SOFTWARE TESTING, VERIFICATION AND RELIABILITY

Softw. Test. Verif. Reliab. 2018; **00**:1–26

Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/stvr

Empirical Evaluation of Mutation-based Test Case Prioritization Techniques

Donghwan Shin[1]*, Shin Yoo[1], Mike Papadakis[2], Doo-Hwan Bae[1]

[1]*KAIST, 291 Daehak-ro Yuseong-gu, Daejeon, Republic of Korea*
[2]*Interdisciplinary Center for Security, Reliability and Trust (SnT), University of Luxembourg, Luxembourg*

SUMMARY

In this paper, we propose a new test case prioritization technique that combines both mutation-based and diversity-aware approaches. The diversity-aware mutation-based technique relies on the notion of mutant distinguishment, which aims to distinguish one mutant's behavior from another, rather than from the original program. The relative cost and effectiveness of the mutation-based prioritization techniques (i.e., using both the traditional mutant kill and the proposed mutant distinguishment) are empirically investigated with 352 real faults and 553,477 developer-written test cases. The empirical evaluation considers both the traditional and the diversity-aware mutation criteria in various settings: single-objective greedy, hybrid, and multi-objective optimization. The results show that there is no single dominant technique across all the studied faults. To this end, the reason why each one of the mutation-based prioritization criteria performs poorly is discussed, using a graphical model called Mutant Distinguishment Graph (MDG) that demonstrates the distribution of the fault-detecting test cases with respect to mutant kills and distinguishment.

Received . . .

KEY WORDS: Mutation testing, Test case prioritization, Regression testing

1. INTRODUCTION

Regression testing is performed when changes are made to existing software; the purpose of regression testing is to provide confidence that the performed changes do not obstruct the behavior of the existing, unchanged parts of the software [1]. In regression testing, test case prioritization finds an ordering of test cases that maximizes a desirable property, such as the rate of fault detection. To achieve this goal, test case prioritization needs to "predict" which test cases will detect faults.

Although test case prioritization techniques have been extensively studied in the literature [2, 3], most of them rely on the use of various types of structural coverage [1]. Little attention has been paid to advanced test elements like mutants (i.e., artificial faults). We believe the mutation-based criteria call for further attention, given that mutants have been shown to be effective at revealing

*Correspondence to: KAIST, 291 Daehak-ro Yuseong-gu, Daejeon, Republic of Korea. E-mail: donghwan@se.kaist.ac.kr

faults [4, 5] and that mutant killing (i.e., detecting the deference between a mutant and its original program) ratios are similar with the fault detection ratios [6, 7, 8, 9]. Yet very few approaches study the mutation-based test case prioritization and none of them evaluates them with real-world applications and faults.

Recent advances in test case prioritization focus on identifying and promoting the diversity of the selected test cases [10, 3, 11], rather than maximizing the coverage. This trend can provide several advantages, especially in cases where there is no source code availability [3]. Therefore, the combination of mutation-based and diversity-based approaches could provide substantial benefits by increasing early fault detection. Investigating such a combination is the primary aim of this study.

In this paper, we propose and empirically investigate a new diversity-aware mutation-based test case prioritization technique. The technique relies on the diversity-aware mutation adequacy criterion, which is recently proposed by Shin *et al.* [12, 13]. The diversity-aware criterion aims at distinguishing the behavior of every mutant from that of all the others (including the original program), in contrast to the traditional mutation adequacy criterion which aims at distinguishing only the behavior of the mutants from that of the original program. According to Shin *et al.*, distinguishing mutants improves the fault detection capabilities of mutation testing. Therefore, our diversity-aware mutation-based prioritization gives higher priority to those test cases that help distinguish all mutants as early as possible.

Our study investigates the relative cost and effectiveness of two mutation-based prioritization techniques, i.e., one using traditional mutant kill and another using distinguishement, with real-world applications and faults. For this, we use 352 real faults and 553,477 developer-written test cases in the `Defects4J` data set [14]. The empirical evaluation considers both the traditional kill-only and the proposed diversity-aware mutation-based prioritization criteria in various settings: single-objective greedy, single-objective hybrid, as well as multi-objective optimization that seeks to prioritize using both criteria simultaneously. We find that there is no single superior technique. To this end, we provide a graphical model called Mutant Distinguishment Graph (MDG) to help us understand how a set of test cases that kills and distinguishes mutants related with fault detection. This visualization scheme demonstrates why and when each one of the mutation-based prioritization criteria performs poorly.

Overall, the technical contributions of this paper can be summarized as follows:

- We present a large empirical study that investigates the relative cost and effectiveness of mutation-based prioritization techniques with real faults.
- We investigate two different mutation-based test prioritization techniques under both single (greedy and hybrid) and multi-objective prioritization schemes.
- We investigate and identify the reasons behind the differences between the traditional kill-only mutation and distinguish mutation prioritization schemes, using intuitive graphical models.

The rest of this paper is organized as follows. Section 2 provides background for mutation adequacy criteria and test case prioritization. Section 3 explains the mutation-based test case prioritization techniques that are studied in this paper. Section 4 explains our experimental settings, including research questions, measures and variables, subject faults, test, and mutants. The results of the empirical evaluation are given in Section 5, together with the threats to validity. Section 6 presents the related work, and Section 7 concludes this paper.

DOI: 10.1002/stvr

2. BACKGROUND

2.1. Mutation Adequacy Criteria

In the late 1970s, DeMillo *et al.* [15] proposed the mutation adequacy criterion as a way to assess the quality of a test suite. The criterion focuses on the differences between the original program version and its mutant versions (i.e., artificially mutated programs) in the program outputs to measure the mutant kills. This technique relies on the idea that a test suite capable of distinguishing the behavior of mutants from those of the original programs are also capable of revealing faults. This idea was recently extended by Shin *et al.* [13], who proposed distinguishing the behavior of mutants among themselves (in addition to the original program). This forms a diversity-aware mutation adequacy criterion that caters for the diversity of behaviors introduced by the mutants.

To be precise, we formally represent and discuss the mutation adequacy criteria using the essential elements of an existing formal framework (for the mutation-based testing methods) [16]. Let P be a set of programs which includes the program under test. There are an original program $p_o \in P$ and a mutant $m \in M \subset P$ generated from p_o. For a test case t in a set of test cases T, if the behaviors of p_o and m are different for t, it is said that t *kills* m. Note that the notion of behavioral difference is an abstract concept. It is formalized by a testing factor, called a test differentiator, which is defined as follows:

Definition 1 (Test Differentiator)
A *test differentiator* $d : T \times P \times P \rightarrow \{0, 1\}$ is a function,[†] such that

$$d(t, p_x, p_y) = \begin{cases} 1\ (true), & \text{if the behaviors of } p_x \text{ and } p_y \text{ are } \textit{different} \text{ for } t \\ 0\ (false), & \text{otherwise} \end{cases}$$

for all test cases $t \in T$ and programs $p_x, p_y \in P$.

By definition, a test differentiator concisely represents whether the behaviors of $p_x \in P$ and $p_y \in P$ are different for t. In addition to a differentiator which formalizes the difference of two programs for a single test, it will be helpful to consider whether the two programs are different for a set of tests. A *d-vector* is defined to represent such difference of the programs as follows:

Definition 2 (d-vector)
A *d-vector* $\mathbf{d} : T^n \times P \times P \rightarrow \{0, 1\}^n$ is an n-dimensional vector, such that

$$\mathbf{d}(TS, p_x, p_y) = \langle d(t_1, p_x, p_y), ..., d(t_n, p_x, p_y) \rangle$$

for all $TS = \{t_1, \cdots, t_n\} \subseteq T^n$, $d \in D$, and $p_x, p_y \in P$.

In other words, a differentiator d returns Boolean value (i.e., 0 or 1) from a single test, whereas a d-vector $\mathbf{d}$ returns n-dimensional Boolean vector from n test cases. Note that a test suite TS is used to denote the order of test cases in the test suite, while T denotes a set of tests without any particular order of test cases.

Using the test differentiator and d-vector, we define the notion of mutant kill as follows:

[†]This function-style definition is replaceable by a predicate-style definition, such as $d \subseteq T \times P \times P$.

DOI: 10.1002/stvr

Test case	$d(t_i, p_o, m_1)$	$d(t_i, p_o, m_2)$	$d(t_i, p_o, m_3)$	$d(t_i, p_o, m_4)$
t_1	1	1	1	1
t_2	0	0	1	1
t_3	0	1	0	1

Figure 1. A working example for mutation adequacy criteria. The table represents whether a test case kills a mutant. For example, $d(t_1, p_o, m_1)$ is 1 which means that t_1 kills m_1.

Definition 3 (Mutant Kill)
A mutant m generated from an original program p_o is *killed* by a test case t when the following condition holds:

$$d(t, p_o, m) \neq 0.$$

Similarly, m generated from p_o is *killed* by a test suite TS when the following condition holds:

$$\mathbf{d}(TS, p_o, m) \neq \mathbf{0}.$$

Based on the notion of mutant kill, the traditional mutation adequacy criterion is defined as follows:

Definition 4 (Traditional Mutation Adequacy Criterion)
For a set of mutants M generated from an original program p_o, a test suite TS is *mutation-adequate* when the following condition holds:

$$\forall m \in M, \mathbf{d}(TS, p_o, m) \neq \mathbf{0}.$$

This definition means that a test suite TS is mutation-adequate for M if and only if all mutants $m \in M$ are killed by at least one test case $t \in TS$. For example, consider four mutants and three test cases that provide the the killing matrix of Figure 1. The matrix entries are binary (0 or 1) and represent the outcome of the test-mutant execution, i.e., whether the test case kills a mutant or not. According to the definition of the traditional mutation adequacy criterion, a test suite $TS_1 = \{t_1\}$ is mutation-adequate for $M = \{m_1, m_2, m_3, m_4\}$ because all the mutants in M are killed by t_1.

