Adaptive Test-Case Prioritization Guided by Output Inspection

Dan Hao, Xu Zhao, Lu Zhang
Institute of Software, School of Electronics Engineering and Computer Science
Key Laboratory of High Confidence Software Technologies (Peking University), Ministry of Education
Peking University, Beijing, 100871, China
{haod, zhaoxu08, zhanglu}@sei.pku.edu.cn

Abstract—Test-case prioritization is to schedule the execution order of test cases so as to maximize some objective (e.g., revealing faults early). The existing test-case prioritization approaches separate the process of test-case prioritization and the process of test-case execution by presenting the execution order of all test cases before programmers start running test cases. As the execution information of the modified program is not available for the existing test-case prioritization approaches, these approaches mainly rely on only the execution information of the previous program before modification. To address this problem, we present an adaptive test-case prioritization approach, which determines the execution order of test cases simultaneously during the execution of test cases. In particular, the adaptive approach selects test cases based on their fault-detection capability, which is calculated based on the output of selected test cases. As soon as a test case is selected and runs, the fault-detection capability of each unselected test case is modified according to the output of the latest selected test case. To evaluate the effectiveness of the proposed adaptive approach, we conducted an experimental study on eight C programs and four Java programs. The experimental results show that the adaptive approach is usually significantly better than the total test-case prioritization approach and competitive to the additional test-case prioritization approach. Moreover, the adaptive approach is better than the traditional approach on some subjects (e.g., replace and schedule).

Keywords—software testing; regression test; test-case prioritization; adaptive approach

I. INTRODUCTION

Software testing plays an important role in assuring the quality of software systems [1]. However, it is estimated that software testing consumes more than half of the cost in software development and maintenance [2], [3]. Therefore, many researchers focus on how to automate software testing and thus improve the efficiency of software testing.

Test-case prioritization [4], [5], [6], [7], [8], [9], [10] is firstly proposed in regression testing [3], [11], which aims to test the changed software during software evolution by reusing the test cases of its previous version (before modification). To facilitate regression testing, test-case prioritization schedules the execution order of test cases so as to maximize some objective (e.g., revealing faults early). Given a previous program $P$ and its modified version $P'$, the process of regression testing includes constructing a test suite for $P'$ based on the existing test suite $T$ of program $P$ and running $P'$ with the constructed test suite. Test-case prioritization is to determine the execution order of test cases within $T$ so that the faults in $P'$ will be revealed early during the execution of $T$ on $P'$.

As test-case prioritization is proposed in regression testing, most existing test-case prioritization approaches [12], [13], [14], [15] use the execution information of the previous program to estimate the fault-detection capability of test cases on the modified program and prioritize test cases on their calculated fault-detection capability. In particular, during the execution of the previous program $P$, its execution information (denoted as $\text{Execution}(P)$, which usually refers to some structural coverage like statement coverage or method coverage) is recorded and is later used to calculate the fault-detection capability of test cases in $T$ on the modified program $P'$. However, as $P$ and $P'$ are two versions developed during software evolution, they may have non-trivial difference and thus cause the inconsistency between $\text{Execution}(P)$ and $\text{Execution}(P')$ for any specified $T$. As these approaches rely on only previous execution information, the prioritized test suite of these approaches can be applied to any modified version of the previous program. These approaches are general to all the modified versions, and thus their prioritized test suites may be not effective in revealing faults early for some specified modified version $P'$ of the previous program $P$ due to the inconsistency between $\text{Execution}(P)$ and $\text{Execution}(P')$. To address this problem, some version-specific test-case prioritization approaches [16], [17] have been proposed by taking into account the information (e.g., the code) of the specified modified version $P'$. Generally speaking, these approaches analyze the difference between $P$ and $P'$, and then prioritize test cases based on the execution coverage information on their difference (i.e., the modified code). However, as the existing test-case prioritization approaches schedule the execution order of test cases before programmers run test cases, the execution information of the modified version $P'$ is not available for test-case prioritization.

The existing test-case prioritization approaches, including the general approaches and the version-specific approaches, view test-case prioritization and test-case execution as two separate processes. With the existing test-case prioritization...
approaches, programmers cannot start running the prioritized test suite until the execution order of test cases is determined. In this paper, we present an adaptive test-case prioritization approach, which combines the test-case prioritization process and the test-case execution process. In particular, the adaptive test-case prioritization approach does not present the execution order of all test cases before programmers run test cases, but tells programmers which test case to execute as soon as the latest test case finishes running. Consequently, the execution information (i.e., the output\(^1\) of test cases) of the modified version \(P'\) is available and can be used to improve test-case prioritization.

