Postprint of article in *IEEE Transactions on Reliability*

Beating Random Test Case Prioritization

Zhi Quan Zhou, Chen Liu, Tsong Yueh Chen, T. H. Tse, and Willy Susilo

Abstract—Existing test case prioritization (TCP) techniques have limitations when applied to real-world projects, because these techniques require certain information to be made available before they can be applied. For example, the family of input-based TCP techniques are based on test case values or test script strings; other techniques use test coverage, test history, program structure, or requirements information. Existing techniques also cannot guarantee to always be more effective than random prioritization (RP) that does not have any precondition. As a result, RP remains the most applicable and most fundamental TCP technique. This paper proposes an extremely simple, effective, and efficient way to prioritize test cases through the introduction of a dispersity metric. Our technique is as applicable as RP. We conduct empirical studies using 43 different versions of 15 real-world projects. Empirical results show that our technique is more effective than RP. Our algorithm has a linear computational complexity and, therefore, provides a practical solution to the problem of prioritizing very large test suites (such as those containing hundreds of thousands, or millions, of test cases), where the execution time of conventional nonlinear prioritization algorithms can be prohibitive. Our technique also provides a practical solution to TCP when neither input-based nor execution-based techniques are applicable due to lack of information.

Index Terms—Dispersity, dispersity metric, dispersity-based prioritization, dissimilarity, random prioritization, natural distance, adaptive random testing, adaptive random sequence.

1 Introduction

Test case prioritization (TCP) is a major challenge in software testing. It attempts to find an optimal ordering of test case executions, which can maximize the tester's effort even if the testing is prematurely terminated, such as when the testing resources have been exhausted [1]–[4].

The majority of automated test case prioritization methods involve the use of structural coverage information (such as control flow, data flow, call-tree paths, and relevant slices information) of the test cases [3], or an estimate of such information [5]. One of the intuitions is that early fulfillment of structural coverage should increase the chance of fault detection. Among the various prioritization methods, the *additional* algorithm (which is an instance of additional greedy algorithms) has been considered as one of the most frequently used benchmark methods [2], [3], [6], [7].

Other approaches to automated test case prioritization involve the use of requirements specifications [3], [8], [9], system models [10], test case execution history in previous runs [11]–[13], mutation testing [14], [15], cost-awareness information such as the severity of faults and the costs of test case executions [3], [12], [16], test case distance metrics based on their execution profiles [6], [17], [18], or differences between different versions of the SUT [19]. More recently, Busjaeger and Xie reduced the TCP problem to that of learning to rank [20].

The above approaches have their advantages under different assumptions and situations, and yet none of them or their combinations can be as applicable as, or can always be more cost-effective than, random prioritization. This is because the required information (such as test case coverage, test case execution history, program structure, or requirements specifications) may not always be available in practical situations [21]. To address this problem, several *input-based* TCP techniques have been developed [21]–[23], making use of the test case values/test script strings rather than their code-coverage or other test performance information. Compared with most other approaches (which are *execution-based*), the input-based TCP strategy has better applicability; nevertheless, its application requires the tester to be able to access the concrete input values of the test cases, and to design *effective* distance (or dis/similarity) metrics to measure the dissimilarity between the concrete values of two test cases. This means that the tester must have thorough knowledge of "the input structure and semantics of the application under test" [21, p. 95]. Even with such knowledge, to design good distance metrics for any arbitrary input structure of any arbitrary program can be challenging. Therefore, input-based TCP techniques are still not as applicable as random prioritization, as the latter does not even require the tester to know the concrete values of the test cases.

Furthermore, as pointed out by Zhou et al. in 2012 [18], traditional TCP research did not consider the fact that real-world test suites could become very large with "millions of test cases." In this situation, the computational overhead of test case prioritization is a major concern. The *additional* statement (or branch) coverage algorithm [2], for example, has a time complexity of $O(n^2m)$, where

- *Zhi Quan Zhou, Chen Liu, and Willy Susilo are with the Institute of Cybersecurity and Cryptology, School of Computing and Information Technology, University of Wollongong, NSW 2522, Australia. E-mails: zhiquan@uow.edu.au, cl565@uowmail.edu.au, wsusilo@uow.edu.au.*
- *Tsong Yueh Chen is with the Department of Computer Science and Software Engineering, Swinburne University of Technology, Hawthorn, VIC 3122, Australia. E-mail: tychen@swin.edu.au.*
- *T. H. Tse is with the Department of Computer Science, The University of Hong Kong, Pokfulam, Hong Kong. E-mail: thtse@cs.hku.hk.*

n is the number of test cases and m is the number of statements (or branches) of the program under test. In our empirical studies with real-world large test suites, we find that their execution time can become prohibitive.

Following up on Zhou et al.'s observations [18], Miranda et al. [24] also reported the following:

> The number of test cases to prioritize grows in size up to millions ... for real-world software the size of a test suite can often exceed the size of the system under test. In contrast, the time available for test execution cycles decreases ...every day at Google an amount of 800K builds and 150M test runs are performed on more than 13K code projects ...**most TCP approaches in the literature cannot handle such scale. Our experimental results show that some TCP approaches become soon inefficient even for small-medium size benchmarks**.

To address these problems, Miranda et al. [24] introduced a *FAST* family of *scalable* test case prioritization techniques based on similarity, and compared *FAST* with other similarity-based TCP techniques (including adaptive random test case prioritization techniques introduced by Jiang et al. [6] and Zhou et al. [17], [18]) among others. *FAST* can be used either as a white-box prioritization technique based on code coverage information, or as a black-box prioritization technique based on the string representation of the actual test cases—in this aspect, the black-box *FAST* is essentially an *input-based* TCP strategy because it needs to know the input values/test scripts. As with the other TCP techniques discussed earlier in this section, therefore, *FAST* is still not as applicable as random prioritization that requires neither white-box coverage information nor the values of the concrete test cases/scripts.

This paper, therefore, raises the following practical research question:

> ***RQ1*** *In real-world software testing, can there be a lightweight test case prioritization method that has the following properties: (i) it is more effective than random prioritization; (ii) it can more quickly detect failures (in terms of execution time) than random prioritization; (iii) it is as efficient as random prioritization; and (iv) it is as readily applicable as random prioritization?*

Remarks

1) The objective of this research is to provide practical guidelines to software engineers in testing. Therefore, we are interested in real-world software projects.
2) In the context of test cases prioritization, method A is more "effective" than method B if the ordering of test case executions generated by the former can enable an earlier detection of problems, such as the detection of failures. To measure effectiveness, we need to record the number of test case executions, without considering the actual execution time.
3) To check property (*ii*), we need to record the overall, actual execution time for failure detection, without considering the number of test case executions.
4) The "efficiency" of a TCP algorithm can be evaluated either by analyzing its computational complexity or by measuring its actual execution time of prioritizing a given test suite, without considering failure detections. "It is as efficient as random prioritization" means that it should at least have the same order of time and space complexity as random prioritization, namely, linear complexity.
5) "It is as readily applicable as random prioritization" means that it can be applied whenever random prioritization can be applied. For example, random prioritization can be applied without the knowledge of the concrete input values of the test cases, test case performance/coverage information, or requirements specifications. Therefore, the new TCP method should also have this property.
6) If all of the above requirements are met, the new method should have a big chance to replace random prioritization in practical situations.

If the answer to RQ1 is positive, we further raise the following research question:

> ***RQ2*** *How does the lightweight technique of RQ1 compare with the heavyweight "additional" algorithm, with respect to the same properties, namely, (i) effectiveness, (ii) actual execution time for failure detection, (iii) efficiency, and (iv) applicability?*

The *additional* algorithm is considered because it is representative as one of the most frequently used benchmark TCP techniques and is easy to implement. Readers are referred to Section 9.4 for further discussion on benchmark TCP techniques.

2 Summary of Contributions

The contributions of this paper are summarized as follows:

1) To the best of our knowledge, this is the first work to point out that neighboring test cases in a real-world test suite often have similarities in certain ways while more dispersed test cases tend to be dissimilar.
2) Based on the above observation, we propose a concept of *natural distance* that can be used as a universal distance metric for measuring the dispersity among test cases in real-world test suites. The measurement requires neither the knowledge of the source code, requirements specifications, designs, input types, etc., of the program under test, nor knowledge of coverage data, execution history, concrete input values, etc., of the test cases.
3) Using the concept of natural distance, we propose a simple and practical test case selection / prioritization strategy to address the research question RQ1. The general form of our strategy is given as Hypothesis **I**, and a

specific implementation of our strategy is given as an algorithm shown in Fig. 4, which has a *linear* time and space complexity. Our strategy, therefore, provides a practical solution to the problem of prioritizing very large test suites—for large test suites containing hundreds of thousands, or millions, of test cases, the execution time of conventional nonlinear prioritization algorithms can be prohibitive.

4) We conduct a series of empirical studies using 43 different versions of software under test from 15 real-world software projects. The empirical results show that our method significantly improves the effectiveness and reduces failure detection time of random prioritization and that, even in the worst case, the performance of our method is still close to that of random prioritization.
5) The results also show that our lightweight approach outperforms the heavyweight branch-coverage-based *additional* algorithm with respect to applicability, execution time for failure detection, as well as efficiency, whereas the *additional* algorithm outperforms our approach in the majority of situations with respect to effectiveness. This finding addresses the research question RQ2.

The rest of this paper is organized as follows: Section 3 presents our observation of the nature of real-world test suites, and proposes a *natural distance metric* based on this observation. Section 4 describes the design of empirical evaluation as well as introduces our test case prioritization method. Section 5 analyzes the empirical results. Section 6 further compares the efficiency of our method with that of random prioritization. Section 7 discusses threats to validity, and Section 8 describes related work. Section 9 contains further discussion of related topics, and Section 10 concludes the paper.

3 A Natural Distance Metric for Real-World Test Suites

We first present our observation of real-world test suites as follows:

Observation I: Real-world test suites have one important commonality that is, to the best of our knowledge, never exploited for test case prioritization: Neighboring test cases often have similarities in certain ways while more dispersed test cases tend to be dissimilar.

Consider how test cases in a real-world test suite would have been generated. In white-box testing, for instance, test cases are generated to cover the statements, branches, functions, etc., of the source code. After testers have designed a test case to cover the *true* branch of an *if* statement S, they would normally consider a second test case to cover the *false* branch of the same statement. Therefore, these two consecutively designed test cases may execute similar paths or branches before statement S is reached. For real-world programs with complex loop and branch structures, a basic technique of generating white-box test cases is to traverse the *execution tree* of the program under test [25], [26]. Sen et al. [26] observed, for instance, that "the feasible executions of a program can be represented as a tree, where the branch points in a program are internal nodes of the tree. The goal is to generate concrete values for inputs which would result in different paths being taken. The classic approach is to use depth first exploration of the paths by backtracking." Consecutive test cases generated in this way will also exhibit similarities. Consider the example shown in Fig. 1 and Fig. 2. This simple illustration enables readers to understand Observation **I** easily.

```
void testme(int a, int b, int c, int d, int e){
if(a>b){ /* Condition 1 */
  printf("1T "); /* Condition 1, True Branch */
    if(b>c){ /* Condition 2 */
      printf("2T "); /* Condition 2, True Branch */
        if(c>d){ /* Condition 3 */
          printf("3T "); /* Condition 3, True Branch */
          if(d>e) /* Condition 4 */
            printf("4T "); /* Condition 4, True Branch */
          else
            printf("4F "); /* Condition 4, False Branch */
        }
        else
          printf("3F "); /* Condition 3, False Branch */
    }
    else
      printf("2F ");  /* Condition 2, False Branch */
}
else{
  printf("1F "); /* Condition 1, False Branch */
    if(b==1){ /* Condition 5 */
      printf("5T "); /* Condition 5, True Branch */
        if(c==1){ /* Condition 6 */
          printf("6T "); /* Condition 6, True Branch */
          if(d==1) /* Condition 7 */
            printf("7T "); /* Condition 7, True Branch */
          else
            printf("7F "); /* Condition 7, False Branch */
        }
        else
          printf("6F "); /* Condition 6, False Branch */
    }
    else
      printf("5F ");  /* Condition 5, False Branch */
 }
 printf("\n");
 return;
}
```

Fig. 1: An illustrative example of a program under test, named "a".

Fig. 1 shows the source code of a C program "a.c". The program has seven *if* statements, marked as "Condition 1," "Condition 2," ..., "Condition 7." As a result, the program has a total of $7 \times 2 = 14$ branches. In each of these 14 branches, a *printf* statement is first executed to print the unique ID of the current branch. For example, the third line prints an ID "1T" to the console to indicate that a *true* branch of Condition 1 has been taken. The IDs "1F," "2T," "2F," ..., "7F" can be explained similarly.

Fig. 2 shows a screenshot of using an automatic test case generator "CUTE" for the above program. CUTE (*Concolic Unit Testing Engine for C*) was developed by Sen et al. [26]. It combines concrete and symbolic execution techniques to automatically generate and execute white-box test cases. The first line "cute a −i 100" is a command entered by the user to run CUTE, where "a" indicates the name of the program under test, "−i 100" requests that the total number of test cases be no more than 100. The

```
$ cute a -i 100
[Iteration 1] a.exe  -m 2
1F 5F
[Iteration 2] a.exe
1F 5T 6F
[Iteration 3] a.exe
1F 5T 6T 7F
[Iteration 4] a.exe
1F 5T 6T 7T
[Iteration 5] a.exe
1T 2F
[Iteration 6] a.exe
1T 2T 3F
[Iteration 7] a.exe
1T 2T 3T 4F
[Iteration 8] a.exe
1T 2T 3T 4T
One complete search is over
```

Fig. 2: Screenshot: CUTE generated and executed a total of eight consecutive test cases for a complete search of the execution tree of the program under test, whose source code is shown in Fig. 1. The executable code of the program under test is named a.exe.

second line, starting with "[Iteration 1]," is an output line of CUTE, indicating that test case 1 is being generated and executed. The message "−m 2" indicates the work mode, which is not relevant to our present discussion. The output of the program under test is shown in the next line, that is, "1F 5F," which indicates that the execution path of test case 1 is branch 1F followed by 5F. The next line, starting with "[Iteration 2]," is again an output line of CUTE, meaning that test case 2 is being generated and executed. The output of the program under test against test case 2 is shown in the next line, which is "1F 5T 6F." This indicates that both test cases 1 and 2 took the *false* branch at Condition 1. In the end, a total of eight test cases have been generated and executed by CUTE. At the bottom of the screenshot, CUTE prints that the search in the execution tree of the program under test is complete. Let S_i denote the set of branches covered by test case i, $i = 1, 2, \ldots, 8$. It can be found that $|S_1 \cap S_2| = 1$, $|S_2 \cap S_3| = 2$, $|S_3 \cap S_4| = 3$, $|S_4 \cap S_5| = 0$, $|S_5 \cap S_6| = 1$, $|S_6 \cap S_7| = 2$, and $|S_7 \cap S_8| = 3$. In most situations, any two consecutively generated test cases have something in common in the branches that they cover (and in the paths that they execute), with the exception of test cases 4 and 5. For more dispersed test cases, they tend to be dissimilar (such as $|S_2 \cap S_8| = 0$ for test cases 2 and 8). This situation is further illustrated using Fig. 3, which gives all of the values of $|S_i \cap S_j|$, $i, j = 1, 2, \ldots, 8$. In the figure, because the matrix is symmetric, only half of it is colored to show that, when the value of $|i - j|$ increases from 1 to 7, the average of $|S_i \cap S_j|$ drops from 1.71 to 0.

