2015 IEEE International Conference on Software Quality, Reliability and Security

PORA: Proportion-Oriented Randomized Algorithm for Test Case Prioritization*

Bo Jiang
School of Computer Science and Engineering
Beihang University
Beijing, China
jiangbo@buaa.edu.cn

W.K. Chan†
Department of Computer Science
City University of Hong Kong
Tat Chee Avenue, Hong Kong
wkchan@cityu.edu.hk

T.H. Tse
Department of Computer Science
The University of Hong Kong
Pokfulam, Hong Kong
thtse@cs.hku.hk

***Abstract*—Effective testing is essential for assuring software quality. While regression testing is time-consuming, the fault detection capability may be compromised if some test cases are discarded. Test case prioritization is a viable solution. To the best of our knowledge, the most effective test case prioritization approach is still the additional greedy algorithm, and existing search-based algorithms have been shown to be visually less effective than the former algorithms in previous empirical studies. This paper proposes a novel *Proportion-Oriented Randomized Algorithm* (PORA) for test case prioritization. PORA guides test case prioritization by optimizing the distance between the prioritized test suite and a hierarchy of distributions of test input data. Our experiment shows that PORA test case prioritization techniques are as effective as, if not more effective than, the total greedy, additional greedy, and ART techniques, which use code coverage information. Moreover, the experiment shows that PORA techniques are more stable in effectiveness than the others.**

***Index Terms*—Test case prioritization, randomized algorithm, proportional sampling strategy, multi-objective optimization**

I. Overview

A. Introduction

Regression testing [17][18][19][27] is important [6] and practical [15][16] but is also time-consuming [18][32]. It uses an existing test suite to verify a changed program to assure that the latter is not adversely affected by amendments. As reported in [25], individual developers often use the retest-all strategy [17][27] that executes all the test cases, which can nonetheless be further optimized [18]. In continuous integration, in particular, the same test cases may be repeatedly applied to the program whenever new builds are synthesized, possibly many times a day.

To explore the optimization dimension in regression testing, different aspects have been scientifically studied in the literature. The techniques may include, for instance, executing only a proper subset of the existing test suite against the changed program (referred to as regression test selection [27]), removing some test cases from the existing test suite permanently (test suite reduction [13]), or assigning an execution priority to the test cases in a test suite (test case prioritization [6][7]). The effects of combinations of these aspects on regression testing have also been reported in empirical studies [32].

Test case prioritization (TCP) is an aspect of regression testing research. The target is to assign an execution priority to the given test cases. In essence, it permutes, but does not discard, test cases in the given test suite. As such, it does not compromise the fault detection ability of the given test suite. This makes TCP a *sound* aspect of regression testing.

A simple TCP strategy is to reorder a test suite randomly [6]. However, the resultant test suite is often ineffective, such as having a low weighted Average of the Percentage of Faults Detected (APFD). (See [6] and Subsection III.E for more details of APFD.) In fact, random reordering of the test suite often results in a "lower bound" in previous experiments [6][17]. Moreover, many previous experiments (such as [6] [17]) also show that the effectiveness of such resultant test suites in terms of APFD may vary significantly even when applied to the same programs. This result is as expected due to the random nature of the simple TCP strategy.

If a nonrandom TCP technique exhibits a large variation in fault detection capability (in terms of APFD), that technique is unstable in terms of its effectiveness from the developers' viewpoint. Nevertheless, the results presented in [6][17] show that all the studied TCP techniques suffer from this problem.

It is generally infeasible to know in advance whether a test case will reveal a failure due to a fault in a changed program. Accurately optimizing the ordering of the test cases so that failure-revealing test cases can receive higher priority (so that they can be executed earlier) than the other test cases in the same test suite is an impossible goal. The code coverage achieved by a test case (or a test suite) is not a substitute of the (unknown) failure-triggering conditions of the corresponding test cases. Thus, although guiding the permutation of test cases using code coverage has a certain merit, *over-fitting* the permutation of a test suite for such type of approximation may lead to a suboptimal rate of fault detection, or sometimes even an unexpectedly poor rate of fault detection, which further results in large variations in performance of the corresponding TCP techniques.

Researchers have studied code-coverage based *total greedy* and *additional greedy* algorithms [6][19] for test case prioritization. They are popularly used to benchmark the performance of other techniques [6][7][17][19][32][36]. To the best of our knowledge, the additional greedy algorithm is widely consid-

* This research is supported in part by the National Natural Science Foundation of China (project no. 61202077) and the Early Career Scheme and the General Research Fund of the Research Grants Council of Hong Kong (project nos. 111313, 11201114, 123512, 125113, 716612, and 717811).

† Corresponding author.

978-1-4673-7989-2/15 $31.00 © 2015 IEEE
DOI 10.1109/QRS.2015.28

ered as the *most effective* TCP approach to quickly expose faults from a program in terms of APFD [11]. In addition, search-based algorithms and genetic algorithms have been proposed, which often permute test suites to optimize code-coverage-based fitness functions [19][30]. Li et al. [19] has concluded, however, that the genetic algorithm proposed in their paper is not the best among all the five cases considered, and in most situations, the difference in effectiveness in terms of various *rates of code coverage* between their genetic algorithm and the additional greedy algorithm is *not* statistically significant. They do not report any results on rate of fault detection (such as APFD). Other empirical results [36] further show that time-aware genetic algorithms [30] can be inferior to the additional greedy algorithm [6] in terms of APFD and the time to prioritize test cases.

The ART algorithm [17] for test case prioritization is a kind of search-based algorithm. To the best of our knowledge, ART is the first search-based algorithm that reports results in terms of APFD. Nonetheless, empirical results show that the rates of fault detection (in terms of APFD) between the best ART technique [17] and the best additional greedy technique [6] are still not statistically significant, and that ART is visually less effective than the additional greedy techniques. Another comparison between ART and a heuristic algorithm [9] shows that, out of 49 cases, 29 are tie cases, 13 are ART losing cases, and only five are ART winning cases in terms of APFD. Moreover, neither the ART algorithm [17] nor the heuristic algorithm [9] is significantly more effective than the additional greedy algorithm [28] or random ordering in terms of APFD.