In particular, the adaptive test-case prioritization approach calculates the fault-detection capability of each test case based on the faulty potential (which measures to what extent a statement is likely to contain faults) of its executed statements. During regression testing, as soon as a selected test case finishes running, the adaptive approach modifies the faulty potential of all the statements executed by this test case based on its output, and then modifies the fault-detection capability of all unselected test cases. The adaptive approach selects a test case with the largest fault-detection capability and programmers run the selected test case. The preceding process repeats until all the test cases are selected and run. Generally speaking, the adaptive approach schedules test cases and executes test cases simultaneously. This is also the main difference between the adaptive approach and existing test-case prioritization approaches.

To evaluate the effectiveness of the proposed adaptive approach, we conducted an experimental study on eight C objects and four Java objects. The experimental results show that the adaptive approach is usually significantly better than the total test-case prioritization approach. The adaptive approach is statistically comparable to the additional test-case prioritization approach. Furthermore, the adaptive approach sometimes is better than the additional approach.

The contributions of this paper are summarized as follows.

- This paper presents an adaptive test-case prioritization approach. To our knowledge, this is the first work in test-case prioritization that combines the process of test-case prioritization and test-case execution so that more precise execution information can be used to improve test-case prioritization.
- To evaluate the effectiveness of the proposed approach, we performed an experimental study by comparing with the total and additional test-case prioritization approaches.

\(^1\)In this paper, the output of a test case refers to whether the test case is executed as expected. In particular, if the actual output of a test case is as expected, the output of a test case is called passed; otherwise, it is called failed.

II. RELATED WORK

As the work presented in this paper targets at test-case prioritization, we will present the most related work in Section II-A. Our work is also related to test-suite reduction and thus we will briefly discuss the work in test-suite reduction in Section II-B.

A. Test-Case Prioritization

Depending on whether the generated prioritized test suite is general for all the modified versions of \(P\) or its specific modified version \(P'\), Rothermel et al. [12] divided the existing test-case prioritization approaches into two categories: general test-case prioritization approaches and version-specific test-case prioritization approaches. We group the existing test-case prioritization approaches into these two categories and briefly review the approaches of each category.

1) General Test-Case Prioritization: Given the previous program \(P\) and test suite \(T\), general test-case prioritization [12] schedules the test cases of \(T\) only based on the execution information of \(T\) on program \(P\). The generated prioritized test suite can be used for any modified version of \(P\). As general test-case prioritization relies on only the previous program \(P\), the approach is not specific for its specific modified version, but general for all its modified versions.

Most general test-case prioritization approaches [12], [13], [14], [15] schedule the order of test cases based on some structural coverage (e.g., statement coverage and branch coverage) of test cases on the previous program. To evaluate the effectiveness of test-case prioritization on various structural coverage, Rothermel et al. [12], [15] conducted an empirical study comparing several approaches on statement coverage, branch coverage, and approximated fault-exposing-potential. Furthermore, Elbaum et al. [13] conducted several a series of empirical studies to investigate other research questions such as the effectiveness of fine granularity and coarse granularity test-case prioritization approaches. Later, Jones and Harrold [14] proposed modified condition/decision coverage (abbreviated as MC/DC) based test-case prioritization approach. MC/DC is a stricter form of branch coverage [14] and thus the test-case prioritization approach based on MC/DC is expected and has been evaluated to be effective. As the preceding approaches ignored constraints (e.g., time and resource constraints) in real software development, many test-case prioritization approaches [10], [18], [19] have been proposed by considering the time limit.

Recently, Bo et al. [20] proposed an adaptive random test-case prioritization, which selects test cases by calculating the distance between selected test cases and remaining unselected test cases based on their structural coverage. Their approach is close to the additional approach, and sometimes is statistically comparable to the additional approach. As
the total and additional approaches are two typical complementary test-case prioritization approaches, Zhang et al. [21] presented models to unify the total approach and additional approach and then generates a spectrum of test-case prioritization approaches. Our approach is similar to their approach, because both of the two approaches modify the weights of unselected test cases during test-case prioritization based on the latest selected test case. However, their approach modifies the weight based on the coverage of the latest selected test case on the previous program, whereas our approach modifies the weight based on the output of the latest selected test case on the current program.

2) Version-Specific Test-Case Prioritization: Version-specific test-case prioritization [12] schedules the order of test cases within $\mathcal{T}$ specially for some modified program $\mathcal{P}'$, not for any modified version of $\mathcal{P}$. Therefore, version-specific test-case prioritization approaches usually reply on some information of the modified program $\mathcal{P}'$ besides the previous execution information. Our proposed adaptive test-case prioritization approach is a version-specific approach, not a general approach. Based on the type of information used in test-case prioritization, existing works on version-specific test-case prioritization can be classified into code-based approaches and model-based approaches.