For real-world large and complex programs with complicated control flow and data structures, a bounded depth-first search strategy can be used for test case generation [26], and the similarities between consecutively generated test cases can become more evident as the lengths of execution paths become large. Generally

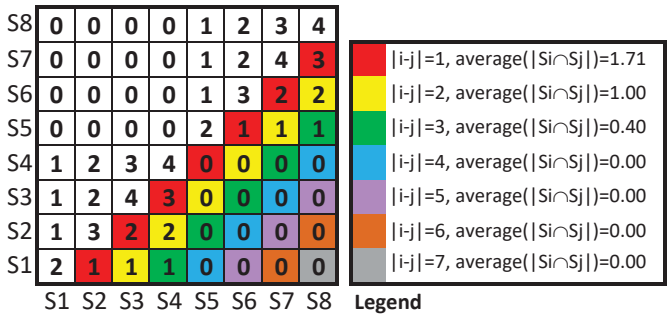

Fig. 3: $|S_i \cap S_j|$, $i, j = 1, 2, \ldots, 8$.

speaking, it is our observation that *test cases that are dispersed with respect to their positions in a real-world test suite often have a higher degree of dissimilarity than those that are close together*[1]. This statement should not, of course, be absolute as there are always exceptions.

The above observation can also be made when test cases are generated using other techniques, such as black-box testing, model-based testing, fault-based testing, or in the context of regression testing [27]. This is because test designers or automated test case generators normally follow a logical or systematic approach when designing / generating test cases. Neighboring test cases, therefore, tend to have similarities in their logic or purpose. For example, test cases that are close together in their relative positions in a black-box test suite may have similarities in the functions that they exercise or the value combinations that they take, and those that are close together in a regression test suite may have been designed around the same period of time for a specific version of the software under test.

The above observation may not hold true if all of the test cases are randomly generated. However, in real-world software testing, random testing is often used in combination with other techniques such as partition testing and testing with special values. In these situations, nonrandom neighboring test cases may still have similarities. Even in situations where all the test cases are purely random, our testing method (which will be introduced shortly) will still do no harm to the effectiveness and efficiency of the original random testing method.

Definition I: Let $T = (t_1, t_2, \ldots, t_n)$, where $n > 0$, be a sequence of test cases. The *natural distance* between test cases t_i and t_j, where $1 \leq i, j \leq n$, is defined as $|i - j|$, the absolute value of $i - j$.

Definition **I** specifies a *natural distance metric*. In short, the natural distance between two test cases is the difference in their positions in the test suite.

Let $T = (t_1, t_2, \ldots, t_{10000})$ be a real-world test suite. Suppose that test case t_9 is selected and executed first, and

1. This observation is made through our participation in more than ten real-life software projects in collaboration with the Australian IT industry. The projects involve the development and testing of Web Application Programming Interfaces (APIs) and Graphical User Interfaces (GUIs), where most test cases are manually generated.

then no failure is detected. According to Observation **I**, t_9 and t_{10} could be similar and, therefore, if t_9 does not reveal a failure, it is not wise to select t_{10} as the next test case. Instead, the next test case had better be farther apart from the previously executed test cases that have not yet detected a failure.

We propose the following hypothesis:

Hypothesis I: Consider software testing in the real world where test cases are selected from, or prioritized using, a (possibly very large) suite of test cases. If the test cases are selected in such a way that they are evenly spread over the test suite in terms of natural distance, then the test effectiveness will be at least as good as random testing.

Note that the concept of evenly spreading test cases also forms the basic intuition of *Adaptive Random Testing* (*ART*) [28]–[30]. ART has been proposed as an enhancement to Random Testing (RT) based on the observation that failure-causing inputs tend to form contiguous failure regions. If certain test cases do not reveal any failure, ART recommends the selection of subsequent test cases that are far away from those already executed. In this way, ART generates test cases that are more evenly spread over the input domain than RT. There is a fundamental difference between ART and the present approach: The former generates concrete values of test cases by evenly spreading them across the *input domain* (and hence needs to develop different distance calculation methods/metrics for different types of input domains). On the other hand, the latter does not consider the input domain—it generates an *execution sequence* of test cases (rather than actual test cases with concrete values) based on IDs assigned to them when they were first added to the test pool.

Furthermore, our approach is different from *adaptive random test case prioritization*, which uses ART algorithms and coverage information to prioritize test cases [6], [17], [18]. Our approach does not need any coverage information.

There are different effectiveness metrics for test case selection and prioritization, such as the P-measure, the F-measure, and the average percentage of faults detected (APFD) [15], [29], [31]. While Hypothesis **I** is not restricted to any specific effectiveness metric, we will adopt APFD and the F-measure in our empirical studies, as will be explained later in the paper.

The next question is: how can the order of the test cases in a real-world test suite be identified? To answer this question, we have the following observation:

Observation II: An order of the test cases in a real-world test suite can often be decided easily.

Observation **II** states that deciding the position (ordinal rank) of a test case in a real-world test suite can often be easy. In automated testing, for instance, a test driver (such as a shell script file) is normally provided to run all the test cases. The order of test case executions can therefore be regarded as the order of the test cases in the test suite. The majority of the subject packages used in our empirical studies are of this category. For instance, in the "mochitest-devtools" test suite of Firefox [2], one of the test cases assigns a property "unchecked" to the "record snapshot" button; the immediate next test case assigns a property "enabled" to the same button; and the test case that follows makes the same button "visible," and so on. All these neighboring test cases involve setting certain properties of the same button.

In situations where a test driver is not provided, the test case names (such as function names, filenames, and directory structures) can often provide hints on a potentially useful ordering of the test cases. For example, the alphanumeric order of the names can often be a useful indicator of a suitable order of the test cases. The following example shows typical test case names of the Apache Commons Text, a library of algorithms working on strings [3]:

testContains_char,
testContains_String,
testContains_StringMatcher,
testDeleteAll_char,
testDeleteAll_String,
testDeleteFirst_char,
testReplaceAll_char_char,
testReplaceAll_String_String,
testReplaceFirst_char_char,

and so on. See Section 4.4 for more discussions. The sequence of test cases may also be revealed by the dates and times that the test case files were first created.

Testers may also apply more than one strategy to identify test case sequences. For example, in a collaborative project (including open source ones), multiple contributors may be working on different parts of the code and adding test cases to the repository simultaneously. In this scenario, we may need to first partition the test cases into different groups according to the contributor ID, and then identify each group's test case sequence using the strategies discussed above.

4 Design of Empirical Evaluation

A series of empirical studies with real-world software packages have been conducted to validate Hypothesis **I** in the context of test case prioritization. This section describes the design of the empirical evaluation, including dependent and independent variables, our test case prioritization algorithm, subject packages, and how the orders of test cases in the test suites were decided.

4.1 A summary of dependent and independent variables for the empirical studies

The independent variable for the empirical studies is the test case prioritization algorithm, namely, (i) random prioritization, (ii) dispersity-based prioritization, and (iii) the *additional* algorithm based on the branch coverage information of the test cases collected from an earlier

2. https://www.mozilla.org
3. https://commons.apache.org/proper/commons-text/

version of the program under test. Algorithm (ii) is our approach, based on the natural distance metric and will be elaborated in Section 4.2. Algorithm (iii) is widely considered to be one of the best test case prioritization algorithms.

The dependent variables for the empirical studies are: (i) applicability of the test case prioritization algorithm (that is, whether the algorithm can be applied to the object under study), (ii) test case prioritization effectiveness, evaluated using APFD and the F-measure, and (iii) execution time for the detection of the first failure. These will be elaborated in Section 4.3.

The efficiency of a test case prioritization algorithm is evaluated by referring to its computational complexity. Because random prioritization and dispersity-based prioritization are both linear algorithms, their efficiency is further compared in Section 6 through an additional set of experiments.

The objects include the programs under test and their test suites, which will be described in Section 4.3.

4.2 Our dispersity-based prioritization (DBP) algorithm

As explained earlier, our dispersity-based approach is not an ART technique as we do not consider the input domain of the program under test. Nevertheless, for the purpose of empirical evaluation, we have applied an efficient ART algorithm to generate an *adaptive random sequence* of integers in the range $[1,\ n]$, where n is the number of test cases, to serve as a sequence of test case IDs. In this way, we can achieve an even spread of test case IDs, which will enable us to validate Hypothesis **I**.

The algorithm we have applied is FSCS-ART enhanced with the "forgetting by consecutive retention" strategy [29], [32], as explained in the next paragraph. Section 8 discusses ART algorithms in more detail.

FSCS-ART is a member of the ART family of algorithms that works as follows: Whenever a new test case is needed, c candidates are first generated randomly, where c is a constant. The distances between each candidate and all the already executed test cases are calculated, and the minimum distance is recorded. The candidate having the largest minimum distance is then chosen as the next test case, and all the other candidates are discarded. To generate n test cases, the time complexity of FSCS-ART is in $O(n^2)$. The "forgetting by consecutive retention" strategy improves the complexity to $O(n)$ [32]. To apply this strategy, instead of calculating the distances between each candidate and *all* the already executed test cases, the distance calculation is limited to the last k already executed test cases, where k is a constant known as the *memory parameter*.

It was previously reported that the effectiveness of FSCS-ART improves as c increases up to about 10, and then does not improve much further [33]. Another study found that the "forgetting by consecutive retention" strategy is more effective than random testing even when k is as small as 10 [34]. Therefore, in the present study of generating test case IDs, c and k are both given a constant value of 10. The enhanced algorithm in combination with the natural distance metric is shown in Fig. 4.

The algorithm accepts one single input parameter n, which is a positive integer. It is assumed that the original test suite is a sequence of test cases $(t_0, t_1, \ldots, t_{n-1})$, where i is the ID of test case t_i, $i = 0, 1, \ldots, n-1$.

Statement 1 of the algorithm uses an array A to store the test case IDs. Statement 2 randomly selects the first test case ID, and statement 3 moves the selected test case ID to A[0] (which can be implemented by the following three statements: temp=A[0]; A[0]=A[m]; A[m]=temp;). In this way, the selected test case ID is stored in A[0], and the rest of the array is the set of not-yet-selected test case IDs, from which future test case IDs will be selected. Statement 4 sets nbOfSelectedTestCases (for "number of selected test cases") to 1, as one test case ID has been selected. In this way, the following invariant is created for the *while* loop starting from statement 5: {A[0], A[1], ..., A[nbOfSelectedTestCases−1]} is always the set of selected test case IDs, and {A[nbOfSelectedTestCases], A[nbOfSelectedTestCases+1], ..., A[n−1]} is always the set of not-yet-selected test case IDs.

Statement 6 means that the memory parameter to be used in "forgetting by consecutive retention" [32] is 10, that is, the distance calculation will be applied only to the last 10 executed test cases. The variable memoryParameter is set to "minimum(nbOfSelectedTestCases, 10)" because in the beginning the number of selected test case IDs is less than 10. Statement 7 applies FSCS-ART to select the next test case ID as follows: First, select 10 candidates randomly from the set of not-yet-selected test case IDs. Let the 10 candidates be A[c_1], A[c_2], ..., A[c_{10}]. (In the situation where the number of not-yet-selected test cases is smaller than 10, select all of them as candidates.) For each candidate A[c_i], calculate d_i, which is the minimum of:

|A[c_i]−A[nbOfSelectedTestCases−memoryParameter]|,
|A[c_i]−A[nbOfSelectedTestCases−memoryParameter+1]|,
...,
|A[c_i]−A[nbOfSelectedTestCases−1]|.

Let d_j be the maximum value among $\{d_1, d_2, \ldots, d_{10}\}$. Then, A[$c_j$] will be selected to be the next test case ID. This A[c_j] is referred to as "A[r]" in statement 7. Because only up to 10 candidates and up to 10 executed test cases are involved in the distance calculation, the time complexity of statement 7 is constant. Statement 8 again moves the selected test case ID to the left part of the array, and statement 9 moves the boundary between the prioritized and un-prioritized sets rightward.

Finally, statements 11 to 14 print the IDs of test cases in their prioritized order. In other words, the prioritized order of test cases is $(t_{A[0]}, t_{A[1]}, \ldots, t_{A[n-1]})$.

The above algorithm has a linear $O(n)$ time and space complexity, which is in the same order of complexity as

```
Purpose: This algorithm performs test case prioritization in linear time and space complexity.
The FSCS-ART algorithm with the "consecutive retention" forgetting strategy is applied
together with the natural distance metric to generate a sequence of test case IDs, where the
size of candidate set is 10 and the Memory Parameter in "forgetting" is also 10 (that is, the
distance calculation is applied to only the last 10 executed test cases).
Input: A positive integer n.
Precondition: The real-world test suite to be prioritized is a sequence of n test cases,
denoted by (t0, t1, ..., tn-1).
Output: Array A, which is a sequence of prioritized test case IDs. Therefore, the prioritized
order of test cases will be (tA[0], tA[1], ..., tA[n-1]).

Begin Algorithm
1.  For i = 0, 1, ..., n-1, set A[i] to i;
    /*A[i] is initialized to store the ID of test case ti.*/

2.  Randomly select an integer m in the range [0, n-1];
    /*Select the first test case ID, A[m], randomly.*/

3.  Swap(A[0], A[m]);
    /*Let A[0] store the ID of the first selected test case.*/

4.  Set nbOfSelectedTestCases to 1;

5.  While(nbOfSelectedTestCases < n-1)
6.    Set memoryParameter to minimum(nbOfSelectedTestCases, 10);
7.    Apply FSCS-ART to select the next test case ID from the set
         {A[nbOfSelectedTestCases], A[nbOfSelectedTestCases+1], ... A[n-1]}, using
         {A[nbOfSelectedTestCases-memoryParameter], A[nbOfSelectedTestCases-
         memoryParameter+1], ..., A[nbOfSelectedTestCases-1]} as the set of selected test case
         IDs; The distance between A[x] and A[y] is given by |A[x]-A[y]|; Let A[r] be the
         finally selected test case ID, where nbOfSelectedTestCases <= r <= n-1;
8.    Swap(A[nbOfSelectedTestCases], A[r]);
9.    nbOfSelectedTestCases = nbOfSelectedTestCases + 1;
10. EndWhile;
11. Print("The IDs of the prioritized order of test cases are in the following sequence:");
12. For i from 0 to n-1
13.   print(A[i]);
14. EndFor;
End of Algorithm
```

Fig. 4: Our dispersity-based algorithm for test case prioritization, which is in the same order of time and space complexity as random prioritization, namely, $O(n)$ where n is the number of test cases prioritized.

random prioritization. This algorithm is also as applicable as random prioritization because it demands as little information as the latter.

