Test Case Prioritization Using Lexicographical Ordering

Sepehr Eghbali and Ladan Tahvildari, *Senior Member, IEEE*

Abstract—Test case prioritization aims at ordering test cases to increase the *rate of fault detection*, which quantifies how fast faults are detected during the testing phase. A common approach for test case prioritization is to use the information of previously executed test cases, such as coverage information, resulting in an iterative (greedy) prioritization algorithm. Current research in this area validates the fact that using coverage information can improve the rate of fault detection in prioritization algorithms. The performance of such iterative prioritization schemes degrade as the number of ties encountered in prioritization steps increases. In this paper, using the notion of lexicographical ordering, we propose a new heuristic for breaking ties in coverage based techniques. Performance of the proposed technique in terms of the rate of fault detection is empirically evaluated using a wide range of programs. Results indicate that the proposed technique can resolve ties and in turn noticeably increases the rate of fault detection.

Index Terms—Regression testing, test case prioritization, lexicographical ordering

1 Introduction

Software evolution and modification are essential ingredients of any software development process. While changes are continuously made to the software in order to introduce new functionalities and/or to repair detected faults, they can adversely inject new faults into the unchanged parts. Software *regression testing* attempts at reducing this risk by re-running a set of test cases after modifying the software. The dominant strategy in regression testing is to re-run all test cases applied to the earlier versions of the software. However, the size of the test suite tends to grow as the software evolves which can make the cost of executing the test suite prohibited. For example, running the entire test suite for an industrial project reported in [1] and [2] has taken seven weeks. As another example, at Google, developers modify code more than 20 times per minute requiring 100 million test executions per day [3].

Researchers have studied techniques to reduce the cost of regression testing. These techniques can be categorized into two groups: *Regression Test case Selection* (RTS) and *Regression Test case Prioritization* (RTP). Test case selection techniques focus on covering the changed code between versions of the software under test [4], [5], [6], [7], [8], [9], [10], [11], [12]. On the other hand, test case prioritization techniques concern the identification of ideal ordering of test cases that enhances the rate of fault detection [2], [13], [14], [15], [16], [17], [18], [19], [20], [21], [22]. While these two approaches differ in their ideas and goals, they share common features. Techniques in both classes rely on extracting relevant features from test cases. Prioritization and selection techniques incorporate such features to differentiate between test cases. Therefore, different techniques may favour dissimilar orderings based on their selected features.

The test case prioritization problem is defined as finding a permutation of test cases for which the value of an objective function is maximized. Rothermel et al. [18] formally define the test case prioritization problem as follows:

Given: T, a test suite; PT, the set of permutations of T; g, a function from PT to the real numbers.

Problem: Find $T' \in PT$ such that

$$\forall T'' \in PT, \quad g(T') \geq g(T'').$$

Ideally, test case prioritization should search for an ordering of test cases that results in early fault detection. However, the faults are not known before execution of the test suite. Therefore, a surrogate has to be used which statistically or heuristically correlates with the faults, expecting that maximizing the surrogate will lead to maximizing the rate of fault detection. Many techniques for test case prioritization adopt code coverage information obtained through instrumentation and execution of the software under test [17], [23], [24]. Accordingly, these are referred to as *coverage-based* techniques. No consensus has emerged yet whether there is a high correlation between coverage and fault detection. While many studies have emphasized the presence of such a correlation [25], [26], [27], others indicate that coverage is not always positively correlated with fault detection [28], [29].

Two well-studied iterative and greedy coverage-based techniques are *Total Technique* (TT) and *Additional Technique* (AT) [18], [21], [30]. Both techniques iterate n times, where n is the number of test cases. In each iteration (step), they select one test case (from the pool of not-so-far-selected test cases) to be inserted into the ordering as the next item. TT prioritizes test cases through maximizing the total number of covered entities, while AT chooses the test case that covers the highest number of entities that have not been

- *The authors are with the Department of Electrical and Computer Engineering, 200 University Ave West, University of Waterloo, Waterloo, Ontario N2L 3G1. E-mail: {s2eghbal, ltahvild}@uwaterloo.ca.*

Manuscript received 2 Dec. 2014; revised 13 Jan. 2016; accepted 6 Mar. 2016. Date of publication 20 Apr. 2016; date of current version 16 Dec. 2016. Recommended for acceptance by M. Cohen.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TSE.2016.2550441

0098-5589 © 2016 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission.
See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

covered in previous steps. Therefore, at each step, TT searches for the test case with the maximum coverage. As a result, each test case selection is independent of previous selections. However, in AT, each test case selection takes into account the effect of test cases already in the ordering. The current literature on RTP abounds with various coverage based techniques. The available evidence seems to suggest that additional-type techniques tend to outperform other coverage-based alternatives [17], [21], [31]. However, a particular test case prioritization technique often cannot offer a superior performance in relation to all programs.

At each step of AT, there is a possibility that a tie occurs which means more than one test case have the highest coverage of not-yet-covered entities. If a tie occurs, AT implicitly assumes that all remaining candidates are equally important and selects one of them randomly. Previous empirical studies indicate that random test case ordering can be ineffective. Traditionally, random ordering has been considered as a lower bound control technique [18], [32], [33]. In Section 2.2, we empirically show that, surprisingly, ties occur with high likelihood.

Some earlier studies address the problem of tie breaking in regression testing tasks. Jiang et al. [33] incorporate *Adaptive Random Testing* (ART) for prioritizing test cases which can also be applied for breaking ties. Although their algorithm outperforms random ordering, their results indicate that AT outperforms ART-based algorithms. Sampath et al. [34] propose a multi-criteria algorithm in RTS. They use the coverage information as the primary criterion and usage-based requirements as the tie-breaker. Similarly, Lin and Huang [35] incorporate an additional criterion for breaking ties. They use branch coverage as the primary criterion and *definition-use pair coverage* as the tie-breaker.

To break ties, instead of random selection or incorporating additional criteria, we propose a coverage-based technique for test case prioritization based on the notion of lexicographical ordering. The proposed technique uses the coverage information of previously executed test cases to break ties. Our proposed technique, unlike traditional coverage based alternatives, does not categorize software entities into distinct groups as *covered* and *not-yet-covered*. Instead, different entities of the software are ranked based on the number of times they are covered until the current step. Using this approach, all entities are considered in the selection of the next test case; while entities that are covered less are given a higher priority.

To simplify the explanation, the article first presents the main idea behind our proposed technique in the form of a basic algorithm. Subsequently, the basic algorithm is modified to reduce the computational cost while maintaining the same functionality. Then, we prove that these algorithms are equivalent and return the same ordering. Next, the proposed technique is compared with some other existing prioritization techniques on some well-known programs.

In particular, the contributions of this paper are as follows:

- An algorithm, called Generalized AT using Lexicographical Ordering (GeTLO), is proposed for breaking ties in AT using the notion of lexicographical ordering.
- As the proposed algorithm may face ties of its own, a recursive technique is proposed for further breaking of ties.
- The performance of the proposed technique is gauged on 6 Java programs available from Software-artifact Infrastructure Repository (SIR) [32].

The rest of the paper is organized as follows. Section 2 presents the proposed technique for test case prioritization. Section 3 introduces the enhanced algorithm for breaking ties. Section 4 presents the evaluation of the proposed technique and its comparison with some existing alternatives. Section 5 describes the related work. Finally, Section 6 concludes the paper and presents some potential directions for future research.

2 Proposed Technique

2.1 Notations

Sets are shown by calligraphic letters such as $\mathcal{X}$. The cardinality of a set is shown by $|\mathcal{X}|$, i.e., $\mathcal{X} = \{x_i\}_{i=1}^{|\mathcal{X}|}$. Matrices appear in bold capital letters e.g., $\mathbf{X}$. More specifically, a matrix of size $a \times b$ is shown by $\mathbf{X} = [x(i,j)]_{a\times b}$. Vectors are shown using bold small letters e.g., $\mathbf{x}$. A vector of size $1 \times a$ is shown by $\mathbf{x} = [x(i)]_{1\times a}$. The transpose of matrix $\mathbf{X}$ is shown by $\mathbf{X}^t$.

Consider a certain version of a program composed of n software entities (class, method or statement), $\mathcal{O} = \{o_i\}_{i=1}^{n}$, and a test suite, $\mathcal{T}$, which consists of m test cases, $\mathcal{T} = \{t_i\}_{i=1}^{m}$. Also, a binary coverage matrix $\mathbf{C} = [c(i,j)]_{m\times n} \in \{0,1\}^{m\times n}$ denotes entities that each test case covers. For example, $c(i,j)$ is equal to one if test case t_i covers entity o_j, and zero otherwise. Also, we can write $\mathbf{C} = [\mathbf{cv}_1^t, \mathbf{cv}_2^t, \ldots, \mathbf{cv}_m^t]^t$ where $\mathbf{cv}_k = [c(i)]_{1\times n} \in \{0,1\}^{1\times n}$ is the coverage vector of the kth test case, $k \in \{1, 2, \ldots, m\}$.

2.2 A Motivating Example

Before explaining the details of the proposed technique, we first review the steps that AT takes to prioritize a set of test cases. In the first step, AT chooses the test case with the maximum coverage. To continue, in the second step, AT finds the test case with the maximum coverage of not-yet-covered entities (not covered by the first test case). In the third step, AT searches for the test case which yields the maximum coverage of entities that the first and the second test case have not covered. In other words, in each step, AT selects the test case that provides the maximum coverage for the not-yet-covered entities. This procedure continues until the ordering is complete. However, there is a chance that a tie occurs at each step.

A tie occurs when more than one test case covers the maximum number of not-yet-covered entities. In such cases, AT resorts to random selection.

Example A.1. Consider a test suite with four test cases, t_1, t_2, t_3, t_4, with the following coverage vectors:

$$\mathbf{C} = \begin{array}{c} \\ t_1 \\ t_2 \\ t_3 \\ t_4 \end{array} \begin{pmatrix} o_1 & o_2 & o_3 & o_4 & o_5 & o_6 & o_7 & o_8 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}.$$

If AT is applied to $\mathbf{C}$, the first selected test case is t_1, as it covers more entities than t_2, t_3, and t_4. In the next step,

TABLE 1
Average Percentage (Averaging Is Over Different Versions of Programs) and Number of Candidates for AT

	Method level		Basic block level		
Program	**Ave. % steps with tie**	**Ave. # candidates**	**Ave. % steps with tie**	**Ave. # candidates**	**Ave. # test cases**
Ant	93%	213.2	83%	48.5	399.5
Galileo	93%	242.2	94%	210.2	865.9
Jmeter	83%	18.7	49%	4.1	72.4
Jtopas	95%	54.1	86%	26.7	154.3
Nano	99%	111.2	95%	45.7	213.6
XML	87%	52.3	71%	7.6	81.6

AT selects t_2 since it covers two of the not-yet-covered entities, namely o_3 and o_4. At the end of the second step, entities o_1 and o_2 are not covered so far. In the third step, AT faces a tie as both t_3 and t_4 cover one of the not-yet-covered entities. Our idea to break the tie is to also consider the covered entities. In this example, t_3 covers o_5 which is covered twice until the current step. However, t_4 covers o_8 which is covered only once , thus our algorithm selects t_4 as the next test case.

To see how often ties occur, we have implemented AT and ran it on 45 versions of 6 different Java programs from SIR [32]. The details of these programs are discussed in Section 4. Table 1 shows the average percentage of steps with ties, as well as the average number of candidate test cases involved in a tie at different coverage granularity levels. For each version of programs, the number of steps with tie and also the average number of candidates involved in each tie is counted. Then, these values are averaged over different versions of each program. Surprisingly, ties occur in a large number of steps of the AT algorithm. Also, the number of the candidates involved at any given tie is typically high on the average. With the exception of *Galileo*, Table 1 shows that the number and the size of ties decrease for finer levels of granularity.

Resorting to random selection for breaking ties can significantly degrade the performance of prioritization algorithms because of two basic observations: i) there are many cases of ties, and ii) there are typically many potential candidates to select from in the case of a tie. Consequently, random selection results in blind selection of one candidate from a large pool. Therefore, it would be advantageous if test cases are selected in a systematic way when ties occur.

To break a tie, we consider not only the not-yet-covered entities, but also covered ones. The key idea of our technique for resolving ties is as follows: *Less covered entities should have higher priority for coverage*. To implement this idea, among the candidates that form any given tie, the one with the maximum coverage of one-time-covered entities is selected. Likewise, if a tie occurs again (or if there is no one-time-covered entity), entities that are covered twice are considered, and so on. It turns out that the proposed procedure can be described in the language of lexicographical ordering.