In this paper, we do not compare our proposed approach with evolutionary algorithms, even though it is generally believed that the latter may have a chance to be superior to simple heuristics. It is widely conjectured that evolutionary algorithms suffer from the limitation of "no free lunch" [31]. The law of "no free lunch" indicates that if a type of algorithm is more effective than another type of algorithm for one problem, then there is another problem for the latter type to be more effective than the former; and by summing up all problems, these two algorithms are the same in effectiveness. To the best of our knowledge, however, there is still no concrete evolutionary algorithm that is more effective than the additional greedy algorithm for the test case prioritization problem. It remains as an open problem whether there is any evolutionary algorithm with the potential to outperform the additional greedy algorithm in terms of the rate of fault detection.

B. Summary of our novel TCP algorithm

In this paper, we present the *first* work showing that our search-based TCP algorithm can be as effective as, if not outperforms, the additional greedy algorithm. Our algorithm has a theoretical foundation, which we describe informally in this paper. We will leave the formal presentation to future work.

In mathematics, the grouping of a set into subsets is called partitioning. Partition testing refers to a common testing approach in which a number of test cases are selected from each input subdomain. In general, partition testing for test case generation may be less effective than random testing [10] unless a well-designed methodology to select test cases from disjoint/overlapping subdomains is used [4]. A proven approach is to use the proportional sampling strategy [3], which proportionally selects test cases from each disjoint subdomain according to the relative sizes of the subdomains. It is theoretically shown to have higher probability in detecting at least one failure than random testing even if the failure rate of the domain is unknown (and hence modeled as a uniform distribution). However, the cost of proportionally partitioning the input domain can be expensive. We are inspired by the existing strategy but want to get rid of the high cost incurred in domain partitioning.

A closer look at the theory [3][4] reveals that the mathematical definition of a domain is a set of discrete elements, which does not need to be restricted to the input domain of a program. For ease of presentation, we will refer to the generalized proportional sampling concept as *size-proportional allocation*.

In this paper, we propose a novel **P**roportion-**O**riented **R**andomized **A**lgorithm (**PORA**) for test case prioritization. Suppose that we want to prioritize an inputted test suite with n test cases into a sequence L of n prioritized test cases. A *prefix* of the sequence L is a subsequence of L starting from the first element of L. Thus, if L is of length n, it has n possible prefixes. If $p1$ and $p2$ are prefixes of the same sequence L and $p1$ is shorter than $p2$, then $p1$ is also a prefix of $p2$. For ease of presentation, we simply write "$p1 < p2$".

PORA expresses L in terms of the n prefixes. It aims to allocate test cases proportionally to the "sizes of the subdomains" determined by the lengths of the prefixes. In other words, for each prefix of length i, PORA will allocate i test cases. An interesting point is that for perfect size-proportional allocations, all the prefixes in L must be interconnected such that if $p1$ and $p2$ are prefixes and $p1 < p2$, then all the test cases in $p1$ must also appear in $p2$. An exact solution may not even exist, however. Thus, in actual allocation, the subdomains determined by $p1$ may or may not be related to the subdomains determined by $p2$. It indicates that the size-proportional allocation for the prefix $p1$ (so that it can probabilistically outperform random allocation) may or may not be part of the size-proportional allocation for the prefix $p2$. PORA minimizes the overall discrepancy among all prefixes of L. Moreover, to generate the subdomains for each prefix with i test cases ($i = 1, 2, ..., n$), PORA needs to divide the inputted test suite into i clusters based on the test input data of all the test cases. The above scheme is, nonetheless, highly inefficient (defying the goal of finding efficient search-based algorithms) for two reasons: (1) numerous numbers of subdomain partitioning incur owing to numerous rounds of evolutions of L in order to generate a good solution via a search-based approach to finding a solution, and (2) there is a need to measure the distance *between* the input data of each test case included in each prefix of each evolved L *and* every cluster for the same prefix. PORA addresses these challenges through innovative strategies including prefix evolution and efficient approximation in test case-cluster distance measurement.

Main PORA Algorithm

Inputs: T: a set of test cases $\{t_1, t_2, ..., t_n\}$
M: the maximum number of trials
Output: P: a sequence of prioritized test cases $\langle p_1, p_2, ..., p_n \rangle$

```
1    for r = 1, 2, ..., n
        s_r = f_1(T, r)  // Each iteration generates a scenario s_r, which is a sequence of r centroids
2    S = ⟨s_1, s_2, ..., s_n⟩
3    Randomly generate a permutation P = ⟨p_1, p_2, ..., p_n⟩ of the test suite
4    Randomly select a test case T_0 from P
5    trial = 0, Min_D = MAXVALUE
6    while (trial < M) {  // For up to M trials
        Randomly select two test cases t_1 and t_2 from P such that t_1 ≠ T_0 and t_2 ≠ T_0
7       T_1 = f_2(t_1, t_2, T_0)  // Select t_1 or t_2 as T_1, to be swapped with T_0
                                  according to the specific strategy pora-random or pora-distance
8       Swap the positions of T_0 and T_1 in P to produce P'
9       D = f_3(P', S)  // Calculate the distance D between P' and S
10      if (D < Min_D) {
11         Min_D = D  // Update the minimum distance Min_D found so far
12         P = P'  // Update the best permutation P found so far
13         trial = 0  // Reset trial
14      } else
15         trial++  // Continue to look for a better permutation
16      T_0 = f_4(P, S)  // Find another test case T_0 with maximum distance
17   }  // End of while
18   return P
```

Figure 1. PORA Test Case Prioritization Algorithm

We further refine PORA with two techniques, namely, pora-random and pora-distance. We evaluate both techniques on four medium-scale UNIX utility programs and compare them with random ordering as well as nine state-of-the-art code-coverage-based techniques based on the total greedy, additional greedy, and ART algorithms. The empirical results show that the PORA techniques are always as effective as, if not more effective than, the nonPORA techniques studied. Moreover, our PORA techniques are remarkably more stable in terms of the distributions of the resultant APFD results across all benchmarks. The overall results show that PORA is promising.

The main contribution of this paper is twofold: (i) It proposes PORA, a novel approach to generating effective test case sequences for test case prioritization based on a generalized proportional sampling strategy. (ii) It reports an empirical study that validates the effectiveness, stability, and efficiency of PORA.

We organize the rest of paper as follows: In Section II, we present PORA for test case prioritization. After that, we report an empirical study in Section III. Finally, we discuss related work in Section IV followed by the conclusion in Section V.

II. The PORA Approach

In this section, we present our PORA approach.