The code-based approaches [16], [17] usually use the source code change between the modified program $\mathcal{P}'$ and its previous program $\mathcal{P}$ besides the previous execution information to schedule the order of test cases. In particular, Wong et al. [17] proposed a modification-based approach, which analyzes the source code change and schedules test cases based on the “increasing cost per additional coverage” [17]. Mei et al. [7], [22] proposed a static approach to prioritizing JUnit test cases based on the callees of each JUnit test case, whose effectiveness is close to dynamic approaches.

Model-based approaches [23], [24], [25] schedule the order of test cases based on the modification on the system model and its execution information. Korel et al. [24] proposed the first model-based test-case prioritization approach, which schedules the order of test cases based on the collected execution information of the modified model together with the previous system model and the modified system model. As the changes are usually made in source code, not the system model, Korel et al. [23] later proposed several model-based test-case prioritization approaches, which identify the modification on the system model based on the modified source code.

For a specific modified program, general test-case prioritization approaches can also be helpful, but version-specific test-case prioritization approaches are expected to be better. Our proposed adaptive test-case prioritization approach is a version-specific test-case prioritization approach because the adaptive approach uses the execution information (i.e., the passed or failed output) of the modified program. Different from the existing code-based or model-based approaches, the adaptive approach replies on the output of a test case, which is naturally generated during regression testing without any extra cost.

B. Test-Suite Reduction

In regression testing, as it is time-consuming to run the aggregated test cases, many test-suite reduction approaches have been proposed whose aim is to reduce the number of test cases used in regression testing so as to reduce the cost of regression testing. Besides research on coverage criteria used in test-suite reduction [14], most research on test-suite reduction focuses on test-suite reduction algorithms, including greedy algorithms [26], [27], genetic algorithms [28], [29], and integer linear programming based algorithms [30], [31]. Furthermore, some empirical studies [32], [33] have been conducted to investigate the impact of various factors on the effectiveness of test-suite reduction. Moreover, some researchers have applied test-suite reduction to facilitate other activities in software testing and debugging (e.g., fault localization [34], [35], [36]).

III. ADAPTIVE TEST-CASE PRIORITIZATION

In this section, we first present the adaptive process of the proposed test-case prioritization approach by showing its basic difference with the existing test-case prioritization approaches in Section III-A, and then give the details of the adaptive approach in Sections III-B and III-C.

For ease of presentation, we present the adaptive test-case prioritization approach in terms of statement coverage, which can also be implemented on other coverage criteria (e.g., method coverage, branch coverage).

A. Adaptive Process

Generally speaking, most existing test-case prioritization approaches [12], [13], [14], [15] schedule the execution order of test cases based on the execution information of the previous program, which occurs before running test cases on the current program. In the application of the existing test-case prioritization approaches, test-case prioritization and test-case execution are two separated phrases (separated by the dotted line in Figure 1(a)), and test-case prioritization occurs before test-case execution. Moreover, the existing test-case prioritization approaches give the complete execution order of test cases all at once. Therefore, although the execution information of the previous program may have much difference from that of the current program, the existing test-case prioritization approaches mainly rely on the former since the latter is not available.

Different from the existing test-case prioritization approaches, in this paper, we propose an adaptive test-case prioritization approach, which schedules the execution order of test cases during the execution of test cases. In particular, the process of adaptive test-case prioritization and test-case
execution is shown by Figure 1(b), which mainly consists of the following steps.

1) Calculate the fault-detection capability of unselected test cases based on the execution information of the previous program and the output of the latest selected test case, and then select any test case \( t \) with the largest fault-detection capability.

2) Run test case \( t \) on the modified program, recording whether its output is passed or failed.

3) Repeat the preceding two steps until all the test cases within the given test suite have been prioritized and run.

The adaptive approach selects the first test case based on only the execution information of the previous program, whereas the execution order of the remaining test cases is determined based on the outputs of all selected test cases on the modified program as well as the execution information of the previous program. In Figure 1(b), the adaptive approach selects a test case to run immediately after programmers finish running a test case. That is, in adaptive test-case prioritization, test-case prioritization occurs simultaneously during the execution of test cases. Moreover, the execution order of test cases using the adaptive test-case prioritization approach is determined during the execution of test cases on the modified program, whereas the execution order of all test cases using the existing test-case prioritization approaches is available before programmers start running test cases on the modified program.