4.3 Subject programs, test suites, and evaluation metrics

In this section, we first provide an overview of the subject programs, and then discuss the evaluation metrics. We also show the challenges in designing the controlled experiments and provide our solutions. Finally, we present more details of the subject packages.

4.3.1 Overview

In the empirical evaluation, we investigated 15 real-world software projects, involving a total of 43 different versions of software under test (SUT), which are summarized in Table 1. The SUTs are written in different programming languages and have various sizes and functionality. Their test suites also have various sizes ranging from a few hundred test cases to very large (which can be larger than the number of statements in the source code of the SUT). This set of projects, therefore, can be considered representative of real-world projects.

The 15 projects listed in Table 1 are SQLite (row #1), g++ (row #2), gcc (row #3), gfortran (row #4), libmudflap (row #5), libstdc++ (row #6), commons-lang (row #7), commons-math (row #8), jfreechart (row #9), joda-time (row #10), Firefox (row #11), Autoconf (rows #12 to #16, five versions), Automake (rows #17 to #21, five versions), MySQL (rows #22 to #26, five versions), and Space (rows #27 to #43, seventeen versions). All the packages, including the SUTs and test suites, were downloaded from the project websites listed in the third column of Table 1. Most packages contain programs, scripts, or other types of files, written in different programming, scripting, or markup languages. The fifth column of Table 1 lists only the main languages, which are not exhaustive. Furthermore, a tool named SLOCCount (http://manpages.ubuntu.com/manpages/precise/man1/sloccount.1.html) was used to count the source lines of code (SLOC) of the packages.

4.3.2 Evaluation metrics

Because the F-measure [29], [35] and APFD [15] are the most common metrics for the evaluation of the *effectiveness* of test case prioritization approaches, they are used in our empirical evaluation. The F-measure refers to the expected number of test case executions that need to be run in order to detect the first failure. APFD also measures how quickly failures can be detected, and takes

TABLE 1
Summary of software packages used in empirical evaluation.

row #	project name & version	project website	total size of package (SLOC)	main language(s)	size of test suite	test suite version	number of failing test cases
1	SQLite v3.7.10	http://sqlite.org	140,603	c	787,530	v3.7.15	13
2	g++, GCC v4.8.0	http://gcc.gnu.org	4,781,336	c, c++ ada java	51,829	GCC v4.8.0	143
3	gcc, GCC v4.8.0				92,603		93
4	gfortran, GCC v4.8.0				43,032		16
5	libmudflap, GCC v4.8.0				1,436		3
6	libstdc++, GCC v4.8.0				8,474		4
7	commons-lang v3.0.1	http://commons.apache.org/proper/commons-lang	56,617	java	2,047	v3.1	2
8	commons-math v3.1	http://commons.apache.org/proper/commons-math	161,968	java	4,534	v3.1.1	4
9	jfreechart v1.0.14	http://jfree.org/jfreechart	147,590	java	2,203	v1.0.15	10
10	joda-time v2.0	http://www.joda.org/joda-time	86,337	java	3,888	v2.1	8
11	Firefox v31.0	https://www.mozilla.org	6,177,736	c++, c	480,575	v31.0	168
12	Autoconf v2.64	http://gnu.org/software/autoconf	7,896	sh perl lisp	503	v2.69	82
13	Autoconf v2.65		7,571				75
14	Autoconf v2.66		7,634				35
15	Autoconf v2.67		7,645				28
16	Autoconf v2.68		7,670				19
17	Automake v1.13	http://gnu.org/software/automake	67,840	sh perl c	1,313	v1.14	336
18	Automake v1.13.1		68,121				336
19	Automake v1.13.2		68,476				329
20	Automake v1.13.3		68,264				56
21	Automake v1.13.4		68,391				47
22	MySQL v5.6.7	http://dev.mysql.com/downloads/mysql	1,721,667	c++ c java	2,554	v5.6.12	361
23	MySQL v5.6.8		1,725,920				257
24	MySQL v5.6.9		1,727,633				195
25	MySQL v5.6.10		1,728,713				129
26	MySQL v5.6.11		1,731,524				34
27	Space v3	http://sir.unl.edu	about 6,199	c	13,551	n/a	645
28	Space v7		about 6,199				163
29	Space v8		about 6,199				95
30	Space v12		about 6,199				33
31	Space v16		about 6,199				503
32	Space v17		about 6,199				196
33	Space v18		about 6,199				33
34	Space v20		about 6,199				210
35	Space v21		about 6,199				210
36	Space v22		about 6,199				64
37	Space v23		about 6,199				274
38	Space v27		about 6,199				34
39	Space v33		about 6,199				32
40	Space v35		about 6,199				212
41	Space v36		about 6,199				90
42	Space v37		about 6,199				92
43	Space v38		about 6,199				36

multiple faulty versions into consideration.

Another effectiveness metric is the P-measure, which is the probability of detecting at least one failure using a set of test cases [31]. By definition, the P-measure will increase when the number of selected test cases increases, and the difference in P-measure between two techniques also depends on the number of selected test cases. Hence, a range of sizes of test sets need to be used when comparing P-measures. Normally, the P-measure is used to evaluate test case selection (rather than prioritization) techniques where the size of the test set is supposed to be meaningful. Consider, for example, a scenario in branch coverage testing in which the test set achieves 100% branch coverage. Then, we can select the same number of random test cases and use the P-measure to compare the effectiveness of the branch coverage and random testing techniques. For the present study, which is on test case prioritization, the P-measure is obviously not as suitable as the F-measure.

For each TCP technique, we record the total CPU time (including the time spent in test case selection / prioritization and in SUT execution) consumed to detect the first failure.

To make statistically meaningful comparisons, $10,000$ trials were conducted every time random prioritization or dispersity-based prioritization (which is an improved random technique) was performed, and the F-measure and the mean APFD of the two prioritization methods were computed.

For random testing by sampling *with* replacement, its F-measure is given by $\frac{1}{\theta}$, where θ is the failure rate. In the context of test case prioritization, however, test case sampling should be performed *without* replacement. In this situation, the F-measure of random prioritization can be easily calculated using the approach developed by Zhou [17, Equation (2)]. Despite the existence of this analytical solution, in our empirical study, for both the random and the dispersity-based prioritization algorithms, we recorded the actual number of test cases executed to detect the first failure, for each of the $10,000$

trials (this treatment allowed us to conduct further statistical analysis on the results). This number of test case executions in a particular trial is referred to as the "F-count" [35] of that trial. In the rest of this paper, we use the term "F-measure" less rigorously to refer to the mean F-count over all trials performed using a particular TCP algorithm for a particular version of the SUT.

Compared with the random and dispersity-based prioritization, the *additional* algorithm is basically a deterministic algorithm. Therefore, any experiment with the *additional* algorithm involved only one trial.

To evaluate the applicability of a test case prioritization technique, we inspect the subject packages to decide whether the technique can be applied. For random and dispersity-based prioritization algorithms, this process is trivial: We just need to ensure that an ordering of test case executions can be generated. For the *additional* algorithm, we need to check whether test case coverage data can be collected.

4.3.3 Challenges

There are challenges in controlled experiments with real-world software packages. First, in order to compare the fault-detection effectiveness of different test case prioritization methods, the test suite must be able to detect at least one failure of the SUT. For some of the software packages investigated, we found that if the test suite and the SUT were from the same version, no failure could be detected. This is because the SUT had already been thoroughly tested against the test suite and passed all of the tests before the package was released.

In a typical development project, when changes are made to existing software, *regression testing* is normally performed by running "old" test cases created for previous versions. This is to ensure that the changes do not harm the existing functionality. Following this practice, we tried to apply old test suites to newer versions of the SUT to address the "no failure" problem described above. However, we found that this approach could not detect any failure either. This is because newer versions of the SUT must have already gone through regression testing and passed all of the test cases before they were released.

When no failures could be detected, researchers in software testing typically seed artificial defects into the SUT [3]. However, in the present research, our objective is to investigate real-life projects with real-life test suites that can detect real failures. In empirical software engineering research, the subjects and objects must be representative of the population [36]. Therefore, instead of seeding artificial faults into the SUT, we decide to use the following strategy: If no failure can be detected, then the test suite will be applied to an earlier version of the SUT. We find that this strategy can produce failures and that this is probably the best solution if one wants to experiment without seeding artificial faults. This treatment is valid as TCP techniques can be applied in many different scenarios that include but are not limited to regression testing—for instance, when the testing objective is *program comprehension* or *change impact analysis* by means of dynamic analysis with test cases [37], or to find *behavioral differences* between different versions of the SUT [38], [39].

Generally speaking, applying a test suite to a different version of the SUT could have compatibility problems during test case executions. In this research, we have carefully considered the compatibility issue and managed to select compatible SUT and test suite versions so that the experiments can be successfully carried out.

Using the same test suite to test different versions of the SUT is also a basic technique and requirement in order to measure APFD [15], which imposes another (and greater) challenge. The measurement of APFD requires that a number of different versions of the SUT be tested using the same test suite, but in some practical situations this turned out to be impossible due to compatibility problems between the test suite and multiple versions of the SUT (such as abortion of execution caused by unrecognized parameters). As a result, APFD was not applied to such programs, listed in rows #1 to #11 in Table 1.

4.3.4 Subject packages

The following is a brief introduction of the projects listed in Table 1. Readers may refer to the project websites, as given in the table, for more information. Our host machine runs Microsoft Windows 7 Ultimate, on top of which we use a virtual machine (VMware Workstation) to run Ubuntu. On this platform the SUTs are installed.

The first project, SQLite (row #1), is claimed to be "the most widely deployed SQL database engine in the world." The test suites of SQLite meet many adequacy criteria such as "100% branch test coverage," "boundary value tests," and so on [40]. SQLite has three independent test harnesses. We were able to download one of them, namely, the TCL Tests, which is in the public domain. The test suite consists of 787,530 test cases, many of which can run multiple times with different parameters to generate several million more test cases. In this study, we focus on the original 787,530 test cases only (where the test oracle is embedded). Note that the size of the test suite (that is, the number of test cases) of SQLite is much larger than the size of the SUT measured in SLOC (140,603). Readers may refer to related literature [18], [24] for more discussions about the importance of using large test suites in empirical studies of test case prioritization techniques.

Rows #2 to #6 list five projects. They are sub-projects of the GNU Compiler Collection (GCC), where g++ and gcc have emphases on compiling C++ and C programs, respectively, gfortran is a Fortran compiler, libmudflap is a runtime library, and libstdc++ is a standard C++ Library. The total size of GCC (containing all of the above five projects) is 4,781,336 SLOC. For these five projects, the test suites and the SUTs are of the same version, namely, GCC v4.8.0.

GCC provides a test driver to run test cases. The test driver has an automated oracle to verify the output of each test case execution, in terms of "expected pass" (this is the result yielded by the majority of the test cases), "expected failure," "unexpected failure," "unexpected pass," "unresolved," and "unsupported." In this research, we treat both "expected pass" and "expected failure" as a *passed test* because they are both expected behavior, and "unexpected failure" and "unexpected pass" as a *detected failure* because they are both unexpected behavior. An "unexpected failure" or "unexpected pass" may not necessarily indicate a fault of the SUT. Instead, they are more likely related to the environmental issues of the platform. Further discussion on the causes of the failures is beyond the scope of the paper. Readers may refer to the GCC website for more information. The small number of "unresolved" and "unsupported" test cases were excluded from the study.

The test oracles of projects listed from row #7 to row #26 are also embedded in their test suites. Rows #7 and #8 list the Apache software projects Commons Lang and Commons Math. The former provides extra methods for the manipulation of core Java classes. The latter is a library of lightweight, self-contained mathematics and statistics components. JFreeChart in row #9 is a Java chart library for developers to display professional quality charts in their applications. Joda-Time in row #10 provides a quality replacement for the Java date and time classes.

Firefox in row #11 is a popular web browser. It is free and open source software. The Firefox package includes several test suites, and we used the largest one, named "mochitest-plain," to conduct experiments. Firefox provides a test driver to run all test cases in the test suite, together with an automated test oracle to verify each and every test result in terms of pass, known failure, unexpected failure, and unexpected pass. Similar to the case of GCC, we treat both "pass" and "known failure" as a *passed test*, and "unexpected failure" and "unexpected pass" as a *detected failure*.

Autoconf (rows #12 to #16) is, according to its website, "an extensible package of M4 macros that produce shell scripts to automatically configure software source code packages. These scripts can adapt the packages to many kinds of UNIX-like systems without manual user intervention." We were able to run the same test suite on five versions of Autoconf to calculate APFD. Automake (rows #17 to #21) is a tool for "automatically generating Makefile.in files compliant with the GNU Coding Standards. Automake requires the use of Autoconf." We were able to run the same test suite on five versions of Automake. Rows #22 to #26 list five versions of MySQL (Community Server), which is claimed to be "a freely downloadable version of the world's most popular open source database that is supported by an active community of open source developers and enthusiasts." We were able to run the same test suite on five versions of MySQL.

The last project is Space (rows #27 to #43), downloaded from SIR [41], [42]. The Space program was developed by Ingegneria Dei Sistemi, Pisa, Italy for the European Space Agency [43]. The program consists of about 6,199 SLOC written in C, and works as an interpreter for an array definition language (ADL). The downloaded package includes a base version and 38 faulty versions. Each faulty version contains a real fault discovered during the development of the program. In addition, the downloaded package contains a pool of $13,551$ test cases. According to SIR [42], the test pool was constructed in two stages: An initial pool of $10,000$ test cases was created using a test case generator, capable of generating "random" ADL input files [43]. Then more test cases were added to the pool so that each executable statement of the base program or edge of its control flow graph would be exercised by at least 30 test cases. In our empirical study with Space, the testing objective is to detect failures of the faulty versions as quickly as possible by running test cases from the existing test suite. In order to detect failures, the base version (not shown in Table 1) was used as a test oracle: Every time a test case is run, the output of the faulty version is compared with that of the base version, and any discrepancy means a failure. [4] It was found that, out of the 38 faulty versions, some were equivalent to the base version for all the test cases and, hence, they were excluded from the study. Furthermore, we excluded those faulty versions whose failure rate is higher than 5%. This is because, from the practicing testers' perspective, programs with small failure rates are more interesting than those with high failure rates, as failures of the latter can be detected easily by any testing technique. As a result, only 17 faulty Space versions with failure rates between 0 and 5% were included in our study. The failure rate threshold was set to 5% because "1 in 20" is normally considered a small probability, which forms the basis of modern statistics [44].

