the ART technique (to our knowledge, those two techniques have never been compared before).

Third, when the version used for coverage collection is far from the version for prioritization (i.e., massive code changes occur), all the four techniques become close in performance, demonstrating the importance of using up-to-date coverage information for test prioritization. For example, for la4j, the total, additional, ART, and search-based techniques have median APFD values of 0.53, 0.85, 0.80, and 0.83, respectively, when using coverage of v_8 to prioritize tests for v_9. In contrast, the median APFD value of the total technique increases to 0.67, while the median APFD values of all the other techniques drop to around 0.71, when using coverage of v_1 to prioritize tests for v_9.

3.8.2 RQ2: Impacts of Software Evolution on Time-Aware Test Prioritization

Figure 2 presents the APFD values for all the four time-aware test prioritization techniques (together with the random prioritization) using the 50% time constraint based on the statement coverage[9]. Each sub-figure presents the detailed APFD results when using each version's coverage information to prioritize the tests for the latest version for each project using the 50% time constraint. The settings for each sub-figure are the same with those in Figure 1. The only difference is that now we use white, blue, green, orange, and red boxes to represent the random, total, additional, ILP-total, and ILP-additional techniques, respectively. From the results, we have the following observations:

First, the additional, ILP-total, and ILP-additional techniques have a clear growth trend when using more up-to-date coverage information for test prioritization. For example, for jopt-simple (with the 5% time constraint) the median APFD value of the additional strategy is 0.38 when using v_1's statement coverage to prioritize tests for v_4, and it grows to 0.58 when using v_4's statement coverage to prioritize tests for v_4.

Second, similar as traditional test prioritization without time constraints, the total technique also does not have a clear growth trend when using more up-to-date coverage information. For example, for java-apns when using 50% time constraint for test prioritization, surprisingly, when we use code coverage of v_8 to prioritize tests for v_8, the median APFD value even falls below 0, while it becomes larger than 0.6 when using less up-to-date coverage information (i.e., from v_7). We looked into the code, and found the reason to be that version v_8 introduced a newly-added huge test (i.e., `handleTransmissionErrorInQueuedConnection` of test class `ApnsConnectionCacheTest`), which has an average execution time of 248ms (far more than the execution time of other tests) and covers a huge amount of code elements (statements/methods/branches). Therefore, when the total technique uses v_8's coverage to prioritize tests, it tends to put that huge test at the very beginning. Then, under the 50% time constraint, only 4 tests can be run when using v_8's coverage to prioritize tests for v_8. On the contrary, when using less up-to-date coverage information, the newly added huge test does not have coverage at all, and will be put into the end of the test execution sequence, enabling more tests to be executed. For example, under the 50% time constraint, when using the coverage information from v_1 to v_7 to prioritize tests for v_8, 41, 57, 67, 67, 73, 73, 75 tests can be executed, respectively. Therefore, the APFD values for those coverage versions are much higher.

Third, when using the most up-to-date coverage information, the ILP-additional technique always performs the best, which confirms previous study on time-aware test prioritization [16]. Furthermore, our results also demonstrate that the ILP-additional technique is the most stable and effective

[8]http://utdallas.edu/~yxl131230/icse16support.html

[9]Note that due to the space limitation, we present the results for other coverage criteria and time constraints (similar to the results shown in this paper) in our project homepage.

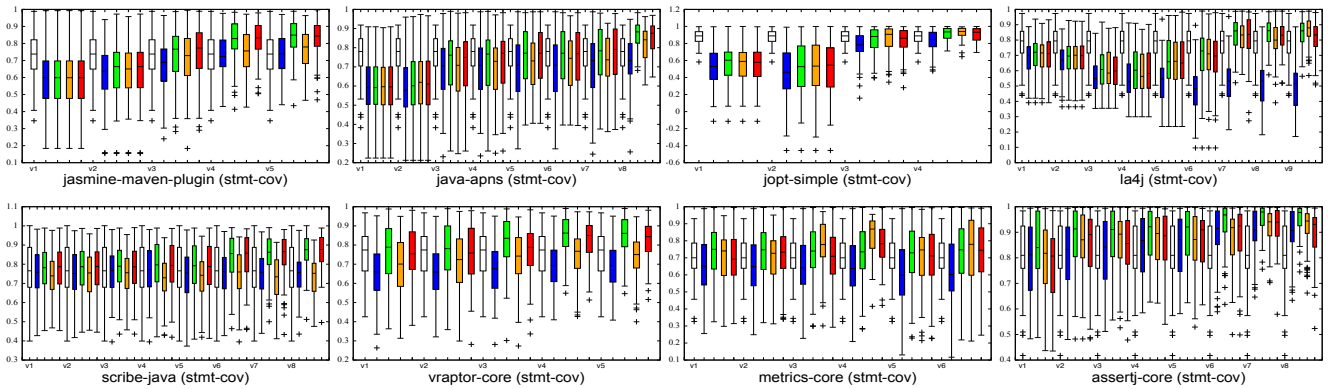

Figure 1: Results for traditional test prioritization techniques (i.e., random ○, total ●, additional ●, ART ●, and search-based ●) based on statement coverage (with new tests)

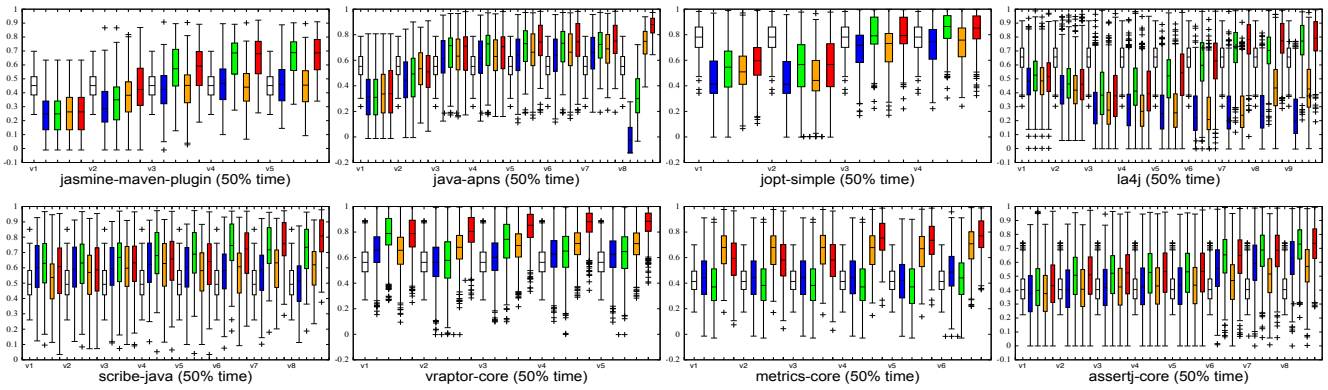

Figure 2: Results for time-aware test prioritization techniques (i.e., random ○, total ●, additional ●, ILP-total ●, and ILP-additional ●) with statement coverage and 50% time constraint (with new tests)

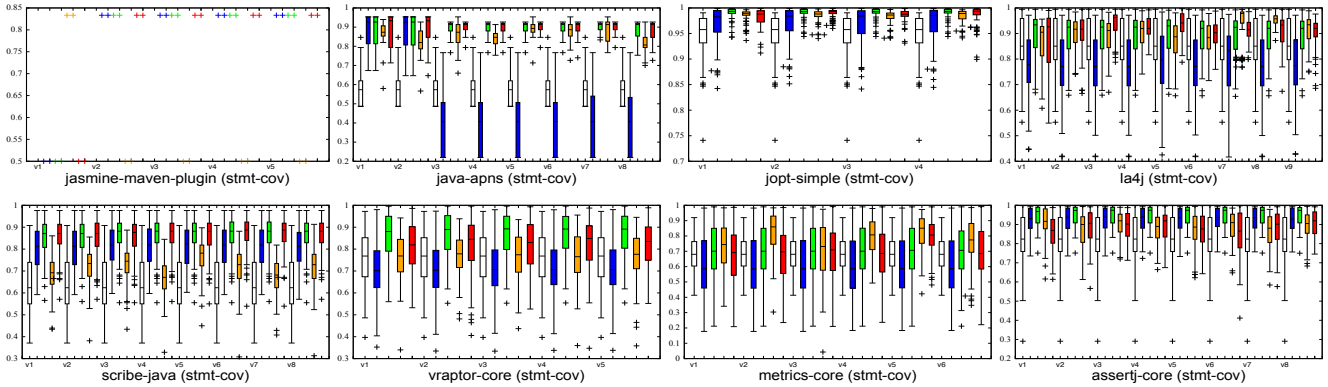

Figure 3: Results for traditional test prioritization techniques (i.e., random ○, total ●, additional ●, ART ●, and search-based ●) based on statement coverage (excluding new tests)

technique even when using less up-to-date coverage information.