However, as the adaptive test-case prioritization approach occurs during the execution of test cases, the execution information of the modified program is available. As more precise information is taken as input during test-case prioritization, the adaptive approach is expected to be more reliable and effective in revealing faults in the modified program. Moreover, our proposed test-case prioritization approach is called the adaptive approach because it recalcualtes the fault-detection capability of unselected test cases and selects test cases based on its latest collected execution information.

### B. Overview of the Adaptive Approach

Algorithm 1 shows the algorithm of the adaptive test-case prioritization approach. Initially, the adaptive approach calculates the initial fault-detection capability (denoted as \( \text{Priority}(t) \)) of each test case \( t \) based on the statement coverage of test cases on the previous program and selects any test case \( t_s \) with the largest \( \text{Priority} \). Then test-case prioritization and execution occur simultaneously. In particular, programmers run the selected test case \( t_s \), recording whether its output is passed or failed. Then based on its output, the adaptive approach modifies the \( \text{Priority} \) of unselected test cases and selects the test case with the largest modified \( \text{Priority} \). The adaptive approach repeats the preceding process until all the test cases within \( T \) are selected and prioritized.

The adaptive test-case prioritization approach has two characteristics. First, its test-case prioritization process and test-case execution process occur simultaneously, different from the existing test-case prioritization approaches. Second, the adaptive approach recalculate the fault-detection capability of test cases as soon as programmers finish running a test case. Section III-A shows its first characteristic. To complete Algorithm 1, we will explain how to calculate or recalculate fault-detection capability of test case in the following section.

#### Algorithm 1 Algorithm of the Adaptive Approach

**Input:** Test suite \( T \)

**Output:** Prioritized test suite \( T' \)

**Declaration:** \( t_s \) represents the latest selected test case

**General Process:**

**Begin**

\[
\text{for each test case } t \text{ in } T \\
\quad \text{calculate initial } \text{Priority}(t) \\
\text{end for}
\]

\[
\text{select the test case (i.e., } t_s\text{) with the largest Priority in } T \\
\quad \text{add } t_s \text{ to } T' \\
\quad T \leftarrow T - \{t_s\} \\
\quad \text{run } t_s \\
\text{while } T \text{ is not empty do} \\
\quad \text{for each test case } t \text{ in } T \\
\quad \quad \text{modify Priority}(t) \text{ based on the output of } t_s \\
\quad \text{end for} \\
\quad \text{select the test case (i.e., } t_s\text{) with the largest Priority in } T \\
\quad \quad \text{add } t_s \text{ to } T' \\
\quad \quad T \leftarrow T - \{t_s\} \\
\quad \quad \text{run } t_s \\
\text{end while} \\
\text{return } T'
\]

**End**

#### C. Fault-Detection Capability

The adaptive approach determines the execution order of test cases based on their fault-detection capability, which
measures how likely each test case reveals faults. During the execution of a test case, some statements will be executed, faults are invoked and finally cause a test case fail. Therefore, whether a test case reveals faults can be measured based on the statements it executes. Based on this assumption, the adaptive approach defines the fault-detection capability of each test case \( t \) by the following equation.

\[
\text{Priority}(t) = \sum_{s \text{ is executed by } t} \text{Potential}(s) \quad (1)
\]

where \( \text{Potential}(s) \) represents how likely statement \( s \) contains faults that have been not revealed by the existing selected test cases. \( \text{Potential}(s) \) of any statement \( s \) is within the scope of \([0,1]\).

Initially, all the statements are likely to have contained new faults since no test cases have been selected and revealed any faults before. Thus, initially, \( \text{Potential} \) of any statement \( s \) is 1. As more test cases are selected and run, the probability of each statement on containing new faults tends to decrease no matter whether the selected test cases are passed or failed. However, the output of selected test case, whether it is passed or failed, may have different influence on the probability of statements on containing new faults. Therefore, the adaptive approach modifies the \( \text{Potential}(s) \) of any statement \( s \) based on the passed or failed output of selected test cases. In particular, as soon as any test case \( t' \) finishes running, the adaptive test-case prioritization approach modifies the \( \text{Potential} \) of each statement \( s \) based on the following equation and then updates the \( \text{Priority} \) of any unselected test case \( t \) based on Equation (1).

\[
\text{Potential}(s) = \left\{ \begin{array}{ll}
\text{Potential}'(s), & s \text{ is not executed by } t' \\
\text{Potential}(s) \times q, & s \text{ is executed by } t' \land \text{t' is passed} \\
\text{Potential}(s) \times p, & s \text{ is executed by } t' \land \text{t' is failed}
\end{array} \right.
\]

where \( \text{Potential}'(s) \) represents the probability of any statement \( s \) on containing new faults before running the test case \( t' \). \( p \) and \( q \) are two non-negative constants, whose values are between 0 and 1. The influence of passed/failed output on the \( \text{Potential}(s) \) of any statement \( s \) is measured by \( q/p \) in the preceding equation.