Note that TS_1 is mutation-adequate for M, despite the fact that only one of the mutants in M is, in effect, used to assess the adequacy of the test suite. In other words, the diversity (differences between the mutant versions) offered by the four mutants in M is ignored. This is because the traditional mutation adequacy criterion simply checks whether each one of the mutants is killed (or not) without considering the differences between the mutants (diversity offered by the mutants). To address this limitation of the traditional mutation adequacy criterion, we first formalize the notion of mutant distinguishment is defined as follows:

Definition 5 (Mutant Distinguishment)
Two mutants m_x and m_y generated from an original program p_o are *distinguished* by a test case t when the following condition holds:

$$d(t, p_o, m_x) \neq d(t, p_o, m_y).$$

DOI: 10.1002/stvr

Similarly, m_x and m_y generated from p_o are *distinguished* by a test suite TS when the following condition holds:

$$\mathbf{d}(TS, p_o, m_x) \neq \mathbf{d}(TS, p_o, m_y).$$

In other words, a test suite distinguishes mutants when it distinguishes their d-vectors. In the working example of Figure 1, it is clear that $TS_1 = \{t_1\}$ cannot distinguish the four mutants in M.

We now introduce the diversity-aware mutation adequacy criterion, called the *distinguishing mutation adequacy criterion*, based on the mutant distinguishment as follows:

Definition 6 (Distinguishing Mutation Adequacy Criterion)
For a set of mutants M generated from an original program p_o, a test suite TS is *distinguishing mutation-adequate* when the following condition holds:

$$\forall m_x, m_y \in M', \mathbf{d}(TS, p_o, m_x) \neq \mathbf{d}(TS, p_o, m_y)$$

where $m_x \neq m_y$ and $M' = M \cup \{p_o\}$.

In other words, a test suite TS is distinguishing mutation-adequate if and only if TS distinguishes all the d-vectors of the mutants in $M' = M \cup \{p_o\}$. In Figure 1, $TS_3 = \{t_1, t_2, t_3\}$ is distinguishing mutation-adequate because it distinguishes all the d-vectors of the mutants in $M' = \{p_o, m_1, \cdots, m_4\}$ as $\mathbf{d}(TS_3, p_o, p_o) = \langle 0, 0, 0\rangle$, $\mathbf{d}(TS_3, p_o, m_1) = \langle 1, 0, 0\rangle$, $\mathbf{d}(TS_3, p_o, m_2) = \langle 1, 0, 1\rangle$, $\mathbf{d}(TS_3, p_o, m_3) = \langle 1, 1, 0\rangle$, and $\mathbf{d}(TS_3, p_o, m_4) = \langle 1, 1, 1\rangle$.

Further, it is possible to quantitatively measure the adequacy of test suites in terms of the distinguishing mutation adequacy criterion. To do so, the distinguishing mutation criterion requires computing the number of mutants distinguished by test suites. In the working example in Figure 1, the number of mutants distinguished by $TS_1 = \{t_1\}$ is 2, because $M' = \{p_o, m_1, m_2, m_3, m_4\}$ is distinguished as $\{p_o\}$ and $\{m_1, m_2, m_3, m_4\}$ by TS_1. After t_2 is added, the number of distinguished mutants increases to 3, because $TS_2 = \{t_1, t_2\}$ distinguishes $\{p_o\}$, $\{m_1, m_2\}$, and $\{m_3, m_4\}$. The number of distinguished mutants becomes 5 when all the mutants in M' are distinguished from each other by $TS_3 = \{t_1, t_2, t_3\}$, which is distinguishing mutation-adequate.

For the sake of simplicity, let d-criterion hereafter refers to the distinguishing mutation adequacy criterion (i.e., diversity-aware) and, similarly, k-criterion to the traditional mutation adequacy criterion (i.e., kill-only).

By definition, the d-criterion *subsumes* the k-criterion: for a set of mutants M generated from an original program p_o, if a test suite TS is adequate to the d-criterion, it is guaranteed that TS is adequate to the k-criterion as well. In other words, the d-criterion is stronger than the k-criterion. For more information, please refer to the recent study of Shin *et al.* [13].

2.2. Test Case Prioritization

Rothermel *et al.* [8] formally define the test case prioritization problem as follows:

Definition 7 (Test Case Prioritization Problem)
Given: A test suite, TS, the set of permutations of TS, Π, and an objective function from Π to real numbers, $f : \Pi \rightarrow \mathbb{R}$.
Problem: Find a permutation $\pi \in \Pi$ such that $\forall \pi' \in \Pi, (\pi' \neq \pi) \wedge (f(\pi) \geq f(\pi'))$.

DOI: 10.1002/stvr

Test case	Fault detected by test case				
	1	2	3	4	5
t_1	x				
t_2		x	x		
t_3	x			x	x

Figure 2. Example test suite with fault detection information. Executing t_3 first followed by t_2 is clearly the most beneficial ordering for early fault detection.

In Definition 7, Π represents all possible orderings of the given test cases in TS, and f represents an objective function that calculates an award value for an ordering $\pi \in \Pi$. Consider a test suite with fault detection information in Figure 2 as a simple example. It is clear that an ordering $\pi_1 = \langle t_3, t_2, t_1 \rangle$ is better than another ordering $\pi_2 = \langle t_1, t_2, t_3 \rangle$, since π_1 detects the faults earlier than π_2. However, it is not clear when we do not know the fault detection information. We need to consider a surrogate for fault detection based on the historical information of the test cases instead of re-executing them, hoping that early maximization of the surrogate will result in early maximization of fault detection. Therefore, while the goal of test case prioritization remains the early maximization of fault detection, it actually aims for the early maximization of the chosen surrogate. Naturally, the test case prioritization techniques vary depending on the chosen surrogate.

The structural coverage information, such as statement coverage, of test cases is one of the widely-used surrogates in test case prioritization [8, 17, 18]. For example, the statement-total approach prioritizes test cases according to the number of statements covered by individual test cases. In other words, a test case covering more statements has higher priority. Similarly, the statement additional approach prioritizes test cases according to the additional number of statements covered by individual test cases.

Mutants are also used as another surrogate for test case prioritization [8, 19, 20]. Instead of using the structural coverage of individual test cases, the mutant kill of individual test cases is utilized. For example, Rothermel *et al.* [8] consider the Fault Exposing Potential (FEP)-total approach that prioritizes test cases according to the number of mutants killed by individual test cases. Similarly, the FEP-additional approach prioritizes test cases according to the additional number of mutants killed by individual test cases. Note that, to kill a mutant, a test case not only needs to cover the location of mutation but also to execute the mutated part [1]. It means the mutation-based approaches can be constructed at least as strong as coverage-based approaches. In this paper, we focus on the mutation-based test case prioritization, using the two mutation-based adequacy criteria (i.e., kill and distinguish), while we use the coverage-based and random approaches as baselines.

Finally, we want to note that the main usage scenario of the prioritization techniques is to be used for the test of the program changes made on subsequent program versions. Recent research [3] has shown that the effectiveness degradation of the prioritization techniques over subsequent program versions is small and that taking into account the code changes performed on a subsequent version does not provide any important information [21]. Therefore, testers need to obtain the required information at a specific point and then use it to prioritize the relevant test suites in the subsequent program versions. In Section 4.2, we will discuss more about the usage scenario of mutation-based test case prioritization and its modification for our experiments.

Prepared using stvrauth.cls
DOI: 10.1002/stvr

2.3. Multi-Objective Test Case Prioritization

The essence of the multi-objective optimization is the notion of Pareto optimality. Given multiple objectives, an ordering of test cases is said to be *non-dominated* if none of the objectives can be improved in value without degrading the other objective values. Otherwise, an ordering of test cases is said to be *dominated* by another ordering that has at least one higher objective value without decreasing any other objective values. Formally, let O be the number of different objectives. For $i \in \{1, 2, \cdots, O\}$, each objective function is represented as $f_i : \Pi \rightarrow \mathbf{R}$. An ordering π is said to dominate another ordering π' if and only if the following is satisfied:

$$(\forall i \in \{1, 2, \cdots, O\}, f_i(\pi) \geq f_i(\pi')) \wedge (\exists i \in \{1, 2, \cdots, O\}, f_i(\pi) > f_i(\pi'))$$

When evolutionary algorithms are applied to multi-objective optimization, they produce a set of orderings that are not dominated by each other. Such a set is called a Pareto front. The number of orderings in a Pareto front is determined by the number of population in the evolutionary algorithms. For example, the Non-Dominated Sorting Genetic Algorithm II (NSGA-II) [22], one of the most widely studied multi-objective evolutionary algorithm, generates K number of Pareto optimal solutions in a Pareto front, where K is the predefined population size.

3. MUTATION-BASED TEST CASE PRIORITIZATION TECHNIQUES

In this paper, we consider six different test case prioritization techniques as described in Table I. The first column represents the mnemonic for each technique that will be used throughout this paper. The second column represents the prioritization objective of each technique. The letters for the mnemonic are capitalized. The third column represents the tie-breaking rule when there are multiple candidate test cases (for greedy and hybrid) or orderings (for multi-objective optimization) satisfying the same level of the objective(s). The last column summarizes each technique. Additional details regarding the techniques listed in Table I can be found in the following subsections.

3.1. Greedy and Hybrid Techniques

We first describe the single-objective greedy techniques: GRK, GRD, and HYB. Algorithmically, these techniques are in essence instances of additional greedy algorithms [23]. The additional greedy test case prioritization technique iteratively selects a test case that maximizes the additional achievement of the objective at a time. Note that the hybrid technique is also an instance of the single-objective additional greedy because its only objective is the form of the weighted sum of GRK and GRD.

GRK and GRD: Based on the k-criterion, GRK iteratively selects a test case that maximizes the number of additionally killed mutants. Formally, let $\kappa(t)$ be the number of additional mutants killed by a test case t. GRK iteratively selects t in a test suite TS that satisfies $\arg\max_{t \in TS}(\kappa(t))$. If there are multiple test cases additionally killing the same number of mutants, one of them is randomly selected. If there are no more test cases that can kill mutants, one of the remaining test cases is randomly selected.

Prepared using stvrauth.cls
DOI: 10.1002/stvr

Similarly, GRD iteratively selects a test case that maximizes the number of additionally distinguished mutants, based on the d-criterion as explained in Section 2.1. Formally, let $\delta(t)$ be the number of additional mutants distinguished by a test case t. GRD iteratively selects t in a test suite TS that satisfies $\arg\max_{t \in TS}(\delta(t))$. If there are multiple test cases additionally distinguishing the same number of mutants, one of them is randomly selected. If there are no more test cases that can distinguish mutants, one of the remaining test cases is randomly selected.