2.3 Lexicographical Order

Definition. *Consider vectors* $\mathbf{x} = [x(1), \ldots, x(k)] \in \mathbb{R}^k$ *and* $\mathbf{y} = [y(1), \ldots, y(k)] \in \mathbb{R}^k$ *where* $\mathbb{R}$ *is the set of real numbers.* $\mathbf{x}$ *is said to have a higher lexicographical rank as compared to* $\mathbf{y}$*, shown as* $\mathbf{x} \succ \mathbf{y}$ *or* $\mathbf{y} \prec \mathbf{x}$*, if considering the vectors' components from left to right, the first component which is not equal in* $\mathbf{x}$ *and* $\mathbf{y}$ *has a larger value in* $\mathbf{x}$*. In mathematical notations, this means:*

$$\exists m > 0 \quad \text{s.t.} \quad \forall n < m, x(n) = y(n) \text{ and } x(m) > y(m).$$

Example B. Consider the following four vectors:

$$\mathbf{x} = [1, 2, 5]$$
$$\mathbf{y} = [1, 2, 6]$$
$$\mathbf{z} = [1, 4, 7]$$
$$\mathbf{u} = [8, 2, 1].$$

These vectors would be ordered lexicographically as follows:

$$\mathbf{u} \succ \mathbf{z} \succ \mathbf{y} \succ \mathbf{x}.$$

Example C. Consider the following coverage vectors:

$$\mathbf{cv}_1 = [1, 0, 0]$$
$$\mathbf{cv}_2 = [1, 1, 0]$$
$$\mathbf{cv}_3 = [0, 1, 0]$$
$$\mathbf{cv}_4 = [0, 1, 1].$$

These coverage vectors would be ordered lexicographically as follows:

$$\mathbf{cv}_2 \succ \mathbf{cv}_1 \succ \mathbf{cv}_4 \succ \mathbf{cv}_3.$$

Lexicographical ordering is not restricted to vectors with elements in $\mathbb{R}$. The definition can be easily extended to any vectors composed of elements in partially ordered sets [36].

2.4 Basic Algorithm

Similar to AT, the basic algorithm prioritizes a test suite comprising of n test cases in n steps. In each step, it selects and includes one of the test cases in the ordering. To perform such a selection, it considers all the entities even if they have been covered in the previous steps. However, the entities that are covered less will be given a higher priority. This procedure can be implemented efficiently using the notion of lexicographical ordering.

Let set $\mathcal{A}^s$ denote the set of test cases in the ordering until step s, where $s \in \{1, \ldots, n\}$. It is clear that $|\mathcal{A}^s| = s$. Also, the total coverage of selected test cases until step s forms a vector, called *Cumulative Coverage vector*, denoted by $\mathbf{cc}^s$ where $\mathbf{cc}^s \in \mathbb{N}^n$, $\mathbb{N}$ is the set of non-negative integers. Formally, we have:

$$\mathbf{cc}^s = \left[\sum_{i\in\mathcal{A}^s} c(i,1), \sum_{i\in\mathcal{A}^s} c(i,2), \ldots, \sum_{i\in\mathcal{A}^s} c(i,n)\right]. \quad (1)$$

Thus, for all $j \in \{1, 2, \ldots, n\}$, $cc^s(j)$ shows that number of times that the jth entity is covered by the test cases in $\mathcal{A}^s$.

Example A.2. In Example A.1, if the algorithm selects test cases t_1 and t_2 by the end of step 2, then we have $\mathbf{cc}^2 = [0, 0, 1, 2, 1, 1, 1, 1]$.

The proposed algorithm favours test cases that maximize the smallest element(s) of the vector $\mathbf{cc}^s$. Note that several elements of $\mathbf{cc}^s$ may have the same smallest value. The smallest element(s) represent the entities that have been covered less than others. If there exist some entities that are not covered so far, then the smallest element(s) of $\mathbf{cc}^s$ will be equal to zero. AT chooses the test case that can cover the highest number of such not-yet-covered entities. Nevertheless, if there are more than one test case satisfying this criterion (tie occurs), AT selects randomly, whereas our proposed technique exploits this freedom to maximize the second smallest element of $\mathbf{cc}^s$, and so on. To implement this procedure, it sorts the entities base on their so far cumulative coverage value and favours increasing the cumulative coverage of entities with smaller values.

By sorting the elements of $\mathbf{cc}^s$ in ascending order, we form the vector *Ordered Cumulative Coverage*, denoted by $\mathbf{occ}^s$ where the smallest element of $\mathbf{cc}^s$ is the leftmost element of $\mathbf{occ}^s$. Without loss of generality, we assume that the ordering of the elements in $\mathbf{occ}^s$ preserves the order of elements $\mathbf{cc}^s$ which have the same value. The basic algorithm relies on the lexicographical ranks of **occ** vectors. At step s, it searches for the test case that maximizes the lexicographical rank of $\mathbf{occ}^{s+1}$. For this purpose, starting with $\mathbf{cc}^0$ equal to a vector of all zeros, at each step, say step s, the algorithm adds the coverage vector of each of the remaining test cases to $\mathbf{cc}^s$ (to form the set of potential $\mathbf{cc}^{s+1}$ vectors). Then, it sorts the resulting vectors and finds the one with the highest lexicographical rank. This vector is a result of adding up $\mathbf{cc}^s$ with a coverage vector such as $\mathbf{cv}_i$. This means that selecting the ith test case, t_i, in the current step results in $\mathbf{occ}^{s+1}$ with the highest lexicographical rank. Therefore, t_i is selected as the next test case to maximize the lexicographical rank of $\mathbf{occ}^{s+1}$. Finally, the value of $\mathbf{cc}^s$ is updated to $\mathbf{cc}^{s+1}$ by adding the coverage vector of the selected test case to $\mathbf{cc}^s$. The algorithm continues in the same manner for the remaining test cases. Note that this procedure may still result in ties of its own.

Example A.3. In Example A.1, at the end of step 2, test cases t_1 and t_2 are selected and we have $\mathbf{cc}^2 = [0, 0, 1, 1, 2, 1, 1, 1]$, thus $\mathbf{occ}^2 = [0, 0, 1, 1, 1, 1, 1, 2]$. Selecting t_3 as the next test case results in $\mathbf{occ}^3 = [0, 1, 1, 1, 1, 1, 1, 3]$, while selecting t_4 results in $\mathbf{occ}^3 = [0, 1, 1, 1, 1, 1, 2, 2]$. As the latter has a higher lexicographical rank, t_4 is selected as the next test case.

Algorithm I shows the pseudo-code of the basic algorithm. Lines 1 and 2 perform initialization. Lines 3 to 16 construct the main loop of the algorithm. In each iteration of this loop, a test case is selected and added to the list $\mathcal{A}$ which is the output of the algorithm. Lines 5 to 13 add the coverage vector of each of the remaining test cases to **cc** and find the one yielding the highest lexicographical order. Line 14 inserts the selected test case into the order. Finally, line 15 updates **cc**. Note that the output of Algorithm I and AT is the same if no ties occur.

Algorithm 1. Basic Algorithm

Input: Coverage matrix $\mathbf{C} = [c(i,j)]_{m\times n}$,
test suite $\mathcal{T} = \{t_1, \ldots, t_m\}$
Output: Order of test cases $\mathcal{A}$

```
1: cc ← 0_{1×n}
2: A ← empty list
3: while A.size() < m do
4:    h ← −1_{1×n}
5:    for i = 1 to m do
6:       if t_i ∉ A then
7:          tmp ← sort(cv_i + cc)
8:          if h ≺ tmp then
9:             h ← tmp
10:            tbest ← t_i, cbest = cv_i
11:         end if
12:      end if
13:   end for
14:   A.add(tbest)
15:   cc ← cc + cbest
16: end while
17: return A
```

The main loop iterates for m times where in each iteration the test case with the highest lexicographical rank is selected. In particular, each coverage vector is added to the current **cc** in $O(m)$ time, sorted in $O(n \log n)$ time and compared with other vectors in $O(n)$ time. Therefore, the total complexity of Algorithm I is $O(m^2 n^2 \log n)$.

2.5 Generalized AT Using Lexicographical Ordering

In this section, we present a greedy algorithm called *Generalized AT using Lexicographical Ordering* which is equivalent to the basic procedure but has lower computational cost.

To explain this algorithm, we need some extra notations. Recall that $\mathbf{cc}^s$ and $\mathbf{occ}^s$ comprise of identical elements but in different orderings. In other words, they are just a permutation of each other. To keep the correspondence between these two vectors, we use the following notation

$$\Pi^s(\mathbf{cc}^s) = \mathbf{occ}^s. \quad (2)$$

where Π^s is a permutation function which specifies the ordering relationship between $\mathbf{cc}^s$ and $\mathbf{occ}^s$. As mentioned before, the permutation function preserves the order of equal elements (those that have the same value) in $\mathbf{cc}^s$. Since $\mathbf{occ}^s$ and $\mathbf{cv}_i$ vectors have the same size, we can apply the same permutation to each coverage vector, $\mathbf{cv}_i$ $i = \{1, 2, \ldots, m\}$, to form the *Ordered Coverage Vector*, $\mathbf{oc}_i^s$:

$$\Pi^s(\mathbf{cv}_i) = \mathbf{oc}_i^s. \quad (3)$$

where $\mathbf{oc}_i^s$ is the permutation of $\mathbf{cv}_i^s$ according to Π^s.

Using $\mathbf{occ}^s$, two other vectors are formed as explained in steps (1) and (2) below:

Step 1: Since $\mathbf{occ}^s$ is sorted, the elements with the same value are adjacent to each other. We group the elements of $\mathbf{occ}^s$ according to their values into r^s partitions, indexed from left to right by $1, \ldots, r^s$, such that all the elements within a partition have the same value. Based on these partitions, we form the vector $\mathbf{q}^s = [q^s(1), \ldots, q^s(r^s)]$ where $q^s(i)$ indicates the number of elements in the ith partition.

Example A.4. In Example A.3, we have $\mathbf{occ}^2 = [0, 0, 1, 1, 1, 1, 1, 2]$, thus $\mathbf{q}^2 = [2, 5, 1]$.

Step 2: Based on the values $q^s(1)$ to $q^s(r^s)$, we partition the elements of all the remaining $\mathbf{oc}_i^s$ vectors into r^s partitions from left to right corresponding to the partitions of $\mathbf{occ}^s$. Using these values, we form the vector $\mathbf{w}^s = [w^s(1), \ldots, w^s(r^s)]$, called the *Weight Vector*. The element $w^s(i)$ of the vector $\mathbf{w}^s$ is the weight of the ith partition, where weight is the sum of the binary values within that partition:

$$\forall i \in \{1, 2, \ldots, r^s\}, \quad w^s(i) = \sum_{j=b_i}^{b_i+q^s(i)-1} \mathbf{oc}^s(j), \tag{4}$$

where b_i is the starting index of the ith partition, i.e.,:

$$b_i = \begin{cases} 1 & \text{if } i = 1 \\ b_{i-1} + q^s(i-1) & \text{otherwise.} \end{cases} \tag{5}$$

Example A.5. In Example A.3, we have $\mathbf{occ}^2 = [0, 0, 1, 1, 1, 1, 1, 2]$, thus $\mathbf{oc}_3^2 = [0, 1, 0, 0, 0, 0, 0, 1]$, and $\mathbf{oc}_4^2 = [1, 0, 0, 0, 1, 0, 0, 0]$. Consequently, the weight vectors corresponding to t_3 and t_4 are $\mathbf{w}_3^2 = [1, 0, 1]$ and $\mathbf{w}_4^2 = [1, 1, 0]$, respectively.

As a result, corresponding to each $\mathbf{oc}_t^s$, we have a vector composed of r^s non-negative integer elements. Finally, let ψ_i^s denotes the set of indices in the ith partition.

Algorithm 2. GeTLO

```
Input: Coverage matrix C = [c(i,j)]_{m×n},
       test suite T = {t_1, ..., t_m}
Output: Order of test cases A
 1: cc ← 0_{1×n}
 2: A ← empty list
 3: while A.size() < m do
 4:   occ = sort(cc)
 5:   Find permutation Π such that Π(cc) = occ
 6:   q = computeq(occ)
 7:   h ← −1_{1×r}
 8:   for i = 1 to m do
 9:     if t_i ∉ A then
10:       oc_i ← Π(c_i)
11:       w_i ← computew(oc_i, q)
12:       if h ≺ w_i then
13:         tbest ← t_i, cbest = c_i
14:       end if
15:     end if
16:   end for
17:   A.add(tbest)
18:   cc ← cc + cbest
19: end while
20: return A
```

To prioritize test cases using Algorithm 1, first each of the $\mathbf{cv}_i$ vectors is added to $\mathbf{cc}^s$. Then, the resulting vectors are sorted and the one with the highest lexicographical rank is chosen. However, it will be shown that the sorting step is not necessary and can be replaced with a permutation and addition to compute the weight vectors. As will be proved later, this modification does not change the functionality, but results in a lower computational cost. GeTLO prioritizes the test suite based on the lexicographical ranks of weight vectors; at each step, it chooses the test case such that the corresponding $\mathbf{w}_i^s$ has the highest lexicographical rank.

Algorithm 2 shows the pseudo code of the procedure with lower computational cost. Lines 1 and 2 are used for initializations. Lines 3 to 19 form the main loop. In each iteration of the main loop, first, permutation Π and vector $\mathbf{q}$ are computed by the function computeq(.). This function simply takes the $\mathbf{occ}$ vector as the input and returns the vector $\mathbf{q}$. Afterwards, to select a test case, the $\mathbf{w}_i$ vector of each test case is computed based on the corresponding value of $\mathbf{oc}$ by the function computew(.). The inputs of this function are oc^i and $\mathbf{q}$ vectors. It adds the values in each partition to form $\mathbf{w}_i$. Then, lines 8 to 16 find the test case that corresponds to $\mathbf{w}_i$ with the highest lexicographical rank. Lines 17 and 18 insert the selected test case into the ordering and update $\mathbf{cc}$. Note that, if the highest lexicographical rank is shared among several test cases, instead of resorting to random selection, the algorithm selects the candidate with the smallest index. The reason for such a selection is to avoid randomness in the final outcome.