Moreover, when \( p=q=0 \), the adaptive approach becomes the additional statement-coverage based test-case prioritization approach, whereas when \( p=q=1 \), the adaptive approach becomes the total statement-coverage based test-case prioritization approach. That is, the total or additional statement-coverage based test-case prioritization approach can be viewed as an instance of the adaptive approach. The existing research on test-case prioritization has fully evaluated the effectiveness of the total approach and the additional approach. Although \( p \) and \( q \) in the preceding equation are two independent variables, to facilitate evaluation of the proposed adaptive approach, currently we assume \( p + q = 1 \) in this paper and evaluate the effectiveness of the adaptive approach by setting \( q=0, 0.2, 0.4, 0.6, 0.8, \) or 1, separately. However, as \( p \) and \( q \) are independent, we will evaluate the effectiveness of the adaptive approach by using various values of \( p \) and \( q \) without assuming that \( p + q = 1 \).

IV. EXPERIMENTAL STUDY

Our experimental study is designed to answer the following two research questions.

- **RQ1**: Is our proposed adaptive test-case prioritization approach more effective than the additional and total test-case prioritization approaches?
- **RQ2**: Among various values of variable \( q \), which makes our proposed adaptive approach generate more effective prioritization results?

A. Objects

The objects used in our experimental study are the seven programs of the siemens suite, space, ant, jmeter, jtopas, and xmlsec. These programs are widely used in the evaluation of software testing and analysis [36], [37], [38], and are available in the software-artifact infrastructure repository (abbreviated as SIR and accessible at http://sir.unl.edu/portal/index.html).

Siemens consists of seven small\(^2\) C programs, which are print_tokens, print_tokens2, replace, schedule, schedule2, tcas, and tot_info. Space is a middle-size C program, whose lines of code are 9564. Each of these programs has a collection of test cases. Specifically, although Siemens are small programs, each of its programs has suited with more than 1,000 test cases. Space has 13,585 test cases in its test pool. Moreover, each of these programs has several faulty versions, each of which contains one injected fault. As our experimental study is designed to evaluate test-case prioritization approaches in regression testing, for each program, we took its correct version as the previous program and constructed its multiple-fault version as the modified program. In particular, for each object, our experimental study isolated all faults from its single-fault versions and constructed its multiple-fault version by injecting randomly selected faults on the correct version. Note that the number of injected faults is also randomly determined. Following the preceding process, our experimental study constructed 10 multiple-fault versions for each object.

Ant, jmeter, jtopas, and xmlsec are Java programs, whose lines of code are from 1,894 to 80,444. In particular, ant has eight versions, jmeter has five versions, whereas both jtopas and xmlsec have three versions. As these versions are collected during software evolution, various versions have non-trivial differences and are taken as different objects by our experimental study. Consequently, our experimental study has 19 Java objects and 8 C objects. As each Java object has also been injected with several faults, our experimental study constructed 10 multiple-fault versions for each Java

\(^2\)The lines of code of these programs are from 173 to 565.
object following the same construction process of C objects. Each Java object has suited with many test cases, which are written in the JUnit framework.

**B. Compared Approaches**

During the scheduling and execution process of test cases, as soon as a test case finishes running, our proposed approach modifies the *Priority* of each test case based on variable \( q \) and selects test cases based on their modified *Priority*. Since variable \( q \) influences the effectiveness of test-case prioritization, our experimental study implemented the proposed adaptive approach with various \( q \). In particular, our experimental study chose \( q = 0, 0.2, 0.4, 0.6, 0.8, \) and 1.0.

Besides the proposed adaptive test-case prioritization approach, our experimental study also implemented the following test-case prioritization approaches in comparison.

- Total statement-coverage based test-case prioritization (abbreviated as the **total approach**) schedules test cases based on the descendental order of the statement coverage of test cases. The larger statement coverage a test case has, the earlier the test case is scheduled to be executed.
- Additional statement-coverage based test-case prioritization (abbreviated as the **additional approach**) schedules test cases based on the number of statements that are not covered by existing selected test case but are covered by this test case.
- Original test-case prioritization (abbreviated as the **original approach**) keeps the original order of test cases as they are provided.
- Reverse test-case prioritization (abbreviated as the **reverse approach**) reverses the original order of test cases.