Semantically, GRK distinguishes mutants from its original program as early as possible, whereas GRD distinguishes all mutants from each other as early as possible. In other words, GRK is essentially based on the concept of intensification, whereas GRD is essentially based on the concept of diversification. Such difference may lead the effectiveness difference between GRK and GRD in prioritization. Section 5.6 discusses this issue in more detail.

As we explained in Section 2.2, GRK is another name of FEP-additional used by Rothermel *et al.* [8]. Since they have already report that FEP-additional is more effective than FEP-total, we only consider the additional approaches for the our greedy techniques.

HYB-w: This hybrid prioritization technique is a weighted sum of GRK and GRD. It iteratively selects a test case that maximizes the number of the weighted sum of additionally killed mutants and additionally distinguished mutants. Formally, for a weight factor $w \in [0, 1]$, HYB-w iteratively selects a test case t in a test suite TS that $\arg\max_{t \in TS}(w \times \kappa(t) + (1 - w) \times \delta(t))$. By definition, $w = 1$ refers the GRK technique and $w = 0$ refers the GRD technique.

3.2. Multi-Objective Optimization Techniques

Unlike the greedy techniques, which iteratively select a test case that suits its objective in a given situation, a multi-objective prioritization technique optimizes an ordering of test cases as a whole to both kill and distinguish mutants as early as possible.

MOK and MOD: To represent the two mutation-based objectives (i.e., *kill* mutants as early as possible and *distinguish* mutants as early as possible) as two measurable functions (i.e., fitness functions in an evolutionary algorithm), we define metrics called APMK (Average Percentage of Mutants Killed) and APMD (Average Percentage of Mutants Distinguished), respectively. The core of these metrics are in APFD (Average Percentage of Faults Detected) [17] that is the most commonly used test case prioritization evaluation metric. The APFD implies how quickly faults are detected by a given ordering of test cases. It is defined as the area under the curve connecting the points where (x, y) = (test suite fraction, percentage of faults detected) for a given ordering of test cases. The APFD value ranges from 0 to 1; higher APFD means more effective test case prioritization. We extract the core concept of the APFD as a template and call it APXX (Average Percentage of XX) that implies how quickly XX is satisfied by a given ordering of test cases. Figure 3 visualizes the APXX. To be precise, let $\pi(i)$ be the ordering fraction of the first i test cases for an ordering of n test cases, and let $PXX(\pi(i))$ be the percentage of XX for $\pi(i)$. Note that $PXX(\pi(n)) = 1$ by definition. For an ordering of n test cases, the APXX value as the area

DOI: 10.1002/stvr

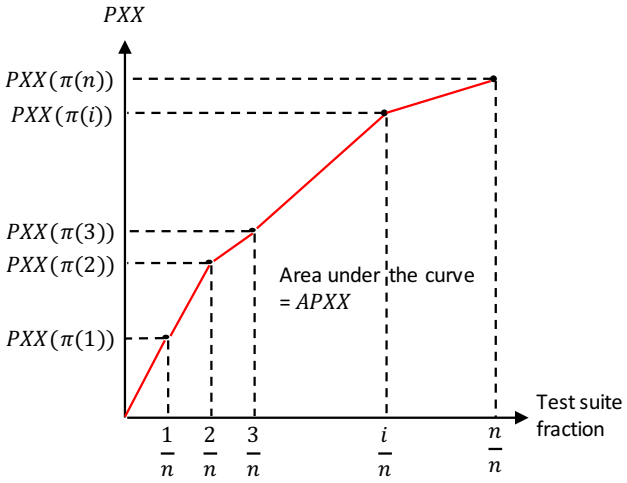


Figure 3. The concept of APXX. The area under the curve is the APXX value.

under the curve is calculated as follows:

$$APXX = \frac{1}{n}\sum_{i=1}^{n} PXX(\pi(i)) - \frac{1}{2n}$$

Using the APXX template, we define APMK and APMD as follows:

Definition 8 (APMK and APMD)
For an ordering π of a test suite TS, the APMK and APMD values are calculated as follows:

$$APMK = \frac{1}{n}\sum_{i=1}^{n} PMK(\pi(i)) - \frac{1}{2n}$$

$$APMD = \frac{1}{n}\sum_{i=1}^{n} PMD(\pi(i)) - \frac{1}{2n}$$

where $n = |TS|$ and $\pi(i)$ is the ordering fraction that contains the first i test cases.

In other words, the APMK and APMD imply how quickly mutants are killed and distinguished by a given ordering of test cases, respectively. As a result, the multi-objective prioritization technique optimizes an ordering of test cases to maximize both APMK and APMD values using a multi-objective optimization algorithm such as NSGA-II. As described in Section 2.3, NSGA-II returns a set of Pareto optimal orderings, and an additional rule is necessary to choose one of these orderings. MOK selects one of the Pareto optimal orderings that has the highest APMK value. Similarly, MOD selects one of the Pareto optimal orderings that has the highest AMPD value.

3.3. Techniques for Comparison

To facilitate our empirical studies, we introduce two simple but widely studied techniques as baselines.

DOI: 10.1002/stvr

RND: We consider random prioritization that randomly prioritizes test cases as a minimum prioritization baseline.

SCV: As an additional control in our studies, we apply the statement-coverage-based test case prioritization. As explained in Section 2.2, the structural coverage information is widely used surrogate in test case prioritization. We implement the statement-additional that iteratively selects a test case that maximizes the number of additionally covered statements, which is the most effective coverage-based prioritization scheme [3]. If there are multiple test cases that additionally covers the same number of statements, one of them is randomly selected.

4. EXPERIMENTAL DESIGN

4.1. Research Questions

In the experiments, we investigate the following five research questions:

- RQ1: How do the mutation-based prioritization techniques compare with the random and coverage-based prioritization in terms of early fault detection?
- RQ2: What is the superior mutation-based prioritization technique in terms of early fault detection?
- RQ3: What is the effect of using different weight values in the hybrid (single-objective) test prioritization scheme?
- RQ4: How effective are the Pareto front solutions of the multi-objective prioritization scheme?
- RQ5: How much time does it take to perform each one of the examined techniques?

RQ1 compares the effectiveness of the mutation-based prioritization techniques with that of the random and coverage-based prioritization. Specifically, we count the number of faults where each of the prioritization techniques is statistically significantly superior, equal, or inferior with the random ordering and the coverage-based ordering, respectively. We also measure the effect size of the effectiveness differences of the techniques with the controls.

RQ2 compares the effectiveness of the studied techniques among each other with the aim of identifying the best performing technique. Similar to RQ1, we count the number of faults where a technique A is statistically significantly superior, or equal, inferior to another technique B, as well as their exact effectiveness difference.

RQ3 focuses on the hybrid prioritization techniques that uses both the k-criterion (i.e., kill) and the d-criterion (i.e., distinguish). We examine different weight factors (between kill and distinguish) and see how it impacts the prioritization effectiveness.

RQ4 considers the effectiveness of orderings of test cases in a Pareto front given by the multi-objective test case prioritization techniques. For the multi-objective prioritization, all orderings of test cases in a Pareto front are equally good in terms of the their objectives. However, since the objectives are proxies, the important question is how these orderings perform in terms of the prioritization effectiveness. Thus, for the Pareto front orderings, we investigate the relationship between the prioritization objectives and the prioritization effectiveness.

DOI: 10.1002/stvr

RQ5 attempts to answer the cost of mutation-based prioritization techniques. One obvious cost of a prioritization technique is the execution time of the technique. We compare the execution times of all the mutation-based prioritization techniques including greedy, hybrid, and multi-objective.

4.2. Usage Scenario of Mutation-based Test Case Prioritization

To clearly explain the suggested usage scenario of mutation-based test case prioritization and the experimentally modified scenario, consider an original program p_o and a test suite TS for p_o. The suggested usage scenario consists of the following 4 steps:

- Step 1. Mutate the entire program p_o and record the killed mutants of each test case in TS by performing mutation analysis for TS. Also, record the coverage (e.g., covered classes) of each test case in TS.
- Step 2. Change happens on p_o: identify which classes are changed from p_o.
- Step 3. Only consider "relevant" test cases in TS whose coverage overlaps with the changes classes identified in Step 2. Similarly, only consider "relevant" mutants from the results of Step 1 in the changed classes.
- Step 4. Using the relevant test cases and mutants, chosen in Step 3, perform mutation-based test case prioritization based on the mutant kill record of Step 1.

Obviously, a developer cannot know a priori what classes will be modified, and mutants should be generated in all classes of p_o beforehand and mutation analysis should be performed for all test cases in TS in Step 1. While it looks expensive, it is rarely performed in practice as it supports a number of subsequent regressions with the small effectiveness degradation of prioritization [21]. Also, there is enough time to perform mutation analysis for TS in Steps 1 and 2 (i.e., the time from the code analysis to the testing time of the new changes).

However, in the experiments, we know what was changed and thus, there is no reason to generate "irrelevant" mutants and perform mutation analysis for "irrelevant" test cases in TS as we consider only those mutants and test cases "relevant" to the changed classes in Step 3 anyway. To speedup the experiments, we omit Step 1 and consider only those mutants and test cases "relevant" to the changed classes.

4.3. Test Subjects and Faults

For the purposes of the present study, we consider the Java applications in the `Defects4J` database [14]. These are all open source software systems and are accompanied by 357 developer-fixed and manually verified real faults. In total, we use the following five applications: JFreeChart (Chart), Closure compiler (Closure), Commons Lang (Lang), Commons Math (Math), and Joda-Time (Time). In `Defects4J`, each fault is given as an independent fault-fix pair of the program versions.

Out of 357 faults, five faults are excluded because they are not able to give mutation analysis results within a practical time limit (i.e., one-hour per each test case). As a result, we consider the remaining 352 faults, which are summarized in Table II. Detailed information for each subject fault is available from our webpage at `http://se.kaist.ac.kr/donghwan/downloads`.

DOI: 10.1002/stvr

4.4. Test Suites

For each fault, `Defects4J` provides "relevant" JUnit test cases that touch the modified classes between the faulty version and the fixed version. Test prioritization is performed when testing newly introduced changes. Thus, it is reasonable to use the information about the modified classes, and focus on them instead of the whole program. This is common practice in industry and is performed by retrieving the test cases that have a dependence with the files that were changed [24]. To account for this issue in our experiments, we compose a test suite of relevant test cases for each fault we consider.