A. Overview of PORA

The main PORA algorithm is shown in Figure 1. It uses four auxiliary functions f_1 to f_4. We will present the main algorithm in this subsection followed by the four auxiliary functions in the next subsections.

Suppose that the inputted test suite T contains n test cases. By calling function f_1 (in line 1) n times with different counts ($i = 1, 2, ..., n$), PORA generates (in line 2) a sequence $S = \langle s_1, s_2, ..., s_n \rangle$ of scenarios. Each *scenario* $s_r = \langle c_{r1}, c_{r2}, ..., c_{rr} \rangle$ in S is a sequence of *centroids* c_{ri} of clusters with i test cases from T. For example, the first scenario $s_1 = \langle c_{11} \rangle$ in S corresponds to clusters with one test case while the second scenario $s_2 = \langle c_{21}, c_{22} \rangle$ corresponds to clusters of one and two test cases.

Informally, the goal of PORA is to find a permutation L of T such that the first test case of L is "close to" the only centroid in the clusters corresponding to the first scenario $s_1 = \langle c_{11} \rangle$, the first two test cases of L are "close to" the two centroids of the clusters corresponding to the second scenario $s_2 = \langle c_{21}, c_{22} \rangle$, and so on. To do so, after the creation of S, PORA randomly produces a candidate permutation P of the inputted test suite T and randomly selects one test case T_0 from P (lines 3 to 4). As we are going to explain, we use this test case as the seed of modifying the latest P.

The algorithm then iteratively performs the following activities (lines 6 to 18): It uses the function f_2 to select from P one of two random test case t_1 or t_2 (which should be different from T_0) and denotes the selected test case by T_1. The purpose of f_2 is to define different test case selection strategies that results in different test case prioritization techniques. The current version of PORA limits the number of test cases to two (which can be further generalized) because this number is the

minimal size that a test case selection strategy may produce non-unique results. PORA then copies P to P' and swaps the positions of T_0 and T_1 in P' (line 8). It further calculates a distance D between P' and S using the function f_3 (line 9). If the distance D of the newly generated permutation is smaller than the previously recorded minimum distance Min_D (which is initialized as the maximum positive value in line 5), the algorithm updates Min_D to D and copies P' back to P (lines 10−12). Otherwise, PORA considers that this iteration results in a failed attempt to optimize P and increments the counter *trial* by 1 (line 14). Finally, within each iteration, PORA uses f_4 to select a test case from P' and assigns it to T_0 to facilitate the next iteration. If the number of failed attempts reaches the given bound M (line 6), the algorithm exits from the loop and returns the latest permutation P, which produces the resultant prioritized sequence to approximate L.

In the next five subsections, we describe how PORA computes the distance between test inputs and present functions f_1 to f_4 in turn. Moreover, there are two strategies in function f_2. By using one of them, PORA can be refined into a TCP technique. We refer to the two refined techniques as pora-random and pora-distance, which correspond to the use of the random and distance-based strategies, respectively.

B. Measuring the Distance of Test Cases Based on Input Data

PORA defines the distance between two test cases based on input data. To faciliate input distance calculation, PORA uses feature extraction techniques to map the input data of each test case to a high-dimensional numerical vector and normalize it within the range of [0, 1].

PORA considers each test input data as a string and uses the linear-time textual analysis technique [8] to break it up into words. For instance, the string "`grep -i grep < file1`" is broken up into five words, namely, "`grep`", "`-i`", "`grep`", "`<`", and "`file1`". It further treats each input string as a bag of words (that is, a multiset in which any element may appear more than once) to form a high-dimensional vector such that it maps an input string to a high-dimensional numerical vector of values (using the occurrence frequency). In the above string, for instance, the word "`grep`" occurs twice and the other three words occur once. It forms a four-dimensional vector. The order of words in the vector is unimportant as long as the numerical vector of values is mapped correctly. PORA processes all the inputs in the given test suite T in the same way. Thus, PORA knows the number of occurrences of each word in the entire test suite. It further normalizes the high-dimensional vector of each test case by dividing the number of occurrences of each word in the test case by the corresponding total number of occurrences of the same word in the entire test suite. For instance, if the word "`file1`" has occurred 10 times in all the test cases and a test case only has one occurrence of this word, then the normalized number of occurrences of the word in this particular test case is $1 \div 10 = 0.1$. The purpose of normalization is to allow a fair comparison among test inputs.

Having known the entire test suite, if we are given an input string u, it is not difficult to expand the corresponding vector of u to contain all the distinct words that have occurred in the given test suite T. If a word is not originally in u, we assign 0 as the normalized number of occurrence of the word. We use the vector $O(u)$ (which we call the *occurrence frequency vector* of u) to represent the normalized number of occurrences of each word $\langle o_1, o_2, ..., o_m \rangle$ of u, where m is the number of distinct words in T. For ease of presentation, we assume that the vector entries are sorted in descending order of the alphabet of words.

The distance between two test cases t_1 and t_2 is defined as the Euclidean distance between $O(t_1)$ and $O(t_2)$:

$$\text{Distance}(t_1, t_2) = \text{Euclidean}(O(t_1), O(t_2))$$

C. Generating Scenarios by Clustering (*Function* f_1)

As shown in Figure 2, function f_1 generates scenarios, which are sequences of centroids as explained below. The function accepts two parameters: a test suite T and a specified number of clusters r. It first performs k-means++ clustering [2] with the number of clusters k equal to the parameter r. It clusters the test suite T based on the set of occurrence frequency vectors (see Section B above). By the property of the k-means++ clustering algorithm[3], the within-cluster variances among test cases are minimized. The function then collects the whole set of test cases within each cluster and takes the arithmetic mean of the corresponding occurrence frequency vectors of these test cases. It produces an occurrence frequency vector by the arithmetic mean to denote the *centroid* of the cluster. PORA aims to generate a hierarchy of clusters of different sizes so that it can also use hierarchical agglomerative clustering, which can naturally generate the set of clusters in one round to make it efficient.