4.4 Deciding the order of test cases

The computation of the natural distance depends on the position of the test cases in the test suite, that is, the order of the test cases. Section 3 described several approaches to identifying such an order, and some of these approaches have been applied to our subject test suites.

For SQLite (row #1 of Table 1), test case executions are performed by a set of test scripts spread over 707 files. We found that seven files do not contain their own test cases, 12 files do not print proper test results, and one file is unstable in the sense that it executes a different number of test cases every time. These files were, therefore, excluded from the study. As a result, a total of 687 files were used to run a total of 787,530 test cases. For test cases within a test script file, their order is defined by their execution order, whereas the

4. In controlled experiments where different testing techniques are compared, it is a common practice to use the base version as an oracle to enable quick verification of large amounts of outputs of the faulty versions.

```
Purpose: To decide the order of test cases.
Begin Procedure
1.  If (there is no test driver file) then
2.      use the alphanumeric order of the test case files;
                /*This is because, for our subject packages, the original file creation dates
                  are unknown.*/
3.      return;
4.  Endif;
5.  If (there is only one test driver file) then /*So, there is only one test suite.*/
6.      use the test case execution order given by the test driver;
7.      return;
8.  Endif;

    /*Now there are multiple test driver files (hence multiple test suites), each of which
      runs a number of test cases. This is the case of SQLite.*/

9.  set the between-group order (the order of test suites) to be the alphanumeric order of
    the test driver filenames;
10. set the within-group order (the order of test cases within a test suite) to be the test
    case execution order given by the respective test driver;
11. return;
End of Procedure
```

Fig. 5: Deciding the order of test cases for each subject package used in our empirical studies.

order of test script files is defined by the alphanumeric order of their filenames. The alphanumeric order that we adopted is similar to alphabetical order except that the latter simply sorts the values left to right, character by character, whereas the former recognizes numeric values in filenames. For example, we used the alphanumeric order (a.test, a1.test, a2.test, a100.test) rather than the alphabetical order (a.test, a1.test, a100.test, a2.test). It is our intuition that the alphanumeric order of filenames carries useful information about the similarity among test script files. For instance, "corrupt8.test," "corrupt9.test," "delete2.test," and "delete3.test" are some of the filenames of real test scripts. Intuitively, "corrupt8.test" and "corrupt9.test" should have some similarity in certain ways (such as the testing purpose); "delete2.test" and "delete3.test" should also be similar in certain ways (such as in the operations performed by the test cases). The same treatment was also applied to the Space package (rows #27 to #43). Each test case of the Space program is an ADL file, and the alphanumeric order of the filenames was taken as the order of the test cases.

The alphanumeric order of filenames was also used to list the test case files for Automake (rows #17 to #21). Each of these files is a test script with a filename extension ".sh."

It is straightforward to identify the order of the test cases for projects listed from row #2 to row #16 and from row #22 to row #26 because these projects have a test driver and, therefore, the sequence used by the driver to execute the test cases is directly taken as the order of the test cases.

The procedure of deciding the order of test cases for the above subject packages is summarized in Fig. 5.

5 Empirical Results

We first compare our test case prioritization approach (DBP) with random prioritization (RP) in Section 5.1, and then with the *additional* algorithm in Section 5.2.

5.1 Comparing DBP with random prioritization (RP)

The research question RQ1 raised in Section 1 identified four aspects of a test case prioritization method. We first look at applicability. DBP and RP have the same applicability, and both these two techniques are applicable to all 43 subject programs. This is because both DBP and RP can be applied whenever an original sequence of test cases—(t_0, t_1, ..., t_{n-1}), as shown in Fig. 4—is given.

We next discuss effectiveness, efficiency, and execution time for failure detection.

5.1.1 Effectiveness

We first present the F-measure results and then the APFD results.

5.1.1.1 *The F-measure results*: Empirical results of testing effectiveness are summarized in Table 2. Columns #3 and #4 show the F-measures of random prioritization (F.RP) and DBP (F.DBP), respectively (out of $10,000$ trials each). To compare these two scores, the ratio F.DBP $\div$ F.RP is calculated and listed in column #5. When the ratio is smaller than 1, F.DBP is better. When it is greater than 1, F.RP is better. To conduct statistically meaningful comparisons, independent-samples t-tests at a significance level of 5% are performed and the (two-tailed) p-values are listed in column #6, together with the respective *effect size* (Cohen's d) [45]: d is the absolute value of the difference of the two means divided by the square root of the mean of the two variances. A p-value below 0.05 indicates that the difference between F.RP and F.DBP is statistically significant, and a p-value above 0.05 indicates that these two F-measures are equal (that is, there is no statistically significant difference). For ease of reading, the corresponding cells in columns #5, #6, #8, #9, and #10 are highlighted when the respective p-values are below 0.05—in this way, all statistically significant results

TABLE 2
F-measure and APFD results. F.RP: F-measure of random prioritization (RP) out of 10,000 trials; F.DBP: F-measure of dispersity-based prioritization (DBP) out of 10,000 trials; F.Add: F-measure of the *additional* algorithm; APFD.RP: mean APFD of RP out of 10,000 trials; APFD.DBP: mean APFD of DBP out of 10,000 trials; APFD.Add: APFD of the *additional* algorithm. Where there is a statistically significant difference between the RP and DBP means (with a p-value below 0.05), the corresponding cells are highlighted. "p=0.000" means p$<$0.0005. Where the effect size (Cohen's d) is 0.10 or larger, the corresponding cells are starred (*). "d=0.00" means d$<$0.005.

1	2	3	4	5	6	7	8	9	10	11
row #	project name & version	F.RP	F.DBP	F.DBP ÷ F.RP	p-value (2-tailed) for F-measure, and effect size (d)	F.Add	APFD.RP	APFD.DBP	p-value (2-tailed) for APFD, and effect size (d)	APFD.Add
1	SQLite v3.7.10	57,401.31	55,304.55	0.96	p=0.005, d=0.04	n/a	n/a	n/a	n/a	n/a
2	g++, GCC v4.8.0	363.36	151.51	0.42	p=0.000, d=0.76 *	n/a	n/a	n/a	n/a	n/a
3	gcc, GCC v4.8.0	1,001.68	979.18	0.98	p=0.104, d=0.02	n/a	n/a	n/a	n/a	n/a
4	gfortran, GCC v4.8.0	2,575.75	2,570.67	1.00	p=0.882, d=0.00	n/a	n/a	n/a	n/a	n/a
5	libmudflap, GCC v4.8.0	360.25	358.74	1.00	p=0.699, d=0.01	n/a	n/a	n/a	n/a	n/a
6	libstdc++, GCC v4.8.0	1,704.33	1,302.08	0.76	p=0.000, d=0.32 *	n/a	n/a	n/a	n/a	n/a
7	commons-lang v3.0.1	686.26	688.97	1.00	p=0.693, d=0.01	n/a	n/a	n/a	n/a	n/a
8	commons-math v3.1	911.13	920.81	1.01	p=0.360, d=0.01	n/a	n/a	n/a	n/a	n/a
9	jfreechart v1.0.14	201.43	205.57	1.02	p=0.115, d=0.02	n/a	n/a	n/a	n/a	n/a
10	joda-time v2.0	436.74	437.31	1.00	p=0.917, d=0.00	n/a	n/a	n/a	n/a	n/a
11	Firefox v31.0	2,876.61	1,663.14	0.58	p=0.000, d=0.52 *	n/a	n/a	n/a	n/a	n/a
12	Autoconf v2.64	6.19	5.93	0.96	p=0.000, d=0.05	n/a				
13	Autoconf v2.65	6.72	6.48	0.96	p=0.003, d=0.04	n/a				
14	Autoconf v2.66	14.13	14.01	0.99	p=0.519, d=0.01	n/a	0.972397	0.973838	p=0.000, d=0.07	n/a
15	Autoconf v2.67	17.29	17.47	1.01	p=0.424, d=0.01	n/a				
16	Autoconf v2.68	25.41	24.59	0.97	p=0.013, d=0.04	n/a				
17	Automake v1.13	3.94	3.75	0.95	p=0.000, d=0.06	1				
18	Automake v1.13.1	3.92	3.74	0.95	p=0.000, d=0.06	1				
19	Automake v1.13.2	4.01	3.79	0.94	p=0.000, d=0.07	1	0.990710	0.991223	p=0.000, d=0.07	0.999619
20	Automake v1.13.3	22.93	21.58	0.94	p=0.000, d=0.06	1				
21	Automake v1.13.4	27.88	27.32	0.98	p=0.132, d=0.02	1				
22	MySQL v5.6.7	7.03	7.02	1.00	p=0.929, d=0.00	n/a				
23	MySQL v5.6.8	9.89	9.65	0.98	p=0.066, d=0.03	n/a				
24	MySQL v5.6.9	13.26	12.74	0.96	p=0.004, d=0.04	n/a	0.990453	0.991448	p=0.000, d=0.15 *	n/a
25	MySQL v5.6.10	20.01	18.93	0.95	p=0.000, d=0.06	n/a				
26	MySQL v5.6.11	72.68	64.79	0.89	p=0.000, d=0.12 *	n/a				
27	Space v3	21.21	19.48	0.92	p=0.000, d=0.09	5				
28	Space v7	82.78	79.26	0.96	p=0.002, d=0.04	8				
29	Space v8	141.50	141.92	1.00	p=0.831, d=0.00	17				
30	Space v12	405.76	231.97	0.57	p=0.000, d=0.54 *	380				
31	Space v16	27.08	26.03	0.96	p=0.004, d=0.04	2				
32	Space v17	68.82	67.98	0.99	p=0.374, d=0.01	244				
33	Space v18	394.83	231.62	0.59	p=0.000, d=0.52 *	380				
34	Space v20	63.95	64.76	1.01	p=0.365, d=0.01	230				
35	Space v21	64.39	64.12	1.00	p=0.758, d=0.00	230	0.986551	0.990271	p=0.000, d=0.88 *	0.984683
36	Space v22	207.88	132.36	0.64	p=0.000, d=0.44 *	8				
37	Space v23	49.74	49.62	1.00	p=0.862, d=0.00	9				
38	Space v27	388.47	246.21	0.63	p=0.000, d=0.44 *	1,802				
39	Space v33	408.16	195.58	0.48	p=0.000, d=0.67 *	143				
40	Space v35	63.60	63.50	1.00	p=0.912, d=0.00	43				
41	Space v36	150.54	148.30	0.99	p=0.279, d=0.02	6				
42	Space v37	148.21	148.80	1.00	p=0.773, d=0.00	1				
43	Space v38	367.45	331.01	0.90	p=0.000, d=0.11 *	29				
			avg	0.90						
			max	1.02						
			min	0.42						

are highlighted. While the p-value measures the statistical significance, the effect size (d) measures the practical significance and should be judged in context [46]. In the context of comparing test case prioritization techniques, a d value of 0.1 or above can be considered nontrivial [47]. Therefore, in columns #6 and #10 of Table 2, cells with $d \geq 0.10$ are marked with an asterisk (*).

A total of 23 cells are highlighted in columns #5 of Table 2, and the values of these 23 cells are all below 1. This means that DBP outperformed RP in the F-measure across all 23 statistically significant cases. Furthermore, all 10 starred cells in column #6 (indicating a practical significance) fall within these 23 cases.

For the remaining 20 subject programs, the F-measures of RP and DBP can be considered equal. In the best case, the ratio F.DBP ÷ F.RP can be as low as 0.42 (row #2), which means that, on average, DBP used 58% fewer test cases than RP, or RP used 2.4 ($= 363.36 \div 151.51$) times as many test cases as DBP, to detect the first failure. This indicates that DBP achieved a saving that is both statistically significant and practically significant. In the worst case, the ratio F.DBP ÷ F.RP is 1.02 (row #9). This means that, even in the worst case, DBP used on average only 2% more test cases than RP to detect the first failure, and this difference was neither statistically nor practically significant (and hence can be ignored). The above results prove Hypothesis **I**.

It would be useful to look further into the different

TABLE 3
Execution time results (time to the first failure, in seconds). T.RP: mean time to the first failure by RP, out of 10,000 trials (= test case selection time + test case execution time); T.DBP: mean time to the first failure by DBP, out of 10,000 trials (= test case selection time + test case execution time); Exe.Add: time to the first failure by the *additional* algorithm (= test case selection time + test case execution time); Preprocessing.Add: preprocessing time taken by the *additional* algorithm to collect test case coverage data. Where there is a statistically significant difference between the RP and DBP means (with a p-value below 0.05), the corresponding cells are highlighted (light gray: DBP outperformed RP; dark gray: RP outperformed DBP). "p=0.000" means p$<$0.0005. Where the effect size (Cohen's d) is 0.10 or larger, the corresponding cells are starred (*). "d=0.00" means d$<$0.005.