**C. Measurement**

Our experimental study used the average percentage faults detected (usually abbreviated as APFD) to measure the effectiveness of test-case prioritization approaches. APFD is first proposed by Rothermel et al. [15] and is defined based on the following equation.

\[
APFD = 1 - \frac{(TF_1 + TF_2 + \ldots + TF_m)}{nm} + \frac{1}{2n},
\]

where \( n \) denotes the total number of test cases, \( m \) denotes the total number of detected faults, and \( TF_i (1 \leq i \leq m) \) denotes the smallest number of test cases programmers have to go through in sequence until fault \( i \) is exposed. APFD values range from 0 to 1. For any given test suite, its \( n \) and \( m \) are fixed so that higher APFD value implies that the average value of \( TF_i \) (variable \( i \) is from 1 to \( m \)) is lower and thus implies a higher fault-detection rate.

**D. Setup and Process**

For each object, our experimental study took its correct version as the previous program and used any one of its multiple-fault versions as the modified program. Since our experimental study totally has 27 objects\(^3\) and constructed 10 multiple-fault versions for each object, there are totally 27*10=270 pairs of previous (i.e., unmodified) and modified programs.

For each pair of unmodified and modified programs, based on the following process, our experimental study constructed 60 test suites to be prioritized. In particular, our experimental study repeated randomly selecting test cases from its test pool until the test suite contained the desired number (\( n \)) of test cases. Our experimental study chose \( n=10, 20, \) and 30. Our experimental study repeated the preceding process 20 times and thus constructed 3*20=60 test suites for each pair of unmodified and modified programs.

For each constructed test suite, our experimental study ran the corresponding unmodified program with its test suite recording its execution information (i.e., the statement coverage of each test case), and then applied the total approach, the additional approach, the original approach, as well as the reverse approach. Then our experimental study applied the execution information of each unmodified program to our proposed adaptive approach, and used the adaptive approach during the execution of the corresponding modified program.

Our experimental study has 270 pairs of unmodified and modified programs and constructed 60 test suites for each pair. After applying test-case prioritization, each compared approach has 270*60=16200 experimental results. To summarize these experimental results, for each pair of unmodified and modified programs, our experimental study averaged its APFD results for the specified \( n \), which is the number of test cases within the constructed test suite. Therefore, each pair of unmodified and modified programs, our experimental study has 3 average results. The results and analysis in the following section are based on the 270*3=810 average APFD results.

**E. Threats to Validity**

The threat to internal validity lies in the implementation of the experimental study. To reduce this threat, the authors of this paper reviewed and tested the code we wrote.

The threats to external validity are from the objects and their test suites used by our experimental study. To reduce this threat in the objects, our experimental study used 19 Java programs and 8 C programs, whose numbers of lines of code are from 0.1 KLOC to 80.4 KLOC. Moreover, these programs have been widely used in the evaluation of 3The 27 objects include print_tokens, print_tokens2, replace, schedule, schedule2, teas, tot_info, space, ant-v1, ant-v2, ant-v3, ... , ant-v8, jmeter-v1, jmeter-v2, ..., jmeter-v5, jtopas-v1, jtopas-v2, jtopas-v3, xmlsec-v1, xmlsec-v2, and xmlsec-v3.
software testing and analysis. The second external threat lies in the faults in these objects. Although the multiple-fault programs are constructed by injecting faults, many of these faults are collected during the development of corresponding program. Since the constructed multiple-fault programs may be not represent real fault programs, we plan to conduct the experimental study on programs with real faults in practice. The third external threat is the test suite to be prioritized. The test suites used in our experimental study are constructed based on test cases in the test pool. That is, the constructed test suites are not real test suite. To reduce this threat, our experimental study constructed 60 test suites by randomly selecting test cases from the existing test pool for each object.

The threat to construct validity lies in whether our experimental results are measured in a correct way. To reduce this threat, our experimental study used APFD [15] to measure the effectiveness of a prioritized test suite since APFD has been widely used in the evaluation of test-case prioritization.

F. Results and Analysis

Figure 2 shows the box plots\(^4\) across all the C programs (including space and the seven programs of Siemens). Table I presents the abbreviation names of the compared test-case prioritization approaches. In general, our adaptive approach, the total and additional approaches outperform the original and reverse approaches. Moreover, the medians of the former approaches are larger than 0.25, whereas the medians of the latter two approaches are only 0.20.