JUnit test cases are Java classes that contain one or more test methods. It leads to two different test suite granularity by considering JUnit test cases as the test-class level and the test-method level [19]. We use the test-method level because it is finer and more informative than the test-class level. In Table II, the column Test Cases (sum) shows the sum of the number of test cases for each fault. For example, there are a total of 5,806 test cases for the 25 Chart faults. The column dT (sum) shows the sum of the number of fault-detecting test cases for each fault. For example, there are total 91 fault-detecting test cases for the 25 Chart faults. Total 553,477 test cases including 810 fault-detecting test cases are considered for the 352 subject faults.

4.5. Mutants

We use `Major` [25] mutation analysis tool for generating and executing all mutants to the test cases for each fault. It provides a set of commonly used set of mutation operators [9, 26] including the AOR (Arithmetic Operator Replacement), LOR (Logical Operator Replacement), COR (Conditional Operator Replacement), ROR (Relational Operator Replacement), ORU (Operator Replacement Unary), STD (STatement Deletion), and LVR (Literal Value Replacement). We applied all the mutation operators. Since the use of sufficient mutation operators may affect on the experimental results, we will discuss this issue in Section 5.7.

We generate mutants out of the fixed (i.e., clean) version of each fault. To perform a controlled experiment, we assume the fixed version is the norm, and perform mutation analysis on it: subsequently, we "reverse" the fix patch to recreate the fault, and evaluate our prioritization. We will discuss this in Section 5.7 as well.

We generate mutants only from the modified classes between the fixed version and the faulty version, as we considered only the relevant test cases. In Table II, the column aM (sum), kM (sum), and dM (sum) show the sum of the number of all generated mutants, killed mutants by the test cases, and distinguished mutants by the test suite for each fault, respectively. For example, for the 25 faults in the Chart program, 8,614 mutants and 1,462 mutants among 21,611 mutants are killed and distinguished by the test cases, respectively.

4.6. Multi-Objective Algorithm Configuration

For NSGA-II, we set the population size as 100. The chosen genetic operators are ones that are widely used for permutation type representation: partially matched crossover, swap mutation, and binary tournament selection [27, 28]. The crossover rate is set to 0.9, and the mutation rate is set to 0.2. The maximum fitness evaluation is set to 100,000. Since finding the best configuration for the mutation-based test case prioritization falls out of the scope of our work, we simply follow the

DOI: 10.1002/stvr

default configuration and parameter values, which are commonly used and tuned. Using the default parameter values is a common practice and has been found to be suitable for our context [29], i.e., search-based testing.

4.7. Variables and Measures

For independent variables, RQ1, RQ2, RQ5 manipulate all the prioritization techniques listed in Table I, whereas RQ3 and RQ4 focus on the hybrid techniques and the multi-objective techniques, respectively.

For dependent variables, we mainly measure the quality and the cost of the test case prioritization techniques. For the quality of the prioritization, we measure the APFD value for each ordering of test cases. For the cost of the prioritization, we measure the execution time for each ordering of test cases. To provide statistical analysis, we independently generate 100 orderings of test cases for each of the greedy, hybrid, and control techniques. For each of the multi-objective techniques, we independently generate 30 orderings of test cases because it takes too long (more than hours for one ordering in the longest case). All our experiments were performed on the Microsoft Azure Clould Platform using the Ubuntu 16.04 operating system on 8 DS3v2 (4 vcpus, 14 GB memory) virtual machines.

To compare the effectiveness of two prioritization techniques, we perform statistical hypothesis tests following the guideline provided by Arcuri and Briand [30]. We perform the Mann-Whitney U-test to assess the difference in stochastic order, that is, whether the APFD values in one technique are more likely to be greater than the APFD values in the other technique. Note that the Mann-Whitney U-test is a non-parametric test which makes no assumption about the distribution of the data. To reduce Type I error, the significance level is $\alpha = 0.001$. We also measure the Vargha and Delaneys $\hat{A}_{AB}$ statistics [31] to represent the effect size of the effectiveness difference between the compared prioritization techniques, A and B. It measures the number of times that the technique A yields higher APFD values than the technique B. For example, $\hat{A}_{AB} = 0.7$ means that the technique A outperforms the technique B in 70% of the runs. Usually, the differences between the compared techniques, as measured by $\hat{A}_{AB}$, can be characterized as small, medium, and large when the $\hat{A}_{AB}$ value exceeds 0.56, 0.64, and 0.71, respectively [31]. Note that $\hat{A}_{AB} = 1 - \hat{A}_{BA}$ and $\hat{A}_{AB} = \hat{A}_{BA} = 0.5$ means the two compared techniques are stochastically equivalent.

For the calculation of APFD values, we use the following equation:

$$APFD = \frac{1}{n}\sum_{i=1}^{n} PFD(\pi(i)) - \frac{1}{2n} \tag{1}$$

where $PFD(\pi(i))$ is the percentage of faults detected by the ordering fraction $\pi(i)$. We should note that there is another commonly used equation provided by Elbaum *et al.* [17] as follows:

$$APFD = 1 - \frac{TF_1 + \cdots + TF_n}{nm} + \frac{1}{2n} \tag{2}$$

where TF_j is the first test case position among n test cases which detects the jth fault among m faults. Both (1) and (2) give the same APFD value, whereas (1) uses the percentages of faults

DOI: 10.1002/stvr

detected by test suite fraction and (2) uses the positions of the first test case that detects each of faults.

To investigate the relationship between the prioritization effectiveness (i.e., APFD) and objectives (i.e., APMK[‡]), we measure the Pearson linear correlation and Spearman rank correlation between APFD and APMK for the orderings in Pareto fronts. When Pearson (or Spearman) correlation is 1, it means that APFD perfectly linearly (or monotonically) increases as APMK increases for the Pareto optimal orderings. When Pearson (or Spearman) correlation is -1, it means that APFD perfectly linearly (or monotonically) decreases as APMK increases for the Pareto optimal orderings. Consequently, the closer to +1 the correlation is, the more effective MOK is, and the closer to -1 the correlation is, the more effective MOD is.

5. RESULTS AND ANALYSIS

5.1. RQ1: Comparison with Controls

Table III records the results for the comparison of the prioritization techniques with the random orderings. For every compared pair (A, B), the column Superiority provides the number of subject faults where the effectiveness of A is statistically superior (+), equal (=), or inferior (-) to B, based on the Mann-Whitney U-tests with $\alpha = 0.001$. The column Effect size provides the average $\hat{A}_{AB}$ statistics to represent how much one technique outperforms the other in average. In terms of superior cases, HYB-010 is the best where 86.4% (304/352) of the subject faults show that the effectiveness of HYB-010 is statistically superior than that of random. In terms of inferior cases, GRD is the best where only 2.27% (8/352) of the subject faults show that the effectiveness of GRD is statistically inferior than that of random. In terms of effect size, HYB-015 is the best where the $\hat{A}_{AB}$ value is 0.8520. Overall, the mutation-based test case prioritization techniques are statistically superior than or equal to random for 95.5% of the subject faults. The average $\hat{A}_{AB}$ value is 0.8452, which means that the differences between the mutation-based prioritization techniques and random are large enough.

Table III also shows that hybrid techniques are at least effective as the simple greedy techniques GRD and GRK. Specifically, HYB-095 is more effective than GRK (i.e., HYB-100), even the weight for the GRD is only 0.05. This signifies that it is more effective to consider the k-criterion and the d-criterion together than to consider the k-criterion only.

Interestingly, multi-objective techniques are relatively ineffective than the hybrid techniques. It means that, in comparison with random, multi-objective optimization techniques using the k-criterion and the d-criterion are less beneficial than merely merging the two greedy techniques.

Table IV records the results related to the comparison of the prioritization techniques with SCV. The structure of the table is the same as Table III. In terms of superior cases, HYB-015 is the best where 76.1% (268/352) of the subject faults show that the effectiveness of HYB-015 is statistically superior than that of SCV. In terms of inferior cases, HYB-070, HYB-080, and HYB-090 are the best where 15.6% (55/352) of the subject faults show that the effectiveness of them are statistically

[‡]We do not need to additionally investigate the relationship between APMD and APFD because there is a clear inverse relationship between APMK and APMD in Pareto fronts.

Prepared using stvrauth.cls
DOI: 10.1002/stvr

inferior than that of SCV. In terms of effect size, HYB-090 is the best where the $\hat{A}_{AB}$ value is 0.7838. Overall, the mutation-based test case prioritization techniques are statistically superior than or equal to the coverage-based prioritization technique at least 83.3% of the subject faults. The average $\hat{A}_{AB} = 0.7699$.

> The mutation-based prioritization is superior to or equal to the random prioritization for 95.5% of the faults, and is superior to or equal to the coverage-based prioritization for 83.3% of the faults.

5.2. RQ2: Comparison between the Techniques

This section investigates whether there is a superior technique or not among the mutation-based test case prioritization techniques. We only consider GRD, HYB-010, HYB-050, HYB-090, GRK, MOK, and MOD, because there are too many pairs containing all weights for the hybrid techniques. Table V contains the comparison results for the pair of the techniques. The structure of the table is the same as Table III.

Comparing GRK and GRD in Table V, GRK is more effective at 60.8% (214/352) faults, whereas GRD is more effective at 25% (88/352) faults. There is no statistical difference for the remaining 14.2% (50/352) faults. For all subject faults, the average effect size $\hat{A}_{AB}$ is 0.6441, which means that GRK outperforms GRD with the probability of an average 64.41% of all runs. While GRK is more effective than GRD in general, GRD outperforms GRK for some faults. Section 5.6 discusses the effectiveness difference between GRK and GRD in more detail.

Interestingly, in comparison to GRD, the average $\hat{A}_{AB}$ values of the hybrid techniques are greater than 0.64, while that of the multi-objective techniques are less than 0.56. In other words, the hybrid techniques outperform GRD with medium effect sizes, while the multi-objective optimization techniques are stochatically almost equivalent to GRD. In terms of considering mutant kill and distinguishment together, simple hybrid is more effective than multi-objective optimization in mutation-based test case prioritization.