3.8.3 RQ3: Impacts of Only Source Code Changes on Test Prioritization

Similar with Figure 1, Figure 3 presents the APFD values for all the four traditional test prioritization techniques using statement coverage criteria on all projects. Similar with Figure 2, Figure 4 presents the APFD values for all the four time-aware test prioritization techniques (together with the random prioritization) using the 50% time constraints based on the statement coverage. The only change is that in Figure 3 and Figure 4, we consider only the tests that exist for all versions in order to study the influence of coverage change alone (i.e., excluding the influence of test additions)[10]. Note that the corresponding figures for other coverage and time constraint settings are all available from our webpage. From the two figures, we observe that for traditional test prioritization, the additional and search-based techniques perform the best across all versions. For example, for la4j, when using the coverage from the earliest version to prioritize tests of the latest version, both the additional and search-based techniques have median APFD results of 0.93, whereas the total and ART techniques have median APFD results of 0.78 and 0.91, respectively. For time-aware test prioritization, the ILP-additional technique performs the best nearly

[10]Note that there are only 3 tests in common for all versions of jasmine-maven, thus the boxplots cannot be shown.

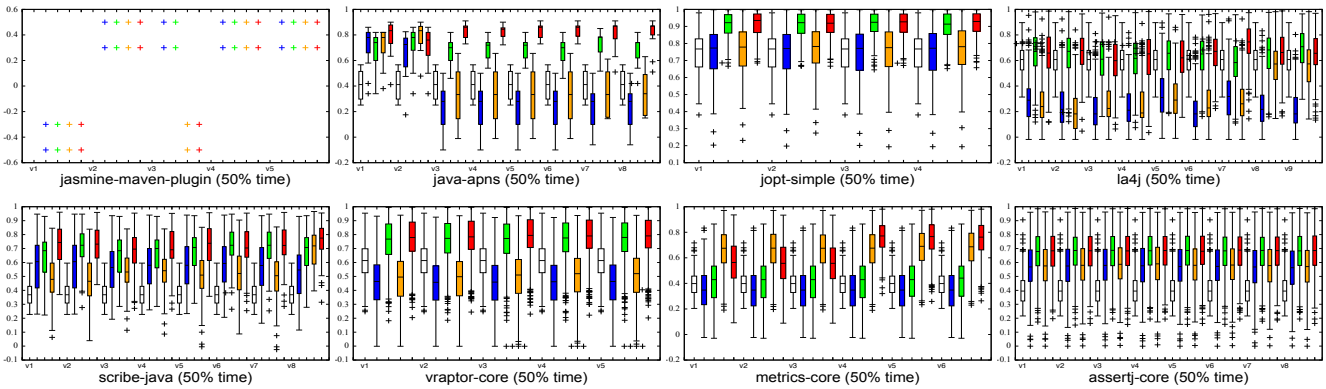

Figure 4: Results for time-aware test prioritization techniques (i.e., random ○, total ●, additional ●, ILP-total ●, and ILP-additional ●) with statement coverage and 50% time constraint (excluding new tests)

Tech	Sub	With new tests										Without new tests									
		p-value	v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8	v_9	p-value	v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8	v_9
Total	jopt-simple	$p<0.05$	b	b	a	a	–	–	–	–	–	0.987	a	a	a	a	–	–	–	–	–
	vraptor-core	0.147	a	a	a	a	a	–	–	–	–	0.999	a	a	a	a	a	–	–	–	–
	metrics-core	0.339	a	a	a	a	a	a	–	–	–	1.000	a	a	a	a	a	a	–	–	–
	java-apns	$p<0.05$	b	b	a	a	a	a	a	a	–	$p<0.05$	a	a	b	b	b	b	b	b	–
	assertj-core	$p<0.05$	b	b	b	b	b	a	a	a	–	1.000	a	a	a	a	a	a	a	a	–
	jasmine-maven	$p<0.05$	c	bc	b	a	a	–	–	–	–	$p<0.05$	b	a	a	a	a	–	–	–	–
	scribe-java	0.854	a	a	a	a	a	a	a	a	–	0.960	a	a	a	a	a	a	a	a	–
	la4j	$p<0.05$	a	a	b	bc	bcd	d	bcd	bcd	cd	0.956	a	a	a	a	a	a	a	a	a
Addit.	jopt-simple	$p<0.05$	b	b	a	a	–	–	–	–	–	0.998	a	a	a	a	–	–	–	–	–
	vraptor-core	$p<0.05$	b	b	ab	a	a	–	–	–	–	0.987	a	a	a	a	a	–	–	–	–
	metrics-core	0.991	a	a	a	a	a	a	–	–	–	1.000	a	a	a	a	a	a	–	–	–
	java-apns	$p<0.05$	c	c	b	b	b	b	b	a	–	0.842	a	a	a	a	a	a	a	a	–
	assertj-core	$p<0.05$	d	c	c	bc	bc	ab	a	a	–	1.000	a	a	a	a	a	a	a	a	–
	jasmine-maven	$p<0.05$	c	c	b	a	a	–	–	–	–	$p<0.05$	b	a	a	a	a	–	–	–	–
	scribe-java	$p<0.05$	c	c	c	bc	bc	ab	a	a	–	0.998	a	a	a	a	a	a	a	a	–
	la4j	$p<0.05$	b	b	c	c	bc	b	a	a	a	0.997	a	a	a	a	a	a	a	a	a
ART	jopt-simple	$p<0.05$	b	b	a	a	–	–	–	–	–	0.119	a	a	a	a	–	–	–	–	–
	vraptor-core	$p<0.05$	b	ab	ab	a	ab	–	–	–	–	0.931	a	a	a	a	a	–	–	–	–
	metrics-core	$p<0.05$	bc	c	b	a	bc	bc	–	–	–	$p<0.05$	c	a	c	ab	a	bc	–	–	–
	java-apns	$p<0.05$	d	cd	bc	b	b	b	b	a	–	$p<0.05$	a	a	a	a	a	a	a	a	–
	assertj-core	$p<0.05$	c	b	b	b	b	ab	a	a	–	$p<0.05$	a	ab	ab	ab	b	ab	b	ab	–
	jasmine-maven	$p<0.05$	b	b	a	a	a	–	–	–	–	$p<0.05$	a	b	b	b	b	–	–	–	–
	scribe-java	0.499	a	a	a	a	a	a	a	a	–	$p<0.05$	bcd	bc	ab	d	a	bcd	cd	bc	–
	la4j	$p<0.05$	c	c	de	e	cd	c	ab	b	a	$p<0.05$	e	cde	cde	cd	de	de	ab	a	bc
Search	jopt-simple	$p<0.05$	b	b	a	a	–	–	–	–	–	$p<0.05$	b	a	ab	ab	–	–	–	–	–
	vraptor-core	$p<0.05$	b	b	b	a	a	–	–	–	–	0.933	a	a	a	a	a	–	–	–	–
	metrics-core	$p<0.05$	b	ab	b	a	b	ab	–	–	–	$p<0.05$	b	b	b	b	a	b	–	–	–
	java-apns	$p<0.05$	c	c	b	b	b	b	b	a	–	0.973	a	a	a	a	a	a	a	a	–
	assertj-core	$p<0.05$	d	bc	c	bc	bc	abc	a	ab	–	$p<0.05$	b	ab	ab	ab	b	b	ab	a	–
	jasmine-maven	$p<0.05$	d	c	b	a	a	–	–	–	–	$p<0.05$	b	a	a	a	a	–	–	–	–
	scribe-java	$p<0.05$	c	c	c	bc	bc	ab	a	a	–	0.951	a	a	a	a	a	a	a	a	–
	la4j	$p<0.05$	b	b	cd	d	bc	b	a	a	a	$p<0.05$	c	b	a	ab	a	bc	ab	b	b

Table 2: ANOVA analysis and Tukey's HSD test among using statement coverage of different versions for traditional prioritization

across all versions for all time constraints, and the additional technique can be competitive when the time constraints are not tight (e.g., >=75%).