Example A.6. Consider the coverage matrix of Example A.1. Until step 3 no ties occurs, therefore, the first two selected test cases by GeTLO are identical to those selected by AT. These are t_1 and t_2 resulting in $\mathbf{cc}^2 = [0, 0, 1, 1, 2, 1, 1, 1]$ and $\mathbf{occ}^2 = [0, 0, 1, 1, 1, 1, 1, 2]$. To select the third test case, the weight vectors are formed as follows:

$$\mathbf{cc}^2 = [0,0,1,1,2,1,1,1] \Rightarrow$$

$$\mathbf{occ}^2 = \left[\underbrace{0,\ 0,}_{1^{\text{st}}\text{partition}} \underbrace{1,1,1,1,1,}_{2^{\text{nd}}\text{partition}} \underbrace{2}_{3^{\text{rd}}\text{partition}} \right]$$

$$\Rightarrow \begin{cases} r^s = 3 \\ \mathbf{q}^s = [2,\ 5,\ 1] \\ b_1 = 1, b_2 = 3, b_3 = 8 \\ \psi_1^s = \{1,2\}, \psi_2^s = \{3,4,5,6,7\}, \psi_3^s = \{8\} \end{cases}$$

$$\mathbf{cv}_3 = [0,1,0,0,1,0,0,0] \Rightarrow$$
$$\mathbf{oc}_3^2 = \Pi^2(\mathbf{cv}_3) = [0,1,0,0,0,0,0,1] \Rightarrow \mathbf{w}_3^2 = [1,\ 0,\ 1]$$

$$\mathbf{cv}_4 = [1,0,0,0,0,0,0,1] \Rightarrow$$
$$\mathbf{oc}_4^2 = \Pi^2(\mathbf{cv}_4) = [1,0,0,0,0,0,1,0] \Rightarrow \mathbf{w}_4^2 = [1,\ 1,\ 0].$$

Recall that to form vectors $\mathbf{w}_i^s$, the entities with the same cumulative coverage values are grouped together as they are of the same preference in terms of being covered in the next iteration. If Algorithm 1 is applied to the coverage vectors of Example A.1, in the third step, $s = 3$, it would choose t_4 which would result in the $\mathbf{occ}^3$ with a higher lexicographical order. The same conclusion could be derived by using Algorithm 2 as we have $\mathbf{w}_3^2 \prec \mathbf{w}_4^2$.

Next, we analyze the time complexity of GeTLO algorithm. To choose a test case at each iteration, first vector **cc** is sorted and then permutation Π is defined in $O(n \log n)$ time. Computing the vector q takes $O(n)$ time. Then, for each of the remaining test cases, the values of $\mathbf{w}_i$ is compared lexicographically with each other to find the one with the highest rank. These operations take $O(mb)$ time where b is the number of partitions. This selection and readjustment must be performed for each test case selection. Therefore, the total complexity of GeTLO is $O(m(n \log n + mz))$ which can be written as $O(mn \log n + m^2 z)$, z is the maximum size of **q** which in the worst case is equal to n. As typically the number of test cases is more than the number of entities, $m > n$, the complexity of Algorithm 2 can be written as $O(m^2 n)$ which is the same as AT. However, in practice, size of **q** is usually smaller than n and Algorithm 2 takes less than $O(m^2 n)$ time. Therefore, Algorithm 2 has lower computational cost in comparison to Algorithm 1 which was shown to take $O(m^2 n^2 \log n)$ time.

2.6 Equivalence of Algorithm 1 and Algorithm 2

Theorem 1 proves that Algorithms 1 and 2 are equivalent and produce identical orderings.

Theorem 1. *Algorithms 1 and 2 result in the same ordering of test cases.*

Proof. To carry on with the proof, for the sake of notation simplicity, superscript s is removed. □

The basic algorithm prioritizes test cases based on the lexicographical ranks of vectors $sort(\mathbf{occ} + \mathbf{oc}_i)$, see lines 6 to 12 of Algorithm 1, while GeTLO proceeds relying on the lexicographical ranks of the weight vectors, see lines 9 to 15 of Algorithm 2. Therefore, one way to prove that the two algorithms are equivalent, is to show that both result in the same lexicographical ordering. In other words, we aim to prove that:

$$\forall \mathbf{occ}, \mathbf{oc}_a, \mathbf{oc}_b \quad sort(\mathbf{occ} + \mathbf{oc}_a) \prec sort(\mathbf{occ} + \mathbf{oc}_b), \tag{6}$$

is equivalent to:

$$\mathbf{w}_a \prec \mathbf{w}_b. \tag{7}$$

Expressions (6) and (7) mean that test case t_b has higher priority by Algorithms 1 and 2, respectively.

In expression (6) computing $sort(\mathbf{occ} + \mathbf{oc}_a)$ and $sort(\mathbf{occ} + \mathbf{oc}_b)$ is dependent on the actual value of vector **occ**, however computing $\mathbf{w}_a$ and $\mathbf{w}_b$ in expression (7) is only dependent on the number of elements in each partition of **occ**, and not on its actual value. Consequently, to proceed with the proof, we aim to relax the role of **occ** in expression (6) in order to establish the relation between expressions (6) and (7). To this end, we show that the explicit dependency of expression (6) on the value of **occ** can be relaxed to a milder, but equivalent condition on the number of elements in each partition of **occ**.

To remove the explicit appearance of **occ** in equation (6), we define an operator, called *Sort Partition*, denoted as $sp_{\mathbf{q}}(\mathbf{x})$ (sort **x** based on the partitions specified in **q**) where each element of vector **q**, such as $q(i)$, denotes the size of the i-th partition of the vector **occ**. This operator reorders the elements of **x** such that the resulting vector is sorted partition-wise. In other words, $sp_{\mathbf{q}}(\mathbf{x})$ sorts the elements within each partition independent of others.

Example D.1. Let $\mathbf{occ} = [1, 1, 1, 2, 2, 4]$ and $\mathbf{oc}_1 = [0, 1, 0, 1, 0, 1]$ and $\mathbf{oc}_2 = [1, 0, 0, 1, 1, 0]$ then we have:

$$\mathbf{occ} = \left[\underbrace{1,\ 1,\ 1,}_{1^{\text{st}}\text{partition}} \underbrace{2,\ 2,}_{2^{\text{nd}}\text{partition}} \underbrace{4}_{3^{\text{rd}}\text{partition}} \right].$$

The corresponding partitioning applied to **oc** results in:

$$\mathbf{occ} = \left[\underbrace{0,\ 1,\ 0,}_{1^{\text{st}}\text{partition}} \underbrace{1,\ 0,}_{2^{\text{nd}}\text{partition}} \underbrace{1}_{3^{\text{rd}}\text{partition}} \right].$$

Therefore, **occ** has three partitions and we have $\mathbf{q} = [3, 2, 1]$ where $q(i)$ shows the number of elements in the i-th partition of **occ**. Thus, we have:

$$sp(\mathbf{oc})_{\mathbf{q}}$$

$$\begin{aligned} &= [sort(1^{\text{st}}\text{partition}), \quad sort(2^{\text{nd}}\text{partition}), \quad sort(3^{\text{rd}}\text{partition})] \\ &= [sort([0,1,\ 0]), \qquad sort([1,\ 0]), \qquad sort([1])] \\ &= [\quad 0, 0, 1, \qquad 0, 1, \qquad 1]. \end{aligned}$$

Following similar steps for t_2 results in $sp_{\mathbf{q}}(\mathbf{oc}_2) = [0, 0, 1, 1, 1, 0]$.

Using this operator, we show that $sort(\mathbf{occ} + \mathbf{oc}_a)$ is equal to $\mathbf{occ} + sp_{\mathbf{q}}(\mathbf{oc}_a)$. Since **occ** is a sorted vector with non-negative integer elements, the difference between two elements in adjacent partitions in **occ** is always greater than or equal to one. In other words:

$$\forall j \text{ and } \forall i \in \psi_j, \forall p \in \psi_{j+1}, \quad occ(i) + 1 \leq occ(p). \tag{8}$$

Recall that: (1) Each element of **occ**, such as $occ(i)$, shows the number of times that the i-th entity of the software is covered until the current step, and (2) ψ_i is the set of indices in **occ** that belong to its ith partition.

Considering the fact that each element of $\mathbf{oc}_a$ is either zero or one ($oc_a(t) \in \{0, 1\}$ for $t \in \{1, 2, \ldots, n\}$), we can rewrite (8):

$$\begin{aligned} &\forall j \text{ and }, \forall i \in \psi_j, \forall p \in \psi_{j+1}, \\ &occ(i) + oc_a(i) \leq occ(p) + oc_a(p). \end{aligned} \tag{9}$$

In simple words, equation (9) entails the following: The elements within a single partition of $\mathbf{occ} + \mathbf{oc}_a$ may not be sorted, however, any two elements that are chosen from different partitions of $\mathbf{occ} + \mathbf{oc}_a$ are sorted. Therefore, to sort $\mathbf{occ} + \mathbf{oc}_a$, one can instead sort each partition of it separately. That is, sorting a vector of size n is broken into r simpler sorting operations, each with the size of its corresponding partition, where r is the number of partitions.

Example D.2. In the case of Example D.1, we have:

$$\mathbf{occ} + \mathbf{oc}_1 = \left[\underbrace{1,\ 2,\ 1,}_{1^{\text{st}}\text{partition}} \underbrace{3,\ 2,}_{2^{\text{nd}}\text{partition}} \underbrace{5}_{3^{\text{rd}}\text{partition}} \right]$$

$$\mathbf{occ} + \mathbf{oc}_2 = \left[\underbrace{2,\ 1,\ 1,}_{1^{\text{st}}\text{partition}} \underbrace{3,\ 3,}_{2^{\text{nd}}\text{partition}} \underbrace{4}_{3^{\text{rd}}\text{partition}} \right].$$

We can observe that sorting vectors $\mathbf{occ} + \mathbf{oc}_1$ and $\mathbf{occ} + \mathbf{oc}_2$ can be reduced to sorting their partitions, that is:

$$\begin{aligned} sort(\mathbf{occ} + \mathbf{oc}_1) &= [sort([1,2,1]), sort([3,2]), sort([5])] \\ &= [1,1,2,2,3,5] \\ sort(\mathbf{occ} + \mathbf{oc}_2) &= [sort([2,1,1]), sort([3,3]), sort([4])] \\ &= [1,1,2,3,3,4]. \end{aligned} \tag{10}$$

Recall that the partitioning of **occ** is based on grouping elements with the same value under the same partition. By considering the fact that entries of **occ** within a partition are all equal, and consequently, do not affect the outcome of the sorting operation, we can write:

$$sort(\mathbf{occ} + \mathbf{oc}_a) = sp_{\mathbf{q}}(\mathbf{occ} + \mathbf{oc}_a) = \mathbf{occ} + sp_{\mathbf{q}}(\mathbf{oc}_a). \tag{11}$$

Example D.3. In the case of Example D.1, we can see that:

$$\begin{aligned} \mathbf{occ} + sp_{\mathbf{q}}(\mathbf{oc}_1) &= [1,1,1,2,2,4] + [0,0,1,0,1,1] \\ &= [1,1,2,2,3,5] \\ \mathbf{occ} + sp_{\mathbf{q}}(\mathbf{oc}_2) &= [1,1,1,2,2,4] + [0,0,1,1,1,0] \\ &= [1,1,2,3,3,4] \end{aligned}$$

which are equal to the values calculated in equation (10).

Now we can rewrite the expression $sort(\mathbf{occ} + \mathbf{oc}_a) \prec sort(\mathbf{occ} + \mathbf{oc}_b)$ using expression (11) as:

$$\mathbf{occ} + sp_{\mathbf{q}}(\mathbf{oc}_a) \prec \mathbf{occ} + sp_{\mathbf{q}}(\mathbf{oc}_b). \tag{12}$$

Adding or subtracting the same value from the two sides does not affect the lexicographical ordering. Thus, we can reduce 12 to:

$$sp_{\mathbf{q}}(\mathbf{oc}_a) \prec sp_{\mathbf{q}}(\mathbf{oc}_b). \tag{13}$$

In other words, the condition in expression (6) is equivalent to the condition in expression (13). As a result, it suffices to prove that:

$$sp_{\mathbf{q}}(\mathbf{oc}_a) \prec sp_{\mathbf{q}}(\mathbf{oc}_b) \Leftrightarrow \mathbf{w}_a \prec \mathbf{w}_b. \tag{14}$$

First, we prove that $sp_{\mathbf{q}}(\mathbf{oc}_a) \prec sp_{\mathbf{q}}(\mathbf{oc}_b) \Rightarrow \mathbf{w}_a \prec \mathbf{w}_b$. To this aim, let $\mathbf{s}_a$ denote the vector returned by applying the sp operator to $\mathbf{oc}_a$, i.e., $\mathbf{s}_a = sp_{\mathbf{q}}(\mathbf{oc}_a)$. Using lexicographical ordering definition, we can infer that if $sp_{\mathbf{q}}(\mathbf{oc}_a) \prec sp_{\mathbf{q}}(\mathbf{oc}_b)$, then:

$$\exists i \quad \text{s.t.} \quad \forall j < i, \quad \mathbf{s}_a(j) = \mathbf{s}_b(j) \text{ and } \mathbf{s}_a(i) < \mathbf{s}_b(i). \tag{15}$$

Considering the fact that $\mathbf{oc}_a$ and $\mathbf{oc}_b$ in expression (15) are binary vectors, the only possible values are $\mathbf{s}_a(i) = 0$ and $\mathbf{s}_b(i) = 1$. Without loss of generality, assume that index i in expression (15) is in the kth partition, namely $i \in \psi_k$. Since both $\mathbf{s}_a$ and $\mathbf{s}_b$ are sorted partition-wise, we have:

$$\begin{aligned} &\forall z \in \psi_k \text{ and } z \geq i, \quad \mathbf{s}_b(z) = 1 \\ &\forall z \in \psi_k \text{ and } z \leq i, \quad \mathbf{s}_a(z) = 0. \end{aligned} \tag{16}$$

This means for the kth partition, we have:

$$\begin{aligned} &k\text{th partition of } \mathbf{oc}_a = [0,0,\ldots,0, \quad 0, \quad \text{✘},\ldots,\text{✘}] \\ &k\text{th partition of } \mathbf{oc}_b = [0,0,\ldots,0, \quad \underbrace{1,}_{i\text{th element}} \quad 1,\ldots,1] \end{aligned}$$

where ✘ is either 0 or 1.