Function $f_1(T, r)$ to Generate Scenario
Inputs: T: a set of test cases $\{t_1, t_2, ..., t_n\}$
r: number of clusters required
Output: s: scenario (which is a sequence of r centroids)
1 Use *k-means*++ to produce r clusters
2 $s = \langle c_1, c_2, ..., c_r \rangle$, where each c_i is the centroid of the i-th cluster (i = 1, 2, ..., r)
3 $N = \langle n_1, n_2, ..., n_r \rangle$, where each n_i is the number of test cases in the i-th cluster (i = 1, 2, ..., r)
4 Sort s in descending order of n_i (i = 1, 2, ..., r)
5 return s

Figure 2. Generating a Scenario

Next, the function stores the centroids of all the i clusters into a tuple s and stores the number of test cases within each cluster into a tuple N. After that, it sorts s in descending number of test cases in each cluster. In other words, a centroid is ranked higher if it is the centroid of a larger cluster. Tie cases are resolved arbitrarily.

D. Selecting Candidate Test Cases for Position Exchange (*Function* f_2)

In this section, we propose two strategies to decide which test case to be used to exchange with T_0 in the main PORA algorithm in Figure 1. The first strategy is to exchange with a test case picked randomly from the two given test cases. We

[3] In this paper, we use k-mean++ for the purpose of demonstrating PORA. This can be replaced by other clustering algorithms.

recall that PORA aims to reduce the overall distances between the scenarios and the resultant ordered test suite. This strategy intentionally does not use any distance information. Intuitively, it serves as a lower bound of PORA. The second strategy is to select the test case further away from T_0. In case of a tie, the strategy resolves it randomly. This strategy minics a typical strategy in randomized algorithms to select a local minimium/ maximum for permutations. The two stratgies are represented by the function $f_2(t_1, t_2, T_0)$.

Function $f_2(t_1, t_2, T_0)$

$$f_2(t_1, t_2, T_0) = \begin{cases} \text{return } t_1 \text{ or } t_2 \text{ randomly} & (1) \\ \begin{cases} t_1 & \text{if Distance}(t_1, T_0) > \text{Distance}(t_2, T_0) \\ t_2 & \text{if Distance}(t_2, T_0) > \text{Distance}(t_1, T_0) \\ \text{return } t_1 \text{ or } t_2 \text{ randomly} & \text{otherwise} \end{cases} & (2) \end{cases}$$

where case (1) represents the *random strategy* to select one test case randomly and case (2) represents the *distance strategy* to select the test case further away from the chosen test case T_0.

E. Measuring the Distance between P and S (Function f_3)

We recall that PORA aims to select among test cases from T so that the (final) permutation P and S are "close to" each another. To find out whether P and S are close to each other, we measure the distance between each prefix of P and the corresponding prefix of S, and then take the *harmonic mean*. To compute the distance between a prefix of P and a prefix of S, we measure the distances between their respective elements using the distance function defined in Section 3.2 and then take the *arithmetic mean*.

We use the harmonic mean to compute the distances between prefixes because in mathematics, the harmonic mean is less affected by extreme values and is proven to be smaller than or equal to the arithmetic mean, thus giving more stable results. Moreover, we want to improve the rate of fault detection. In mathematics, harmonic mean is the most appropriate way to compute the average of rates. We use the arithmetic mean to compute the average distances between test cases and centroids corresponding to the same prefix because there is no sense of "rate" in mind.

Formally, let $P = \langle p_1, p_2, ..., p_n \rangle$ be a sequence of test cases and $S = \langle s_1, s_2, ..., s_n \rangle$ be a sequence of scenarios produced by the function f_1. Let $s_r = \langle c_{r1}, c_{r2,...,} c_{rr} \rangle$, where $r = 1, 2, ..., n$ and each c_{ri} is the centroid in the corresponding cluster ($i = 1, 2, ..., r$). The distance between P and S is defined by the following function:

Function $f_3(P, S)$

$$f_3(P, S) = \frac{n}{\sum_{r=1}^{n} \frac{1}{\sum_{i=1}^{r} \text{Distance}(c_{ri}, p_i)/r}}$$

where $\text{Distance}(c_{ri}, p_i)$ is the same as the distance function between two test cases as defined in Subsection II.B.

F. Finding a Test Case with Maximum Distance (Function f_4)

At the end of an iteration in the main algorithm in Figure 1, PORA needs to pick a new candidate for T_0 to prepare for the next iteration. It finds the sequence of scenarios (i.e., a prefix of S) that is the least matched with the current permutation P, and returns the last element of the corresponding prefix p of P. We choose the last element of this particular prefix p as the candidate rather than other elements of p because any other prefix of P shorter than p should be no worse than p.

Let $P = \langle p_1, p_2, ..., p_n \rangle$ be the current permutation of test cases and $S = \langle s_1, s_2, ..., s_n \rangle$ be the sequence of scenarios s_r returned by the function $f_1(T, r)$ for $r = 1, 2, ..., n$. We use the following function $f_4(P, S)$ to find the last element p_r of the prefix that is least matched with P:

Function $f_4(P, S)$

$f_4(P, S) = p_r$ such that

$$SceneError(s_r, \langle p_1, p_2, \dots, p_r \rangle) = \max_{i=1.,2,\dots,n} SceneError(s_i, \langle p_1, p_2, \dots, p_i \rangle)$$

where

$$SceneError(s_i, \langle p_1, p_2, \dots, p_i \rangle) = \sum_{j=1}^{i} \text{Distance}(s_{ij,} p_j)$$

In case of a tie, the function returns the smallest p_r in the tie set. Note that PORA will not be trapped easily at a local maximum because of the random selection of candidate test cases in line 7 of the main algorithm.

III. Empirical Study

In this section, we perform an empirical study to evaluate the effectiveness, stability, and efficiency of PORA.

A. Research Questions

We aim to find the answers to the following research questions:

RQ1. Is PORA effective and stable in improving the rate of fault detection?

To know whether PORA is effective, we compare it with some of the best code-coverage-based techniques in terms of APFD.

RQ2. Is PORA efficient in prioritizing test cases in practice?

PORA requires many distance measurements. We want to find out whether PORA is sufficiently efficient. Hence, we compare the time taken to permute test cases by PORA and that taken by some of the best code-coverage-based techniques. Because code coverage profiling can be costly and depends on the kind of tools to retrieve the statistics, we do not include the cost of obtaining code coverage information when computing the time needed for prioritizing test cases for code-coverage-based techniques.