Comparing MOK and MOD, they are equally effective at 73.6% (259/352) faults, and it is almost the same when MOK is more effective and MOD is more effective for the remaining faults. This implies that orderings of test cases in a Pareto front may have similar prioritization effectiveness. This issue will be investigated in Section 5.4.

Overall, Table V shows that all pairs have both superior and inferior cases that cannot be ignored. Also, the average $\hat{A}_{AB}$ of all pairs are not large. It means that, in terms of mutation-based test case prioritization using the k-criterion and the d-criterion, there is no single superior technique among greedy, hybrid, and multi-objective.

> Among greedy, hybrid, and multi-objective strategies using the traditional kill-only mutation adequacy and the diversity-aware mutation adequacy, there is no single superior test case prioritization technique.

DOI: 10.1002/stvr

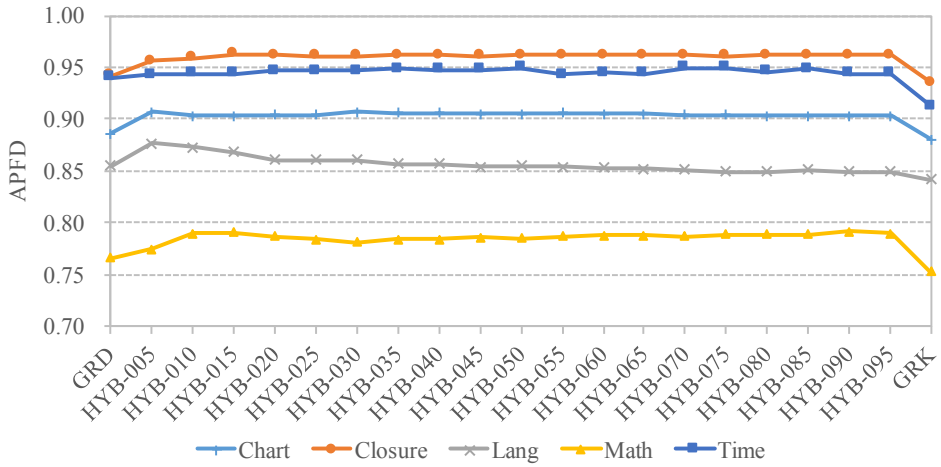


Figure 4. Effect of changing the weight factor in hybrid technique. The effectiveness is maximized when the weight is between 0 and 1 and not on the extreme values 0 or 1.

5.3. *RQ3: Effect of Changing Weight between Kill and Distinguish*

To investigate the effect of weight w change on APFD for the HYB-w prioritization techniques, the average APFD is obtained by changing w from 0 to 1 in steps of 0.05. Figure 4 shows the results; the x-axis is w and the y-axis is the APFD.

In Figure 4, all the subject programs show the same result: the highest APFD is when w is between 0 and 1 (i.e., neither 0 nor 1). This means that the combination of GRK and GRD has a positive effect on the test case prioritization effectiveness.

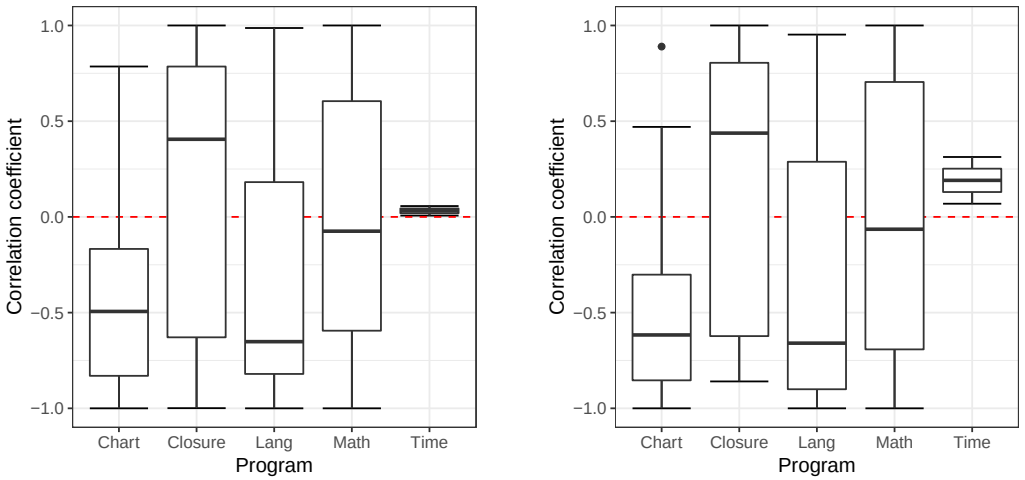
There is no single w value showing the highest APFD for all programs. For Chart and Lang, $w = 0.05$ shows the highest APFD. For Closure, $w = 0.15$ shows the highest APFD. For Math and Time, $w = 0.90$ and $w = 0.75$ shows the highest APFD, respectively. It means that the best weight w between GRK and GRD depends on the program characteristics.

> The test case prioritization effectiveness is maximized when the weight is between 0 and 1 and not on the extreme values 0 and 1. This means that the combination of GRK and GRD increases the effectiveness. The optimal weight depends on the subject programs.

5.4. *RQ4: Effectiveness of Orderings in Pareto Fronts*

This section investigates the effectiveness of orderings in Pareto fronts. To do that, we measure the Pearson and Spearman correlation coefficients between APMK (i.e., how quickly mutants are killed) and APFD (i.e., how quickly faults are detected) for the orderings in Pareto fronts. For 207 among the 352 subject faults (i.e., 58.8%), the correlation coefficients are undefined because the variance of APFD is zero. It means that, for the 207 fault, all the orderings in a Pareto front are equally good in terms of APFD. This partially explains the fact that MOK and MOD are statistically equally effective at 74.1% faults as noted in Section 5.2. Remaining 145 faults have correlation coefficients ranging from -1 to +1.

Prepared using stvrauth.cls
DOI: 10.1002/stvr



(a) Pearson correlation

(b) Spearman correlation

Figure 5. Correlation coefficients between APMK and APFD for orderings in Pareto fronts. Each point represents the (a) Pearson or (b) Spearman correlation coefficient of a fault. The bottom and top of the box are the first and third quartiles, and the band inside the box is the median. The top and bottom of the whisker are the highest and lowest datum still within 1.5 IQR of the upper and lower quartile. The correlation coefficients widely vary depending on the studied faults, except for the Time program. The box for the Pearson correlation of the faults in the Time program looks different because the orderings in Pareto fronts for each of the faults in the Time program have almost no linear correlation between APMK and APFD.

Figure 5 summarizes the distribution of correlation coefficients of the 145 (=352-207) faults. Each point represents the Pearson (in Figure 5a) or Spearman (in Figure 5b) correlation coefficient of a fault. The bottom and top of the box are the first and third quartiles, and the band inside the box is the median. The bottom and top of the whiskers are the lowest and highest datum still within 1.5 IQR of the lower and upper quartile. For example, Figure 5a shows that all faults in the Time program have very similar Pearson correlation coefficients, which is nearly zero. This zero Pearson coefficient means that there is no linear correlation between APMK and APFD for the orderings in Pareto fronts. Except the Time program, Figure 5 shows that both correlation coefficients are widely distributed from -1 to +1. This implies that there is no superiority between MOK and MOD on average for all programs. The faults in Chart and Lang tend to have correlation coefficients close to -1, whereas the faults in Closure tend to have correlation coefficients close to +1. It implies that MOK is often more effective than MOD for Chart and Lang, whereas MOD is often more effective than MOK for Closure. The faults in Math and Time tend to have zero correlation, which implies that the effectiveness of MOK and MOD is similar.

For 58.8% of the subject faults, the orderings of test cases in Pareto fronts are equally effective in terms of APFD. For the remaining faults, the correlation coefficient between APMK (or APMD) and APFD vary from -1 to +1, depending on the studied faults.

Prepared using stvrauth.cls
DOI: 10.1002/stvr

5.5. RQ5: Execution Time of the Techniques

Table VI shows the average test case prioritization time for each technique. For example, the GRK prioritization technique takes 2253.9 ms to prioritize a test suite on average. There is no time difference between MOK and MOD since both MOK and MOD simply select one orderings of test cases in a Pareto front, and the information needed for selection (i.e., APMK and APMD) is calculated beforehand.

In Table VI, GRD takes around 3.4 times more time than GRK. This is because the computation for mutant distinguishment (i.e., whether a mutant's d-vector is unique) is more complex than the computation for mutant kill (i.e., whether a mutant's d-vector is non-zero). Specifically, for a test suite TS and a set of mutants M generated from an original program p_o, the computation for mutant distinguishment takes $O(|TS| \times |M|^2)$ time to evaluate $\exists t \in TS, d(t, p_o, m_x) \neq d(t, p_o, m_y)$ per pair $m_x, m_y \in M \cup \{p_o\}$, whereas the computation for mutant kill takes $O(|TS| \times |M|)$ time to evaluate $\exists t \in TS, d(t, p_o, m) \neq 0$ per $m \in M$ [13]. HYB is similar to GRD, because it also requires the computation for mutant distinguishment. While GRD and HYB techniques take more time than GRK, it is within 8 seconds for each prioritization on average. On the other hand, MOK (or MOD) takes far much time; approximately 37 minutes for each prioritization. This is mainly because the number of test cases and mutants are too large to optimize permutations as a whole.

We also investigate the effect of the total number of test cases (i.e., the size of a test suite) and the total number of mutants on the execution time. It turns out that the product of the total number of test cases and the total number of mutants is linearly proportional to the time for all the subject test case prioritization technique. The average Pearson correlation coefficient between the product and the time for all the techniques is 0.930.

Note that Table VI only reports the execution time of the prioritization, not the time for mutation analysis. On average, mutation analysis takes 651.8 seconds per fault. However, there are several test cases that do not give mutation analysis results within one-hour time limit. While such test cases are excluded in our controlled experiment, it can be problematic in practice. Fortunately, mutation analysis for each test case can be easily parallelized. Further, it is possible to prepare the mutation analysis results independently from the future changes and regression testing.

> On average, multi-objective techniques requires approximately 37 minutes, whereas greedy and hybrid techniques require less than 8 seconds. The prioritization execution time for all techniques has an exact linear relationship with the product of the number of test cases and mutants.