1	2	3	4	5	6	7	8
row #	project name & version	T.RP	T.DBP	T.DBP ÷ T.RP	p-value (2-tailed) and effect size (d)	Exe.Add	Preprocessing.Add
1	SQLite v3.7.10	71.346673	67.751497	0.950	p=0.000, d=0.05	n/a	n/a
2	g++, GCC v4.8.0	7.632750	3.433284	0.450	p=0.000, d=0.70 *	n/a	n/a
3	gcc, GCC v4.8.0	26.334649	26.229569	0.996	p=0.777, d=0.00	n/a	n/a
4	gfortran, GCC v4.8.0	81.503814	81.359644	0.998	p=0.894, d=0.00	n/a	n/a
5	libmudflap, GCC v4.8.0	14.379306	14.618975	1.017	p=0.129, d=0.02	n/a	n/a
6	libstdc++, GCC v4.8.0	304.885681	234.552912	0.769	p=0.000, d=0.32 *	n/a	n/a
7	commons-lang v3.0.1	32.739562	32.151085	0.982	p=0.074, d=0.03	n/a	n/a
8	commons-math v3.1	70.945570	70.646528	0.996	p=0.718, d=0.01	n/a	n/a
9	jfreechart v1.0.14	0.629425	0.616548	0.980	p=0.362, d=0.01	n/a	n/a
10	joda-time v2.0	1.060908	1.075443	1.014	p=0.506, d=0.01	n/a	n/a
11	Firefox v31.0	16.024432	10.076629	0.629	p=0.000, d=0.41 *	n/a	n/a
12	Autoconf v2.64	4.836720	4.564996	0.944	p=0.000, d=0.06	n/a	
13	Autoconf v2.65	5.089850	4.812579	0.946	p=0.000, d=0.06	n/a	
14	Autoconf v2.66	10.907391	10.617755	0.973	p=0.047, d=0.03	n/a	n/a
15	Autoconf v2.67	13.418242	13.312348	0.992	p=0.557, d=0.01	n/a	
16	Autoconf v2.68	19.816506	18.837227	0.951	p=0.000, d=0.05	n/a	
17	Automake v1.13	2.684771	2.432545	0.906	p=0.000, d=0.06	5.536	
18	Automake v1.13.1	2.721153	2.504413	0.920	p=0.000, d=0.05	5.580	
19	Automake v1.13.2	2.822524	2.551866	0.904	p=0.000, d=0.06	5.600	9269
20	Automake v1.13.3	16.530596	15.554517	0.941	p=0.000, d=0.06	**5.620**	
21	Automake v1.13.4	19.992300	19.456912	0.973	p=0.073, d=0.03	**5.564**	
22	MySQL v5.6.7	20.215466	20.109030	0.995	p=0.863, d=0.00	n/a	
23	MySQL v5.6.8	21.489901	21.112889	0.982	p=0.260, d=0.02	n/a	
24	MySQL v5.6.9	29.073387	28.535064	0.981	p=0.222, d=0.02	n/a	n/a
25	MySQL v5.6.10	37.554342	35.821509	0.954	p=0.002, d=0.04	n/a	
26	MySQL v5.6.11	132.062247	118.960734	0.901	p=0.000, d=0.10 *	n/a	
27	Space v3	0.004122	0.003869	0.939	p=0.000, d=0.07	1.537	
28	Space v7	0.014429	0.014070	0.975	p=0.070, d=0.03	1.845	
29	Space v8	0.023454	0.023929	1.020	p=0.151, d=0.02	2.806	
30	Space v12	0.087596	0.050711	0.579	p=0.000, d=0.53 *	36.713	
31	Space v16	0.005111	0.004974	0.973	p=0.0498, d=0.03	1.188	
32	Space v17	0.010594	0.010680	1.008	p=0.563, d=0.01	24.387	
33	Space v18	0.069770	0.041618	0.596	p=0.000, d=0.50 *	36.768	
34	Space v20	0.025857	0.025962	1.004	p=0.772, d=0.00	22.965	
35	Space v21	0.012606	0.012803	1.016	p=0.270, d=0.02	22.900	5
36	Space v22	0.045247	0.029189	0.645	p=0.000, d=0.43 *	1.854	
37	Space v23	0.014726	0.015314	1.040	p=0.006, d=0.04	1.955	
38	Space v27	0.077955	0.050109	0.643	p=0.000, d=0.43 *	160.595	
39	Space v33	0.074699	0.036271	0.486	p=0.000, d=0.66 *	14.863	
40	Space v35	0.011852	0.012003	1.013	p=0.361, d=0.01	5.376	
41	Space v36	0.025377	0.025727	1.014	p=0.319, d=0.01	1.633	
42	Space v37	0.045219	0.045650	1.010	p=0.493, d=0.01	1.088	
43	Space v38	0.063167	0.058255	0.922	p=0.000, d=0.08	4.037	
			avg	0.905			
			max	1.040			
			min	0.450			

orders of test case executions. Consider SQLite, for example. We first ordered the 687 test script files according to their filenames, and then the test cases within each file according to their execution order. Do either or both of these two strategies play an important role? We find that both these two strategies have contributed to the test case prioritization. If we apply DBP/RP to prioritize the 687 test script files only and then the test cases within each file are still executed sequentially, then there is no significant difference between the F-measures of DBP and RP. This is because the 13 failure-causing test cases of SQLite are distributed over three test script files and these three files are *not* clustered (that is, they are not neighbors in the test file sequence). It is known that, for non-clustered failure patterns, ART and RT have similar F-measures [48]. Likewise, if we execute the 687 test script files sequentially and apply DBP/RP to the test cases within each test script file only, then there is no significant difference between the F-measures of DBP and RP either. This is because the first test script file in the sequence to fail SQLite includes only one failure-causing test case and, therefore, applying either DBP or RP to

the test cases within this file makes no difference. Due to the sheer size and complexity of the real-world test suites involved in our experiments, this research will not attempt to further investigate the inner workings of the different orderings of test case executions—such an investigation would require a separate study that is beyond the scope of this paper.

5.1.1.2 *The APFD results*: As explained previously, APFD was not applied to programs listed in rows #1 to #11 of Table 2 because of compatibility problems of the real-world test suites across multiple versions of the SUT. This observation indicates that the APFD metric may not be as applicable as the F-measure metric.

For RP and DBP, available mean APFD scores are given in rows #12 to #43 of columns #8 and #9 of Table 2 for four projects. In all four projects, DBP outperformed RP with a constantly higher mean APFD score. Furthermore, as shown in column #10, the APFD differences between RP and DBP were statistically significant across all four projects and, therefore, the relevant cells are highlighted. Furthermore, two of these four projects are starred as they had a nontrivial d value (as shown in column #10).

While we report that DBP outperformed RP in APFD with a statistical significance across all four projects and with a practical significance in two of these four projects, we also wish to point out that all observed APFD scores are very large and that the differences between APFD.RP and APFD.DBP are very small. **This observation does not mean that RP and DBP have little difference in TCP effectiveness, because their F-measures differ a lot.** This phenomenon of large APFD scores associated with large test suites was first reported by Zhou et al. [18], where it was suggested that "**APFD may not necessarily be a suitable effectiveness measure or may need to be adjusted in certain situations, such as when the test suites are very large.**" Nevertheless, the outcomes of the APFD comparisons and those of the F-measure comparisons have been quite consistent: Both these metrics show that DBP outperformed RP with a statistical significance in testing effectiveness, and that 43 to 50 percent of these statistically significant cases were also practically significant.

5.1.2 Efficiency

DBP and RP have the same order of computational complexity, $O(n)$, where n is the number of test cases prioritized. Further comparison of their efficiency is presented in Section 6.

5.1.3 Execution time for failure detection

We next compare their actual execution times for the detection of the first failure. The empirical results are shown in Table 3. T.RP (column #3) and T.DBP (column #4) are the mean execution times (out of $10,000$ trials) spent by RP and DBP, respectively, to detect the first failure, including the SUT execution time and the test case selection time.

To compare T.RP and T.DBP, the ratio T.DBP $\div$ T.RP is calculated and listed in column #5. When the ratio is smaller than 1, T.DBP is better. When it is greater than 1, T.RP is better. To conduct statistically meaningful comparisons, independent-samples t-tests at a significance level of 5% are performed and the (two-tailed) p-values are listed in column #6. Cells in columns #5 and #6 are highlighted when their respective p-values are below 0.05, indicating statistically significant differences. The cells highlighted in light gray indicate that DBP outperformed RP with a statistical significance, and that highlighted in dark gray indicates that RP outperformed DBP with a statistical significance. Cells in column #6 are starred when their respective d values are 0.10 or larger, indicating a nontrivial practical significance.

A total of 23 cells are highlighted in column #5 of Table 3, including 22 in light gray and only one in dark gray (in row #37). A total of 9 cells are starred, all of which fall within the 22 light gray cases.

The above results indicate that DBP outperformed RP in terms of spending less time to detect the first failure, quite consistent with the F-measure results. For the remaining 20 subject programs, the mean execution times of RP and DBP can be considered equal. In the best case, the ratio T.DBP $\div$ T.RP can be as low as 0.45 (row #2), hence a saving of 55%. In the worst case, the ratio is 1.04 (row #37), indicating only 4% extra time.

5.1.4 Summary

In summary, DBP has the same applicability and the same order of computational complexity as RP, but significantly outperformed RP in both effectiveness (in terms of the F-measure and APFD) and execution time for failure detection. DBP often achieved large savings in both the F-measure and the execution time. Overall, its F-measure performance (in terms of the F.DBP $\div$ F.RP ratio) was slightly better than its execution time performance (in terms of the T.DBP $\div$ T.RP ratio), and this was expected because the DBP test case selection algorithm involves more computations than RP although they are both linear algorithms. Nevertheless, it is interesting to observe that, for some programs, such as Autoconf and Automake, DBP's execution time performance was better than its F-measure performance. This was because DBP selected test cases that consumed less execution time than those selected by RP.

5.2 Comparing DBP with the *additional* algorithm

This section addresses the research question RQ2.

5.2.1 Applicability

While DBP was applicable to all the subject programs, we found that the *additional* algorithm has limitations in its applicability. This is because the *additional* algorithm requires that the test case coverage data be known *prior to* test case prioritization. In the context of regression testing, if a test suite was used in the past on a previous

version of the SUT, then the previous test coverage data could be used by the *additional* algorithm, in order to prioritize the test cases for the new version of the SUT [15]. We found that such a strategy has its limitations when applied to our real-world subject programs. As explained previously, for packages listed from row #1 to row #11 in Table 2, the same test suites could not run across multiple SUT versions due to compatibility problems. Therefore, the *additional* algorithm could not be applied to these packages. Furthermore, for the packages Autoconf and MySQL, test case coverage data could not be obtained (such as by using *gcov*, a standard utility used in concert with GCC). As a result, we were only able to apply the *additional* algorithm to the Automake programs (rows #17 to #21) and the Space programs (rows #27 to #43).

For Automake, we managed to run the test suite on a previous version of the SUT, namely, v1.12.6, to collect coverage data so that the *additional* algorithm could be applied. For Space, the base version Space program was used to collect the test case coverage data so that the *additional* algorithm could be applied to the faulty Space versions.

5.2.2 Effectiveness

The effectiveness of the test case prioritization methods was evaluated using the F-measure and APFD. For the Automake package, the *additional* algorithm had a perfect F-measure of 1 across all five versions of the SUT (see the F.Add scores in column #7 of rows #17 to #21, Table 2), as well as the best APFD (column #11).

For the Space package, the results are more involved. On the one hand, out of the 17 faulty versions of the SUT, the *additional* algorithm outperformed DBP in the F-measure for 11 versions, whereas DBP outperformed the *additional* algorithm for only six versions; in the best case, the *additional* algorithm again achieved a perfect F-measure of 1 (row #42). On the other hand, DBP outperformed the *additional* algorithm in terms of APFD: the APFD of DBP (column #9) is 0.990271, which is the best among all three algorithms. As mentioned previously, it is our observation that, when the test suites are large, the APFD scores of different TCP techniques are very close, and therefore APFD may not necessarily be a suitable effectiveness measure in this situation. In any case, in terms of *effectiveness*, the *additional* algorithm had better F-measures than DBP in the majority of cases. This is because the *additional* algorithm makes use of test case coverage information, whereas DBP does not use such information.

It should be noted that *t*-tests were not performed for comparisons with the *additional* algorithm. This is because, as a reminder, *additional* is basically a deterministic algorithm and, therefore, only one trial was run for each version of the SUT; in contrast, 10,000 trials of DBP were run and a mean was calculated.

5.2.3 Efficiency

As mentioned previously, the (branch-coverage-based) *additional* algorithm has a time complexity of $O(n^2m)$, where n is the number of test cases in the test suite and m is the number of branches of the SUT. In comparison, DBP has a linear complexity $O(n)$, where n is the number of prioritized test cases.

5.2.4 Execution time for failure detection

Exe.Add (column #7 of Table 3) is the execution time spent by the *additional* algorithm to detect the first failure, which includes the SUT execution time and the test case selection time.

Furthermore, as explained previously, the *additional* algorithm requires that the test case coverage data be made available prior to test case prioritization, as thus we collected such data by running all the test cases on a previous (instrumented) version of the SUT, and the time taken is shown as Preprocessing.Add in column #8 (it may not necessarily be counted as a cost of the *additional* algorithm as we assume that such information is available before the start of the TCP process). For Automake, each of its test case files allowed the user to either enable or disable the test coverage generation feature. To collect T.RP and T.DBP, we disabled this feature in order to run the test cases as fast as possible; to collect Preprocessing.Add, we enabled this feature and, subsequently, the system generated a total of 1,313 HTML files corresponding to the coverage profiles of the 1,313 individual test cases, at a cost of 9,269 seconds as shown in Table 3. Note that this does not include the execution time of our own code that read the 1,313 HTML files to extract the branch coverage data. For the Space program, we used the *gcov* tool to collect the test coverage data. The time (5 seconds) shown in Table 3 does not include the execution time of our own code that processed the intermediate system files to extract the branch coverage data after each test case execution. Even if we do not consider this Preprocessing.Add, DBP still outperformed the *additional* algorithm very obviously across all subject programs except on Automake v1.13.3 and v1.13.4 (rows #20 and #21), where the two Exe.Add scores are printed in bold.

5.2.5 Summary

In summary, the empirical results show that DBP strongly outperformed the *additional* algorithm with respect to *applicability* and *execution time for failure detection*, in addition to *efficiency*. On the other hand, it is hardly surprising from the results that the heavyweight *additional* algorithm was generally more *effective* than our lightweight approach. In any case, it is pleasing to find that DBP was more *effective* than the *additional* algorithm in 27.3% $(= 6 \div (5+17))$ of the situations where the latter was applicable.

6 Further Comparison of Test Case Prioritization Efficiency between RP and DBP

Although DBP has the same order of computational complexity as RP, the former involves more steps in selecting a test case. In this section, therefore, we compare the execution times of the DBP and RP algorithms for prioritizing the same number of test cases, **without involving actual test suites, SUTs, or failure detections**. We considered the following test suite sizes: 1000, $10,000$, $100,000$, $1,000,000$, and $10,000,000$. For each test suite size, we ran the RP and DBP algorithms to prioritize the entire test suite[5] and recorded the times taken. For each test suite size and each algorithm, 1000 trials were conducted. The mean values of the execution times are compared as shown in Table 4. They are named "driver time" because the results do not involve any actual SUT execution.

The results of Table 4 indicate that RP can be up to 63 times faster than DBP to prioritize the same number of test cases. However, if we compare Tables 2 and 3, we can see that this advantage of RP does not help much with its overall failure-detection time as compared with DBP. This is because, for real-world large and complex software, the time consumed by the linear-complexity RP and DBP test drivers for test case selection is trivial when compared with the time spent for the actual test case execution.

More formally, let x be the average time consumed by the RP test driver to select a test case, then, according to Table 4, that of DBP can be up to $63x$. Let y and z be the average time to execute a test case and to verify a test result, respectively. The total time consumed by RP and DBP to process a test case is, therefore, $x+y+z$ and $63x+y+z$, respectively. Let k be the F-measure of RP. According to the "avg" result shown in Table 2, the F-measure of DBP can be estimated as $0.9k$. For RP to more quickly detect a failure than DBP, the following relation must be satisfied: $k(x+y+z) < 0.9k(63x+y+z)$, which gives $y+z < 557x$. This means that, if, on average, the total time of executing a test case and verifying a test result is smaller than $557x$ (which is mainly the time of generating 557 random numbers), then RP will detect a failure more quickly; otherwise, DBP will detect a failure more quickly. For real-world large and complex programs, the test case execution time plus the result verification time is generally far longer than the time of generating 557 random numbers. In these situations, therefore, DBP can be used to replace RP. For instance, in our platform, it takes 0.0000088 seconds to generate 557 random numbers. We have also calculated the mean time of executing a test case for each of the 43 programs under test. Of these 43 mean results, the minimum, maximum, and average are 0.0001507 ($= 0.0000088 \times 17.125$) seconds, 2.7974794 seconds, and 0.4333023 seconds, respectively.