Table 1. Prioritization Techniques used in our Empirical Study

Algorithm	Technique	Brief Description	Code Coverage Granularity Used
	random	Randomly select test cases one by one	–
Total Greedy	total-st	Sort test cases in descending order of the total number of program constructs covered	statement
	total-fn		function
	total-br		branch
Additional Greedy	addtl-st	Sort test cases in descending order of the coverage of program constructs not yet covered by the selected test cases with reset capability.	statement
	addtl-fn		function
	addtl-br		branch
ART	ART-st-maxmin	Iteratively select test cases that a lower priority test case maximizes its distance with the higher priority test cases	statement
	ART-fn-maxmin		function
	ART-br-maxmin		branch
PORA	pora-random	Our Proposed Techniques	–
	pora-distance		–

B. Techniques for Comparison

We compare PORA with random ordering and nine existing effective code-coverage-based prioritization techniques [6] [17]. These peer techniques include three total greedy techniques (total-st, total-br, and total-fn) and three additional greedy techniques (addtl-st, addtl-br, and addtl-fn) proposed in [6]. As mentioned in Section I, these techniques are popularly used to benchmark other techniques. The peer techniques also include three ART techniques (ART-st-maxmin, ART-fn-maxmin, and ART-br-maxmin). Note that ART-**-maxmin techniques are chosen because they are the best ART techniques reported in [17]. We do not include techniques based on genetic algorithms in our experiment because we are not aware of published work showing that genetic algorithms outperform the total greedy and additional greedy algorithms.

Table 1 summarizes these 12 techniques. As shown in the table, only the two PORA techniques use test input data to permute test cases. All the total greedy, additional greedy, and ART techniques use code coverage data. Random ordering uses neither of them. To facilitate comparisons, we have implemented a test infrastructure in C++ to support all the above techniques. We set the maximum number of trials of the main PORA algorithm to 50. We will leave further generalization to future work.

C. Subject Programs and Test Pools

We use four real-life UNIX utility programs with real faults as subject programs. They are obtained from SIR [5] at http://sir.unl.edu. Table 2 shows their descriptive statistics.

We use the UNIX tool gcov to collect code coverage data of each test case to support the prioritization process of the code-coverage-based techniques. Following [6][17], we exclude the versions with faults that cannot be revealed by any test case as well as the versions with faults that can be detected by more than 20% of the test cases. In addition, we also exclude those versions that cannot be supported by our platforms.

Table 2. Subject Programs

Subject Program	Description	No. of Faulty Versions	Executable Source LOC	Size of Test Pool
flex	Lexical Analyzer	21	8571–10124	567
grep	Text Searcher	17	8053–9089	809
gzip	File Compressor	55	4081–5159	217
sed	Stream Editor	17	4756–9289	370

D. Test Suites

For each UNIX program, we generate 1000 test suites iteratively from the test pool. In each iteration, we randomly select a test case and add it to the suite as long as the test case can increase the coverage of the suite. The process stops when we have covered all the statements/functions/branches or when the new test case can no longer improve the coverage. This procedure is also used in [17].

E. Metrics

We adopt APFD [6] to measure the rate of fault detection. *APFD* is the weighted Average of the Percentage of Faults Detected over the life of the suite. Although there are weaknesses in APFD (see, for example, the discussions in [19][30] [36]), this metric is widely used in numerous experiments to compare the fault detections rates among techniques.

APFD is defined as follows: Let T be a test suite containing n test cases and let F be a set of m faults revealed by T. Let TF_v be the first test case in the prioritized test suite T' of T that reveals fault v. The APFD value for T' is given by the equation

$$APFD = 1 - \frac{TF_1 + TF_2 + \cdots + TF_m}{nm} + \frac{1}{2n}$$

Time is the next dimension that we compare the techniques. We measure the time (in seconds) taken by a test case prioritization technique from accepting an inputted test suite to producing a permutation of the test suite.

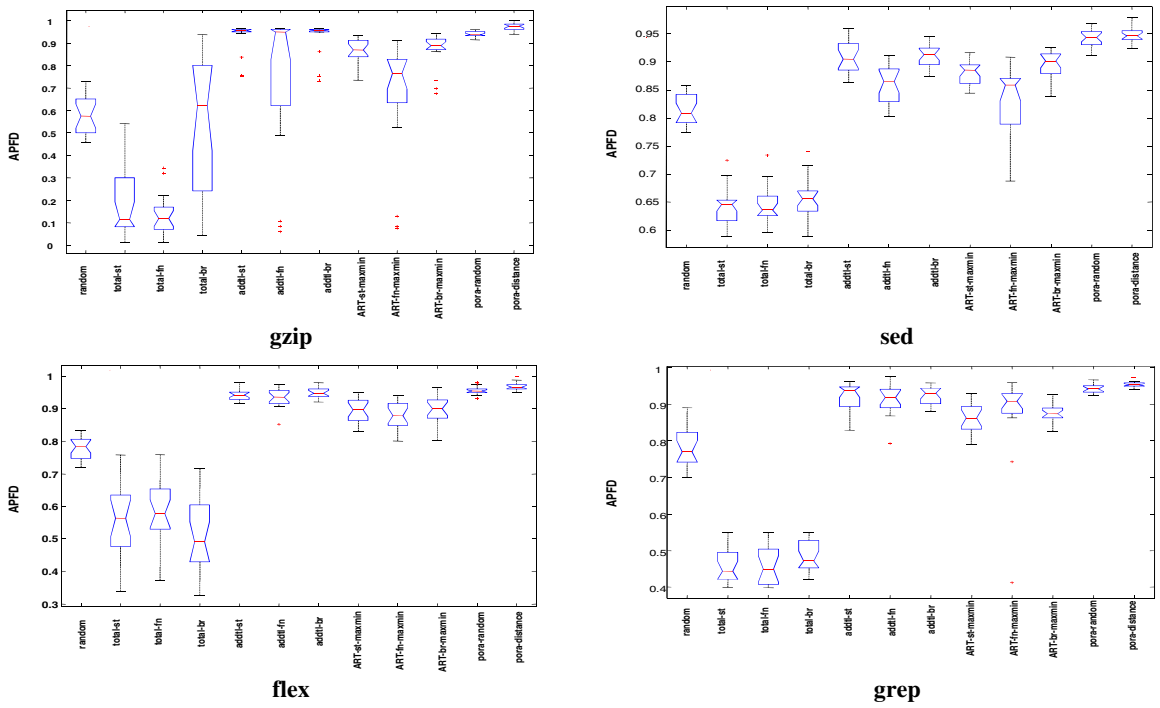

Figure 3. APFD Distributions for Individual UNIX Programs

F. *Empirical Study Environment*

We perform the empirical study on a Dell PowerEdge 1950 server running a Solaris UNIX. The server has two Xeon 5355 (2.66GHz, quad-core) processors with 8GB physical memory.