We also observe that for all the studied techniques, their effectiveness is not influenced much by the use of coverage data from different versions. To illustrate, for the vast majority of the sub-figures in Figures 1 and 2, the effectiveness of all techniques tends to increase when using more up-to-date coverage information. However, for all sub-figures in Figures 3 and 4, the effectiveness of all techniques is stable when using coverage from different versions. To further confirm our observation, we perform *one-way ANOVA analysis* [33] at the significance level of 0.05 to investigate whether using coverage information from different versions incur any significant effectiveness differences inside each techique. In addition, we further perform *Tukey HSD post-hoc test* to rank the effectiveness when using coverage information of different versions as different groups for each technique. The detailed results for traditional/time-aware test prioritization are shown in Table 2/3. In each table, Columns 1 and 2 list all the techniques and subjects. Column 3 presents the one-way ANOVA analysis results, and Column 4 presents the detailed Tukey HSD grouping results ("a" means the group with best effectiveness) on using coverage data of different versions when considering new tests. Similarly, Columns 5 and 6 present the corresponding one-way ANOVA analysis and Tukey HSD test results when excluding new tests.

From the tables, we observe that when considering new tests, the effectiveness of each technique has significant differences (rejecting the NULL hypothesis of one-way ANOVA analysis by $p < 0.05$) when using coverage data from different versions for the vast majority of the studied subjects. To illustrate, the search-based technique for traditional test prioritization has significant differences on all the studied subjects, and all the studied time-aware test prioritization techniques have significant differences on all subjects. In

Tech	Sub	With new tests									Without new tests									
		p-value	v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8	v_9	p-value	v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8	v_9
Total	jopt-simple	$p<0.05$	b	b	a	a	–	–	–	–	–	1.000	a	a	a	a	–	–	–	–	–
	vraptor-core	$p<0.05$	a	d	c	b	b	–	–	–	–	0.942	a	a	a	a	a	–	–	–	–
	metrics-core	$p<0.05$	b	b	b	b	b	a	–	–	–	1.000	a	a	a	a	a	a	–	–	–
	java-apns	$p<0.05$	c	b	a	a	a	a	a	d	–	$p<0.05$	a	a	b	b	b	b	b	b	–
	assertj-core	$p<0.05$	e	d	d	d	d	c	b	a	–	0.785	a	a	a	a	a	a	a	a	–
	jasmine-maven	$p<0.05$	c	b	a	a	a	–	–	–	–	$p<0.05$	b	a	a	a	a	–	–	–	–
	scribe-java	$p<0.05$	a	a	a	a	a	ab	ab	b	–	0.458	a	a	a	a	a	a	a	a	–
	la4j	$p<0.05$	a	b	c	c	cd	de	d	d	e	$p<0.05$	b	c	cd	cd	a	cd	ab	cd	d
Addit.	jopt-simple	$p<0.05$	c	c	b	a	–	–	–	–	–	0.925	a	a	a	a	–	–	–	–	–
	vraptor-core	$p<0.05$	a	d	b	c	c	–	–	–	–	0.271	a	a	a	a	a	–	–	–	–
	metrics-core	$p<0.05$	b	b	b	b	b	a	–	–	–	0.983	a	a	a	a	a	a	–	–	–
	java-apns	$p<0.05$	c	b	a	a	a	a	a	c	–	0.732	a	a	a	a	a	a	a	a	–
	assertj-core	$p<0.05$	e	d	d	d	d	c	b	a	–	1.000	a	a	a	a	a	a	a	a	–
	jasmine-maven	$p<0.05$	d	c	b	a	a	–	–	–	–	$p<0.05$	b	a	a	a	a	–	–	–	–
	scribe-java	$p<0.05$	b	b	b	ab	ab	a	a	a	–	$p<0.05$	b	ab	ab	ab	ab	a	ab	ab	–
	la4j	$p<0.05$	d	f	g	f	e	c	b	b	a	$p<0.05$	ab	ab	de	cde	bc	bcd	e	ab	a
ILP-T	jopt-simple	$p<0.05$	b	c	a	a	–	–	–	–	–	0.933	a	a	a	a	–	–	–	–	–
	vraptor-core	$p<0.05$	c	b	a	a	a	–	–	–	–	$p<0.05$	b	b	ab	a	a	–	–	–	–
	metrics-core	$p<0.05$	b	b	b	b	b	a	–	–	–	0.809	a	a	a	a	a	a	–	–	–
	java-apns	$p<0.05$	d	c	b	b	ab	ab	ab	a	–	$p<0.05$	a	a	b	b	b	b	b	b	–
	assertj-core	$p<0.05$	e	d	d	d	d	c	b	a	–	0.763	a	a	a	a	a	a	a	a	–
	jasmine-maven	$p<0.05$	c	b	ab	a	a	–	–	–	–	$p<0.05$	b	a	b	a	a	–	–	–	–
	scribe-java	$p<0.05$	b	ab	a	a	ab	ab	a	a	–	$p<0.05$	b	b	b	b	b	b	b	a	–
	la4j	$p<0.05$	a	b	c	cd	cd	e	de	b	b	$p<0.05$	c	d	c	c	b	c	bc	a	a
ILP-A	jopt-simple	$p<0.05$	c	d	b	a	–	–	–	–	–	0.970	a	a	a	a	–	–	–	–	–
	vraptor-core	$p<0.05$	d	c	b	a	a	–	–	–	–	$p<0.05$	b	ab	a	a	a	–	–	–	–
	metrics-core	$p<0.05$	c	c	c	ab	b	a	–	–	–	$p<0.05$	b	b	b	a	a	a	–	–	–
	java-apns	$p<0.05$	e	d	c	c	bc	bc	b	a	–	$p<0.05$	ab	b	a	ab	a	a	ab	ab	–
	assertj-core	$p<0.05$	e	d	d	d	d	c	b	a	–	1.000	a	a	a	a	a	a	a	a	–
	jasmine-maven	$p<0.05$	d	c	b	a	a	–	–	–	–	$p<0.05$	b	a	b	a	a	–	–	–	–
	scribe-java	$p<0.05$	d	d	d	cd	cd	bc	ab	a	–	0.209	a	a	a	a	a	a	a	a	–
	la4j	$p<0.05$	c	d	f	e	c	b	a	a	a	$p<0.05$	bcd	b	e	cde	de	bcd	a	bc	bcd

Table 3: ANOVA analysis and Tukey's HSD test among using statement coverage of different versions for time-aware prioritization

addition, for most techniques, Tukey HSD test shows that using more up-to-date coverage data tends to be more effective. For example, the effectiveness of the search-based technique is grouped into "a" or "ab" when using the most up-to-date coverage, while being grouped into "b" to "d" when using the most obsolete coverage data. On the contrary, when excluding new tests, the studied techniques tend not to have significant differences when using coverage data from different versions. In addition, even for the cases that there are statistical differences, the techniques using the most up-to-date coverage data are usually not categorized as the best performance group. This observation further confirms that source code changes alone (i.e., excluding impacts of new tests) do not impact the effectiveness of test prioritization techniques much.