Figure 3 shows the box plots across all the Java programs (including ant, jmeter, jtopas, and xmlsec). We also observe that the adaptive approach outperforms the original approach. The adaptive approach outperforms the reverse approach when \(q=0, q=0.2, q=0.4, q=0.6, \) or \(q=0.8\), but the adaptive approach is less effective when \(q=1\). The reverse approach achieves better results in Java programs (than C programs) because the original test cases in the test pools of some Java objects may have been ordered based on their fault revealing. Although our experimental study constructed test suites by randomly selecting test cases from the test pool, it is possible that the test cases in some constructed test suite still keep their order as they are in the test pool.

1) Effectiveness of the Adaptive Approach: Table II presents the average APFD results for each object. For each object, its best average APFD result of various test-case prioritization approaches is addressed by the bold font. The last row of this table summarizes the number of objects that each compared test-case prioritization approach achieves the best average results. According to Table II, the adaptive approach achieves the best average APFD results on most objects since the proposed adaptive approach outperforms

\[^4\] A boxplot is a statistical graph representing data distribution, whose width spans the central 50 percent of the data and whose ends mark the upper and lower quartiles.

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Names of the Approach</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q1</td>
<td>the adaptive approach with (q=0)</td>
</tr>
<tr>
<td>Q2</td>
<td>the adaptive approach with (q=0.2)</td>
</tr>
<tr>
<td>Q3</td>
<td>the adaptive approach with (q=0.4)</td>
</tr>
<tr>
<td>Q4</td>
<td>the adaptive approach with (q=0.6)</td>
</tr>
<tr>
<td>Q5</td>
<td>the adaptive approach with (q=0.8)</td>
</tr>
<tr>
<td>Q6</td>
<td>the adaptive approach with (q=1)</td>
</tr>
<tr>
<td>T</td>
<td>the total approach</td>
</tr>
<tr>
<td>A</td>
<td>the additional approach</td>
</tr>
<tr>
<td>O</td>
<td>the original approach</td>
</tr>
<tr>
<td>R</td>
<td>the reverse approach</td>
</tr>
</tbody>
</table>

Figure 2. APFD Distributions for C Programs

Figure 3. APFD Distributions for Java Programs
the other two approaches on 10 objects, whereas both the additional approach and the total approach outperform the other approaches on only 2 Java objects. That is, the adaptive approach is competitive compared with the total and additional approaches.

To learn whether there are significant differences between the compared approaches, we performed the Wilcoxon signed ranked test\(^5\) on the experimental results. The Wilcoxon signed ranked test [39] is a nonparametric test on comparing two related or paired samples to see whether there are significant differences between the two groups.

Table III summarizes results of the Wilcoxon signed ranked test on comparing the proposed adaptive approach with the total approach (denoted as T) and the additional approach (denoted as A). For the comparison between approach M and approach N (denoted as M vs. N), “+” represents the number of times that M wins N, “-” represents the number of times that N wins M, whereas “R” represents the result of the Wilcoxon signed ranked test. If there is no significantly difference between M and N, the corresponding cell is marked as “E”, which represents the test results cannot reject the hypothesis that there is no significant difference between the compared two approaches. If the test results reject such hypothesis, there is significant difference between the compared two approaches and the corresponding cell is denoted by the abbreviated name of the approach, which is significantly better.

Most of the results in Table III are “E” or “Qi” (where i=1, 2, 3, 4, 5, and 6). That is, the adaptive approach is usually as effective as or sometimes even better than the total and additional approaches. Moreover, the adaptive approach is sometimes better than the total and additional approaches. To summarize the results in Table III and answer the first research question, we constructed Table IV based on the test results in Table III. Table IV presents the values of q of the adaptive approach when its corresponding results are significantly better than both the total and additional approaches (i.e., \(Q > T \& A\)), its corresponding results are significantly better than only one of the total and additional approaches and are no significantly different from the other approach (i.e., \(Q \geq T \& A\)), its corresponding results are not significantly different from either the total or additional approaches (i.e., \(Q = T \& A\)). The last row summarizes the number of objects whose results by using the adaptive approach with some q are classified into the corresponding group. According to this table, on five objects (i.e., print_tokens2, replace, tot_info, space, and jmeter), the proposed adaptive approach (with some specified q) is significantly better than both the total and the additional approaches. Moreover, the proposed adaptive approach can be at least as effective as the total and the additional approaches for all the objects in this experimental study.

Note that Table III only compares the adaptive approach with the total or additional approach, although we have conducted the Wilcoxon signed ranked test on comparing the adaptive approach with the original or reverse approach. Because according to the signed test results, the adaptive approach (including Q1, Q2, Q3, Q4, Q5, and Q6), the total and additional approaches are significantly better than the original and reverse approaches. This observation is as expected since the latter two approaches schedule test cases based on the order they are provided and thus cannot always guarantee the effectiveness of the generated prioritized test suite.