5.6. Discussion

As described in Section 5.2, there is no clear winner between GRK (i.e., kill mutants as early as possible) and GRD (i.e., distinguish mutants as early as possible) test prioritization schemes; 60.8% of the subject faults show that GRK is statistically superior than GRD, whereas 25% of the subject faults show the opposite result. This is interesting because the d-criterion *subsumes* the k-criterion as explained in Section 2.1. Taken together, the d-criterion is stronger than the k-criterion, whereas the prioritization based on the d-criterion is not superior than the prioritization based on the k-criterion.

Prepared using stvrauth.cls
DOI: 10.1002/stvr

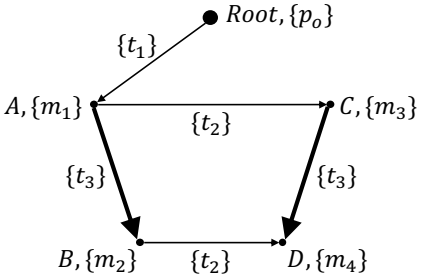


Figure 6. The MDG corresponding to the example in Figure 1

To further understand why this happens, we investigate the relationship between kill and distinguish in test case prioritization.

By definition, the mutant kill concerns the difference between the original program and its mutants, whereas the mutant distinguishment concerns the difference among all programs including the original program and mutants. To see how a set of test cases kills and distinguishes mutants and where the fault-detecting test cases are, we propose a graphical representation called *Mutant Distinguishment Graph* (MDG). In an MDG, each node represents a set of undistinguished mutants, and a directed edge from a node n_x to another node n_y represents a set of test cases that distinguishes n_y from n_x by killing the mutants in n_y not n_x. We call n_y as the child of n_x when there is a directed edge from n_x to n_y. To represent where the fault-detecting test cases are, we draw the edges with thickness to be proportional to the percentage of fault-detecting test cases (among the set of test cases represented by the edge). To avoid zero thickness, we give a default value even if the percentage is zero. There is a special "root" node that has no incoming edge. In other words, all the remaining nodes are the children of the root. The root node refers the original program and the mutants that are not killed by all the test cases in the MDG. The structure of an MDG varies depending on the given set of test cases and mutants.

For example, using the four mutants and three test cases given in Figure 1, we have the MDG as Figure 6. There are five nodes, $Root$, A, B, C, and D, with the five directed edges between the nodes. The $Root$ node at the top only has p_o because all the mutants are killed by the set of test cases $\{t_1, t_2, t_3\}$. The edge from $Root$ to A is labeled with t_1. This shows m_1 is distinguished from p_o by t_1. In other words, t_1 kills m_1. The edge from A to B is labeled with t_3, which shows m_2 is distinguished from m_1 by t_3. We assume t_3 is the fault-detecting test case as an example, and the edges labeled with t_3 are thicker than the others.

Note that there is no direct edge from $Root$ to B labeled with t_1 in Figure 6, while t_1 also kills m_2 as well as m_1. This is because the *transitivity* of the mutant distinguishment [16]. For mutants m_x, m_y, m_z and test cases t_x, t_y, if m_y is distinguished from m_x by t_x and m_z is distinguished from m_y by t_y, then m_z is always distinguished from m_x by both t_x and t_y. When it comes to an MDG, such transitivity implies that the edges between a node and its descendants can be omitted without any information loss. As a result, we are able to know that both t_1 and t_3 kill m_2 while there is no direct edge from $Root$ to B.

The transitivity of an MDG provides an interesting property for the test cases in the edges from $Root$. Among all test cases in an MDG, the test cases in the edges from $Root$ are *sufficient* to kill all the mutants except mutants in $Root$. Furthermore, a test case in the edges from $Root$ kills all the

DOI: 10.1002/stvr

mutants in the distinguished node and its descendants. For the example in Figure 6, $\{t_1\}$ (i.e., the set of test cases in the edges from $Root$) is sufficient to kill all mutants, and t_1 kills all the mutants in A (i.e., the distinguished node) and B, C, and D (i.e., the descendants of A).

The aforementioned property is an important key to understand the effectiveness difference between GRK and GRD in test case prioritization. For GRK, giving the test cases in the edges from $Root$ high priorities is clearly beneficial to kill all mutants as early as possible. In other words, GRK gives the test cases not in the edges from $Root$ low priorities. If there is no thick edges from $Root$, it means the fault-detecting test cases are not in the edges from $Root$, and GRK gives the fault-detecting test cases low priorities. In summary, GRK becomes ineffective when there is no thick edge from $Root$.

GRD tries to distinguish all mutants as early as possible. However, as GRD has to choose among test cases that distinguish mutants (select any edge instead of those from the $Root$) it is likely to give less priority to the test cases in the edges from $Root$. Thus, it is likely to give higher priorities on test cases that distinguish mutants than killing mutants. On the contrary, when fault-detecting test cases are triggered by mutant distinguishement (thick edges are not in the $Root$), we see that GRD becomes effective. In these cases GRD is more likely to outperform GRK.

Figure 7 shows the four representative MDGs from the 352 subject faults: Figure 7a and Figure 7b show the MDGs for the faults that GRD is much more effective than GRK, whereas Figure 7c and Figure 7d show the faults that GRK is much more effective than GRD. For each graph, the big node near the center refers $Root$. It is clear that there is no thick edge from $Root$ in Figure 7a and Figure 7b, whereas there is a thick edge from $Root$ in Figure 7c and Figure 7d.

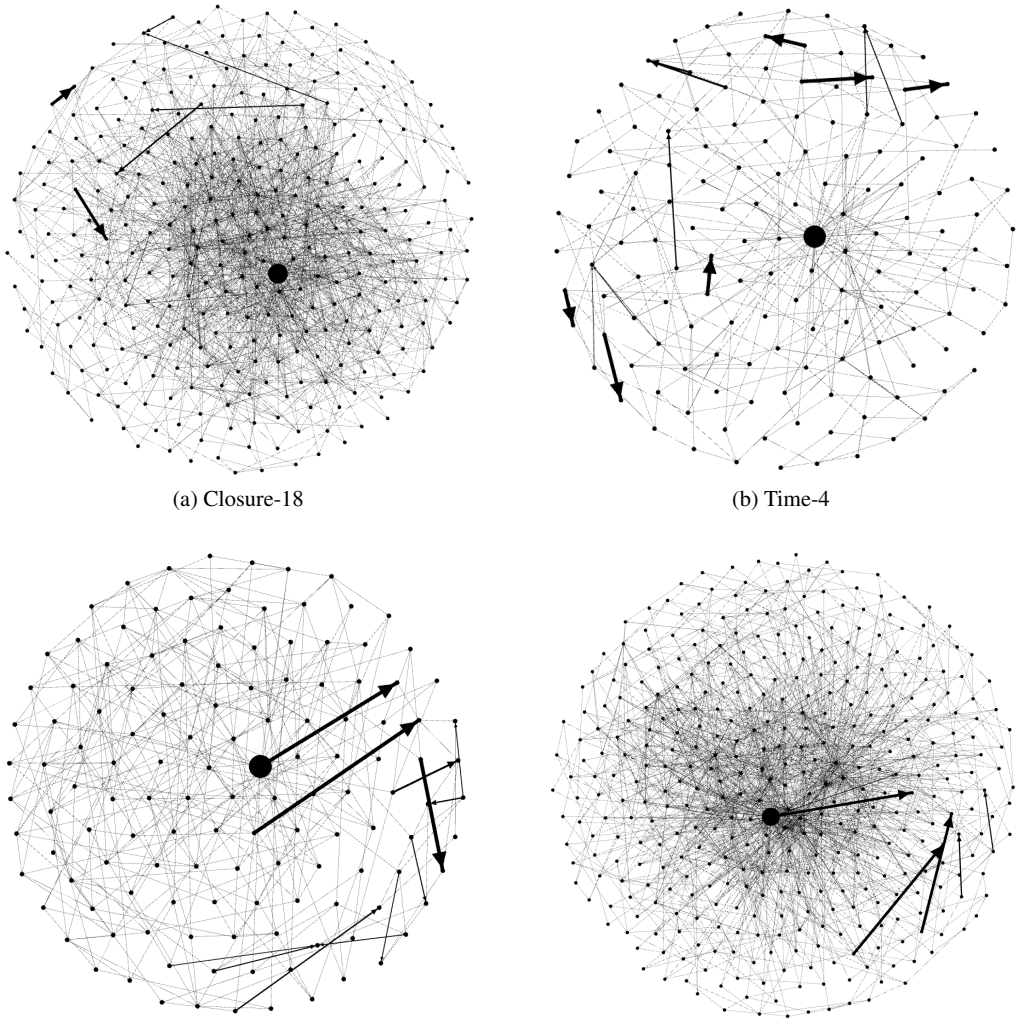
An MDG is useful in explaining the strengths and weakness of the mutation-based test case prioritization schemes based on the position of mutants that are killed or distinguished by fault-detecting test cases. However, an MDG cannot be used to predict which of GRK and GRD will be more effective than the other. This is the because fault-detecting test cases are not known at the time of prioritization, and the thickness of edges in an MDG cannot be determined without the fault detection information. Fortunately, recent studies show that it is possible to predict and prioritize mutants that are more likely to be killed by fault-detecting test cases than the others [32, 33]. Such studies will help to extend the use cases of an MDG. For example, it could be possible to develop another hybrid prioritization technique that dynamically selects between GRK and GRD to kill and distinguish the prioritized mutants.

We should note that an MDG is similar to the Mutant Subsumption Graph (MSG) suggested by Kurtz *et al.* [34], as the mutant distinguishment is closely related to the mutant subsumption as discussed by Shin and Bae [16]. The main difference between an MDG and an MSG is that an MDG additionally contains the information of the fault-detecting test cases in the thickness of edges.

5.7. Threats to Validity

There are several threats to validity for our experimental results. One threat is due to the subject programs and faults that we use. These might not be representative of other programs and faults. While this threat is common to any empirical study and can only be addressed by making multiple and context-related studies, we tried to mitigate it by using a large set of real faults. Thus, we used all the faults of `Defects4J`, which is an independently constructed dataset built to support controlled experimental results.

DOI: 10.1002/stvr



(a) Closure-18 (b) Time-4

(c) Math-19 (d) Closure-35

Figure 7. MDGs for representative cases. The big node near the center refers to *Root*. When there is no thick edge from *Root*, such as Closure-18 and Time-4, GRK is more likely to be ineffective than GRD. On contrary, when there is a thick edge from *Root*, such as Math-19 and Closure-35, GRK is more likely to be effective than GRD.