3.8.4 RQ4: Comparison with Random Prioritization

As simple random techniques can be powerful in software testing research [34, 35], we also investigate the strengths of random technique in test prioritization. More specifically, we compare all the studied techniques against random prioritization both considering new tests and excluding new tests. We chose to use the *Wilcoxon Signed-Rank Test* [36] for the comparison, because it is suitable even for the case that the sample differences may not be normally distributed. Table 4/5 shows the detailed Wilcoxon test results at the 0.05 significance level for traditional/time-aware test prioritization techniques. In each table, Columns 1 and 2 list all the subjects and techniques. Columns 3 lists the statistical test results for comparing each technique against the random prioritization when using each version's coverage to prioritize all tests for the latest version. Column 4 presents similar results with Column 3 but excluding new tests. We use ❍ to denote that there is no statistical difference, ✔ to denote that the studied technique is significantly better, and ✖ to denote that the random prioritization is significantly better.

From the two tables, we observe that random prioritization can be competitive when programs evolve with new tests. For example, when prioritizing tests for v_8 using coverage data from v_7 for java-apns, all traditional test prioritization techniques cannot outperform the random prioritization. The results on time-aware test prioritization further confirms this finding. For example, when prioritizing tests for v_9 of la4j under the 50% time constraint, random prioritization can outperform all techniques when using coverage data earlier than v_7. In contrast, when considering only source code changes (i.e., excluding new tests), a number of techniques can always significantly outperform the random prioritization in both traditional and time-aware test prioritization. For traditional test prioritization, all the studied techniques except the total technique are at least as effective as the random prioritization using coverage data from any version for all subjects. For time-aware test prioritization, the additional and ILP-additional techniques are at least as effective as the random prioritization using coverage data from any version for all subjects.

In summary, random prioritization can be competitive when new tests are involved in software evolution. However, state-of-the-art techniques can outperform random prioritization stably when excluding new tests.

3.9 Practical Guidelines

Our study reveals a number of interesting findings that can serve as the practical guidelines for test prioritization.

Source code changes vs. test additions. Our experimental results demonstrate that the effectiveness of test prioritization techniques can be greatly influenced by software evolution (including source code changes and test additions). However, when only considering source code changes (i.e.,

Sub	Tech	With new tests									Without new tests								
		v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8	v_9	v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8	v_9
jasmine-maven	Total	✖(0.00)	✖(0.00)	✖(0.00)	○(0.94)	○(0.92)	–	–	–	–	○(0.35)	○(0.35)	○(0.35)	○(0.35)	○(0.35)	–	–	–	–
	Addit.	✖(0.00)	✖(0.00)	○(0.60)	✔(0.00)	✔(0.00)	–	–	–	–	○(0.35)	○(0.35)	○(0.35)	○(0.35)	○(0.35)	–	–	–	–
	ART	✖(0.00)	✖(0.00)	○(0.31)	✔(0.03)	✔(0.04)	–	–	–	–	○(0.35)	○(0.35)	○(0.35)	○(0.35)	○(0.35)	–	–	–	–
	Search	✖(0.00)	✖(0.00)	○(0.42)	✔(0.00)	✔(0.00)	–	–	–	–	○(0.35)	○(0.35)	○(0.35)	○(0.35)	○(0.35)	–	–	–	–
java-apns	Total	✖(0.00)	✖(0.00)	✖(0.00)	✖(0.00)	✖(0.00)	✖(0.00)	✖(0.00)	✖(0.00)	✖(0.01)	✔(0.00)	✔(0.00)	✖(0.00)	✖(0.00)	✖(0.00)	✖(0.00)	✖(0.00)	✖(0.00)	✖(0.00)
	Addit.	✖(0.00)	✖(0.00)	○(0.08)	○(0.13)	○(0.84)	○(0.82)	○(0.38)	✔(0.00)	–	✔(0.00)	✔(0.00)	✔(0.00)	✔(0.00)	✔(0.00)	✔(0.00)	✔(0.00)	✔(0.00)	–
	ART	✖(0.00)	✖(0.00)	✖(0.00)	✖(0.02)	○(0.08)	○(0.06)	✔(0.00)	–	–	✔(0.00)	✔(0.00)	✔(0.00)	✔(0.00)	✔(0.00)	✔(0.00)	✔(0.00)	✔(0.00)	–
	Search	✖(0.00)	✖(0.00)	✖(0.04)	○(0.15)	○(0.82)	○(0.75)	○(0.23)	✔(0.00)	–	✔(0.00)	✔(0.00)	✔(0.00)	✔(0.00)	✔(0.00)	✔(0.00)	✔(0.00)	✔(0.00)	–
jopt-simple	Total	✖(0.00)	✖(0.00)	✖(0.00)	✖(0.00)	–	–	–	–	–	✔(0.00)	✔(0.00)	✔(0.00)	✔(0.00)	–	–	–	–	–
	Addit.	✖(0.00)	✖(0.00)	○(0.14)	✔(0.00)	–	–	–	–	–	✔(0.00)	✔(0.00)	✔(0.00)	✔(0.00)	–	–	–	–	–
	ART	✖(0.00)	✖(0.00)	○(0.38)	✔(0.00)	–	–	–	–	–	✔(0.00)	✔(0.00)	✔(0.00)	✔(0.00)	–	–	–	–	–
	Search	✖(0.00)	✖(0.00)	✖(0.03)	✔(0.00)	–	–	–	–	–	✔(0.00)	✔(0.00)	✔(0.00)	✔(0.00)	–	–	–	–	–
la4j	Total	✖(0.00)	✖(0.00)	✖(0.00)	✖(0.00)	✖(0.00)	✖(0.00)	✖(0.00)	✖(0.00)	✖(0.00)	✖(0.00)	✖(0.00)	✖(0.00)	✖(0.00)	✖(0.00)	✖(0.00)	✖(0.00)	✖(0.00)	✖(0.00)
	Addit.	✖(0.00)	✖(0.00)	✖(0.00)	✖(0.00)	✖(0.00)	✖(0.00)	✔(0.00)	✔(0.00)	✔(0.00)	✔(0.03)	○(0.07)	○(0.07)	○(0.07)	✔(0.02)	○(0.07)	✔(0.03)	○(0.07)	○(0.07)
	ART	✖(0.00)	✖(0.00)	✖(0.00)	✖(0.00)	✖(0.00)	✖(0.00)	✔(0.00)	○(0.15)	✔(0.00)	○(0.63)	✔(0.00)	✔(0.00)	✔(0.00)	○(0.08)	✔(0.00)	✔(0.00)	✔(0.00)	✔(0.00)
	Search	✖(0.00)	✖(0.00)	✖(0.00)	✖(0.00)	✖(0.00)	✖(0.00)	✔(0.00)	✔(0.00)	✔(0.00)	○(0.47)	✔(0.00)	✔(0.00)	✔(0.00)	✔(0.00)	✔(0.00)	✔(0.00)	✔(0.00)	✔(0.00)
scribe-java	Total	○(0.73)	○(0.53)	○(0.69)	○(0.88)	○(0.13)	○(0.09)	○(0.18)	✖(0.03)	–	✔(0.00)	✔(0.00)	✔(0.00)	✔(0.00)	✔(0.00)	✔(0.00)	✔(0.00)	✔(0.00)	–
	Addit.	○(0.53)	○(0.60)	○(0.30)	○(0.10)	○(0.19)	✔(0.00)	✔(0.00)	✔(0.00)	–	✔(0.00)	✔(0.00)	✔(0.00)	✔(0.00)	✔(0.00)	✔(0.00)	✔(0.00)	✔(0.00)	–
	ART	○(0.06)	○(0.38)	○(0.23)	✖(0.01)	✖(0.03)	○(0.49)	✖(0.02)	✖(0.02)	–	✔(0.00)	✔(0.00)	✔(0.00)	✔(0.01)	✔(0.00)	✔(0.00)	✔(0.02)	✔(0.00)	–
	Search	○(0.44)	○(0.53)	○(0.31)	○(0.10)	○(0.24)	✔(0.00)	✔(0.00)	✔(0.00)	–	✔(0.00)	✔(0.00)	✔(0.00)	✔(0.00)	✔(0.00)	✔(0.00)	✔(0.00)	✔(0.00)	–
vraptor-core	Total	✖(0.00)	✖(0.00)	✖(0.00)	✖(0.00)	✖(0.00)	–	–	–	–	✖(0.00)	✖(0.00)	✖(0.00)	✖(0.00)	✖(0.00)	–	–	–	–
	Addit.	○(0.10)	✔(0.02)	✔(0.00)	✔(0.00)	✔(0.00)	–	–	–	–	✔(0.00)	✔(0.00)	✔(0.00)	✔(0.00)	✔(0.00)	–	–	–	–
	ART	✖(0.02)	○(0.09)	○(0.52)	○(0.51)	○(0.99)	–	–	–	–	○(0.88)	○(0.19)	○(0.81)	○(0.56)	○(0.39)	–	–	–	–
	Search	○(0.43)	○(0.29)	○(0.08)	✔(0.00)	✔(0.00)	–	–	–	–	✔(0.00)	✔(0.00)	✔(0.00)	✔(0.00)	✔(0.00)	–	–	–	–
metrics-core	Total	✖(0.00)	✖(0.00)	✖(0.00)	✖(0.00)	✖(0.00)	✖(0.00)	–	–	–	✖(0.00)	✖(0.00)	✖(0.00)	✖(0.00)	✖(0.00)	✖(0.00)	–	–	–
	Addit.	✔(0.00)	✔(0.00)	✔(0.00)	✔(0.00)	✔(0.05)	✔(0.00)	–	–	–	✔(0.03)	✔(0.03)	✔(0.03)	✔(0.03)	✔(0.02)	✔(0.03)	–	–	–
	ART	○(0.43)	○(0.91)	✔(0.00)	✔(0.00)	○(0.12)	✔(0.00)	–	–	–	✔(0.00)	✔(0.00)	✔(0.02)	✔(0.00)	✔(0.00)	✔(0.00)	–	–	–
	Search	○(0.70)	✔(0.01)	○(0.54)	✔(0.00)	○(0.88)	✔(0.00)	–	–	–	○(0.92)	○(0.81)	○(0.20)	○(0.33)	✔(0.00)	○(0.16)	–	–	–
assertj-core	Total	○(0.26)	○(0.89)	○(0.64)	○(0.09)	○(0.10)	✔(0.00)	✔(0.00)	✔(0.00)	–	✔(0.00)	✔(0.00)	✔(0.00)	✔(0.00)	✔(0.00)	✔(0.00)	✔(0.00)	✔(0.00)	–
	Addit.	○(0.10)	✔(0.00)	✔(0.00)	✔(0.00)	✔(0.00)	✔(0.00)	✔(0.00)	✔(0.00)	–	✔(0.00)	✔(0.00)	✔(0.00)	✔(0.00)	✔(0.00)	✔(0.00)	✔(0.00)	✔(0.00)	–
	ART	○(0.58)	✔(0.00)	✔(0.00)	✔(0.00)	✔(0.00)	✔(0.00)	✔(0.00)	✔(0.00)	–	✔(0.00)	✔(0.00)	✔(0.00)	✔(0.00)	✔(0.00)	✔(0.00)	✔(0.00)	✔(0.00)	–
	Search	✖(0.01)	✔(0.00)	✔(0.04)	✔(0.00)	✔(0.00)	✔(0.00)	✔(0.00)	✔(0.00)	–	✔(0.00)	✔(0.00)	✔(0.00)	✔(0.00)	✔(0.00)	✔(0.00)	✔(0.00)	✔(0.00)	–