G. *Empirical Study Procedure*

We run each faulty version over its test pool so that we know which test case fails. For every test suite, we run each technique to generate a prioritized test suite. Random ordering as well as ART and PORA techniques are based on random selection. We repeat each of them 50 times for every test suite to obtain averages that can portray typical performance. We then compute the APFD of each prioritized test suite and the time taken by the technique to produce the test suite. The whole experimental procedure is adapted from our previous work on test case prioritization [17].

H. *Results and Analyses*

1) *Answering RQ*1

For every technique, we compute the APFD results across all the faulty versions and draw box-and-whisker plots for each UNIX program, as shown in Figure 3.

Encouragingly, we find from Figure 3 that both PORA techniques perform outstandingly. In every case, we observe that each PORA technique is significantly more effective than random ordering, all the three total greedy techniques, and all the three ART techniques. Moreover, except for the case of pora-random on gzip, we find the median APFD of each PORA technique is visually more effective than every additional greedy algorithm technique on each subject program. In particular, on sed, every PORA technique has a higher median value than each additional greedy technique as indicated by their non-overlapping notches. In fact, we are not aware of existing techniques from the literature that can be visually more effective than all these peer techniques on the benchmark subjects.

We further conduct one-way analyses of variances (ANOVA) to verify whether the means of the APFD distributions for different techniques differ significantly. For the alpha adjustment procedure, we use Tukey's honestly significance difference (HSD), which is more conservative than the least significance difference (LSD) [19] and is also the default option used by MATLAB for multiple comparisons. The ANOVAs return a *p*-value much less than 0.001, which successfully rejects the null hypothesis at a significance level of 5%.

We select the most effective technique (in terms of mean APFD) from random ordering and every peer algorithm to compare with each PORA technique in turn. The peer techniques chosen are random, total-br, addtl-br, and ART-br-maxmin. For ease of reference, we refer to random ordering or a total greedy, additional greedy, or ART technique as a *nonPORA* technique in this paper.

The results are shown in Table 3. There is a column for pora-random (and pora-distance) that shows the results of performing multiple comparisons between pora-random and the four nonPORA techniques selected. There are two subcolumns within this column. The subcolumn entitled "Single Best Technique" shows the name of the technique that is significantly more effective than any other techniques. The subcolumn entitled "Multiple Best Techniques" shows that the

techniques that are comparable to each other in terms of effectiveness and are significantly more effective than any other techniques in the empirical study.

Table 3. Multiple Comparisons between PORA and other Test Case Prioritization Techniques

	pora-random		pora-distance	
Subject Program	**Single Best Technique**	**Multiple Best Techniques**	**Single Best Technique**	**Multiple Best Technique**
gzip		Addtl-br, Addtl-st, **pora-random**		Addtl-br, Addtl-st, **pora-distance**
sed	**pora-random**		**pora-distance**	
flex		Addtl-br, **pora-random**		Addtl-br, **pora-distance**
grep		Addtl-br, **pora-random**		Addtl-br, **pora-distance**

Table 4. Comparisons of Standard Deviations of APFD Results among Different Techniques

Technique (down) Subject (right)	**gzip**	**sed**	**flex**	**grep**
random	0.090	0.027	0.035	0.052
addtl-br	0.016	0.032	0.109	0.048
addtl-fn	0.094	0.033	0.100	0.053
addtl-st	0.018	0.034	0.110	0.040
total-br	0.066	0.027	0.017	0.036
total-fn	0.317	0.036	0.027	0.042
total-st	0.068	0.020	0.015	0.025
ART-br-maxmin	0.062	0.021	0.036	0.040
ART-st-maxmin	0.262	0.059	0.040	0.122
ART-fn-maxmin	0.077	0.024	0.040	0.024
pora-random	**0.015**	**0.014**	**0.011**	**0.010**
pora-distance	**0.012**	**0.010**	**0.011**	**0.007**
Mean	0.091	0.028	0.046	0.042

The results in Table 3 confirms that both pora-distance and pora-random are either more effective than or as effective as nonPORA techniques.

Furthermore, across the four plots, compared to other techniques, both PORA techniques consistently have small bars. We further compare the standard deviation of the APFD results achieved by these techniques on each subject program. As shown in Table 4, pora-random and pora-distance consistently achieve the smallest standard deviations in terms APFD values for each subject. Moreover, before the introduction of PORA, there was no technique consistently performing best across all the subjects. These results show clearly that our PORA techniques are much more stable than existing techniques in generating effective prioritized test suites across all subject programs.

Combining the results above, our empirical study on the subject programs indicates that PORA techniques can be both highly effective and highly stable in generating prioritized test suites and can be competitive candidates in real-world regression testing practice.

2) Answering RQ2

In this section, we further analyze the time cost of PORA prioritization techniques and compare them with other techniques to help guide practical use. The results are shown in Table 5. We observe that the additional greedy techniques incur much more time cost than the mean prioritization time.

The statement-level ART prioritization technique has a time cost comparable with the mean of all techniques in the last row of the table. The PORA techniques, branch-level ART techniques, greedy technique, and random ordering always use much less time than the mean time cost of all techniques.

In general, the PORA techniques are only slightly slower than random ordering and some greedy techniques, but are much more efficient than existing code-coverage-based additional greedy techniques, branch-level techniques, and statement-level ART techniques. As a result, we conclude that PORA can be efficient in prioritizing test cases in practice.

Table 5. Time Comparisons of Different Techniques (seconds)

Technique (down) Subject (right)	**gzip**	**sed**	**flex**	**grep**
random	0.01	0.01	0.01	0.01
addtl-br	13.91	1.39	6.71	7.54
addtl-fn	19.79	1.78	6.49	6.97
addtl-st	43.28	2.79	22.87	21.72
total-br	0.71	0.12	0.48	0.69
total-fn	0.03	0.00	0.03	0.03
total-st	2.44	0.31	1.88	1.84
ART-br-maxmin	1.15	0.12	0.61	0.89
ART-st-maxmin	2.78	0.31	1.88	2.02
ART-fn-maxmin	0.51	0.06	0.29	0.23
pora-random	**0.50**	**0.05**	**0.25**	**0.20**
pora-distance	**0.52**	**0.07**	**0.31**	**0.28**
Mean	7.14	0.58	3.48	3.54

3) Threats to Validity

To conduct the empirical study, we used many tools, which could have added variability to our results and increase the threats to internal validity. We used several procedures to control these sources of variation. We carefully verified and tested our regression testing tools, which are the same set of tools used in [17]. We used C++ to implement our tools for instrumentation, test suite prioritization, and results analysis. To minimize errors, we have carefully tested our tools to assure correctness.