5. It should be noted that no actual test suites were involved in the experiment, because to apply the DBP and RP algorithms does not need to access the actual test cases.

7 Threats to Validity

With regard to the internal validity of this study, which refers to whether causal relations are properly examined, the main concern is the correctness of the software tools that we developed to conduct the experiments. To avoid faults in these software tools, their code has been carefully reviewed and tested. Furthermore, all the resulting F-measures of random prioritization have been checked against the mathematical expectations of F-measures (for random sampling *without* replacement) calculated using the approach developed by Zhou [17, Equation (2)]. The discrepancies are very small and can be ignored. All the intermediate and final empirical data have also been carefully checked for correctness and consistency.

In this research, when performing a t-test, we calculated the p-value and the effect size, but did not analyze the *power* [45]. This is because, when the sample size is large, the observed effect size is a good estimator of the true effect size, but there is no guarantee that the *observed power* is a good estimator of the true power—Yuan and Maxwell [49] provided a detailed examination of the *post hoc power* (the observed power) and concluded that the observed power typically provides little useful information about the true power of a single study.

Independent-samples t-tests have some basic assumptions. First, the observations must be independent. By examining the designs of the experiments, we confirm that this assumption is satisfied. Second, the populations from which the samples are taken should be normally distributed [50]. Note that "it is not in fact necessary for the distribution of the observed data to be normal, but rather the sample values should be compatible with the population (which they represent) having a normal distribution" [51]. If the sample data are approximately normal, then the sampling distribution will be normal, too [50]. Furthermore, according to the *central limit theorem*, the sampling distribution will tend to be normal regardless of the population distribution if the sample size is above 30; in other words, in big samples the sampling distribution tends to be normal anyway regardless of the shape of the data that are actually collected [44]. This means that "if we have samples consisting of hundreds of observations, we can ignore the distribution of the data" [50]. As the sample size used in our experiments (Tables 2 and 3) was 10,000, the assumption of normality is satisfied. The third assumption is *homogeneity of variance*, that is, variances in groups should be approximately equal. We found that our experimental data do not perfectly satisfy this assumption. Nevertheless, for a large number of different situations the t-test is robust enough even if the assumption of homogeneity of variance is untenable [52]. This includes situations, for example, where sample sizes are large (above 25 or 30) and equal or nearly equal, and two-tailed hypothesis is considered

[52]–[54]. In the present research, all sample sizes are equal and large, and two-tailed hypothesis has been used. Furthermore, in situations where equal variances are not assumed in a t-test, the SPSS software package provides users with an alternative t-value which compensates for the fact that the variances are not the same [55].

With regard to external validity, which is concerned with the ability to generalize the findings, it should be noted that all the subject programs used in our empirical studies have been taken from the public domain. To enhance external validity, further studies with programs from other sources will be needed. This will require collaboration work with industry.

8 Related Work

As pointed out earlier in this paper, the present research is fundamentally different from adaptive random testing (ART) because we do not consider the input domain. Nevertheless, it is worthwhile to review some of the research results reported in the ART literature, which may help us to gain a deeper understanding of our TCP approach that generates a sequence of test case IDs based on an ART algorithm (Fig. 4).

ART [28], [29], [56] was developed as an enhancement of random testing (RT). The key intuition of ART is to *evenly* spread test cases throughout the input domain. To implement the concept of "even spread," various approaches have been proposed [29], some of which are summarized as follows: (1) Selection of the best candidate from a set of candidates, such as FSCS-ART; (2) Exclusion [57], which defines exclusion zones around previously executed test cases (that have not detected any failure) to restrict the regions from which the next test case is to be generated. Random inputs are generated one by one until one falls outside of all the exclusion zones and that input is taken as the next test case; (3) Partitioning [58], [59], which divides the input domain into partitions and then selects a partition from which the next test case is to be generated; (4) Test profile [60], which achieves an even spread of test cases using a specially designed and dynamically adjusted test profile (which is different from the uniform test profile of RT); and (5) Metric-driven [61], which uses distribution metrics (that measure the degree of even distribution of a set of points) as criteria for test case selection.

To compare the effectiveness of ART and RT, the F-measure has been the most widely used metric. Various studies have proven that ART is superior to RT in the F-measure and that this advantage of ART "is quite significant and is in no way diminished by any potential challenge to previous experiments' validity" [29].

Our TCP approach shown in Fig. 4 uses the FSCS-ART (with forgetting) algorithm to generate a sequence of test case IDs, where the ART algorithm essentially works on a 1-dimensional input space of integers (namely, $[0,\ n-1]$). With respect to the 1-dimensional input space and the F-measure, previous research [48] revealed that (1) the smaller the failure rate is, the more the FSCS-ART outperforms RT, (2) FSCS-ART has better performance for lower-dimensional input spaces (and the best performance for the 1-dimensional input space), and (3) even in the worst case, the F-measure of FSCS-ART is still very close to that of RT.

9 Discussions

In this section, we revisit the motivation of our research, summarize the findings, present the limitations of our study, and answer several potential questions concerning the validity and completeness of our work.

9.1 Revisit of motivation

While various test case prioritization (TCP) techniques (including input-based ones [21]) have been developed and reported in the literature, their applicability to real-world software projects are limited. This is because, first, different techniques require different types of information and/or dissimilarity metrics on the input values, test cases, or the SUT prior to their application, but in practice such details may not be available. For example, Yoo and Harman [3] pointed out that "component-based software development method tends to result in the use of many black-box components, often adopted from a third party. Any change in the third-party components may interfere with the rest of the software system, yet it is hard to perform regression testing because the internals of the third-party components are not known to their users."

Secondly, the complexity of sophisticated TCP techniques and their computational overhead may be too high for real-life utilization. For example, we have made attempts to use $O(n^2)$ TCP algorithms in SQLite (row #1 of Table 1) but found the execution times to be prohibitive. Yoo and Harman [3] also pointed out that "the shorter life-cycle of software development, such as the one suggested by the agile programming discipline, also imposes restrictions and constraints on how regression testing can be performed within limited resources."

One may argue that the time complexity of TCP algorithms is not too important because test cases are only ordered once and then applied to all subsequent versions of the SUT, so that any excessive time spent in TCP is not an issue. This concept is not valid for the following reasons: First, as reported earlier, we have found that when a test suite contains hundreds of thousands of test cases, the computational overhead of conventional nonlinear TCP algorithms may be prohibitive. Second, test suites are updated throughout the course of software evolution, and hence TCP is not a one-off activity; rather, it is repeated regularly. Finally, if test cases are always prioritized in exactly the same (deterministic) order, some lowly ranked test cases may never be run unless the entire test suite is executed. Therefore, some randomness in TCP techniques is desirable.

In fact, random prioritization (RP) is considered to be one of the most practical solutions in vast majority of

TABLE 4
Comparing the DBP and RP test driver execution times for prioritizing the same number of test cases (1000 trials per algorithm with respect to various test suite sizes). Actual SUTs and failure detections are not involved.

size of (virtual) test suite	mean DBP driver time ÷ mean RP driver time
1,000	40.37
10,000	60.37
100,000	62.65
1,000,000	58.17
10,000,000	32.65

real-life situations, as it is simplest in concept, easiest and cheapest to implement, and most importantly, requires no precondition for adoption. This explains why RP is used as the de facto benchmark in test case prioritization studies.

9.2 Summary of findings

In this research, we raised a challenging question, namely, whether a test case prioritization method can be developed to enhance RP. To achieve this goal, the technique must simultaneously satisfy all the following four requirements as stated in research question RQ1: It should be (i) more effective than RP, (ii) quicker to detect failures than RP, (iii) as efficient as RP, *and* (iv) as readily applicable as RP. We reiterate that *none of the existing techniques in the literature can simultaneously satisfy all these four requirements*.

To address the above research question, we have proposed a solution using the concept of *natural distance* for real-world test suites, and proposed a simple dispersity-based prioritization (DBP) algorithm. Our method does not require any knowledge regarding the software requirements, the SUT, the test history, the test coverage, or even the values of the test cases. Thus, DBP is *non-execution-based* and *non-input-based*, and hence is as applicable as RP.

A series of empirical studies have been conducted with 43 real-world programs collected from the public domain. The results show that our method significantly outperforms RP in effectiveness (in terms of both APFD and the F-measure) and the execution time to detect the first failure. Even in the worst case, the performance of DBP is still close to that of RP, and this observation is consistent with previous research results in the ART literature [48]. It is also shown in Table 2 that the F-measure appears to be a more suitable effectiveness metric than APFD when the test suite is large, as it more clearly shows the differences among different TCP techniques.

In terms of efficiency, it is obvious that the RP algorithm involves fewer computation steps than DBP: To select a test case, the former only needs to generate a random number, whereas the latter needs to generate 10 random numbers and conduct $10 \times 10 = 100$ distance computations. Nevertheless, both of these algorithms have the same order of computational complexity, namely, the linear complexity. Note that we are concerned with the testing of large and complex real-world systems. The SUT execution time, together with the result verification time, is generally far longer than the DBP driver time consumed for test case selection [6]. Hence, the difference in computation times between the RP and DBP algorithms has little impact on their relative overall execution times. This explains why the comparative results of effectiveness and execution times are similar.

The *additional* algorithm is recognized as one of the best traditional TCP algorithms. We have, therefore, compared it with DBP in research question RQ2. We have shown that DBP outperforms the *additional* algorithm in applicability, execution time for failure detection, and efficiency. In terms of effectiveness, it is not surprising to find that our lightweight approach cannot guarantee better effectiveness than the heavyweight additional algorithm. It should be noted that the execution time and effectiveness comparisons with the *additional* algorithm have been performed on only two of 15 projects. Further investigations with more subject programs are warranted in future research.

The comparative results are further summarized in Table 5. In short, with respect to research question RQ1, DBP is shown to simultaneously satisfy all four requirements, and can therefore be considered to be a promising enhancement of RP. In actual practice, testers or developers may often estimate the average time to execute a test case, and such information will be helpful in deciding whether DBP or RP should be used. Our findings further suggest that DBP should be considered as a reference benchmark for the evaluation of new test case prioritization techniques. With respect to research question RQ2, our case studies have also produced useful comparative results.

9.3 Limitations

We have indicated in Observation **II** that it is not difficult to decide on the order among test cases in a real-world test suite. Nevertheless, we have not presented a

6. This is because DBP is a linear algorithm. For nonlinear TCP algorithms, the situation may be very different.

TABLE 5
Summary of results.

	applicability	effectiveness	execution time for failure detection	efficiency
DBP vs. RP	Same.	DBP is better.	DBP is better.	These two algorithms have the same order of (linear) computational complexity, although the RP algorithm consumes less computation time than the DBP algorithm.
DBP vs. *Additional*	DBP is better.	*Additional* is better.	DBP is better.	DBP is better: DBP is in $O(n)$, *Additional* is in $O(n^2m)$.

systematic methodology. Indeed, some of the discussions in this paper may be slightly oversimplified. For instance, the naming convention for test cases varies considerably among different programming languages, projects, frameworks, and organizations. We have not studied the naming patterns or their implications. In fact, test case priority determined by different heuristics may result in distinctive performances. In future research, we plan to improve the DBP strategy by leveraging the structural information of test suites. For example, it is common in Java projects to have one test suite per class with multiple method invocations with the same naming as the source methods. So the tester can easily incorporate such knowledge into their partitioning heuristic without affecting the complexity of the algorithm.

Second, in some frameworks, the order of execution of a test suite is not necessarily sequential. For instance, the default test case execution order of JUnit is unpredictable. As such, it may not be straightforward for *users of such a framework* to directly adopt DBP to prioritize their test cases; however, *developers of the framework* can implement DBP in the platform as an alternative to their original ordering algorithm.

9.4 Why did we not compare DBP with other TCP techniques?

Further comparisons between DBP and various other TCP algorithms are beyond the scope of this paper, as our main research question is RQ1. Such comparisons would actually be unfair because DBP does not require the extra information used by other TCP techniques—it does not even need to know the values of the test cases. Similarly, we have not compared DBP with other TCP techniques reported to perform equally well or better than the *additional* algorithm [6], [18], [19], [24], [62], [63].

In fact, we have attempted to use some of the existing TCP algorithms that have a quadratic time complexity, only to find that when the test suite is large, the test-case-generation time is not only much longer than the test-case-execution time but also prohibitive for any practical purpose. This confirms the finding of Miranda et al. [24] that "some TCP approaches [soon become] inefficient even for small-medium size benchmarks."

9.5 Why did we not report further validations of Observation I?

Observation **I** is the cornerstone of DBP. We have given an illustrative example in Figs. 1 and 2 to support this observation.

It should be noted that the test cases in Fig. 2 have been generated automatically by a tool. Test suites generated by human developers may be different. In some companies, for instance, failure-causing test cases can be added to a regression test suite after a reported bug has been fixed. In other situations, the developers may skip related scenarios. It is thus desirable to have Observation **I** backed by further empirical evidence, such as through a comprehensive examination of both manual and tool-generated real-world test suites. We have inspected in detail some of the test suites in our empirical studies, and have indeed found obvious similarities among neighboring test cases. However, owing to the sheer size and complexity of real-world test suites and test scripts, and also owing to the subjectivity of assessing similarities in test case semantics, it is beyond the scope of this exploratory paper to present a thorough empirical investigation of Observation **I**. Such an investigation is clearly warranted in future research.

It should be noted, however, that ***intuitions and small-scale observations often come before a systematic and large-scale investigation***. For example, DeMillo et al. [64] made the intuitive assumption of a "coupling effect" of program mutations in their seminal work in 1978. Yet, a comprehensive investigation on the validity of the coupling effect was only reported more than ten years later, at which time Offutt [65] pointed out that "Even though the coupling effect has been mentioned by numerous researchers, there has been little effort to verify or disprove the effect." We therefore envisage more empirical investigations into Observation **I** from both industry and academia in the future, to either support it or produce counterexamples.

The effectiveness of DBP and RP will become similar when Observation **I** is violated, that is, when neighboring test cases do not have any similarity (in terms of features tested, code coverage, or failure-detection capability, etc.). This may happen if, for example, all the test cases have been randomly generated/sampled, or added to the

test suite by different anonymous contributors working simultaneously on different parts of the project (in this situation, the ordering information may be unavailable or unreliable, and hence DBP should not be applied). Furthermore, DBP will not be applicable if test case IDs cannot be used—this may happen if the test driver cannot be edited, for example, if the test driver is a binary executable file or is provided as a black-box component by a third party (in this situation, all TCP techniques, except the sequential ordering, will be inapplicable). In addition, sometimes certain test cases may be given a higher priority—in this situation, DBP (as well as any other TCP technique) can only be applied to prioritize the test cases that have the same level of priority. An in-depth investigation into the above scenarios and development of further solutions is warranted in future research.