Our results are also to some extent dependent on the configuration of NSGA-II. Parameter turning is an important and challenging problem in evolutionary algorithms [35]. However, we feel that such a tuning will not impact much our results as the default configurations perform well in our context [29].

The mutation analysis tool `Major` [25] and the mutation operators we use form another source of threats for our study. This is because different types of mutants may result in different behaviour and influence our results. Therefore, the use of another mutation testing tool employing a different set of operators, like the `PIT` [36], may result in different findings. However, we do not consider this threat as vital as our main contribution lies in the relative comparison of the mutation-based test prioritization techniques and not on their optimal performance. Moreover, we expect that using different mutant sets will have a similar influence on all the prioritization techniques we study since

Prepared using stvrauth.cls
DOI: 10.1002/stvr

all of them rely on the same set of mutants. Nonetheless, according to a recent study by Kintis *et al.*[26], the fault detection capabilities of `PIT` are considerably lower than those of `Major`. The same study reports that another version of the tool, named `PIT_RV` (i.e., the research version of `PIT`), is 5% more effective at detecting faults than `Major`. Therefore, we believe that this 5% difference from `PIT_RV` cannot make a major difference on the results we report.

Other threats come from the order of the fixed and faulty versions. While the fixed version comes after the faulty version in the repository timeline, we assume the fixed version as the clean version that previously passes all regression test cases and the faulty version as the change-introduced version that should be tested by the regression test cases. Such reverse order is used to perform our controlled experiment. Still more studies are needed in order to investigate the differences between the results of the controlled experiments and actual practice.

The APFD metric used for representing the effectiveness of test case prioritization has some limitations. It does not account for the severity of faults and test case execution cost. Since `Defects4J` provides many single-fault program versions instead of one multiple-faults program version, we do not need to concern the severity of each fault. To overcome the limitation related to the test case execution cost, we additionally measure the APFD_c values [37] which account for the execution times of individual test cases. The results show that the average difference between the APFD and APFD_c values for each test case ordering is almost negligible (i.e., 3.169e-04). This is because the execution times of test cases in a test suite are almost equivalent for all the subject test suites. As a result, we only use the APFD metric for representing the results.

To allow reproducibility of the results presented in this paper, all the prioritization results and the implementation of the mutation-based test case prioritization techniques are available from our web page at `http://se.kaist.ac.kr/donghwan/downloads`.

6. RELATED WORK

Since the diversity-aware mutation adequacy criterion (i.e., the d-criterion) has been recently proposed by Shin et al. [12], there is no previous study for the diversity-aware mutation adequacy in test case prioritization. However, the d-criterion is experimentally evaluated in test suite selection, compared to the traditional kill-only mutation adequacy criterion (i.e., the k-criterion). The results on 45 real faults in `Defects4J` show that the d-criterion increases the fault detection effectiveness of adequate test suites in comparison with the k-criterion, whereas the d-criterion requires more test cases to be adequate than the k-criterion.

In test case prioritization, the traditional mutation adequacy criterion is already investigated by Rothermel et al. [8]. They investigate the effectiveness of several greedy prioritization techniques using branches, statements, and mutants, respectively. The branch and statement techniques prioritizes test cases according to the number of branches and statements covered by each test case, respectively. They find that there is no single best technique. However, on average across the programs, the mutant-based technique performs most effectively. Later, Elbaum et al. [17] extend the empirical study of Rothermel et al. by including function-level coarser granularity techniques in comparison with the statement-level fine granularity techniques. The empirical results on eight

DOI: 10.1002/stvr

C programs listed in Siemens benchmarks [38] show that the coarser granularity decreases the effectiveness of test case prioritization in general.

For the total greedy approach and the additional greedy approach in test case prioritization, Li et al. [23] report that the additional approach significantly outperforms the total approach. They also study meta-heuristic algorithms for test case prioritization, whereas the prioritization effectiveness difference between the performance of meta-heuristic and that of additional greedy is not significant. Zhang et al. [18] also focus on the total and additional approaches. They develop a unified approach with the total and additional at two extreme instances. The unified model yields a spectrum of genetic approaches ranging between the total and additional approaches depending on a control parameter. The empirical results on four Java programs show that selecting a proper parameter increases the prioritization effectiveness compared to the simple total and additional approaches. However, the additional approach is almost effective as the parametrized approach in all programs.

In multi-objective test case prioritization, Epitropakis et al. [28] present an empirical study of the effectiveness of multi-objective test case prioritization. They mainly investigate two different multi-objective evolutionary algorithms, NSGA-II and Two Archive Evolutionary Algorithm (TAEA) [39], for the objectives including statement coverage and fault detection history. The results show that the multi-objective prioritization techniques are superior to greedy techniques that target each of the objectives of the multi-objective technique.

Perhaps the work that is the closest to ours is that of Lou et al. [20], which studies mutation-based prioritization within software evolution. The study concerns two prioritization schemes; one based on the number of mutants killed and one based on the distribution of the killed mutants. Their results show that ordering tests by the number of mutants killed performs best. This approach is similar to our greedy one. While there are similarities between our and Lou et al. studies, our is based on real faults (while theirs is based on mutant-faults) and we consider the distinguish method with multiple heuristics, while they do not.

Regarding diversity-based test prioritization, there are several studies working mainly in a black-box manner. Henard et al. [10] suggest diversity-aware metric based on the concept of Combinatorial Interaction Testing. This method performs test prioritization by ordering tests according to the dissimilarity of the combinations of the test input parameters. Feldt et al. [11] suggest using a compression utilities to support test prioritization. The techniques measures the dissimilarity distance of test suites using the concept of Normalized Compression Distance. More recently, Hennard et al. [3] compare these techniques with other coverage-based test prioritization and find that they are of similar power despite that they do not use any dynamic information from the tested systems.

7. CONCLUSION

In this paper, we investigate test case prioritization guided by mutants. Based on the recently defined diversity-aware mutation adequacy criterion, we present the new prioritization objective that distinguishing all mutants as early as possible. We evaluate the effectiveness of mutation-based prioritization techniques by considering the new objective as well as the existing objective, which is killing all mutants as early as possible. Based on these two objectives, we investigate greedy, hybrid,

DOI: 10.1002/stvr

and multi-objective prioritization strategies using 352 real faults and 553,477 developer-written test cases.

Our results show that the mutation-based prioritization is more than or equally effective than the random prioritization and the coverage-based prioritization for at least 95.5% and 83.3% of the faults, respectively. Among the greedy, hybrid, and multi-objective optimization strategies using the kill-only and diversity-aware mutation adequacy criteria, there is no single superior test case prioritization technique. Interestingly, while there is no superiority between the kill-only mutation and the diversity-aware mutation adequacy criteria, their combined use improves the effectiveness of the prioritization. For the multi-objective optimization, the effectiveness of orderings in Pareto fronts does not have steady correlation with the prioritization objectives. The prioritization execution time for the multi-objective techniques requires approximately 37 minutes, while the greedy and hybrid techniques require less than 8 seconds.

There are several implications from the results. For example, both distinguishing and killing mutants as early as possible are more effective than covering statements as early as possible. To detect faults as early as possible, the mutation-based prioritization is more beneficial than the coverage-based one. Interestingly, a greedy hybrid approach, which considers the two mutation-based objectives at the same time, is more effective than considering one objective at a time. The same combination of the two mutation-based objectives can be done by using a multi-objective optimization but unfortunately it does not provide any important benefits and requires far more execution time to prioritize test cases than the hybrid.

More research is needed in order to develop a single test case prioritization technique that is clearly superior to killing and distinguishing mutants. To support such attempts, we provide a graphical model called Mutant Distinguishment Graph (MDG), which visualizes how mutants are killed and distinguished by a test suite with respect to the fault-detecting test cases. This way we demonstrate the reasons why simply killing mutants as early as possible is not always effective.

ACKNOWLEDGEMENT

This research was partially supported by the Institute for Information & communications Technology Promotion (IITP) grant funded by the Korea government (MSIP) (No.2015-0-00250, (SW Star Lab) Software R&D for Model-based Analysis and Verification of Higher-order Large Complex System), and also by Next-Generation Information Computing Development Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Science, ICT (2017M3C4A7068179). Mike Papadakis is supported by the Luxembourg National Research Fund (FNR) (C17/IS/11686509/CODEMATES).

REFERENCES

1. Yoo S, Harman M. Regression testing minimization, selection and prioritization: a survey. *Software Testing, Verification and Reliability* 2012; **22**(2):67–120.
2. Catal C, Mishra D. Test case prioritization: a systematic mapping study. *Software Quality Journal* 2013; **21**(3):445–478.
3. Henard C, Papadakis M, Harman M, Jia Y, Le Traon Y. Comparing white-box and black-box test prioritization. *Proceedings of the 38th International Conference on Software Engineering*, ACM, 2016; 523–534.
4. Chekam TT, Papadakis M, Traon YL, Harman M. Empirical study on mutation, statement and branch coverage fault revelation that avoids the unreliable clean program assumption. *Proceedings of the International Conference on Software Engineering (ICSE)*, 2017.