We only chose C programs in our empirical study because they were still widely used in many real-life applications such as Web servers, UNIX tools, and database servers. A further investigation on subject programs written in other programming languages may help generalize our findings. We used APFD to measure the effectiveness of the studied test case

prioritization techniques. Using other metrics may provide different results.

In our empirical study, we compared PORA with existing code-coverage-based techniques. For all the subject programs, the input data of the test suites provided less differentiable values than the code coverage achieved by these test suites. We tend to believe that our comparisons in terms of APFD do not provide PORA with unfair advantage. To validate this assumption, we have run adapted versions of the total greedy, additional greedy, and ART techniques using the input data as the data source. Our preliminary finding is that the comparison results reported in Table 3 are still valid in that the two PORA techniques are more effective than the adapted techniques and random ordering for some subject programs and as effective as the adapted techniques and random ordering for the remaining subject programs. We do not find PORA beaten by the adapted techniques for any subject program. Regarding the time spent on prioritization, we observe that the adapted additional greedy techniques still run significantly slower than the two PORA techniques. However, the ART techniques and the total greedy techniques run faster than PORA (and yet they are less effective than PORA significantly). Owing to the many dimensions in data analyses, we will leave the reporting of the detailed results to future work.

IV. Related Work

Researchers have proposed many test case prioritization techniques in previous work. In this section, we review related work not discussed in the Introduction section. Integrating multiple aspects to improve regression testing is still a trend.

Wong et al. [32] proposed an approach to combining test suite minimization and prioritization to select cases based on the cost per additional coverage. Walcott et al. [30] proposed a time-aware prioritization technique based on a genetic algorithm to reorder test cases under time constraints. Furthermore, Zhang et al. [36] proposed a set of time-aware test case prioritization techniques using integer linear programming. All the above were code-coverage-based techniques that took the cost and time constraints into consideration. Qu et al. [26] proposed a black-box test cases prioritization technique that grouped the test cases based on their failure exposing history and adjusted their priority dynamically during execution. However, their technique required execution history information that may not be available in practice. We will also study the impact of time constraint on black-box test case prioritization techniques in the future.

Li et al. [19] proposed various search algorithms for test case prioritization based on code coverage information. However, since they were focusing only on the goal of maximizing the code coverage rate while we are focusing on increasing the rate of fault detection, their techniques are not directly comparable to ours. We will also study the effectiveness of adopting genetic algorithms as well as other AI-search strategies for black-box test case prioritization in future work. Jiang et al. [17] proposed a family of coverage-based adaptive random testing techniques to evenly spread the test cases across the code coverage domain. Their study showed that ART techniques can be as effective as additional greedy techniques while involving much lower cost. Our techniques are similar to theirs in the sense that all the techniques make use of randomized algorithms. The major difference lies in that our techniques are driven by the size-proportional allocation strategy. Hao et al. [12] proposed a test case prioritization technique guided by dynamic test case execution outputs. In this way, the coverage information of the unselected test cases on the modified program can be more precise. In [11][35], Hao and her collaborators further proposed a unified test case prioritization approach that encompasses both the total and additional strategies. Mei et al. [21] proposed a static approach to guiding the prioritization of JUnit test cases. You et al. [34] performed an empirical study on time-aware test case prioritization techniques. Taneja et al. [29] proposed to use dynamic symbolic execution technique to explore those path affected by code change for generating regression test suite. Yoo and Harman [33] performed a systematic survey on regression testing minimization, selection, and prioritization. Huang et al. [14] proposed a cost-cognizant test case prioritization technique based on historical information. Arafeen and Do [1] proposed a new test case prioritization technique by incorporating the information on both requirement clustering and traditional code analysis information. Industrial case studies on test case prioritization in continuous integration scenarios have also been reported [20].

Researchers also studied the problem of regression testing of service-oriented applications. Mei et al. [22] proposed a hierarchy of prioritization techniques for the regression testing of service-oriented business applications by modeling business process, XPath, and WSDL information. In [24], they also studied the problem of black-box test case prioritization of service-oriented applications based on the coverage information of WSDL tags. In [23], they further proposed a preemptive regression testing technique to address the service evolution problem during regression testing.

V. Conclusion

In this paper, we propose **P**roportion-**O**riented **R**andomized **A**lgorithm (**PORA**) for test case prioritization. The PORA techniques search for a highly effective permutation of the test suite by minimizing its distance against a hierarchy of sequences of centroids. Our experiment shows that PORA techniques are always as effective as, if not more effective than, some of the well accepted nonPORA techniques in the literature, including the total greedy, additional greedy, and ART techniques using code coverage information. Furthermore, the results show that the PORA techniques are consistent more stable than nonPORA techniques evaluated in the empirical study. Finally, the PORA techniques are also efficient, which makes it a good choice for practical use. In future work, we will further investigate how to extend the idea of PORA beyond test case prioritization. In particular, the concept of partition testing based on while-box data is well studied in previous work. It will be interesting to extend the idea of PORA to guide the test case prioritization with code coverage information. We will also generalize PORA and further

evaluate it on more benchmarks using both input data and code coverage.

REFERENCES

[1] M.J. Arafeen and H. Do, "Test case prioritization using requirements-based clustering," *Proceedings of the IEEE 6th International Conference on Software Testing, Verification and Validation* (*ICST '13*), IEEE Computer Society, 2013, pp. 312–321.

[2] D. Arthur and S. Vassilvitskii, "k-means++: The advantages of careful seeding," *Proceedings of the 18th Annual ACM-SIAM Symposium on Discrete Algorithms* (*SODA '07*), Society for Industrial and Applied Mathematics, 2007, pp. 1027–1035.

[3] F.T. Chan, T.Y. Chen, I.K. Mak, and Y.T. Yu, "Proportional sampling strategy: guidelines for software testing practitioners," *Information and Software Technology*, vol. 38, no. 12, 1996, pp. 775–782.