9.6 During test case prioritization, is it appropriate to consider all the available test cases?

In TCP literature, researchers typically construct a set of test suites, each containing a small number of test cases selected from a large test pool created for the research project. Contrary to this research practice, in our present work, we advocate that TCP experiments should be conducted using *real-life* packages, where test suites are large—after all, if the test suites were small, there would be no need for test case prioritization. Test suites with millions of test cases have been extensively reported in the industry (by Microsoft [66], IBM [67], and Google [68], to name a few), and has drawn researchers' attention in recent years [18]. In fact, Miranda et al. [24] noted that most TCP techniques "do not scale up to handle the many thousands or even some millions test suite sizes of modern industrial systems."

9.7 Is DBP really effective for the detection of meaningful software issues?

The experiments conducted in this study involve testing but not debugging. In other words, we have not gone further to investigate the root causes for the failures detected. The debugging process is not included in this study because of the sheer size and complexity of the real-life SUTs and their test cases. Nevertheless, it is known that the failures of all 17 faulty Space programs (in rows #27 to #43 of Table 1) have been caused by *genuine* bugs collected during the *real-life* development of the software. Furthermore, all five GNU Compiler Collection programs listed in rows #2 to #6 of Table 1 (namely, g++, gcc, gfortran, libmudflap, and libstdc++) have been tested using test suite v4.8.0, the same version as the SUTs. This means that failures detected for these five GNU Compiler Collection programs have revealed real and meaningful software issues. In addition, Firefox (row #11 of Table 1), the largest SUT of our study, which contains 6,177,736 source lines of code, has also been tested using a test suite that is of the same version as the SUT (namely, v31.0) and, therefore, its failures also indicate real and meaningful software issues.

Apart from the above programs, there are eight subject packages where a newer version of test suite has been used to test an older version of SUT. As listed in Table 1, these packages are: SQLite in row #1, commons-lang in row #7, commons-math in row #8, jfreechart in row #9, joda-time in row #10, Autoconf in rows #12 to #16, Automake in rows #17 to #21, and MySQL in rows #22 to #26.

One may argue that the "failures" detected in this study for the above eight packages may not necessarily indicate any defect in the SUT or any problem in the environment because these failures might have been caused by compatibility issues (such as the old version SUT not supporting a new input parameter or a new component being absent in the old version) and that such a testing practice is meaningless with respect to fault detection. It should be noted, however, that the purpose of our testing activities with the above eight packages is *not* direct detection of defects. As explained in Section 4.3.3, there could be various other testing objectives in running a newer test suite on an older SUT. These objectives include program comprehension, change impact analysis, or to find behavioral differences (which may or may not be caused by software faults) between two versions of the software, among others [38], [39], [69]–[71].

Consider the results shown in Table 2. We can divide the table into two sub-tables: The **first sub-table** consists of all 17 faulty Space programs (in rows #27 to #43), which includes *genuine* bugs, and all five GNU Compiler Collection programs listed in rows #2 to #6 (namely, g++, gcc, gfortran, libmudflap, and libstdc++), for which the SUT and the test suite are of the same version, as well as row #11 (Firefox, the largest SUT of our study), which has also been tested using the same version of test suite. The **second sub-table** consists of all the remaining programs, that is, the eight subject packages where a newer test suite has been applied to an older SUT (namely, SQLite in row #1, commons-lang in row #7, commons-math in row #8, jfreechart in row #9, joda-time in row #10, Autoconf in rows #12 to #16, Automake in rows #17 to #21, and MySQL in rows #22 to #26).

Examining the respective test results for these two sub-tables reveals that the **first sub-table** results are much better than those of the **second sub-table**. Consider column #5 (F.DBP÷F.RP), for example, the **first sub-table** has mean, maximum, and minimum values of 0.84, 1.01, and 0.42, respectively, whereas the respective statistics of the **second sub-table** are 0.97, 1.02, and 0.89, which means that the **former** is the dominating factor for the observed superior performance of DBP over RP.

When designing the experiments, we also tried to apply the old test suites to the new versions of the SUTs. However, no failures could be detected. As explained in Section 4.3.3, this is because the new SUT versions must have already gone through regression testing and passed all of the old test cases before they were released.

TABLE 6
Results of a supplemental experiment with the *Replace* program. F.RP: F-measure of random prioritization (RP) out of 10,000 trials; F.DBP: F-measure of dispersity-based prioritization (DBP) out of 10,000 trials. Where there is a statistically significant difference between the RP and DBP means (with a p-value below 0.05), the corresponding cells are highlighted (light gray: DBP outperformed RP; dark gray: RP outperformed DBP). "p=0.000" means p$<$0.0005. Where the effect size (Cohen's d) is 0.10 or larger, the corresponding cells are starred (*). "d=0.00" means d$<$0.005.

project name & version	F.RP	F.DBP	F.DBP ÷ F.RP	p-value (2-tailed) for F-measure, and effect size (d)
Replace v1	81.11	68.58	0.85	p=0.000, d=0.17*
Replace v2	145.93	135.96	0.93	p=0.000, d=0.07
Replace v3	41.81	24.98	0.60	p=0.000, d=0.49*
Replace v4	38.55	22.82	0.59	p=0.000, d=0.51*
Replace v5	20.46	19.04	0.93	p=0.000, d=0.07
Replace v6	56.80	54.55	0.96	p=0.004, d=0.04
Replace v7	65.75	67.08	1.02	p=0.145, d=0.02
Replace v9	178.70	181.60	1.02	p=0.242, d=0.02
Replace v10	230.74	234.85	1.02	p=0.193, d=0.02
Replace v13	34.30	21.01	0.61	p=0.000, d=0.48*
Replace v15	91.72	89.36	0.97	p=0.059, d=0.03
Replace v17	225.32	230.15	1.02	p=0.115, d=0.02
Replace v18	26.56	25.54	0.96	p=0.004, d=0.04
Replace v19	1,373.69	1,422.78	1.04	p=0.001, d=0.05
Replace v20	246.15	252.65	1.03	p=0.055, d=0.03
Replace v21	1,372.45	1,139.15	0.83	p=0.000, d=0.24*
Replace v22	285.51	292.11	1.02	p=0.084, d=0.02
Replace v24	32.72	32.54	0.99	p=0.687, d=0.01
Replace v25	1,406.24	1,463.03	1.04	p=0.000, d=0.05
Replace v27	20.69	20.90	1.01	p=0.472, d=0.01
Replace v28	38.86	24.37	0.63	p=0.000, d=0.45*
Replace v29	85.69	77.78	0.91	p=0.000, d=0.10*
		avg	0.91	
		max	1.04	
		min	0.59	

To further confirm the finding that DBP is more effective than RP for the detection of software faults, we have conducted a supplemental experiment using the *Replace* program of the *Siemens suite of programs* [72] downloaded from SIR [41]. According to SIR documentation, the Siemens suite of programs were initially assembled by Tom Ostrand and colleagues at Siemens Corporate Research for a study of the fault detection capabilities of control-flow and data-flow coverage criteria [72] and then modified by other researchers for further studies [73]. The Siemens suite of programs have long been used by the testing community for benchmarking testing strategies [74]. Among all seven Siemens programs, the Replace program, which performs regular expression matching and substitutions, is the most complex; despite having only 512 SLOC in C, it covers the most varieties of logic errors [75]. The Replace package includes a test driver that runs $5,542$ test cases. The order of test case executions in the test driver is therefore used as the order of the test cases in our experiment. The package contains a base version, which is used as the test oracle, and a total of 32 faulty versions—although the faults are created manually, the Siemens researchers have made them as realistic as possible [72]. Of the 32 faulty versions, 22 have recorded a failure rate below 5%. These 22 faulty versions, therefore, are used in our experiment. As with the previous experiments, for each faulty version, we have conducted $10,000$ trials of DBP and $10,000$ trials of RP to estimate their respective F-measures. The experimental results are given in Table 6, which clearly show that DBP has outperformed RP. These results are consistent with those collected from genuine real-life packages presented earlier in this paper.

10 Conclusions

The most important contribution of this research is the discovery of Observation **I**, a very simple but important property of real-world test suites that can support test case prioritization. More specifically, we observe that neighboring test cases in real-world test suites often have similarities in certain ways. As indicated earlier, the validity of this observation should be further examined in future research, by inspecting many more software projects and their test suites.

Based on Observation **I**, we have proposed an extremely simple approach to prioritizing test cases. The algorithm itself is not novel, as it is a direct application of ART. The novelty lies in the dispersity metric, which makes use of test case IDs (which are readily available) rather than concrete input values (of which the distance may not be easy to measure) or test case coverage data (which may not be available).

Our results are consistent with recent studies suggesting that diversity (and therefore dissimilarity) is a key concept underlying the foundations of successful software testing strategies [28], [76]. Our proposed DBP algorithm generates a sequence of test case IDs by using the "FSCS-ART with forgetting" method. Other linear ART algorithms [29] can also be adopted to replace "FSCS-ART with forgetting." An empirical evaluation of these other algorithms in the context of text case prioritization and test case selection using the natural distance is a future research topic.

Our research has shown that DBP is more applicable than the *additional* algorithm. But even when conventional TCP techniques are applicable, DBP can still be a better choice—especially if the test suite is very large, resulting in a high computational overhead. The proposed approach, therefore, provides an innovative direction and practical hints for testing engineers dealing with large test suites. It can be a very simple and yet useful solution to the test case prioritization problem in real life.

In real-world projects, the original order of test cases may be generated according to very different rules, and not all test cases are logically generated by tools.

Furthermore, during software evolution, many new test cases are added to the test suite, making the test order different. However, as reported earlier in this paper, even in such circumstances, applying DBP will do no harm to the test effectiveness and efficiency.

ACKNOWLEDGMENTS

This work was supported in part by a linkage grant of the Australian Research Council (Project ID: LP160101691), an Australian Government Research Training Program scholarship, the General Research Fund of the Research Grants Council of Hong Kong (project number 716612), and a Western River entrepreneurship grant. We wish to thank Morphick Solutions Pty Ltd, Australia, for supporting this research. All correspondence should be addressed to Dr. Z. Q. Zhou at the address shown on the first page of this paper.

REFERENCES

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

[2] G. Rothermel, M. Burnett, L. Li, C. Dupuis, and A. Sheretov, "A methodology for testing spreadsheets," *ACM Transactions on Software Engineering and Methodology*, vol. 10, no. 1, pp. 110–147, 2001.

[3] S. Yoo and M. Harman, "Regression testing minimization, selection and prioritization: A survey," *Software Testing, Verification and Reliability*, vol. 22, pp. 67–120, 2012.

[4] S. Wang, S. Ali, T. Yue, Ø. Bakkeli, and M. Liaaen, "Enhancing test case prioritization in an industrial setting with resource awareness and multi-objective search," in *Proceedings of the 38th International Conference on Software Engineering Companion (ICSE-C 16)*. ACM, 2016, pp. 182–191.

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

[6] B. Jiang, Z. Zhang, W. K. Chan, and T. H. Tse, "Adaptive random test case prioritization," in *Proceedings of the 24th IEEE/ACM International Conference on Automated Software Engineering (ASE'09)*. IEEE Computer Society Press, November 2009, pp. 233–244.

[7] Z. Li, M. Harman, and R. M. Hierons, "Search algorithms for regression test case prioritization," *IEEE Transactions on Software Engineering*, vol. 33, no. 4, pp. 225–237, 2007.

[8] M. B. Cohen, M. B. Dwyer, and J. Shi, "Constructing interaction test suites for highly-configurable systems in the presence of constraints: A greedy approach," *IEEE Transactions on Software Engineering*, vol. 34, no. 5, pp. 633–650, 2008.

[9] H. Srikanth, L. Williams, and J. Osborne, "System test case prioritization of new and regression test cases," in *Proceedings of the 4th International Symposium on Empirical Software Engineering (ISESE'05)*. IEEE Computer Society Press, 2005, pp. 64–73.

[10] B. Korel, G. Koutsogiannakis, and L. Tahat, "Application of system models in regression test suite prioritization," in *Proceedings of the 24th IEEE International Conference on Software Maintenance (ICSM'08)*. IEEE Computer Society Press, 2008, pp. 247–256.

[11] J. M. Kim and A. Porter, "A history-based test prioritization technique for regression testing in resource constrained environments," in *Proceedings of the 24th International Conference on Software Engineering (ICSE '02)*. ACM Press, 2002, pp. 119–129.

[12] Y.-C. Huang, K.-L. Peng, and C.-Y. Huang, "A history-based cost-cognizant test case prioritization technique in regression testing," *Journal of Systems and Software*, vol. 85, pp. 626–637, 2012.

[13] Z. Q. Zhou, A. Sinaga, W. Susilo, L. Zhao, and K.-Y. Cai, "A cost-effective software testing strategy employing online feedback information," *Information Sciences*, vol. 422, pp. 318–335, 2018.

[14] S. S. Hou, L. Zhang, T. Xie, H. Mei, and J. S. Sun, "Applying interface-contract mutation in regression testing of component-based software," in *Proceedings of the 23rd IEEE International Conference on Software Maintenance (ICSM'07)*. IEEE Computer Society Press, 2007, pp. 174–183.

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

[16] S. G. Elbaum, A. G. Malishevsky, and G. Rothermel, "Incorporating varying test costs and fault severities into test case prioritization," in *Proceedings of the 23rd International Conference on Software Engineering (ICSE '01)*. ACM Press, 2001, pp. 329–338.

[17] Z. Q. Zhou, "Using coverage information to guide test case selection in adaptive random testing," in *Proceedings of the 34th Annual International Computer Software and Applications Conference (COMPSAC'10), 7th International Workshop on Software Cybernetics*. IEEE Computer Society Press, 2010, pp. 208–213.

[18] Z. Q. Zhou, A. Sinaga, and W. Susilo, "On the fault-detection capabilities of adaptive random test case prioritization: Case studies with large test suites," in *Proceedings of the 45th Annual Hawaii International Conference on System Sciences (HICSS'12)*. IEEE Computer Society Press, 2012, pp. 5584–5593.

[19] R. K. Saha, L. Zhang, S. Khurshid, and D. E. Perry, "An information retrieval approach for regression test prioritization based on program changes," in *Proceedings of the 37th International Conference on Software Engineering (ICSE '15) - Volume 1*. IEEE Press, 2015, pp. 268–279.

[20] B. Busjaeger and T. Xie, "Learning for test prioritization: An industrial case study," in *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE '16)*. ACM, 2016, pp. 975–980.

[21] B. Jiang and W. K. Chan, "Input-based adaptive randomized test case prioritization: A local beam search approach," *Journal of Systems and Software*, vol. 105, pp. 91–106, 2015.