Example D.4. For the case of Example D.1, we have:

$$\mathbf{s}_1 = [\underbrace{0,\ 0,\ 1,}_{1^{\text{st}}\text{partition}} \underbrace{0,\ 1,}_{2^{\text{nd}}\text{partition}} \underbrace{1}_{3^{\text{rd}}\text{partition}}].$$

$$\mathbf{s}_2 = [\underbrace{0,\ 0,\ 1,}_{1^{\text{st}}\text{partition}} \underbrace{1,\ 1,}_{2^{\text{nd}}\text{partition}} \underbrace{0}_{3^{\text{rd}}\text{partition}}].$$

We can see that $\mathbf{s}_1$ and $\mathbf{s}_2$ differ in the second partition; thus, expression (16) is satisfied for this example with $k = 2$ and $i = 4$.

This means, for the index i satisfying expression (15), all the elements of $\mathbf{s}_b$ with indices higher than i that are in the kth partition are equal to 1. At the same time, $\mathbf{s}_b(i) = 1$ and $\mathbf{s}_a(i) = 0$. Consequently, regardless of the values (0 or 1) taken by the elemets of $\mathbf{s}_a$ for $z > i$, we can conclude that the sum of values in the kth partition of $\mathbf{oc}_a$ (k is the partition of $\mathbf{oc}_a$ that includes index i), is less than the corresponding value in $\mathbf{oc}_b$. Thus, we have $w_a(k) < w_b(k)$. Finally, noting that we have: $\forall i < k$, $w_a(i) = w_b(i)$, by the definition of lexicographical ordering, we can conclude $\mathbf{w}_a \prec \mathbf{w}_b$.

Next, we prove that $\mathbf{w}_a \prec \mathbf{w}_b \Rightarrow \mathbf{s}_a \prec \mathbf{s}_b$. The assumption is that $\mathbf{w}_a \prec \mathbf{w}_b$, therefore, according to the definition of lexicographical ordering, we have:

$$\exists i \quad \text{s.t.} \quad \forall j < i, \quad \mathbf{w}_a(j) = \mathbf{w}_b(j) \text{ and } \mathbf{w}_a(i) < \mathbf{w}_b(i). \tag{17}$$

The inequality $\mathbf{w}_a(i) < \mathbf{w}_b(i)$ in (17) means that the number of '1's in the ith partition of $\mathbf{oc}_b$ is larger than that of $\mathbf{oc}_a$. Consider the ith partition of $\mathbf{s}_a$ and $\mathbf{s}_b$. Since both partitions are sorted, we can write:

$$\begin{aligned} \exists z \quad \text{s.t.} \quad \forall t < z, \quad &\mathbf{s}_a(t) = \mathbf{s}_b(t) = 0 \text{ and} \\ &\mathbf{s}_a(z) = 0 \text{ and } \mathbf{s}_b(z) = 1. \end{aligned} \tag{18}$$

Therefore, we conclude that $\mathbf{w}_a \prec \mathbf{w}_b \Rightarrow \mathbf{s}_a \prec \mathbf{s}_b$.

3 Enhanced GeTLO Using Recursive Tie Breaking

Since the ties that occur in the proposed technique are different from those that occur in AT, we refer to the former by G-ties. Using Algorithm 2, a G-tie occurs when more than

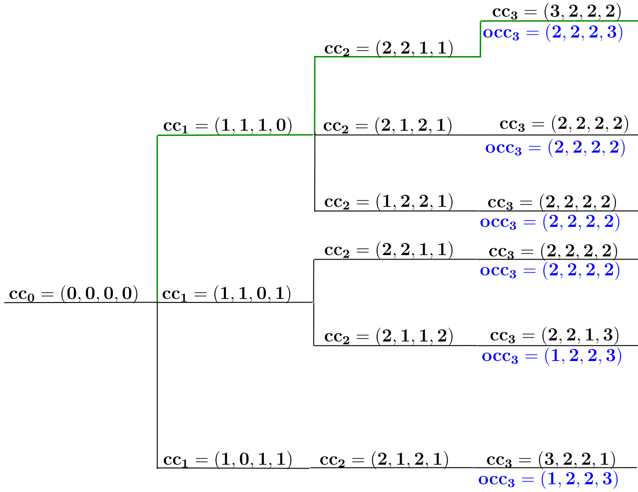

Fig. 1. The process of choosing a test case.

one $\mathbf{w}_i$ vector achieves the highest lexicographical rank. This problem can arise even if all coverage vectors are different. G-ties tend to happen more frequently in the first few steps where there are more not-yet-covered entities. Random selection is one choice, but there might be a selection which can further improve the rate of fault detection. The next example shows that different selections in the case of a tie can result in different **occ** vectors.

Example E. Consider the case where there are four test cases with the following coverage matrix:

$$\mathbf{C} = \begin{pmatrix} 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}.$$

At the first step, there are two candidate test cases to select, namely 1 and 2. If test case 1 is selected, test cases 4 and 3 will be selected in the next two steps resulting in $\mathbf{occ}^3$ equal to $[1,1,1,1,1,1,1,1,1,1,1,2]$. Nevertheless, if test case 2 is chosen at the first step, test cases 3 and 4 will be added to the ordering in subsequent steps which results in the $\mathbf{occ}^3$ equal to $[0,1,1,1,1,1,1,1,1,1,2,2]$. The lexicographical rank of the former $\mathbf{occ}^3$ vector is higher than the latter one, because it covers all the entities at least once. Ideally, we are interested in **occ** vectors with higher ranks.

To remove G-ties, instead of choosing one of the candidates randomly, one could let the algorithm to proceed further for each choice up to a desired depth, d. Assuming a tie is faced at step s, the algorithm proceeds further by considering each candidate test case involved in the tie as if it were the selected test case at this step. At this point, the algorithm continues d steps ahead to step $s+d$ for all candidates. Then, the search algorithm retrieves a test case (among those which have resulted in G-tie at step s) such that the corresponding $\mathbf{occ}^{s+d}$ has the highest lexicographical rank.

This process can be represented as a tree, capturing the correspondence between ties and branches at each node. This concept is explained in the following example.

Example F. Fig. 1 shows the evolution of the aforementioned tree for a system composed of four entities. In this example, an early G-tie occurs at the first step where there are three test cases with equal number of covered entities resulting in the same lexicographical rank. Assume $d = 3$. This means that the algorithm proceeds until the third level of the tree for all candidates and then chooses the one with the highest lexicographical rank. In this example, the **occ** vector with the highest lexicographical rank is $\mathbf{occ} = [2,2,2,3]$, which corresponds to the test case with coverage vector $\mathbf{cv}_1 = [1,1,1,0]$. The algorithm, then, discards all other choices and proceeds in the same manner for the next steps .

It is evident that increasing d results in choosing more promising candidates but at the price of a higher complexity.

Note that Algorithm 2 corresponds to the enhanced algorithm described above for $d = 0$.

4 Empirical Studies

4.1 Research Questions

Following research questions are addressed:

- *RQ1:* What are the benefits of using the proposed algorithm in increasing the rate of fault detection?
- *RQ2:* What are the effects of the depth parameter, d, in the performance of the proposed algorithm?
- *RQ3:* What anomalies can affect the performance of the coverage based techniques?
- *RQ4:* What is the impact of the granularity level on the performance of the proposed technique?
- *RQ5:* How fast can the proposed technique find the hard-to-detect faults?

4.2 Subject Programs & Experiment Setup

Experiments are performed using 45 versions of six different open source Java programs: *Ant, Galileo, Jmeter, Jtopas, NanoXML (Nano), XML-Security (XML)*. The source code of all subject programs and their test cases are downloaded from the Software-artifact Infrastructure Repository [32] which has been extensively used in earlier studies on test case prioritization and selection techniques. This facilitates the comparison of the results with other techniques from the literature. Each program in the SIR contains the folder "version.alt" with subdirectories *V0, V1,. . .,Vk*, where subdirectory *Vj* contains the jth version of the program. This folder also includes the file "VersionMap" that contains the mapping between SIR versions and the original versions of the program.

The details of subject programs are listed in Table 2. It includes the number of versions, the number of source code lines of the programs (including the source code of test cases and the non-executable lines), the number of test cases (all test cases are at the test-method level), the number of faults (all values for the most recent version), as well as the type of test cases which are either JUnit or Test Specification Language (TSL). The size of subject programs range from 5.4 to 80.4 KLoC.

The original versions of these programs do not include any faults. For each program on the SIR, a limited number of faults (hand seeded) is provided. However, to perform an adequate statistical comparison between different

TABLE 2
Details of Subject Programs Used in Empirical Studies

	Details				
	#Vers	KLoC	#TMeth	#Faults	Type
Ant	9	80.4	877	412	JUnit
Galileo	16	15.2	912	2494	TSL
Jmeter	6	43.3	78	386	JUnit
Jtopas	4	5.4	128	1104	JUnit
Nano	6	7.6	216	204	TSL
XML	4	16.3	83	246	JUnit

techniques, the number of faults must be sufficiently large. To increase the number of faults, we have used mutation faults [37]. There have been studies about whether mutation faults can be a representative of real faults. These studies indicate that it is usually appropriate to use mutation faults for studying regression testing techniques [38], [39]. In the context of test case prioritization, Do and Rothermel [40] have empirically investigated the effectiveness of using mutation faults. For five programs (all except *Jtopas*), we have used the same mutation faults as used in [10]. For *Jtopas* (which was not used in [10]), we have used *MuJava* [41] to generate the mutation faults. In particular, to generate the faults for a version of a given program, first the set of all methods that differ from the previous version are extracted. Second, a relatively large number of mutant faults are injected into the modified methods. Finally, all injected mutant faults that cannot be detected at least by one test case are disregarded and different techniques are executed using the remaining ones. A similar procedure has been used for comparison in other relevant studies [10], [40].

The coverage information for the five of the subject programs (all except for *Jtopas*) are obtained from [10] where they use the *Sofya* instrumentation tool [42] to obtain coverage information. For *Jtopas*, we have used the *Fault Tracer* tool [43] to obtain the coverage information. The granularity of coverage information is at the method level and the basic block level. Following the procedure presented in [44], if at least one entity of a method/basic block is covered by a test case, we consider that method/basic block covered, and substitute '1' in its corresponding entry in the coverage matrix, otherwise, the corresponding entry will be '0'.

4.3 Techniques Used for Comparison

For comparison purposes, we have selected a set of coverage-based test case prioritization techniques, described as follows:

- As the proposed algorithm is built on top of TT and AT, we have implemented both of these techniques for the sake of comparison.
- To measure the effect of the depth parameter, d, we have used $d = 0$ and $d = 6$. The maximum depth is chosen based on trial and error. It was observed that in almost all occurrences of a G-tie, using $d = 6$ removes all the ties.
- There are many other techniques which not only use the coverage data, but also rely on other relevant data such as the *change information*. To perform a comparison between the proposed technique and some of such multi-criteria approaches, we include two *Bayesian Network* (BN) based techniques: BN and BNA [45]. The relevant implementation and parameters that have been used in this work are identical to those reported in [45].

We implemented TT, AT and GeTLO in MATLAB and used the implementation of BN and BNA in Java provided by the authors in [46]. An implementation of GeTLO technique is available at `https://github.com/sepehr3pehr/GeTLO`.

4.4 Metric Used for Comparison

To measure the effectiveness of different test case prioritization techniques, we use the widely-accepted *Average Percentage Fault Detection* (APFD) metric [18]. Intuitively, APFD measures how fast faults are detected by the generated ordering. Higher values of APFD indicate faster fault detection. Formally, let T be the set of n test cases and TF_i be the index of the first test case that detects the ith fault. Also, assume that the number of faults that can be detected by the test suite is m, then the value of APFD metric of an ordering is given by following equation [18]:

$$\text{APFD} = 1 - \frac{TF_1 + \cdots + TF_m}{nm} + \frac{1}{2n}. \tag{19}$$

which measures the area under the curve by plotting the percentage of faults detected on the vertical axis and number of test cases on the horizontal axis. The APFD metric ranges from 0 to 1, where the value of 1 indicates that all faults are detected by the first test case.