Table 4: Wilcoxon tests between each studied technique and random technique based on statement coverage for traditional prioritization (p values included in brackets)

Sub	Tech	With new tests									Without new tests								
		v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8	v_9	v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8	v_9
jasmine-maven	Total	✖(0.00)	✖(0.00)	○(0.13)	○(0.40)	○(0.51)	–	–	–	–	○(0.50)	○(0.50)	○(0.50)	○(0.50)	○(0.50)	–	–	–	–
	Addit.	✖(0.00)	✖(0.00)	✔(0.00)	✔(0.00)	✔(0.00)	–	–	–	–	○(0.50)	○(0.50)	○(0.50)	○(0.50)	○(0.50)	–	–	–	–
	ILP-T	✖(0.00)	✖(0.00)	○(0.56)	○(0.71)	○(0.74)	–	–	–	–	○(0.50)	○(0.50)	○(0.50)	○(0.50)	○(0.50)	–	–	–	–
	ILP-A	✖(0.00)	○(0.68)	✔(0.00)	✔(0.00)	✔(0.00)	–	–	–	–	○(0.50)	○(0.50)	○(0.50)	○(0.50)	○(0.50)	–	–	–	–
java-apns	Total	✖(0.00)	✖(0.00)	✔(0.05)	✔(0.03)	✔(0.00)	✔(0.00)	✔(0.00)	✖(0.00)	–	✔(0.00)	✔(0.00)	✖(0.02)	✖(0.02)	✖(0.02)	✖(0.02)	✖(0.02)	✖(0.02)	–
	Addit.	✖(0.00)	✖(0.01)	✔(0.00)	✔(0.00)	✔(0.00)	✔(0.00)	✔(0.00)	✖(0.00)	–	✔(0.00)	✔(0.00)	✔(0.00)	✔(0.00)	✔(0.00)	✔(0.00)	✔(0.00)	✔(0.00)	–
	ILP-T	✖(0.00)	○(0.17)	✔(0.01)	✔(0.01)	✔(0.00)	✔(0.00)	✔(0.00)	✔(0.00)	–	✔(0.00)	✔(0.00)	○(0.35)	○(0.32)	○(0.32)	○(0.32)	○(0.25)	○(0.27)	–
	ILP-A	✖(0.00)	○(0.08)	✔(0.00)	✔(0.00)	✔(0.00)	✔(0.00)	✔(0.00)	✔(0.00)	–	✔(0.00)	✔(0.00)	✔(0.00)	✔(0.00)	✔(0.00)	✔(0.00)	✔(0.00)	✔(0.00)	–
jopt-simple	Total	✖(0.00)	✖(0.00)	✖(0.00)	✖(0.00)	–	–	–	–	–	○(0.73)	○(0.78)	○(0.85)	○(0.76)	–	–	–	–	–
	Addit.	✖(0.00)	✖(0.00)	✔(0.02)	✔(0.00)	–	–	–	–	–	✔(0.00)	✔(0.00)	✔(0.00)	✔(0.00)	–	–	–	–	–
	ILP-T	✖(0.00)	✖(0.00)	✖(0.00)	✖(0.01)	–	–	–	–	–	○(0.09)	○(0.10)	○(0.35)	○(0.20)	–	–	–	–	–
	ILP-A	✖(0.00)	✖(0.00)	✔(0.01)	✔(0.00)	–	–	–	–	–	✔(0.00)	✔(0.00)	✔(0.00)	✔(0.00)	–	–	–	–	–
la4j	Total	✖(0.00)	✖(0.00)	✖(0.00)	✖(0.00)	✖(0.00)	✖(0.00)	✖(0.00)	✖(0.00)	✖(0.00)	✖(0.00)	✖(0.00)	✖(0.00)	✖(0.00)	✖(0.00)	✖(0.00)	✖(0.00)	✖(0.00)	✖(0.00)
	Addit.	✖(0.00)	✖(0.00)	✖(0.00)	✖(0.00)	✖(0.00)	✖(0.00)	✔(0.00)	✔(0.00)	✔(0.00)	✔(0.00)	✔(0.00)	○(0.37)	○(0.23)	✔(0.00)	✔(0.00)	○(0.22)	✔(0.00)	✔(0.00)
	ILP-T	✖(0.00)	✖(0.00)	✖(0.00)	✖(0.00)	✖(0.00)	✖(0.00)	✖(0.00)	✖(0.00)	✖(0.00)	✖(0.00)	✖(0.00)	✖(0.00)	✖(0.00)	✖(0.00)	✖(0.00)	✖(0.00)	✖(0.00)	✖(0.00)
	ILP-A	✖(0.00)	✖(0.00)	✖(0.00)	✖(0.00)	✖(0.00)	✖(0.00)	✔(0.00)	✔(0.00)	✔(0.00)	✔(0.00)	✔(0.00)	○(0.66)	✔(0.01)	○(0.05)	✔(0.00)	✔(0.00)	✔(0.00)	✔(0.00)
scribe-java	Total	✔(0.00)	✔(0.00)	✔(0.00)	✔(0.00)	✔(0.00)	✔(0.00)	✔(0.01)	○(0.97)	–	✔(0.00)	✔(0.00)	✔(0.00)	✔(0.00)	✔(0.00)	✔(0.00)	✔(0.00)	✔(0.00)	–
	Addit.	✔(0.00)	✔(0.00)	✔(0.00)	✔(0.00)	✔(0.00)	✔(0.00)	✔(0.00)	✔(0.00)	–	✔(0.00)	✔(0.00)	✔(0.00)	✔(0.00)	✔(0.00)	✔(0.00)	✔(0.00)	✔(0.00)	–
	ILP-T	○(0.53)	✔(0.02)	✔(0.00)	✔(0.00)	✔(0.01)	✔(0.00)	✔(0.00)	✔(0.00)	–	✔(0.00)	✔(0.00)	✔(0.00)	✔(0.00)	✔(0.00)	✔(0.00)	✔(0.00)	✔(0.00)	–
	ILP-A	✔(0.00)	✔(0.00)	✔(0.00)	✔(0.00)	✔(0.00)	✔(0.00)	✔(0.00)	✔(0.00)	–	✔(0.00)	✔(0.00)	✔(0.00)	✔(0.00)	✔(0.00)	✔(0.00)	✔(0.00)	✔(0.00)	–
vraptor-core	Total	✔(0.00)	○(0.91)	✔(0.00)	✔(0.00)	✔(0.00)	–	–	–	–	✖(0.00)	✖(0.00)	✖(0.00)	✖(0.00)	✖(0.00)	–	–	–	–
	Addit.	✔(0.00)	✔(0.03)	✔(0.00)	✔(0.00)	✔(0.00)	–	–	–	–	✔(0.00)	✔(0.00)	✔(0.00)	✔(0.00)	✔(0.00)	–	–	–	–
	ILP-T	✔(0.00)	✔(0.00)	✔(0.00)	✔(0.00)	✔(0.00)	–	–	–	–	✖(0.00)	✖(0.00)	✖(0.00)	✖(0.00)	✖(0.00)	–	–	–	–
	ILP-A	✔(0.00)	✔(0.00)	✔(0.00)	✔(0.00)	✔(0.00)	–	–	–	–	✔(0.00)	✔(0.00)	✔(0.00)	✔(0.00)	✔(0.00)	–	–	–	–
metrics-core	Total	○(0.20)	○(0.20)	○(0.20)	○(0.16)	○(0.46)	✔(0.00)	–	–	–	✖(0.00)	✖(0.00)	✖(0.00)	✖(0.00)	✖(0.00)	✖(0.00)	–	–	–
	Addit.	✖(0.00)	✖(0.00)	✖(0.00)	✖(0.00)	✖(0.00)	○(0.44)	–	–	–	○(0.35)	○(0.26)	○(0.24)	○(0.35)	○(0.15)	○(0.08)	–	–	–
	ILP-T	✔(0.00)	✔(0.00)	✔(0.00)	✔(0.00)	✔(0.00)	✔(0.00)	–	–	–	✔(0.00)	✔(0.00)	✔(0.00)	✔(0.00)	✔(0.00)	✔(0.00)	–	–	–
	ILP-A	✔(0.00)	✔(0.00)	✔(0.00)	✔(0.00)	✔(0.00)	✔(0.00)	–	–	–	✔(0.00)	✔(0.00)	✔(0.00)	✔(0.00)	✔(0.00)	✔(0.00)	–	–	–
assertj-core	Total	✖(0.00)	○(0.49)	○(0.43)	○(0.13)	✔(0.03)	✔(0.00)	✔(0.00)	✔(0.00)	–	✔(0.00)	✔(0.00)	✔(0.00)	✔(0.00)	✔(0.00)	✔(0.00)	✔(0.00)	✔(0.00)	–
	Addit.	○(0.52)	✔(0.00)	✔(0.00)	✔(0.00)	✔(0.00)	✔(0.00)	✔(0.00)	✔(0.00)	–	✔(0.00)	✔(0.00)	✔(0.00)	✔(0.00)	✔(0.00)	✔(0.00)	✔(0.00)	✔(0.00)	–
	ILP-T	✖(0.00)	○(0.83)	○(0.14)	✔(0.02)	✔(0.00)	✔(0.00)	✔(0.00)	✔(0.00)	–	✔(0.00)	✔(0.00)	✔(0.00)	✔(0.00)	✔(0.00)	✔(0.00)	✔(0.00)	✔(0.00)	–