2) Choice of q in the Adaptive Approach: As our experimental study implemented the adaptive approach with various q, we will analyze their experimental results and try to find how to set q to guarantee the effectiveness of the adaptive approach in this section.

Considering the results for all the objects in Table III, the adaptive approach with q=0 or q=0.2 loses the additional approach 4 times, the adaptive approach with q=0.4, q=0.6, or q=0.8 loses the total approach twice and loses the additional approach 3 times, whereas the adaptive approach with q=1 loses the total approach 3 times and loses the additional approach 5 times. That is, none of the candidate values (i.e., 0, 0.2, 0.4, 0.6, 0.8, and 1) of q can guarantee that the corresponding adaptive approach always wins the total and additional approaches for all the objects. Table IV confirms this conclusion. Generally speaking, the adaptive approach can always be at least as effective as the total and the additional approaches, and the former is significantly better than the latter two approaches for many objects. However, we cannot find a constant q among the six candidate values to guarantee that the corresponding adaptive test-case prioritization approach is always significantly better than the total and additional approaches. That is, although the adaptive approach is generally effective compared with

\(^5\)In particular, our experimental study used the statistical analysis software SPSS 15.0 to perform the Wilcoxon signed ranked test.

\(^6\)Here, M wins N iff the APFD of M is bigger than that of N.
the total and the additional approaches, the value of \( q \) to
guarantee the effectiveness of the corresponding adaptive
approach is not fixed and general for all objects. That is, for
different objects, the adaptive approach may have various
values of \( q \) to achieve the best prioritization results.

According to Table IV, the adaptive approach with \( q=0.8 \)
seems to be more promising than the adaptive approach with
other values of \( q \) because this approach wins both the total
and additional approaches in three objects, and have similar
effectiveness with the total and additional approaches in
another four objects. That is, the adaptive approach with \( q=0.8 \)
produces satisfactory results in most objects. Moreover, the
adaptive approach with \( q=0 \) can produce satisfactory results
in the remaining objects. Therefore, based on the existing
experimental results, when it is hard to determine the value
of \( q \) in using the adaptive approach, \( q=0.8 \) or \( q=0 \) may make
the corresponding approach produce satisfactory results.

**G. Summary**

According to the experimental results, the adaptive
approach is usually significantly more effective than the total
approach and is competitive to the additional approach.
Moreover, the adaptive approach is significantly better than
the additional approach on some objects (e.g., replace and
schedule).
V. CONCLUSIONS AND FUTURE WORK

As the existing test-case prioritization approaches schedule the execution order of all test cases before programmers run test cases on the modified program, the execution information of the modified program is not available for these approaches. Without any clue on the execution information of the modified program, the existing approaches mainly rely on the execution information of the previous program before modification although the latter execution information is not consistent with the former execution information. To address this problem, we presented an adaptive test-case prioritization approach, which determines the execution order of test cases simultaneously during the execution of test cases on the modified program. Consequently, the execution information of the modified program is available following the process of the adaptive approach. In particular, during regression testing, our adaptive approach determines which test case to run based on the output of selected test cases as soon as a test case finishes running. The experimental results show that the adaptive approach is usually significantly better than the total statement-coverage based test-case prioritization approach and comparable to the additional statement-coverage based test-case prioritization approach. Moreover, sometimes the adaptive approach is even better than the additional approach.

The main advantage of the proposed adaptive approach lies in its combination of the process of test-case prioritization and the process of test-case execution so that the execution information of the modified program becomes available in test-case prioritization. The adaptive approach proposed in this paper uses the output of the modified program because the output of the modified program is usually collected during regression testing without any extra cost. Although other execution information like statement coverage can be collected during the execution of the modified program and may improve the adaptive approach, it may consume extra cost (e.g., instrumentation) to collect such information and increment the time cost of regression testing. Considering the balance between efficiency and effectiveness, the adaptive approach in this paper schedules test cases based on the output of the modified program and the statement coverage of the previous program. In the future, we plan to conduct experiments to compare the adaptive approach proposed in this paper with the existing test-case prioritization approaches on their time cost. Furthermore, we will attempt to present another adaptive test-case prioritization approach based on other execution information of the modified program besides the output.

ACKNOWLEDGMENT

This research is sponsored by the National Basic Research Program of China (973) under Grant No. 2011CB302604, the High-Tech Research and Development Program of China under Grant No.2012AA011202, the Key Program of Ministry of Education, China, under Grant No.3130110, and the National Natural Science Foundation of China under Grant Nos. 61121063 and 61272157.

REFERENCES