The APFD metric has some limitations, for example, it does not consider the execution cost and faults severities [47]. Although there are studies which consider cost-benefit models for APFD, we do not incorporate them in this paper.

In practical testing scenarios, a program do not include faults in numbers as high as the size of mutant faults that are available from [10]. Also, to have an adequate statistical comparison, we need to apply different techniques to all versions of subject programs for a sufficiently large number of times. To satisfy these two conditions, in each run (on a specific version), we choose a random number of mutants from the pool. This number should not be too small (such that the effectiveness of different techniques can be measured); it should not be too large either (for different random selection of subsets of faults to be independent of each other). The number of mutation faults in each run is set by randomly picking an integer value in the range of 5 to 15. The same procedure has been applied in [10], [40].

4.5 Results

To provide an overview of the collected data, Figs. 2 and 3 show the box-plots of the APFD metric for different techniques at the method level and at the basic block level, respectively. On each box, the central mark is the median, the edges of the box are the 25th and 75th percentiles, the whiskers (lines extending vertically from the boxes) extend to the most extreme data points that are not considered to be outliers. Figs. 2 and 3 show the results of 30 executions of each technique, each with a different subset of faults. The

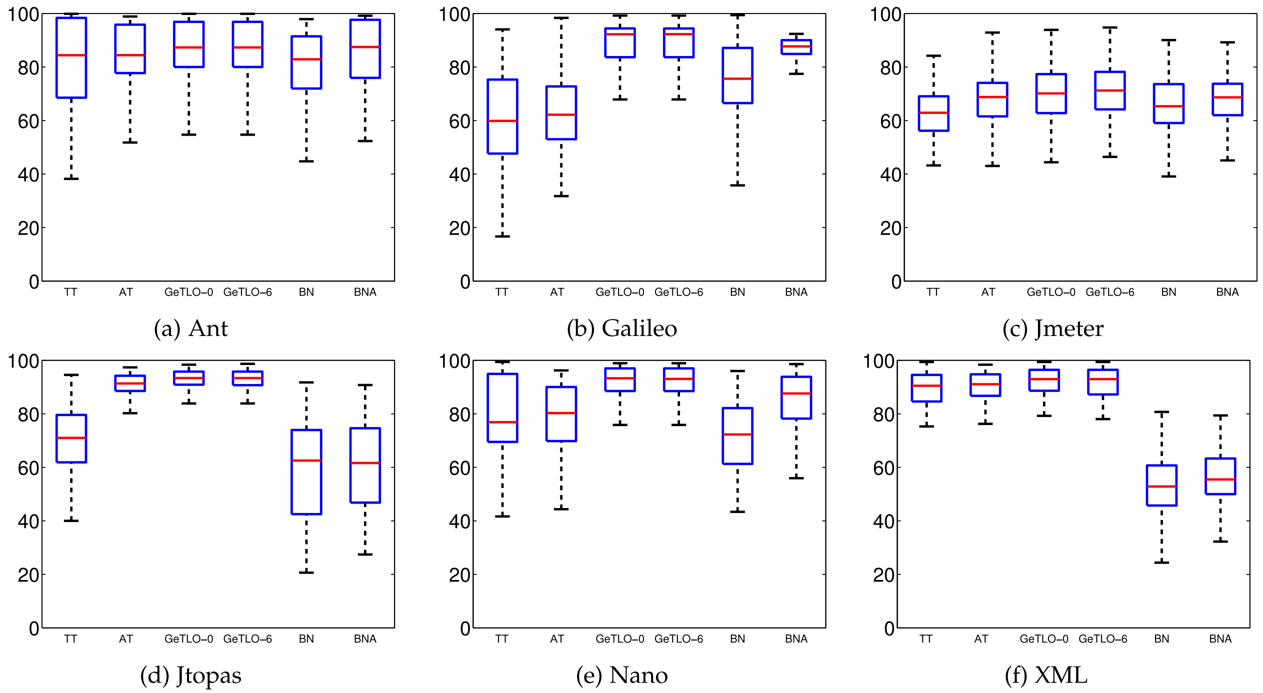

Fig. 2. Box-plots of different methods for all subject programs at the method level.

integer value following the GeTLO shows the value of the depth parameter, e.g., GeTLO-6 is GeTLO technique with $d = 6$. Each plot contains a box for each of the six techniques, representing the distribution of the APFD metric.

To validate the observations, different techniques are compared using the Kruskal-Wallis one-way analysis of variance [48]. We did not use the ANOVA test [49] because our datasets do not have equal variances and do not follow a normal distribution. For multiple comparisons, the Bonferroni method is used due to its generality and conservatism. The null hypothesis is that the mean values of the APFD metric of two techniques are the same. The level of significance for acceptance or rejection of the null hypothesis is set to 0.05. A similar value is used in [10], [17], [18], [40], [50]. Tables 3 and 4 show the results of such statistical comparisons. Different techniques are sorted based on the mean of their APFD metric over different versions. Techniques which are not significantly different share the same rank. Similar procedure for comparison is applied in [10], [18], [40].

It is worth mentioning that these programs were selected based on their popularity in relevant studies without any

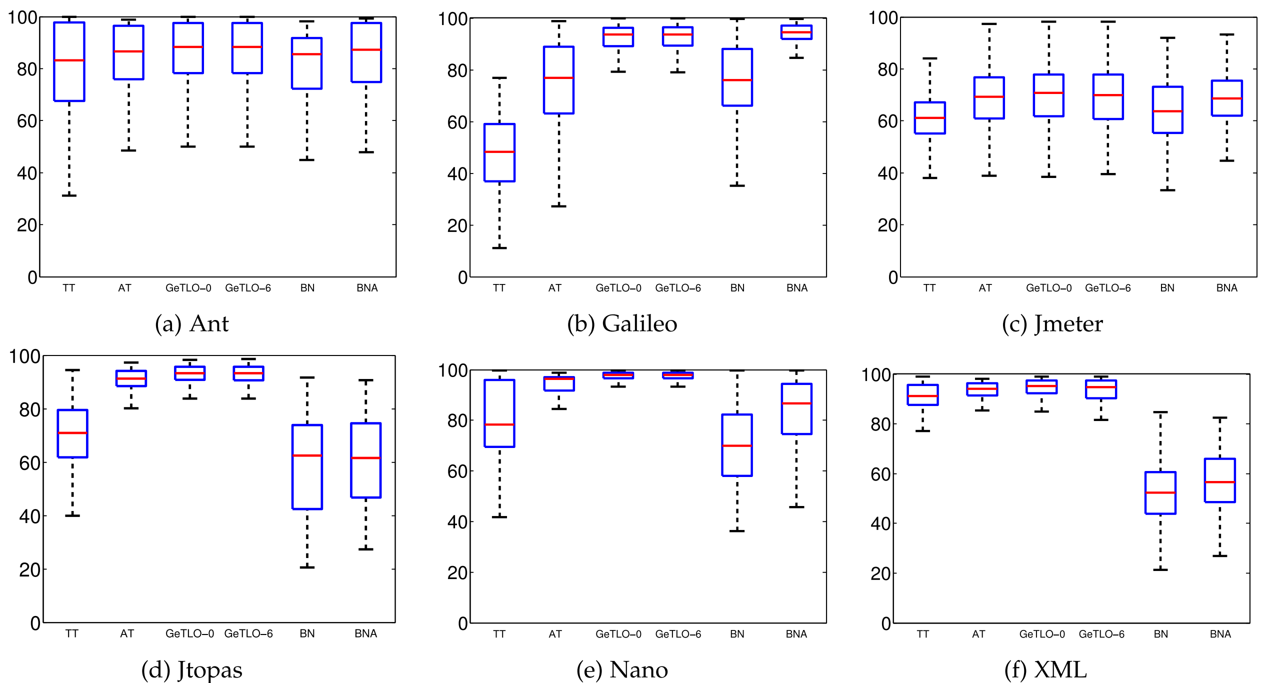

Fig. 3. Box-plots of different techniques for all subject programs at the basic block level.

TABLE 3
Kruskalwallis Test Results at the Method Level

Ant

Tech	Mean	Rank
GeTLO-0	84.69	A
GeTLO-6	84.69	A
AT	84.23	A
BNA	83.31	A
BN	81.81	B
TT	79.20	C

Galileo

Tech	Mean	Rank
BNA	86.87	A
GeTLO-6	86.47	A
GeTLO-0	86.44	A
BN	76.38	B
AT	62.40	C
TT	60.06	C

Jmeter

Tech	Mean	Rank	
GeTLO-6	70.29	A	E
GeTLO-0	69.30	B	E
AT	68.77	B	
BNA	68.07	B	
BN	63.37	C	
TT	60.10	D	

Jtopas

Tech	Mean	Rank
GeTLO-6	93.23	A
GeTLO-0	93.19	A
AT	91.46	B
TT	72.96	C
BNA	63.69	D
BN	61.31	D

Nano

Tech	Mean	Rank
GeTLO-6	92.45	A
GeTLO-0	92.40	A
BNA	84.59	B
AT	80.68	C
TT	80.02	C
BN	71.11	D

XML

Tech	Mean	Rank
GeTLO-6	92.58	A
GeTLO-0	92.31	A
AT	91.60	A
TT	89.61	B
BNA	57.03	C
BN	51.72	D

attention to the corresponding number of ties (number of ties in each of them were not known beforehand). We would expect that the difference between GeTLO and AT decreases as the number and the size of ties decline.

4.6 Discussions on the Results

4.6.1 RQ1: Rate of Fault Detection

Box-plots in Figs. 2 and 3 as well as relative rankings in Tables 3 and 4 indicate that using the notion of lexicographical ordering for test case prioritization can boost the rate of fault detection. Following patterns are observed in the results:

- In all programs, GeTLO-6 and GeTLO-0 are ranked as the highest, although they may share this rank with other alternatives.
- On *Ant*, almost all techniques exhibit similar performance. The reason for such an anomaly will be discussed later.
- On *Jtopas* and *Nano*, all techniques are outperformed by the GeTLO. Note that, according to Table 1, these two projects have the highest percentage of steps with ties indicating that GeTLO is successful at resolving ties to achieve higher rates of fault detection.
- On *Galileo*, the first rank is shared among GeTLO-0, GeTLO-6 and BNA.

These observations clearly demonstrate the benefit of the proposed technique over the existing alternatives.

Fig. 4 shows the average execution time of all techniques over different projects at the method and the basic block levels (the execution time of TT is extremely small and not visible in the figure). The figure indicates that the execution time of GeTLO is higher than that of AT, but it does not exceed 25 seconds even for large projects.

4.6.2 RQ2: Effects of the Depth Parameter

In terms of comparing GeTLO-0 and GeTLO-6, the results show that the difference is not statistically significant in any of the subject programs (GeTLO-0 and GeTLO-6 share the same statistical rank). To further investigate any differences, we have computed the number of G-tie occurrences in each subject program. GeTLO-0 and

TABLE 4
Kruskalwallis Test Results at the Basic Block Level

Ant

Tech	Mean	Rank
GeTLO-0	86.80	A
GeTLO-6	86.80	A
AT	85.31	A
BNA	83.40	A
BN	81.92	B
TT	79.26	B

Galileo

Tech	Mean	Rank
BNA	94.08	A
GeTLO-6	92.24	A
GeTLO-0	92.03	A
BN	76.87	B
AT	75.54	C
TT	46.52	D

Jmeter

Tech	Mean	Rank
GeTLO-6	69.87	A
GeTLO-0	69.01	A
AT	68.80	A
BNA	68.57	A
BN	63.58	B
TT	60.56	C

Jtopas

Tech	Mean	Rank
GeTLO-6	93.18	A
GeTLO-0	93.00	A
AT	91.25	B
TT	72.42	C
BNA	63.99	D
BN	61.91	D

Nano

Tech	Mean	Rank
GeTLO-6	96.77	A
GeTLO-0	96.77	A
AT	94.55	B
BNA	84.02	C
TT	80.21	C
BN	70.51	D

XML

Tech	Mean	Rank	
GeTLO-6	94.08	A	
GeTLO-0	93.78	A	F
AT	92.76	B	F
TT	89.68	C	
BNA	56.58	D	
BN	50.48	E	

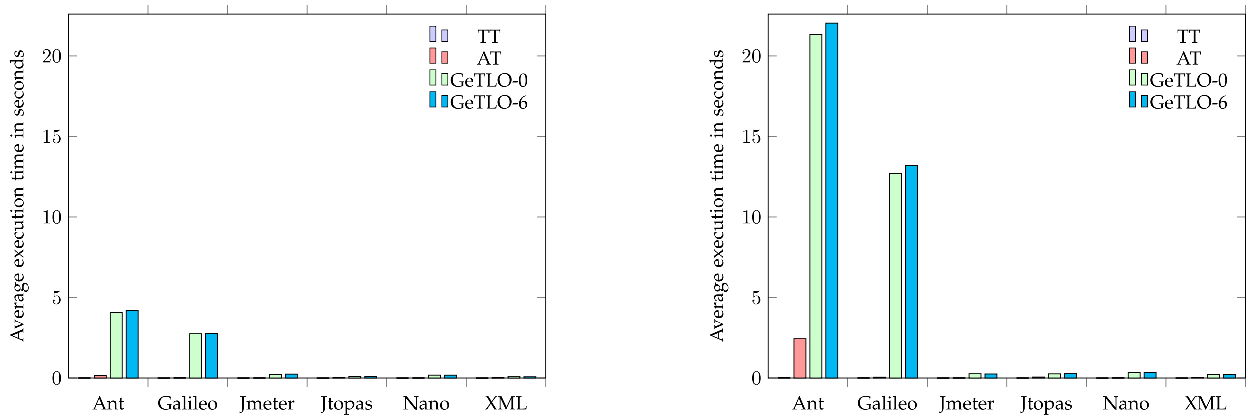

Fig. 4. The average execution time of different techniques at the method (left) and the basic block (right) levels.