	ILP-A	✔(0.00)	✔(0.00)	✔(0.00)	✔(0.00)	✔(0.00)	✔(0.00)	✔(0.00)	✔(0.00)	–	✔(0.00)	✔(0.00)	✔(0.00)	✔(0.00)	✔(0.00)	✔(0.00)	✔(0.00)	✔(0.00)	–

Table 5: Wilcoxon tests between each studied technique and random technique based on statement coverage for time-aware prioritization under the 50% time constraints (p values included in brackets)

excluding test additions), test prioritization techniques perform stably even there are massive changes (i.e., using coverage data of a very early version to prioritize tests for the current version). This finding provides implications for both researchers and developers, i.e., they need to consider the influence of test addition when proposing new test prioritization techniques or applying test prioritization techniques in practice.

Traditional prioritization vs. time-aware prioritization. Our experimental results demonstrate that the impacts of software evolution (both source code evolution,and test additions) are similar for both traditional and time-aware test prioritization.

Prioritization using various coverage criteria. Our experimental results on different coverage criteria demonstrate that the impacts of software evolution are similar for various coverage criteria (detailed data available online.)

Prioritization using various revision granularities. Different revision granularities (i.e., the distance between the version for coverage collection and the version for test prioritization) can greatly impact test prioritization results. For instance, using the finer revision granularity usually produces more effective test prioritization results. The reason is that smaller number of tests may be added for finer revision granularity.

Prioritization techniques and random prioritization. Our experimental results show that random prioritization can be competitive when software evolution involves test additions. However, state-of-the-art test prioritization techniques can always outperform random prioritization when only considering source code changes (i.e., excluding new tests). Among various test prioritization techniques, our study demonstrates that the additional and search-based test prioritization can be the most effective for traditional test prioritization, while the ILP-additional technique is the most effective technique for time-aware test prioritization.

4. RELATED WORK

In the literature, a considerable amount of research focuses on test prioritization, and recently Yoo and Harman [37] conducted a survey on regression test minimization [23, 38–40], selection [6,24,41], and prioritization [3,7,42–46], which provides a broad view of the state-of-art on test prioritization.

Prioritization Strategies. The research in this category mainly focuses on various strategies or algorithms used in test prioritization. Among various prioritization strategies, the traditional total and additional strategies [3, 47] are the most widely used. Based on the basic mechanisms of the total and additional techniques, Zhang et al. [9] presented unified test prioritization to bridge their gap based on the probability that a test case can detect faults and generated a series of prioritization strategies between them. Li et al. [8] transferred the test prioritization problem into a searching problem and presented two searching algorithms (i.e., hill-climbing algorithm and genetic programming algorithm) for test prioritization. Motivated by random test prioritization, Jiang et al. [7] presented adaptive random test prioritization, which tends to select a test case that is the farthest from the already selected tests. Tonella et al. [48] presented a machine learning based technique, which prioritizes tests based on user knowledge. Yoo et al. [45] proposed a cluster-based test prioritization technique, which clusters tests based on their dynamic behavior. Recently, Nguyen et al. [49] and Saha et al. [50] presented to prioritize tests based on information retrieval. In this paper, we do not present a novel test prioritization strategy, but aim to investigate the effectiveness of typical test prioritization techniques through an empirical study.