[22] Y. Ledru, A. Petrenko, and S. Boroday, "Using string distances for test case prioritisation," in *Proceedings of the 24th IEEE/ACM International Conference on Automated Software Engineering (ASE '09)*. Auckland, New Zealand: IEEE, Nov. 16–20, 2009, pp. 510–514.

[23] S. W. Thomas, H. Hemmati, A. E. Hassan, and D. Blostein, "Static test case prioritization using topic models," *Empirical Software Engineering*, vol. 19, pp. 182–212, 2014.

[24] B. Miranda, E. Cruciani, R. Verdecchia, and A. Bertolino, "FAST approaches to scalable similarity-based test case prioritization," in *Proceedings of the 40th International Conference on Software Engineering (ICSE '18)*. Gothenburg, Sweden: ACM, May 27–Jun 3, 2018, pp. 222–232.

[25] S. Khurshid, C. S. Păsăreanu, and W. Visser, "Generalized symbolic execution for model checking and testing," in *Proceedings of the 9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'03)*. Springer-Verlag, 2003, pp. 553–568.

[26] K. Sen, D. Marinov, and G. Agha, "CUTE: A concolic unit testing engine for C," in *Proceedings of the Joint 10th European Software Engineering Conference and 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC-FSE'05)*. ACM Press, 2005, pp. 263–272.

[27] M. Pezzè and M. Young, *Software Testing and Analysis: Process, Principles, and Techniques*. New York: Wiley, 2008.

[28] T. Y. Chen, F.-C. Kuo, R. G. Merkel, and T. H. Tse, "Adaptive random testing: The ART of test case diversity," *Journal of Systems and Software*, vol. 83, no. 1, pp. 60–66, 2010.

[29] S. Anand, E. Burke, T. Y. Chen, J. Clark, M. B. Cohen, W. Grieskamp, M. Harman, M. J. Harrold, and P. McMinn, "An orchestrated survey of methodologies for automated software test case generation," *Journal of Systems and Software*, vol. 86, no. 8, pp. 1978–2001, 2013.

[30] A. C. Barus, T. Y. Chen, F.-C. Kuo, H. Liu, R. Merkel, and G. Rothermel, "A cost-effective random testing method for programs with non-numeric inputs," *IEEE Transactions on Computers*, vol. 65, no. 12, pp. 3509–3523, 2016.

[31] A. Shahbazi, A. F. Tappenden, and J. Miller, "Centroidal Voronoi tessellations – A new approach to random testing," *IEEE Transactions on Software Engineering*, vol. 39, no. 2, pp. 163–183, 2013.

[32] K. P. Chan, T. Y. Chen, and D. Towey, "Forgetting test cases," in *Proceedings of the 30th Annual International Computer Software and*

Applications Conference (COMPSAC'06). IEEE Computer Society Press, 2006, pp. 485–494.

[33] T. Y. Chen, H. Leung, and I. K. Mak, "Adaptive random testing," in *Proceedings of the 9th Asian Computing Science Conference (ASIAN'04), Lecture Notes in Computer Science 3321*. Springer-Verlag, 2004, pp. 320–329.

[34] E. Selay, Z. Q. Zhou, T. Y. Chen, and F.-C. Kuo, "Adaptive random testing in detecting layout faults of web applications," *International Journal of Software Engineering and Knowledge Engineering*, vol. 28, no. 10, pp. 1399–1428, 2018.

[35] T. Y. Chen and R. Merkel, "An upper bound on software testing effectiveness," *ACM Transactions on Software Engineering and Methodology*, vol. 17, no. 3, pp. 16:1–16:27, 2008.

[36] B. A. Kitchenham, S. L. Pfleeger, L. M. Pickard, P. W. Jones, D. C. Hoaglin, K. E. Emam, and J. Rosenberg, "Preliminary guidelines for empirical research in software engineering," *IEEE Transactions on Software Engineering*, vol. 28, no. 8, pp. 721–734, August 2002.

[37] Y. Li, C. Zhu, J. Rubin, and M. Chechik, "Precise semantic history slicing through dynamic delta refinement," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE '16)*. ACM, 2016, pp. 495–506.

[38] W. Jin, A. Orso, and T. Xie, "Automated behavioral regression testing," in *Proceedings of the 3rd International Conference on Software Testing, Verification and Validation (ICST'10)*. IEEE Computer Society Press, 2010, pp. 137–146.

[39] G. Soares, R. Gheyi, and T. Massoni, "Automated behavioral testing of refactoring engines," *IEEE Transactions on Software Engineering*, vol. 39, no. 2, pp. 147–162, February 2013.

[40] SQLite website, "How SQLite is tested," http://www.sqlite.org/testing.html.

[41] H. Do, S. G. Elbaum, and G. Rothermel, "Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact," *Empirical Software Engineering: An International Journal*, vol. 10, no. 4, pp. 405–435, 2005.

[42] Software-artifact Infrastructure Repository, http://sir.unl.edu, 2013.

[43] F. I. Vokolos and P. G. Frankl, "Empirical evaluation of the textual differencing regression testing technique," in *Proceedings of the International Conference on Software Maintenance*, 1998, pp. 44–53.

[44] A. Field, *Discovering Statistics Using IBM SPSS Statistics: North American Edition*, 5th ed. SAGE Publications Ltd, 2017.

[45] J. Cohen, *Statistical power analysis for the behavioral sciences*, 2nd ed. Hillsdale, 1988.

[46] J. A. Durlak, "How to select, calculate, and interpret effect sizes," *Journal of Pediatric Psychology*, vol. 34, no. 9, pp. 917–928, 2009.

[47] M. Abbas, I. Inayat, M. Saadatmand, and N. Jan, "Requirements dependencies-based test case prioritization for extra-functional properties," in *Proceedings of the IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE, 2019, pp. 159–163.

[48] T. Y. Chen, F.-C. Kuo, and Z. Q. Zhou, "On favourable conditions for adaptive random testing," *International Journal of Software Engineering and Knowledge Engineering*, vol. 17, no. 6, pp. 805–825, 2007.

[49] K.-H. Yuan and S. Maxwell, "On the post hoc power in testing mean differences," *Journal of Educational and Behavioral Statistics*, vol. 30, no. 2, pp. 141–167, 2005.

[50] A. Ghasemi and S. Zahediasl, "Normality tests for statistical analysis: A guide for non-statisticians," *International Journal of Endocrinology and Metabolism*, vol. 10, no. 2, pp. 486–489, 2012.

[51] D. G. Altman, "The normal distribution," *BMJ*, vol. 310, pp. 298–298, February 1995. [Online]. Available: https://doi.org/10.1136/bmj.310.6975.298

[52] C. A. Boneau, "The effects of violations of assumptions underlying the t test," *Psychological Bulletin*, vol. 57, no. 1, pp. 49–64, 1960.

[53] J. H. Zar, *Biostatistical Analysis*, 2nd ed. Englewood Cliffs, N.J.: Prentice-Hall, Inc, 1984.

[54] D. M. Levine, D. Stephan, T. C. Krehbiel, and M. L. Berenson, *Statistics for Managers Using Microsoft Excel*, 4th ed. Upper Saddle River, N.J.: Pearson Prentice Hall, 2005.

[55] J. Pallant, *SPSS Survival Manual: A Step By Step Guide to Data Analysis Using SPSS for Windows (Version 15)*, 3rd ed. Crows Nest, NSW, Australia: Allen & Unwin, 2007.

[56] T. Y. Chen, F.-C. Kuo, R. G. Merkel, and S. P. Ng, "Mirror adaptive random testing," *Information and Software Technology*, vol. 46, no. 15, pp. 1001 – 1010, 2004.

[57] K. P. Chan, T. Y. Chen, and D. Towey, "Restricted random testing: adaptive random testing by exclusion," *International Journal of Software Engineering and Knowledge Engineering*, vol. 16, no. 4, pp. 553–584, 2006.

[58] T. Y. Chen, G. Eddy, R. Merkel, and P. K. Wong, "Adaptive random testing through dynamic partitioning," in *Proceedings of the 4th International Conference on Quality Software (QSIC'04)*. IEEE Computer Society Press, 2004, pp. 79–86.

[59] T. Y. Chen, D. H. Huang, and Z. Q. Zhou, "On adaptive random testing through iterative partitioning," *Journal of Information Science and Engineering*, vol. 27, no. 4, pp. 1449–1472, 2011.

[60] H. Liu, X. Xie, J. Yang, Y. Lu, and T. Y. Chen, "Adaptive random testing through test profiles," *Software: Practice and Experience*, vol. 41, no. 10, pp. 1131–1154, 2011.

[61] T. Y. Chen, F.-C. Kuo, and H. Liu, "Adaptive random testing based on distribution metrics," *Journal of Systems and Software*, vol. 82, no. 9, pp. 1419–1433, 2009.

[62] D. Hao, L. Zhang, L. Zhang, G. Rothermel, and H. Mei, "A unified test case prioritization approach," *ACM Transactions on Software Engineering and Methodology*, vol. 24, no. 2, pp. 10:1–10:31, December 2014.

[63] D. Mondal, H. Hemmati, and S. Durocher, "Exploring test suite diversification and code coverage in multi-objective test case selection," in *Proceedings of the IEEE 8th International Conference on Software Testing, Verification and Validation (ICST '15)*. Graz, Austria: IEEE, Apr. 13–17, 2015, pp. 1–10.

[64] R. A. DeMillo, R. J. Lipton, and F. G. Sayward, "Hints on test data selection: Help for the practicing programmer," *IEEE Computer*, vol. 11, no. 4, pp. 34–41, 1978.

[65] A. J. Offutt, "Investigations of the software testing coupling effect," *ACM Transactions on Software Engineering and Methodology*, vol. 1, no. 1, pp. 5–20, January 1992.

[66] V. Vangala, J. Czerwonka, and P. Talluri, "Test case comparison and clustering using program profiles and static execution," in *Proceedings of the 7th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE '09)*. Amsterdam, The Netherlands: ACM, Aug. 24–28, 2009, pp. 293–294.

[67] IBM, "Address system-on-chip development challenges with enterprise verification management," Enterprise verification management solutions white paper, September 2009. [Online]. Available: ftp.software.ibm.com

[68] S. Elbaum, G. Rothermel, and J. Penix, "Techniques for improving regression testing in continuous integration development environments," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE '14)*. Hong Kong, China: ACM, Nov. 16–22, 2014, pp. 235–245.

[69] P. Braione, G. Denaro, O. Riganelli, M. Baluda, and A. Muhammad, "Static/dynamic test case generation for software upgrades via ARC-B and DeltaTest," in *Validation of Evolving Software*, H. Chockler, D. Kroening, L. Mariani, and N. Sharygina, Eds. Heidelberg: Springer, 2015, ch. 11, pp. 147–184.

[70] Y. Noller, "Differential program analysis with fuzzing and symbolic execution," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE '18)*. ACM, 2018, pp. 944–947.

[71] B. Danglot, "Automatic unit test amplification for DevOps," Ph.D. dissertation, University of Lille, France, 2019.

[72] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand, "Experiments on the effectiveness of dataflow- and controlflow-based test adequacy criteria," in *Proceedings of the 16th International Conference on Software Engineering (ICSE '94)*. IEEE Computer Society Press, 1994, pp. 191–200.

[73] SIR. C object biographies. [Online]. Available: https://sir.csc.ncsu.edu/content/bios/tcas.php#siemens

[74] B. Kurtz, P. Ammann, J. Offutt, M. E. Delamaro, M. Kurtz, and N. Gökçe, "Analyzing the validity of selective mutation with dominator mutants," in *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE '16)*. ACM, Nov 2016, pp. 571–582.

[75] T. Y. Chen, T. H. Tse, and Z. Q. Zhou, "Semi-proving: An integrated method for program proving, testing, and debugging," *IEEE Transactions on Software Engineering*, vol. 37, no. 1, pp. 109–125, 2011.

[76] T. Y. Chen, F.-C. Kuo, D. Towey, and Z. Q. Zhou, "A revisit of three studies related to random testing," *Science China Information Sciences*, vol. 58, pp. 052 104:1–052 104:9, 2015.

Zhi Quan Zhou received the BSc degree in computer science from Peking University, China, and the PhD degree in software engineering from The University of Hong Kong. He is currently an associate professor and director of the bachelor of computer science degree with the University of Wollongong, Australia. His current research interests include software testing and debugging; the interplay among software testing, machine learning, and big data; and self-driving vehicles. He was one of the few pioneers who opened up and established the research field of metamorphic testing. In 2016, he co-founded and chaired the IEEE/ACM 1st International Workshop on Metamorphic Testing, in conjunction with ICSE (ICSE MET '16). He was an invited keynote speaker at ICSE MET '17 and at the IEEE International Conference on Artificial Intelligence Testing (AITest), 2019. He was an invited ACM SIGSOFT Webinar speaker, an ICSE '18 Technical Briefings speaker, and an ICSE '16 and ICSE '19 journal-first speaker. He was selected for a Virtual Earth Award by Microsoft Research, Redmond, USA, and a 2018 Researcher of the Year Gold Disruptor Award by the Australian Computer Society.

Chen Liu received the Bachelor of Computer Science (Honours) degree from the University of Wollongong, Australia, where he is currently a PhD student in computer science. His research interest is in software testing and debugging.

Tsong Yueh Chen received the BSc and MPhil degrees from The University of Hong Kong; MSc degree and DIC from the Imperial College of Science and Technology, University of London; and PhD degree from The University of Melbourne. He is a professor of software engineering at Swinburne University of Technology, Australia. Prior to joining Swinburne, he had taught at The University of Hong Kong and The University of Melbourne. His research interests include software testing and analysis. He is the inventor of adaptive random testing and metamorphic testing.

T. H. Tse received the PhD degree from the London School of Economics and was a visiting fellow with the University of Oxford. He is an honorary professor in computer science with The University of Hong Kong after retiring from the full professorship in 2014. His research interest is in program testing and debugging. He is the steering committee chair of the IEEE International Conference on Software Quality, Reliability, and Security, an associate editor of *IEEE Transactions on Reliability*, and an editorial board member of *Software Testing, Verification and Reliability* and *Software: Practice and Experience*. He also served on the search committee for the editor-in-chief of *IEEE Transactions on Software Engineering* in 2013. He is a fellow of the British Computer Society. He was awarded an MBE by The Queen of the United Kingdom.

Willy Susilo received the PhD degree in computer science from the University of Wollongong, Australia. He is a professor, the head of the School of Computing & IT, and the director of the Institute of Cybersecurity and Cryptology at the University of Wollongong. He was previously an Australian Research Council Future Fellow, and was selected for a Researcher of the Year Award by the University of Wollongong in 2016. His main research interests include cybersecurity, cryptography and information security. He is the editor-in-chief of *Information*, and an editorial board member of *International Journal of Information Security*.