GeTLO-6 return the same order if no tie occurs, therefore the only factor that can differentiate these two techniques is the number of ties. Table 5 shows the average number of ties for each program (averaged over its versions). In *Ant*, *Galileo*, *Jmeter* and *Nano*, the probability of a G-tie is relatively low. This causes GeTLO-0 and GeTLO-6 to exhibit similar rates of fault detection.

Whenever a tie occurs, GeTLO-0 randomly chooses a test case among those with the same lexicographical rank. Therefore, if the number of alternatives increases, the likelihood that GeTLO-0 and GeTLO-6 select the same test case decreases. For each program, Table 5 also contains the averaged tie size where the tie size is the number of candidate test cases involved in a tie. It turns out that this number is relatively low in all programs. In other words, choosing a random candidate may not significantly affect the performance. This fact indicates that although the number of tie occurrences in programs *Jtopas* and *XML* is relatively high, in these cases random selection can be efficient since the number of choices is small. In spite of the fact that in all programs GeTLO-0 and GeTLO-6 are not significantly different, in five out six programs mean of APFD metric for GeTLO-6 is higher than that of GeTLO-0 at both the method level and the basic block level.

Note that by comparing Tables 1 and 5, we can infer that the proposed algorithm is very effective in reducing the number of ties.

4.6.3 RQ3: Anomalies

Tables 3 and 4 indicate that for some programs, such as *Ant*, there is no significant difference between most of the compared techniques. To inspect why such an anomaly occurs, the fault information of *Ant* is studied in further details. It turns out that in different versions of *Ant*, a relatively large number of faults are difficult to detect. In other words, these faults can be detected by only a very small fraction of test cases.

To study this issue further, for each version of *Ant*, we partition the faults into two sets: 1) the set $\mathcal{H}$ that includes *Hard-to-Detect* faults, and 2) the set $\mathcal{E}$ that includes *Easy-to-Detect* faults. $\mathcal{H}$ is the set of faults that are detectable by a small number of test cases (less than 4 test cases). $\mathcal{E}$ is the set of faults that are detectable by a large number of test cases (more than 4 test cases). With some misuse of notation, assume that test cases that detect the faults in $\mathcal{H}$ are indexed from 1 to j.

In what follows we further elaborate on the characteristics of test cases and faults that are the source of observed anomaly. For versions 4 and 8 of *Ant*, the relation between faults in $\mathcal{H}$ and test cases are shown in Fig. 5, where test cases and faults are represented by circles and rectangles, respectively. The circles inside a rectangle represent the faults that are detectable by the corresponding test case. The individual faults in $\mathcal{E}$ are shown as well. As the number of test cases that detect faults in $\mathcal{E}$ is relatively high, the test cases that detect them are not shown. In *Ant* V4, which contains 215 test cases, 13 out of 21 faults belong to $\mathcal{H}$ (are hard to detect), four of which, namely h_5, h_6, h_7, h_8, are detectable by only one test case, namely t_2. Also, three faults in $\mathcal{H}$, namely h_9, h_{10}, h_{11}, are only detectable by t_6. In fact, three test cases (out of 215), namely t_1, t_2, t_3, detect more than 50 percent of the faults. As another example, in *Ant* V8 with 878 test cases, 9 out of 17 faults belong to $\mathcal{H}$. In summary, if the small fraction of test cases that are able to detect the faults in $\mathcal{H}$ are not selected in earlier stages, then the detection of a large portion of faults will be significantly postponed. Another important fact is that, although such test cases significantly affect the performance of any given algorithm, they do not have comparatively large coverage values. Therefore, two major issues are identified in *Ant*:

- A small fraction of test cases detect a large fraction of faults.
- A large fraction of faults are only detectable by a small fraction of test cases, which on the other hand, have relatively small coverage values.

TABLE 5
Average Percentage (Averaging Is Over Different Versions of the Programs) and Number of Candidates for GeTLO-0

Program	Method level		Basic block level	
	Ave.% steps with G-ties	Ave. # candidates	Ave.% steps with G-ties	Ave. # candidates
Ant	6%	2.17	1%	2.17
Galileo	3%	2.35	4%	2.29
Jtopas	25%	2.90	10%	2.10
Jmeter	14%	2.68	10%	2.27
Nano	7%	2.19	1%	2.00
XML	28%	3.62	22%	3.67

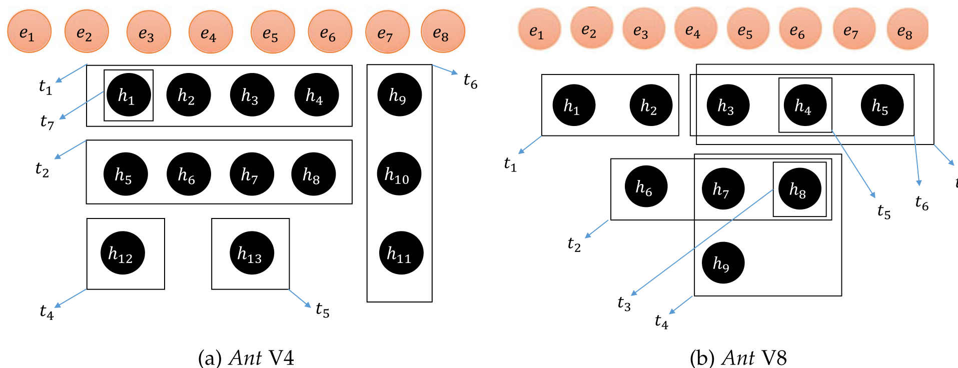

Fig. 5. Relationships between hard-to-detect faults and test cases. Total number of faults are 21 and 17 for *Ant* V4 and *Ant* V8, respectively.

This behaviour is due to the fact that the correlation between coverage and fault is not always high. Therefore, there is always a chance that a test case with a lower coverage detects more faults. In the case of *Ant*, this problem is intensified since some test cases with small coverage values detect a large fraction of faults that other test cases cannot detect. In such cases, the outcome of different coverage-based techniques will have a random nature, and consequently, conclusive comparisons cannot be performed.

4.6.4 RQ4: Impact of the Granularity Level

The box-plots and statistical comparison both indicate that the impact of the granularity level on the rate of fault detection is insignificant. It turns out that the proposed technique is more beneficial when the number of ties and size of the candidate sets are relatively large. However, according to Table 1, increasing the level of granularity from method to basic block reduces the number and the size of ties only marginally. The only exception is *Jmeter* where the number of ties and size of the candidate sets are significantly reduced at the basic block level. As a consequence, the rate of fault detection of GeTLO is similar to AT for *Jmeter* at the basic block level. This conclusion, that the effect of granularity on the rate of fault detection is insignificant, is consistent with previous empirical studies [2], [40].

4.6.5 RQ5: Hard to Detect Faults

It is known that faults do not behave the same in terms of the effort required to detect them; some faults are easier to detect than others. Previous studies indicate that mutant faults are neither easier nor harder to detect than real faults [38], [40], therefore they are appropriate representative for real faults. While easy-to-detect faults can usually be detected by randomly generated test cases, others require more effort to be detected. Here, we investigate the performance of the proposed technique for hard-to-detect faults. The results in Section 4.5 indicate that GeTLO can outperform AT for majority of the subject programs. Here, we are interested to know whether GeTLO also exhibits better performance for the set of hard-to-detect faults. To measure the hardness of detecting a fault, each fault is associated with a *Hardness Index* (HI); the number of test cases that can detect the fault over the total number of test cases. Therefore, a mutation fault with HI=0.01 is considered harder to detect than a mutation fault with HI=0.90.

Figs. 6 and 7 show the average APFD measure for different versions of the subject programs in terms of HI values. To generate the plots, the interval between 0 and 1 (possible values of HI) is uniformly discretized to 30 values. For each version of the subject program, we first calculate the HI value corresponding to each of the mutants. Then, for each value of HI (the horizontal axis) say x, we select all the faults with HI less than x, thus we form 30 sets of faults (not necessarily different) for each version. Finally, we run all techniques with the selected faults and the APFD, averaged over different venison, is reported . Unlike the experiments in Section 4.2, we do not sample the faults here. The reason is that the total number of faults with low HI values is often relatively small. This can make the randomly selected sets of faults dependent on each other. Also, Fig. 8 shows the cumulative percentage of faults in terms of the HI value. While in some programs, such as *Nano*, faults are distributed uniformly, in some others, such as *Jtopas*, almost all faults have small HI values.

Figs. 6 and 7 indicate that the relative performance of different techniques is more or less preserved for different values of HI. In all programs, except *Galileo*, GeTLO exhibits a better performance. More importantly, GeTLO preserves its relative improvement for different values of HI.

In *Ant*, all techniques exhibit similar rates of fault detection due to the phenomena discussed in Section 4.6.3. *Nano* is the only program in which the hardness index does effect the relative performance of some of the techniques. The main reason is that, in *Nano*, the faults are almost uniformly distributed among different HI values (see Fig. 8). This causes the sets of faults corresponding to the different HI values to be different from each other. Recall that the range of HI values is quantized to 30 uniform intervals. As a result the set of faults are grouped into 30 sets each associated with different quantized HI value. The 30 sets of faults do overlap with each other. For *Nano*, we observe that almost all 30 sets of faults for each version are different from each other. However, this is not the case for other subject programs. As an example, in *Jtopas*, all faults have HI values smaller than 0.1, implying that faults are not uniformly distributed.

4.7 Threats to Validity

The major sources of threats to external validity are the subject programs and their faults. We used Java programs in our experiments. Therefore, it is difficult to generalize the

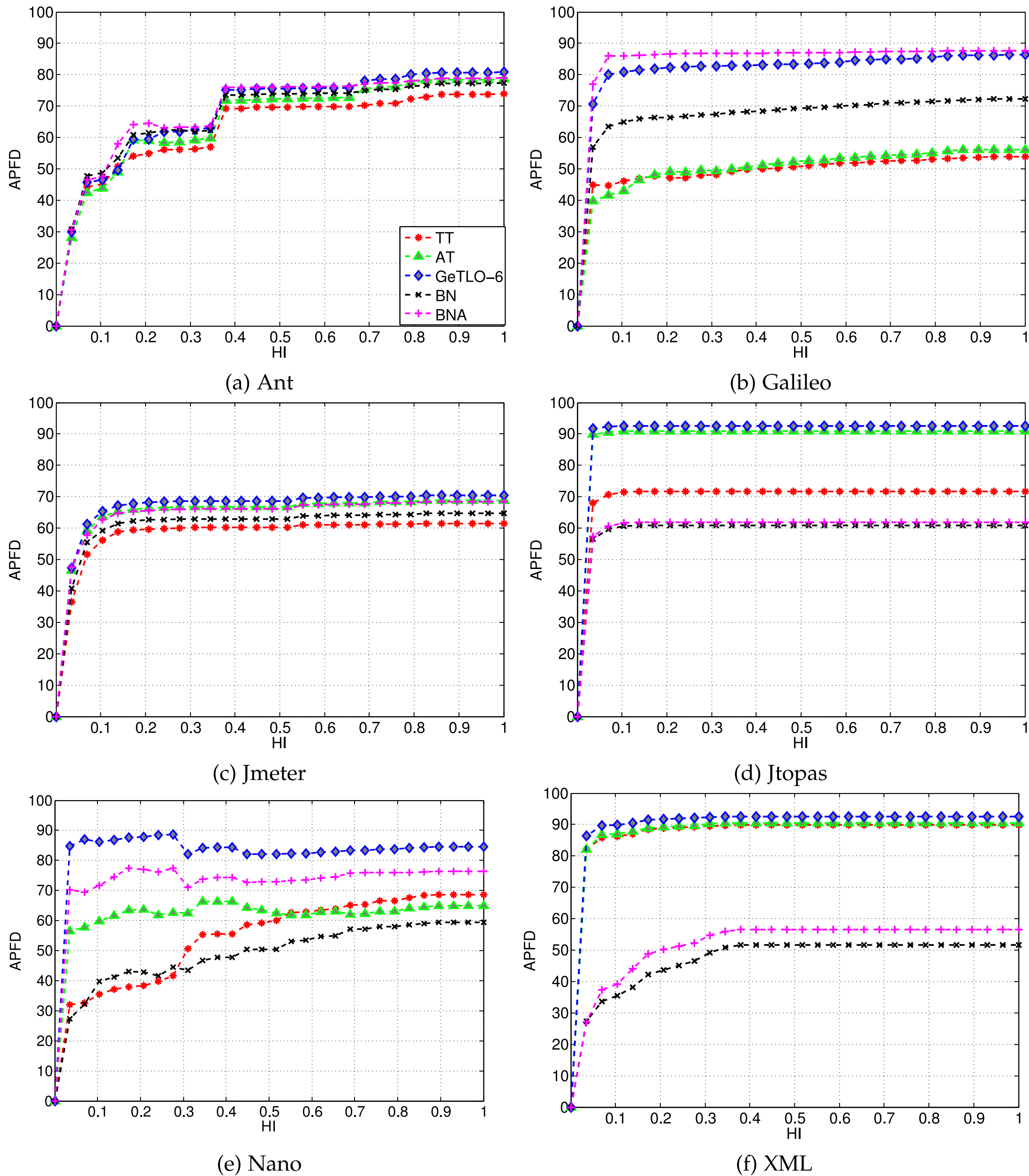

Fig. 6. APFD in terms of HI for six subject programs at the method level.

results for programs in other languages. In the experiments, the number of mutant faults were simply chosen with a uniform distribution. However, faults may follow various distributions in different programs. Moreover, the results are based on artificially injected faults, which may not be the accurate representative of real faults. Additional studies, including different fault types and distributions of faults are required to minimize these threats. Furthermore, the number of ties in almost all programs is relatively high. While all programs were chosen due to their frequent use in other studies, additional studies with fewer ties can reduce this threat.