Coverage Criteria. The research in this category mainly focuses on various coverage criteria used in test prioritization. Besides the most widely used coverage criteria (e.g., statement and method coverage [3], block coverage [51], modified condition and decision coverage [52], and statically estimated method coverage [53, 54]), Elbaum et al. [46] proposed fault-exposing-potential and fault index coverage to guide prioritization. Mei et al. [55] proposed a family of dataflow testing criteria, and three levels of coverage criteria, i.e., workflow, XPath, and WSDL for service-oriented business application [56]. Xu and Ding [57] proposed transition coverage and roundtrip coverage to prioritize aspect tests. Thomas et al. [58] proposed to build topics using the linguistic data in tests so as to guide test prioritization. In this paper, we do not present any new coverage criteria, but investigate the effectiveness of test prioritization based on widely-used coverage criteria (i.e., statement coverage, method coverage, and branch coverage).

Constraints. The research in this category mainly focuses on test prioritization with various constraints, e.g., test cost and fault severity [59, 60]. That is, researchers investigated how to prioritize tests under these constraints. In particular, Hou et al. [61] proposed a test prioritization technique with quota constraints for service-centric systems. Kim and Porter [62] proposed a history-based test prioritization technique with time and resource constraints. To address the time constraint, Walcott et al. [19] proposed a genetic algorithm, Zhang et al. [16] proposed an integer linear programming based technique, and Do et al. [63] presented an empirical study investigating the effect of time constraints. In this paper, we consider test prioritization without any constraint as well as test prioritization with only time constraints in the empirical study.

Measurements. The research in this category mainly focuses on how to measure the effectiveness of test prioritization techniques. Until now, the average percentage faults detected (APFD) [3] is the mostly used metric for evaluating test prioritization. Based on APFD, Elbaum et al. [59, 64] proposed the weighted APFD by assigning higher weights to tests with lower cost and/or with capability on detecting faults with higher severity. Walcott et al. [19] extended APFD by using a time penalty, and Do and Rothermel [65, 66] presented comprehensive economic models to accurately assess tradeoffs between techniques in industrial testing environments. In this paper, we use APFD as a metric to measure the test prioritization results, and will consider other measurements in the future.

Empirical Studies. The research in this category mainly focuses on investigating test prioritization by considering the influence of its internal or external factors through empirical studies. In particular, Rothermel et al. [3] investigated the influence of coverage criterion (e.g.,statement coverage, branch coverage, and probability of exposing faults). Do et al. [63] investigated the influence of time constraints and found that time constraints affected test prioritization techniques differently. Rothermel et al. [67] investigated the influence of fault types (i.e., real faults, mutation faults, and seeded faults). Elbaum et al. [46] investigated the influence of specific versions, coverage criteria (i.e., statement coverage and function coverage), and predictor of fault-proneness. Similar to these work, this paper also presents an empirical study on test prioritization, but it focuses on the influence of the important threat resulting from real software evolution, which has never been investigated before.

5. CONCLUSION

Although test prioritization has been intensively investigated, little effort has focused on the investigation of its threats to validity. In this paper, we pointed out an important threat resulting from real-world software evolution, including both source code changes and test additions, which are usually ignored by the existing work on test prioritization. Without considering this threat, it is unclear whether the existing conclusions drawn by the existing work are still valid for real-world software evolution. Therefore, we conducted the first empirical study to investigate the influence of this threat in test prioritization and got the following main findings: (1) both the traditional and time-aware test prioritization techniques studied in this paper become much less effective in software evolution involving test additions; (2) both traditional and time-aware test prioritization techniques are stable and effective in case of only source code changes (i.e., without test additions).

6. ACKNOWLEDGEMENTS

We thank August Shi from the University of Illinois for sharing the experimental subjects with us. This research was sponsored by faculty startup funds from the University of Texas at Dallas, the National Basic Research Program of China (973) under Grant No. 2014CB347701, the National Natural Science Foundation of China under Grant No.61421091, 91318301, 61225007, and 61522201.

7. REFERENCES

[1] C. Kaner, "Improving the maintainability of automated test suites," in *Proc. QW*, 1997.

[2] P. K. Chittimalli and M. J. Harrold, "Recomputing coverage information to assist regression testing," *IEEE TSE*, vol. 35, no. 4, pp. 452–469, 2009.

[3] G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold, "Test case prioritization: an empirical study," in *ICSM*, pp. 179–188, 1999.

[4] W. E. Wong, J. R. Horgan, S. London, and H. Agrawal, "A study of effective regression testing in practice," in *Proc. ISSRE*, pp. 264–274, 1997.

[5] M. J. Harrold, "Testing evolving software," *JSS*, vol. 47, no. 2–3, pp. 173–181, 1999.

[6] G. Rothermel and M. Harrold, "A safe, efficient regression test selection technique," *TOSEM*, vol. 6, no. 2, pp. 173–210, 1997.

[7] B. Jiang, Z. Zhang, W. K. Chan, and T. H. Tse, "Adaptive random test case prioritization," in *ASE*, pp. 257–266, 2009.

[8] Z. Li, M. Harman, and R. Hierons, "Search algorithms for regression test case prioritisation," *TSE*, vol. 33, no. 4, pp. 225–237, 2007.

[9] L. Zhang, D. Hao, L. Zhang, G. Rothermel, and H. Mei, "Bridging the gap between the total and additional test-case prioritization strategies," in *ICSE*, pp. 192–201, 2013.

[10] D. Hao, L. Zhang, L. Zhang, G. Rothermel, and H. Mei, "A unified test case prioritization approach," *TOSEM*, vol. 10, pp. 1–31, 2014.

[11] S. e Zehra Haidry and T. Miller, "Using dependency structures for prioritization of functional test suites," *TSE*, vol. 39, no. 2, pp. 258–275, 2013.

[12] H. Mei, D. Hao, L. Zhang, L. Zhang, J. Zhou, and G. Rothermel, "A static approach to prioritizing junit test cases," *TSE*, vol. 38, no. 6, pp. 1258–1275, 2012.

[13] C. Zhang, A. Groce, and M. A. Alipour, "Using test case reduction and prioritization to improve symbolic execution," in *Proc. ISSTA*, pp. 160–170, 2014.

[14] L. Zhang, D. Marinov, and S. Khurshid, "Faster mutation testing inspired by test prioritization and reduction," in *Proc. ISSTA*, pp. 235–245, 2013.

[15] H. Do, S. Mirarab, L. Tahvildari, and G. Rothermel, "The effects of time constraints on test case prioritization: A series of controlled experiments," *TSE*, vol. 36, no. 5, pp. 593–617, 2010.

[16] L. Zhang, S.-S. Hou, C. Guo, T. Xie, and H. Mei, "Time-aware test-case prioritization using integer linear programming," in *ISSTA*, pp. 213–224, 2009.

[17] M. Harman, S. A. Mansouri, and Y. Zhang, "Search-based software engineering: Trends, techniques and applications," *CSUR*, vol. 45, no. 1, p. 11, 2012.

[18] M. Harman and B. F. Jones, "Search based software engineering," *IST*, vol. 43, no. 14, pp. 833–839, 2001.

[19] K. R. Walcott, M. L. Soffa, G. M. Kapfhammer, and R. S. Roos, "Time aware test suite prioritization," in *ISSTA*, pp. 1–11, 2006.

[20] S. Alspaugh, K. R. Walcott, M. Belanich, G. M. Kapfhammer, and M. L. Soffa, "Efficient time-aware prioritization with knapsack solvers," in *Workshop on Empirical Assessment of Software Engineering Languages and Technologies*, pp. 17–31, 2007.

[21] L. Zhang, M. Kim, and S. Khurshid, "Localizing failure-inducing program edits based on spectrum information," in *ICSM*, pp. 23–32, 2011.

[22] L. Zhang, M. Kim, and S. Khurshid, "Faulttracer: a change impact and regression fault analysis tool for evolving java programs," in *FSE*, p. 40, 2012.