[4] T.Y. Chen and Y.T. Yu, "On the expected number of failures detected by subdomain testing and random testing," *IEEE Transactions on Software Engineering*, vol. 22, no. 2, 1996, pp. 109–119.

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

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

[7] S.G. Elbaum, G. Rothermel, S. Kanduri, and A.G. Malishevsky, "Selecting a cost-effective test case prioritization technique," *Software Quality Control*, vol. 12, no. 3, 2004, pp. 185–210.

[8] G. Forman and E. Kirshenbaum, "Extremely fast text feature extraction for classification and indexing," *Proceedings of the 17th ACM Conference on Information and Knowledge Management* (*CIKM '08*), ACM, 2008, pp. 1221–1230.

[9] A. Gonzalez-Sanchez, R. Abreu, H.-G. Gross, and A. van Gemund, "A diagnostic approach to test prioritization," Technical Report TUD-SERG-2010-007, Software Engineering Research Group, Delft University of Technology, 2010.

[10] D. Hamlet and R.N. Taylor, "Partition testing does not inspire confidence," *IEEE Transactions on Software Engineering*, vol. 16, no. 12, 1990, pp. 1402–1411.

[11] 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, 2014, pp. 10:1–10:31.

[12] D. Hao, X. Zhao, and L. Zhang, "Adaptive test-case prioritization guided by output inspection," *Proceedings of the IEEE 37th Annual Computer Software and Applications Conference* (*COMPSAC '13*), IEEE Computer Society, 2013, pp. 169–179.

[13] M.J. Harrold, R. Gupta, and M.L. Soffa, "A methodology for controlling the size of a test suite," *ACM Transactions on Software Engineering and Methodology*, vol. 2, no. 3, 1993, pp. 270–285.

[14] 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, no. 3, 2012, pp. 626–637.

[15] B. Jiang, T.H. Tse, W. Grieskamp, N. Kicillof, Y. Cao, and X. Li, "Regression testing process improvement for specification evolution of real-world protocol software," *Proceedings of the 10th International Conference on Quality Software* (*QSIC '10*), IEEE Computer Society, 2010, pp. 62–71.

[16] B. Jiang, T.H. Tse, W. Grieskamp, N. Kicillof, Y. Cao, X. Li, and W.K. Chan, "Assuring the model evolution of protocol software specifications by regression testing process improvement," *Software: Practice and Experience*, vol. 41, no. 10, 2011, pp. 1073–1103.

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

[18] H.K.N. Leung and L.J. White, "Insights into regression testing," *Proceedings of the IEEE International Conference on Software Maintenance* (*ICSM '89*), IEEE Computer Society, 1989, pp. 60–69.

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

[20] D. Marijan, A. Gotlieb, and S. Sen, "Test case prioritization for continuous regression testing: An industrial case study," *Proceedings of the 2013 IEEE International Conference on Software Maintenance* (*ICSM '13*), IEEE Computer Society, 2013, pp. 540–543.

[21] 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, 2012, pp. 1258–1275.

[22] L. Mei, Y. Cai, C. Jia, B. Jiang, W.K. Chan, Z. Zhang, and T.H. Tse, "A subsumption hierarchy of test case prioritization for composite services," *IEEE Transactions on Services Computing*, 2014, doi: 10.1109/TSC.2014.2331683.

[23] L. Mei, W.K. Chan, T.H. Tse, B. Jiang, and K. Zhai, "Preemptive regression testing of workflow-based web services," *IEEE Transactions on Services Computing*, 2014, doi: 10.1109/TSC.2014.2322621.

[24] L. Mei, W.K. Chan, T.H. Tse, and R.G. Merkel, "Tag-based techniques for black-box test case prioritization for service testing," *Proceedings of the 9th International Conference on Quality Software* (*QSIC '09*), IEEE Computer Society, 2009, pp. 21–30.

[25] A.K. Onoma, W.-T. Tsai, M. Poonawala, and H. Suganuma, "Regression testing in an industrial environment," *Communications of the ACM*, vol. 41, no. 5, 1998, pp. 81–86.

[26] B. Qu, C. Nie, B. Xu, and X. Zhang, "Test case prioritization for black box testing," *Proceedings of the 31st Annual International Computer Software and Applications Conference* (*COMPSAC '07*), vol. 1, IEEE Computer Society, 2007, pp. 465–474.

[27] G. Rothermel and M.J. Harrold, "A safe, efficient regression test selection technique," *ACM Transactions on Software Engineering and Methodology*, vol. 6, no. 2, 1997, pp. 173–210.

[28] 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, 2001, pp. 929–948.

[29] K. Taneja, T. Xie, N. Tillmann, and J. de Halleux, "eXpress: Guided path exploration for efficient regression test generation," *Proceedings of the 2011 International Symposium on Software Testing and Analysis* (*ISSTA '11*), ACM, 2011, pp. 1–11.

[30] K.R. Walcott, M.L. Soffa, G.M. Kapfhammer, and R.S. Roos, "TimeAware test suite prioritization," *Proceedings of the 2006 ACM SIGSOFT International Symposium on Software Testing and Analysis* (*ISSTA '06*), ACM, 2006, pp. 1–12.

[31] D.H. Wolpert and W.G. Macready, "No free lunch theorems for optimization," *IEEE Transactions on Evolutionary Computation*, vol. 1, no. 1, 1997, pp. 67–82.

[32] W.E. Wong, J.R. Horgan, S. London, and H. Agrawal, "A study of effective regression testing in practice," *Proceedings of the 8th International Symposium on Software Reliability Engineering* (*ISSRE '97*), IEEE Computer Society, 1997, pp. 264–274.

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

[34] D. You, Z. Chen, B. Xu, B. Luo, and C. Zhang, "An empirical study on the effectiveness of time-aware test case prioritization techniques," *Proceedings of the 2011 ACM Symposium on Applied Computing* (*SAC '11*), ACM, 2011.

[35] L. Zhang, D. Hao, L. Zhang, G. Rothermel, and H. Mei, "Bridging the gap between the total and additional test-case prioritization strategies," *Proceedings of the 2013 International Conference on Software Engineering* (*ICSE '13*), IEEE, 2013, pp. 192–201.

[36] L. Zhang, S.-S. Hou, C. Guo, T. Xie, and H. Mei, "Time-aware test-case prioritization using integer linear programming," *Proceedings of the 18th ACM SIGSOFT International Symposium on Software Testing and Analysis* (*ISSTA '09*), ACM, 2009, pp. 213–224.