In terms of the internal validity, choice of the depth parameter d for tie breaking can affect the results. In this article, the selection of this parameter has been based on is based on try and error. Further investigations can study the effect of the depth parameter. Also, we have used the APFD metric to gauge the effectiveness of different techniques. While APFD has been widely used for comparison of RTP techniques, it is known to have limitations [40]. For example, it does not include the cost of running each test case. Using other comparison metrics may reduce this threat.

5 Related Work

Various RTP techniques have been proposed and studied empirically, among them coverage-based techniques are more prevalent [51], [52]. TT and AT are the most widely applied coverage-based prioritization techniques [21]. Rothermel et al. [18] initially proposed them as part of a family of RTP techniques. Li et al. [17] have applied a set of well-known meta-heuristic algorithms including hill climbing, genetic algorithm and two-optimal greedy algorithms (refer to [17] for definition) for test case prioritization, where the results are compared with TT and AT using different

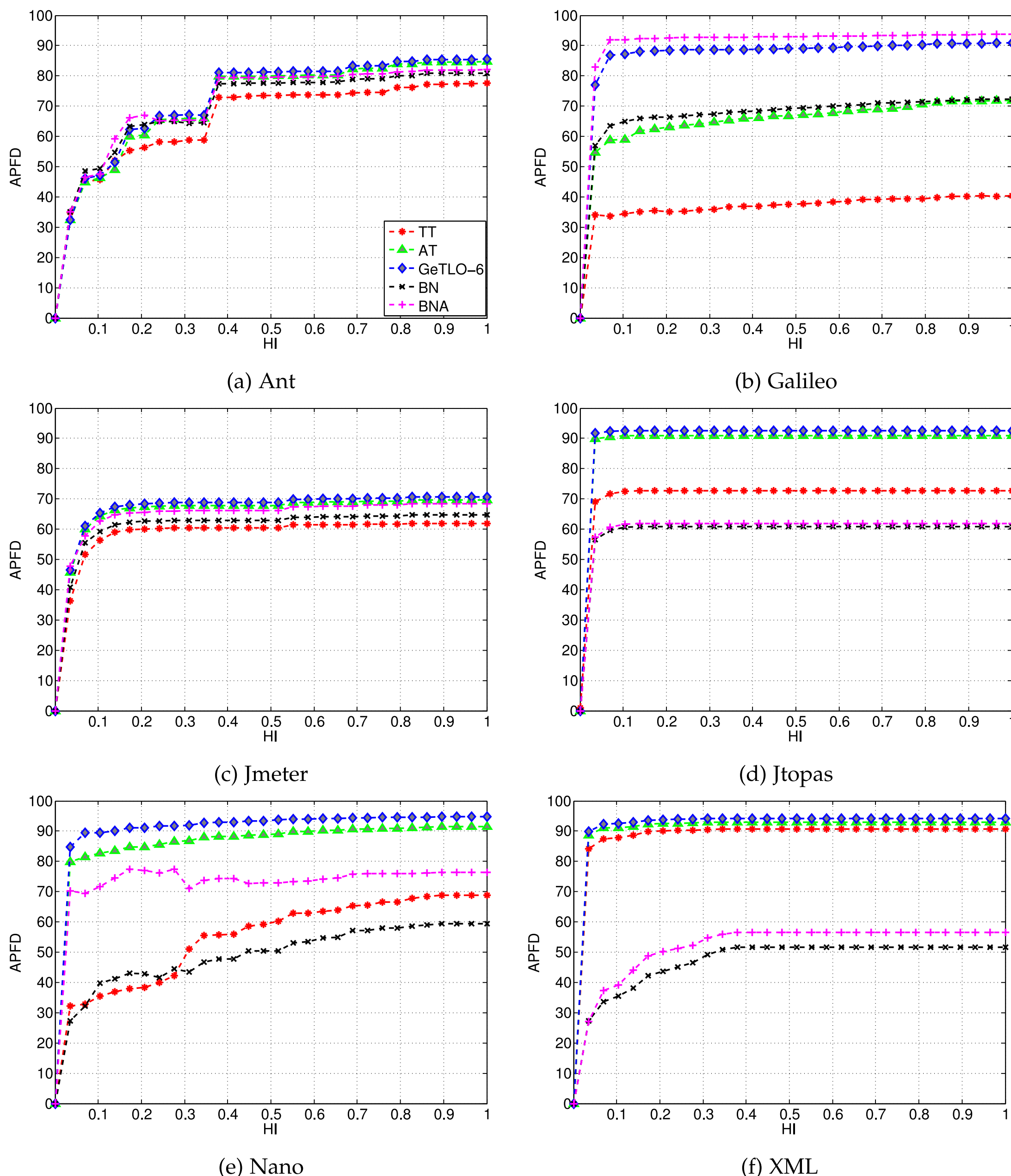

Fig. 7. APFD in terms of HI for six subject programs at the basic block level.

metrics. These results indicate that AT is the most efficient technique in majority of cases. Wong et al. [52] have proposed a method by combining a greedy based approach with test selection based on the cost per additional coverage metric. More recently, Zhang et al. [21] have studied a spectrum of methods between TT and AT. In their work, based on two complexity metrics, blocks are assigned with values that estimate the probability of containing faults. Then, test cases are sorted with respect to their likelihood of detecting faults in the blocks that they cover.

Test case prioritization is an optimization problem by definition. Naturally, various studies have been conducted to use optimization techniques for solving it. Usually, the underlying optimization problem is accompanied by a time or budget constraint. These prioritization problems are referred to as *cost-aware test case prioritization*. Mirarab et al. [10] have presented a multi objective technique which aims at maximizing the minimum coverage, as well as maximizing the total coverage. Similarly, Epitropakis et al. [53] reformulate RTP as a pareto multi-objective optimization problem with three objectives. Specifically, they consider: i) statement coverage, ii) difference of statement coverage, and iii) historical fault information. Zhang et al. [22] have formulated RTP as a constraint integer linear programming. They, then, adopt linear programming to solve it. Do et al. [46] have investigated the benefit of applying prioritization techniques subject to a set of constraints by incorporating a cost-benefit model.

Researches have also aimed at boosting the performance of coverage-based techniques by incorporating other sources of information. In this context, Elbaum et al. [2] have incorporated a fault proneness metric, called fault index, which uses the change data to guide their technique to focus

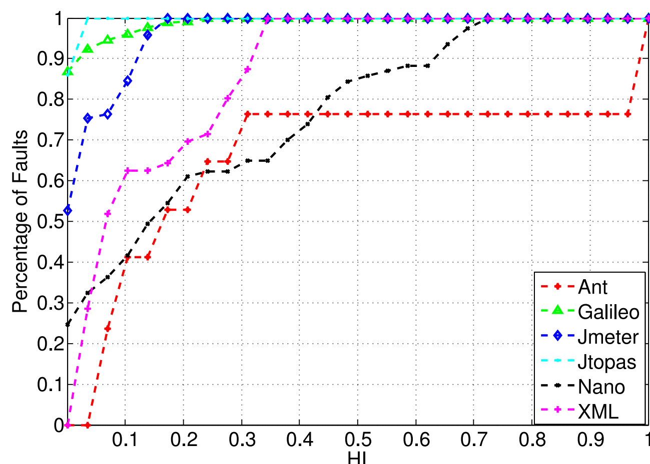

Fig. 8. Cumulative percentage of faults in terms of the HI values for the most recent versions of the programs. This plot is cumulative meaning that each value such as (x, y) on the plot indicates the percentage of faults with HI $<$ x is equal to y for that program.

on entities of code which are more fault prone. Similarly, Srivastava and Thiagarajan have built a test case prioritization system, called *Echelon*, which implements a prioritization technique based on the changes that have been made to the program [54]. Echelon first detects the changes by comparing the binary files of new and previous versions of the code, and then, the prioritization algorithm orders the test cases with respect to the coverage of modified parts of the code. Jerry and Gupta study the use of coverage requirement in the relevant slices of the output of the test cases [55]. Park et al. [56] also have used historical data to propose a prioritization technique based on cost/fault severities. Carlos et al. [57] propose a prioritization technique which incorporates a hierarchical clustering method to cluster the test cases. They have utilized code coverage, code complexity and fault history as the features for clustering. They empirically have shown that test cases within the same cluster tend to have similar fault detection capability. In a related study, Arafeen and Do [58] have used requirement information to cluster and prioritize test cases. Thomas et al. [59] have considered the source code of each test case as a text document. They have applied a text analysis algorithm (topic model) to the source code of test cases to evaluate their capability of fault detection. More Recently, Alshahwan and Harman [60] empirically have shown that *output uniqueness* has positive correlation with statement, branch and path coverage. That is, two test cases that generate different outputs are more likely to traverse different paths thus are likely to cover different entities.

6 Conclusions and Future Works

In this paper, we first argue that acting randomly in the case of ties can degrade the performance of the AT algorithm. We empirically show that it is very likely for AT to face ties. To break the ties, unlike AT-type techniques which only consider the not-yet-covered entities for coverage, entities are assigned with priorities for further coverage. In other words, all entities are considered where less covered entities receive higher priority for coverage. The main advantage of the proposed technique is that it can efficiently breaks ties. Based on this idea, we then propose the basic algorithm using the lexicographical ordering of cumulative coverage vectors. In the next step, we propose GeTLO algorithm by modifying the basic algorithm to reduce its time complexity. We also prove that both algorithms generate the same ordering. Then, we show that even GeTLO algorithm might face ties. Therefore, we present an algorithm which takes into account all possible candidates and further evaluates each one to find the best test case.

To gauge the performance of proposed ideas, they are examined through empirical studies. We evaluate the results of different techniques, including the ones proposed in the current article, using a set of well-known subject programs. The results reveals that GeTLO outperforms all except one technique (AT) in all cases, AT is also outperformed by the proposed technique in 4 out of 6 cases and exhibits a similar performance in others.

Using the notion of lexicographical ordering when the coverage vectors do not take binary values can be a future line of research. Other sources of information, such as change data, can be incorporated along with the proposed technique. Moreover, to study the presented ideas in more depth, new empirical studies can compare the effect of different levels of granularity of the coverage.

All of the coverage-based techniques are based on the high correlation between coverage and rate of fault detection. We observe that for some programs, where this correlation is not high, we cannot expect a desirable performance. Future work can formulate this dependency and then propose bounds on the performance of coverage based techniques based on the value of correlation.

Acknowledgments

This research was supported by Natural Sciences and Engineering Research Council of Canada (NSERC). The authors would like to express their appreciations to the associate editor and the anonymous reviewers for their valuable suggestions.

References

[1] S. Elbaum, A. G. Malishevsky, and G. Rothermel, “Prioritizing test cases for regression testing,” *Softw. Eng. Notes*, vol. 25, no. 5, pp. 102–112, 2000. [Online]. Available: http://doi.acm.org/10.1145/347636.348910.

[2] S. Elbaum, A. G. Malishevsky, and G. Rothermel, “Test case prioritization: A family of empirical studies,” *IEEE Trans. Softw. Eng.*, vol. 28, no. 2, pp. 159–182, Feb.. 2002.

[3] A. Kumar, “Development at the speed and scale of Google,” *Int. Softw. Develop. Conf.*, 2010, Presentation slides are available at: qconsf.com/sf2010/. [Online]. Available: Presentation slides are available at: https://qconsf.com/sf2010/.

[4] L. C. Briand, Y. Labiche, and G. Soccar, “Automating impact analysis and regression test selection based on uml designs,” in *Proc. Int. Conf. Softw. Maintenance*, 2002, pp. 252–261.

[5] E. G. Cartaxo, F. G. O. Neto, and P. D. L. Machado, “Automated test case selection based on a similarity function,” *Lecture Notes Informat.*, vol. 7, pp. 399–404, 2007.