[23] A. Shi, A. Gyori, M. Gligoric, A. Zaytsev, and D. Marinov, "Balancing trade-offs in test-suite reduction," in *Proc. FSE*, pp. 246–256, 2014.

[24] A. Shi, T. Yung, A. Gyori, and D. Marinov, "Comparing and combining test-suite reduction and regression test selection," in *FSE*, pp. 237–247, 2015.

[25] A. Gyori, A. Shi, F. Hariri, and D. Marinov, "Reliable testing: Detecting state-polluting tests to prevent test dependency," in *ISSTA*, pp. 223–233, 2015.

[26] J. H. Andrews, L. C. Briand, and Y. Labiche, "Is mutation an appropriate tool for testing experiments?," in *ICSE*, pp. 402–411, 2005.

[27] H. Do and G. Rothermel, "A controlled experiment assessing test case prioritization techniques via mutation faults," in *ICSM*, pp. 411–420, 2005.

[28] R. Just, D. Jalali, L. Inozemtseva, M. D. Ernst, R. Holmes, and G. Fraser, "Are mutants a valid substitute for real faults in software testing," in *Proc. FSE*, pp. 654–665, 2014.

[29] H. Do and G. Rothermel, "On the use of mutation faults in empirical assessments of test case prioritization techniques," *TSE*, vol. 32, no. 9, pp. 733–752, 2006.

[30] A. Arcuri and L. Briand, "A practical guide for using statistical tests to assess randomized algorithms in software engineering," in *ICSE*, pp. 1–10, 2011.

[31] R. K. Saha, L. Zhang, S. Khurshid, and D. E. Perry, "An information retrieval approach for regression test prioritization based on program changes," 2015. To appear.

[32] R. Just, D. Jalali, and M. D. Ernst, "Defects4j: A database of existing faults to enable controlled testing studies for java programs," in *ISSTA*, pp. 437–440, 2014.

[33] T. H. Wonnacott and R. J. Wonnacott, *Introductory statistics*, vol. 19690. Wiley New York, 1972.

[34] L. Zhang, S.-S. Hou, J.-J. Hu, T. Xie, and H. Mei, "Is operator-based mutant selection superior to random mutant selection?," in *ICSE*, pp. 435–444, 2010.

[35] Y. Qi, X. Mao, Y. Lei, Z. Dai, and C. Wang, "The strength of random search on automated program repair," in *ICSE*, pp. 254–265, 2014.

[36] F. Wilcoxon, "Individual comparisons by ranking methods," in *Breakthroughs in Statistics*, pp. 196–202, 1992.

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

[38] J. Black, E. Melachrinoudis, and D. Kaeli, "Bi-criteria models for all-uses test suite reduction," in *ICSE*, pp. 106–115, 2004.

[39] G. Rothermel, M. J. Harrold, J. Von Ronne, and C. Hong, "Empirical studies of test-suite reduction," *STVR*, vol. 12, no. 4, pp. 219–249, 2002.

[40] L. Zhang, D. Marinov, L. Zhang, and S. Khurshid, "An empirical study of junit test-suite reduction," in *ISSRE*, pp. 170–179, 2011.

[41] T. Ball, "On the limit of control flow analysis for regression test selection," in *ISSTA*, pp. 134–142, 1998.

[42] J. Chen, Y. Bai, D. Hao, Y. Xiong, H. Zhang, L. Zhang, and B. Xie, "A text-vector based approach to test case prioritization," in *ICST*, 2016. To appear.

[43] Y. Lou, D. Hao, and L. Zhang, "Mutation-based test-case prioritization in software evolution," in *ISSRE*, pp. 46–57, 2015.

[44] D. Hao, L. Zhang, L. Zang, Y. Wang, X. Wu, and T. Xie, "To be optimal or not in test-case prioritization," *TSE*, 2015. To appear.

[45] S. Yoo, M. Harman, P. Tonella, and A. Susi, "Clustering test cases to achieve effective and scalable prioritisation incorporating expert knowledge," in *Proc. ISSTA*, pp. 201–212, 2009.

[46] S. Elbaum, A. Malishevsky, and G. Rothermel, "Test case prioritization: A family of empirical studies," *TSE*, vol. 28, no. 2, pp. 159–182, 2002.

[47] S. Elbaum, A. Malishevsky, and G. Rothermel, "Prioritizing test cases for regression testing," in *ISSTA*, pp. 102–112, 2000.

[48] P. Tonella, P. Avesani, and A. Susi, "Using the case-based ranking methodology for test case prioritization," in *Proc. ICSM*, pp. 123–133, 2006.

[49] C. D. Nguyen, A. Marchetto, and P. Tonella, "Test case prioritization for audit testing of evolving web services using information retrieval techniques," in *Proc. ICWS*, pp. 636–643, 2011.

[50] R. K. Saha, L. Zhang, S. Khurshid, and D. E. Perry, "An information retrieval approach for regression test prioritization based on program changes," in *Proc. ICSE*, p. to appear, 2015.

[51] H. Do, G. Rothermel, and A. Kinneer, "Empirical studies of test case prioritization in a JUnit testing environment," in *ISSRE*, pp. 113–124, 2004.

[52] J. A. Jones and M. J. Harrold, "Test-suite reduction and prioritization for modified condition/decision coverage," in *ICSM*, pp. 92–101, 2001.

[53] L. Zhang, J. Zhou, D. Hao, L. Zhang, and H. Mei, "Prioritizing JUnit test cases in absence of coverage information," in *ICSM*, pp. 19–28, 2009.

[54] H. Mei, D. Hao, L. Zhang, L. Zhang, J. Zhou, and G. Rothermel, "A static approach to prioritizing junit test cases," *TSE*, vol. 38, no. 6, pp. 1258–1275, 2012.

[55] L. Mei, W. K. Chan, and T. H. Tse, "Data flow testing of service-oriented workflow applications," in *ICSE*, pp. 371–380, 2008.

[56] L. Mei, Z. Zhang, W. K. Chan, and T. H. Tse, "Test case prioritization for regression testing of service-oriented business applications," in *WWW*, pp. 901–910, 2009.

[57] D. Xu and J. Ding, "Prioritizing state-based aspect tests," in *Proc. ICST*, pp. 265–274, 2010.

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

[59] S. Elbaum, A. Malishevsky, and G. Rothermel, "Incorporating varying test costs and fault severities into test case prioritization," in *ICSE*, pp. 329–338, 2001.

[60] H. Park, H. Ryu, and J. Baik, "Historical value-based approach for cost-cognizant test case prioritization to improve the effectiveness of regression testing," in *SSIRI*, pp. 39–46, 2008.

[61] S.-S. Hou, L. Zhang, T. Xie, and J. Sun, "Quota-constrained test-case prioritization for regression testing of service-centric systems," in *ICSM*, pp. 257–266, 2008.

[62] J. M. Kim and A. Porter, "A history-based test prioritization technique for regression testing in resource constrained environments," in *ICSE*, pp. 119–129, 2002.

[63] H. Do, S. Mirarab, L. Tahvildari, and G. Rothermel, "An empirical study of the effect of time constraints on the cost-benefits of regression testing," in *FSE*, pp. 71–82, 2008.

[64] A. Malishevsky, J. R. Ruthru, G. Rothermel, and S. Elbaum, "Cost-cognizant test case prioritization," tech. rep., Department Computer Science and Engineering of University of Nebraska, 2006.

[65] H. Do and G. Rothermel, "Using sensitivity analysis to create simplified economic models for regression testing," in *Proc. ISSTA*, pp. 51–62, 2008.

[66] H. Do and G. Rothermel, "An empirical study of regression testing techniques incorporating context and lifecycle factors and improved cost-benefit models," in *Proc. FSE*, pp. 141–151, 2006.

[67] G. Rothermel, R. J. Untch, C. Chu, and M. J. Harrold, "Prioritizing test cases for regression testing," *TSE*, vol. 27, no. 10, pp. 929–948, 2001.