[6] T. Y. Chen and M. F. Lau, “Dividing strategies for the optimization of a test suite,” *Inform. Process. Lett.*, vol. 60, no. 3, pp. 135–141, 1996.

[7] T. Y. Chen and M. F. Lau, “Test case selection strategies based on boolean specifications,” *Softw. Testing, Verification Rel.*, vol. 11, no. 3, pp. 165–180, 2001.

[8] M. J. Harrold and M. L. Souffa, “An incremental approach to unit testing during maintenance,” in *Proc. Int. Conf. Softw. Maintenance*, 1988, pp. 362–367.

[9] Y.-C. Huang, K.-L. Peng, and C.-Y. Huang, “A history-based cost-cognizant test case prioritization technique in regression testing,” *J. Syst. Softw.*, vol. 85, no. 3, pp. 626–637, 2012.
[10] S. Mirarab, S. Akhlaghi, and L. Tahvildari, “Size-constrained regression test case selection using multicriteria optimization,” *IEEE Trans. Softw. Eng.*, vol. 38, no. 4, pp. 936–956, Jul./Aug. 2012.
[11] S. Sprenkle, S. Sampath, E. Gibson, L. Pollock, and A. Souter, “An empirical comparison of test suite reduction techniques for user-session-based testing of web applications,” in *Proc. Int. Conf. Softw. Maintenance*, 2005, pp. 587–596.
[12] S. Yoo and M. Harman, “Pareto efficient multi-objective test case selection,” in *Proc. Int. Symp. Softw. Testing Anal.*, 2007, pp. 140–150.
[13] W. Dickinson, D. Leon, and A. Podgurski, “Fin failures by cluster analysis of execution profiles,” in *Proc. Int. Conf. Softw. Eng.*, 2001, pp. 339–348.
[14] D. Hao, X. Zhao, and L. Zhang, “Adaptive test-case prioritization guided by output inspection,” in *Proc. 37th Annu. Comput. Softw. Appl. Conf.*, 2013, pp. 169–179.
[15] J.-M. Kim and A. Porter, “A history-based test prioritization technique for regression testing in resource constrained environments,” in *Proc. Int. Conf. Softw. Eng.*, 2002, pp. 119–129.
[16] D. Leon and A. Podgurski, “A comparison of coverage-based and distribution-based techniques for filtering and prioritizing test cases,” in *Proc. Int. Symp. Softw. Rel. Eng.*, 2003, pp. 442–453.
[17] Z. Li, M. Harman, and R. M. Hierons, “Search algorithms for regression test case prioritization,” *IEEE Trans. Softw. Eng.*, vol. 33, no. 4, pp. 225–237, Apr. 2007.
[18] G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold, “Prioritizing test cases for regression testing,” *IEEE Trans. Softw. Eng.*, vol. 27, no. 10, pp. 929–948, Oct. 2001.
[19] A. Smith and G. Kapfhammer, “An empirical study of incorporating cost into test suite reduction and prioritization,” in *Proc. Symp. Appl. Comput.*, 2009, pp. 461–467.
[20] H. Srikanth, L. Williams, and J. Osborne, “System test case prioritization of new and regression test cases,” in *Proc. Int. Symp. Empirical Softw. Eng.*, 2005, pp. 10.
[21] L. Zhang, D. Hao, L. Zhang, G. Rothermel, and H. Mei, “Bridging the gap between the total and additional test-case prioritization strategies,” in *Proc. Int. Conf. Softw. Eng.*, 2013, pp. 192–201.
[22] L. Zhang, S.-S. Hou, C. Guo, T. Xie, and H. Mei, “Time-aware test-case prioritization using integer linear programming,” *Proc. 18th Int. Symp. Softw. Testing Anal.*, 2009, pp. 213–224.
[23] S. Hong, M. Staats, J. Ahn, M. Kim, and G. Rothermel, “The impact of concurrent coverage metrics on testing effectiveness,” in *Proc. Int. Conf. Softw. Testing, Verification Validation*, 2013, pp. 232–241.
[24] G. Rothermel, S. Elbaum, A. Malishevsky, P. Kallakuri, and B. Davia, “The impact of test suite granularity on the cost-effectiveness of regression testing,” in *Proc. Int. Conf. Softw. Eng.*, 2002, pp. 130–140.
[25] Y. W. Kim, “Efficient use of code coverage in large-scale software development,” in *Proc. Centre Adv. Studies Conf.*, 2003, pp. 145–155.
[26] A. S. Namin and J. H. Andrews, “The influence of size and coverage on test suite effectiveness,” in *Proc. Int. Symp. Softw. Testing Anal.*, 2009, pp. 57–68.
[27] P. Piwowarski, M. Ohba, and J. Caruso, “Coverage measurement experience during function test,” in *Proc. Int. Conf. Softw. Eng.*, 1993, pp. 287–301.
[28] C. Kaner, J. Bach, and B. Pettichord, *Lessons Learned in Software Testing*. Hoboken, NJ, USA: Wiley, 2008.
[29] S. Park, B. Hossain, I. Hussain, C. Csallner, M. Grechanik, K. Taneja, C. Fu, and Q. Xie, “Carfast: Achieving higher statement coverage faster,” in *Proc. Int. Symp. Foundations Softw. Eng.*, 2012, pp. 1–11.
[30] H. Do, S. Mirarab, L. Tahvildari, and G. Rothermel, “The effects of time constraints on test case prioritization: A series of controlled experiments,” *IEEE Trans. Softw. Eng.*, vol. 36, no. 5, pp. 593–617, Sep./Oct. 2010.
[31] H. Mei, D. Hao, L. Zhang, L. Zhang, J. Zhou, and G. Rothermel, “A static approach to prioritizing JUnit test cases,” *IEEE Trans. Softw. Eng.*, vol. 38, no. 6, pp. 1258–1275, Nov./Dec. 2012.
[32] H. Do, S. Elbaum, and G. Rothermel, “Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact,” *Empirical Softw. Eng.*, vol. 10, no. 4, pp. 405–435, 2005.
[33] B. Jiang, Z. Zhang, W. K. Chan, and T. H. Tse, “Adaptive random test case prioritization,” in *Proc. Int. Conf. Automated Softw. Eng.*, 2009, pp. 233–244.
[34] S. Sampath, S. Sprenkle, E. Gibson, and L. Pollock, “Integrating customized test requirements with traditional requirements in web application testing,” in *Proc. Testing, Analysis, Verification Web Services Appl.*, 2006, pp. 23–32.
[35] J.-W. Lin and C.-Y. Huang, “Analysis of test suite reduction with enhanced tie-breaking techniques,” *Inform. Softw. Technol.*, vol. 51, no. 4, pp. 679–690, 2009.
[36] P. C. Fishburn, “Lexicographic orders, utilities and decision rules: A survey,” *Manag. Sci.*, vol. 20, no. 11, pp. 1442–1471, 1974.
[37] R. G. Hamlet, “Testing programs with the aid of a compiler,” *IEEE Trans. Softw. Eng.*, vol. SE-3, no. 4, pp. 279–290, 1977.
[38] J. H. Andrews, L. C. Briand, and Y. Labiche, “Is mutation an appropriate tool for testing experiments?” in *Proc. Int. Conf. Softw. Eng.*, 2005, pp. 402–411.
[39] J. H. Andrews, L. C. Briand, Y. Labiche, and A. S. Namin, “Using mutation analysis for assessing and comparing testing coverage criteria,” *IEEE Trans. Softw. Eng.*, vol. 32, no. 8, pp. 608–624, Aug. 2006.
[40] H. Do and G. Rothermel, “On the use of mutation faults in empirical assessments of test case prioritization techniques,” *IEEE Trans. Softw. Eng.*, vol. 32, no. 9, pp. 733–752, Sep. 2006.
[41] Y.-S. Ma, J. Offutt, and Y. R. Kwon, “Mujava: An automated class mutation system,” *Software Testing, Verification Rel.*, vol. 15, no. 2, pp. 97–133, 2005.
[42] A. Kinneer, M. Dwyer, and G. Rothermel, “Sofya: A flexible framework for development of dynamic program analyses for java software,” Depart. Comput. Sci. Eng., Univ. Nebraska, Lincoln, NE, Tech. Rep., 2006.
[43] L. Zhang, M. Kim, and S. Khurshid, “Faulttracer: A change impact and regression fault analysis tool for evolving Java programs,” in *Proc. Int. Symp. Foundations Softw. Eng.*, 2012, pp. 40–44.
[44] H. Do, G. Rothermel, and A. Kinneer, “Prioritizing JUnit test cases: An empirical assessment and cost-benefits analysis,” *Empirical Softw. Eng.*, vol. 11, no. 1, pp. 33–70, 2006.
[45] S. Mirarab and L. Tahvildari, “An empirical study on Bayesian network-based approach for test case prioritization,” in *Proc. Int. Conf. Softw. Testing, Verification, Validation*, 2008, pp. 278–287.
[46] S. Mirarab and L. Tahvildari, “A prioritization approach for software test cases based on Bayesian networks,” *Proc. 10th Int. Conf. Fundamental Approaches Softw. Eng.*, 2007, pp. 276–290.
[47] S. Elbaum, A. Malishevsky, and G. Rothermel, “Incorporating varying test costs and fault severities into test case prioritization,” in *Proc. Int. Conf. Softw. Eng.*, 2001, pp. 329–338.
[48] W. H. Kruskal and W. A. Wallis, “Use of ranks in one-criterion variance analysis,” *J. Am. Statist. Assoc.*, vol. 47, no. 260, pp. 583–621, 1952.
[49] G. R. Iversen and H. Norpoth, *Analysis of Variance*. Thousand Oaks, CA, USA: Sage, 1987, no. 1.
[50] L. Zhang, D. Marinov, L. Zhang, and S. Khurshid, “An empirical study of JUnit test-suite reduction,” in *Proc. Int. Symp. Softw. Rel. Eng.*, 2011, pp. 170–179.
[51] D. Jeffrey and R. Gupta, “Test suite reduction with selective redundancy,” in *Proc. Int. Conf. Softw. Maintenance*, 2005, pp. 549–558.
[52] E. W. Wong, J. R. Horgan, S. London, and H. Agrawal, “A study of effective regression testing in practice,” in *Proc. Int. Symp. Softw. Rel. Eng.*, 1997, pp. 264–274.
[53] M. G. Epitropakis, S. Yoo, M. Harman, and E. K. Burke, “Pareto efficient multi-objective regression test suite prioritisation,” UCL, London, U.K., Dept. Comput. Sci., Tech. Rep. RN/14/01, 2014.
[54] A. Srivastava and J. Thiagarajan, “Effectively prioritizing tests in development environment,” *Softw. Eng. Notes*, vol. 27, no. 4, pp. 97–106, 2002.
[55] D. Jeffrey and R. Gupta, “Test case prioritization using relevant slices,” in *Proc. Int. Comput. Softw. Appl. Conf.*, 2006, vol. 1, pp. 411–420.
[56] H. Park, H. Ryu, and J. Baik, “Historical value-based approach for cost-cognizant test case prioritization to improve the effectiveness of regression testing,” in *Proc. Int. Conf. Secure Syst. Integr. Rel. Improvement*, 2008, pp. 39–46.
[57] R. Carlson, H. Do, and A. Denton, “A clustering approach to improving test case prioritization: An industrial case study,” in *Proc. Int. Conf. Softw. Maintenance*, 2011, pp. 382–391.

[58] M. J. Arafeen and H. Do, “Test case prioritization using requirements-based clustering,” in *Proc. IEEE Int. Conf. Softw. Testing, Verification Validation*, 2013, pp. 312–321.
[59] S. W. Thomas, H. Hemmati, A. E. Hassan, and D. Blostein, “Static test case prioritization using topic models,” *Empirical Software Eng.*, vol. 19, no. 1, pp. 182–212, 2014.
[60] N. Alshahwan and M. Harman, “Coverage and fault detection of the output-uniqueness test selection criteria,” in *Proc. Int. Symp. Softw. Testing Anal.*, 2014, pp. 181–192.

Sepehr Eghbali received the BSc degree in computer engineering from the Isfahan University of Technology, Iran in 2008 and the MSc degree in artificial intelligence and robotics from the University of Tehran in 2012. He is currently working toward the PhD degree from the University of Waterloo, Waterloo, ON, Canada. His current research interests include fast retrieval on large scale datasets.

Ladan Tahvildari is an associate professor in the Department of Electrical and Computer Engineering at the University of Waterloo and the founder of the Software Technologies Applied Research (STAR) Laboratory. Together with her research team, she investigates methods, models, architectures, and techniques to develop software systems with a higher quality in a cost effective manner. She served as a guest co-editor for the IEEE Transactions on Software Engineering issue July/August 2009. She has also been on the program and organization committees of many international IEEE/ACM conferences. She was the publications chair of the IEEE/ACM ICSE 2009 in Vancouver and the program chair of the IEEE ICSM 2007 in Paris. She has served as Chair of the IEEE Computer Society (CS), and IEEE Women in Engineering Affinity Group in the local chapter since 2004. Various awards have recognized her research accomplishments. Recently, she has been honoured with the prestigious Ontario’s Early Researcher Award (ERA) to recognize her work in self-adaptive software. She is a senior member of the IEEE.

▷ **For more information on this or any other computing topic, please visit our Digital Library at** www.computer.org/publications/dlib.