DOI: 10.1002/stvr

5. Ahmed I, Jensen C, Groce A, McKenney PE. Applying mutation analysis on kernel test suites: an experience report. *Software Testing, Verification and Validation Workshops (ICSTW), 2017 IEEE International Conference on*, IEEE, 2017; 110–115.
6. Papadakis M, Shin D, Yoo S, Bae D. Are mutation scores correlated with real fault detection?: a large scale empirical study on the relationship between mutants and real faults. *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, 2018; 537–548, doi:10.1145/3180155.3180183. URL http://doi.acm.org/10.1145/3180155.3180183.
7. Just R, Jalali D, Inozemtseva L, Ernst MD, Holmes R, Fraser G. Are mutants a valid substitute for real faults in software testing? *Proceedings of the International Symposium on Foundations of Software Engineering (FSE)*, 2014; 654–665.
8. Rothermel G, Untch RH, Chu C, Harrold MJ. Prioritizing test cases for regression testing. *IEEE Transactions on Software Engineering (TSE)* 2001; **27**(10):929–948.
9. Andrews JH, Briand LC, Labiche Y, Namin AS. Using mutation analysis for assessing and comparing testing coverage criteria. *IEEE Transactions on Software Engineering (TSE)* 2006; **32**(8):608–624.
10. Henard C, Papadakis M, Perrouin G, Klein J, Heymans P, Le Traon Y. Bypassing the combinatorial explosion: Using similarity to generate and prioritize t-wise test configurations for software product lines. *IEEE Transactions on Software Engineering* 2014; **40**(7):650–670.
11. Feldt R, Poulding S, Clark D, Yoo S. Test set diameter: Quantifying the diversity of sets of test cases. *Software Testing, Verification and Validation (ICST), 2016 IEEE International Conference on*, IEEE, 2016; 223–233.
12. Shin D, Yoo S, Bae DH. Diversity-aware mutation adequacy criterion for improving fault detection capability. *Proceedings of the International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, 2016; 122–131.
13. Shin D, Yoo S, Bae DH. A theoretical and empirical study of diversity-aware mutation adequacy criterion. *IEEE Transactions on Software Engineering* 2017; .
14. Just R, Jalali D, Ernst MD. Defects4J: A database of existing faults to enable controlled testing studies for Java programs. *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, 2014; 437–440.
15. DeMillo RA, Lipton RJ, Sayward FG. Hints on test data selection: Help for the practicing programmer. *Computer* 1978; **11**(4):34–41.
16. Shin D, Bae DH. A theoretical framework for understanding mutation-based testing methods. *Proceedings of the International Conference on Software Testing, Verification and Validation (ICST)*, 2016; 299–308.
17. Elbaum S, Malishevsky AG, Rothermel G. Test case prioritization: A family of empirical studies. *IEEE transactions on software engineering* 2002; **28**(2):159–182.
18. Zhang L, Hao D, Zhang L, Rothermel G, Mei H. Bridging the gap between the total and additional test-case prioritization strategies. *Proceedings of the 2013 International Conference on Software Engineering*, IEEE Press, 2013; 192–201.
19. Do H, Rothermel G. On the use of mutation faults in empirical assessments of test case prioritization techniques. *IEEE Transactions on Software Engineering* 2006; **32**(9):733–752.
20. Lou Y, Hao D, Zhang L. Mutation-based test-case prioritization in software evolution. *Proceedings of the 26th International Symposium onSoftware Reliability Engineering (ISSRE)*, IEEE, 2015; 46–57.
21. Lu Y, Lou Y, Cheng S, Zhang L, Hao D, Zhou Y, Zhang L. How does regression test prioritization perform in real-world software evolution? *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*, 2016; 535–546, doi:10.1145/2884781.2884874. URL http://doi.acm.org/10.1145/2884781.2884874.
22. Deb K, Pratap A, Agarwal S, Meyarivan T. A fast and elitist multiobjective genetic algorithm: Nsga-ii. *IEEE transactions on evolutionary computation* 2002; **6**(2):182–197.
23. Li Z, Harman M, Hierons RM. Search algorithms for regression test case prioritization. *IEEE Transactions on software Engineering* 2007; **33**(4).
24. Memon AM, Gao Z, Nguyen BN, Dhanda S, Nickell E, Siemborski R, Micco J. Taming google-scale continuous testing. *39th IEEE/ACM International Conference on Software Engineering: Software Engineering in Practice Track, ICSE-SEIP 2017, Buenos Aires, Argentina, May 20-28, 2017*, 2017; 233–242, doi:10.1109/ICSE-SEIP.2017.16. URL https://doi.org/10.1109/ICSE-SEIP.2017.16.
25. Just R. The Major mutation framework: Efficient and scalable mutation analysis for Java. *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, 2014; 433–436.
26. Kintis M, Papadakis M, Papadopoulos A, Valvis E, Malevris N, Le Traon Y. How effective mutation testing tools are? an empirical analysis of java mutation testing tools with manual analysis and real faults. *Technical Report, Luxembourg University: http://pages.cs.aueb.gr/ kintism/papers/preprint1.pdf*, 2017.
27. Goldberg DE, Holland JH. Genetic algorithms and machine learning. *Machine learning* 1988; **3**(2):95–99.

28. Epitropakis MG, Yoo S, Harman M, Burke EK. Empirical evaluation of pareto efficient multi-objective regression test case prioritisation. *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ACM, 2015; 234–245.
29. Kotelyanskii A, Kapfhammer GM. Parameter tuning for search-based test-data generation revisited: Support for previous results. *Quality Software (QSIC), 2014 14th International Conference on*, IEEE, 2014; 79–84.
30. Arcuri A, Briand L. A hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering. *Software Testing, Verification and Reliability (STVR)* 2014; **24**(3):219–250.
31. Vargha A, Delaney HD. A critique and improvement of the cl common language effect size statistics of mcgraw and wong. *Journal of Educational and Behavioral Statistics* 2000; **25**(2):101–132.
32. Chekam TT, Papadakis M, Bissyandé T, Traon YL, Sen K. Selecting fault revealing mutants. *arXiv preprint* 2018; (arXiv:1803.07901).
33. Chekam TT, Papadakis M, Bissyandé T, Traon YL. Predicting the fault revelation utility of mutants. *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*, ACM, 2018; 408–409.
34. Kurtz B, Ammann P, Delamaro ME, Offutt J, Deng L. Mutant subsumption graphs. *Proceedings of the International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, 2014; 176–185.
35. Eiben ÁE, Hinterding R, Michalewicz Z. Parameter control in evolutionary algorithms. *IEEE Transactions on evolutionary computation* 1999; **3**(2):124–141.
36. Coles H, Laurent T, Henard C, Papadakis M, Ventresque A. Pit: a practical mutation testing tool for java. *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ACM, 2016; 449–452.
37. Elbaum S, Malishevsky A, Rothermel G. Incorporating varying test costs and fault severities into test case prioritization. *Proceedings of the 23rd International Conference on Software Engineering*, IEEE Computer Society, 2001; 329–338.
38. Hutchins M, Foster H, Goradia T, Ostrand T. Experiments of the effectiveness of dataflow-and controlflow-based test adequacy criteria. *Proceedings of the 16th international conference on Software engineering*, IEEE Computer Society Press, 1994; 191–200.
39. Deb K, Agrawal S, Pratap A, Meyarivan T. A fast elitist non-dominated sorting genetic algorithm for multi-objective optimization: Nsga-ii. *International Conference on Parallel Problem Solving From Nature*, Springer, 2000; 849–858.

DOI: 10.1002/stvr

Table I. Summary of mutation-based test case prioritization techniques

Mnemonic	Objective	Tiebreaker	Description
GRK	GReedy, Kill	random	iteratively select a test case that maximizes the number of additionally killed mutants
GRD	GReedy, Distinguish	random	iteratively select a test case that maximizes the number of additionally distinguished mutants
HYB-w	HYBrid, $w$eight	random	iteratively select a test case that maximizes the weighted sum of the number of additionally killed mutants and additionally distinguished mutants
MOK	Multi-Objective, kill & distinguish	kill	optimize an ordering of test cases to both kills and distinguishes mutants as early as possible, and select one of the Pareto optimal orderings that kills mutants as early as possible
MOD	Multi-Objective, kill & distinguish	distinguish	optimize an ordering of test cases to both kills and distinguishes mutants as early as possible, and select one of the Pareto optimal orderings that distinguishes mutants as early as possible
RND	RaNDom	random	randomized ordering
SCV	Statement CoVerage	random	iteratively select a test case that maximizes the number of additionally covered statements

Table II. Summary for subject faults, test cases, and mutants. The number of fault-detecting test cases (dT), all mutants (aM), killed mutants by the test cases (kM), and distinguished mutants by the test cases (dM) are also presented.

Program	Faults	Test Cases (sum)	dT (sum)	aM (sum)	kM (sum)	dM (sum)
Chart	25	5,806	91	21,611	8,614	1,462
Closure	133	443,596	347	109,727	82,676	34,685
Lang	65	11,409	124	81,524	63,551	5,467
Math	106	20,661	172	101,978	73,931	14,591
Time	27	72,005	76	19,996	13,665	3,838
Total	352	553,477	810	334,836	242,437	60,043

DOI: 10.1002/stvr

Table III. Comparison of prioritization effectiveness with random. For every pair (A, B), there are the number of cases where the effectiveness of A is statistically superior (+), equal (=), or inferior (-) to B, based on the Mann-Whitney U-tests with $\alpha = 0.001$. The average $\hat{A}_{AB}$ value is given to represent the effect size.

Pair		Superiority			Effect size	Pair		Superiority			Effect size
A	B	+	=	-	$\hat{A}_{AB}$	A	B	+	=	-	$\hat{A}_{AB}$
GRD	RND	294	50	**8**	0.8269	HYB-060	RND	298	37	17	0.8480
HYB-005	RND	298	38	16	0.8447	HYB-065	RND	298	36	18	0.8482
HYB-010	RND	**304**	34	14	0.8519	HYB-070	RND	299	35	18	0.8474
HYB-015	RND	300	39	13	**0.8520**	HYB-075	RND	301	33	18	0.8475
HYB-020	RND	298	40	14	0.8498	HYB-080	RND	299	36	17	0.8473
HYB-025	RND	296	41	15	0.8476	HYB-085	RND	300	34	18	0.8484
HYB-030	RND	296	41	15	0.8468	HYB-090	RND	297	38	17	0.8488
HYB-035	RND	296	39	17	0.8475	HYB-095	RND	300	35	17	0.8478
HYB-040	RND	295	39	18	0.8476	GRK	RND	275	56	21	0.8188
HYB-045	RND	297	38	17	0.8471	MOK	RND	296	43	13	0.8396
HYB-050	RND	298	36	18	0.8479	MOD	RND	294	47	11	0.8401
HYB-055	RND	298	36	18	0.8476	Average	RND	296.8	39.2	16.0	0.8452

Table IV. Comparison of prioritization effectiveness with coverage-based prioritization. For every pair (A, B), there are the number of cases where the effectiveness of A is statistically superior (+), equal (=), or inferior (-) to B, based on the Mann-Whitney U-tests with $\alpha = 0.001$. The average $\hat{A}_{AB}$ value is given to represent the effect size.

Pair		Superiority			Effect size	Pair		Superiority			Effect size
A	B	+	=	-	$\hat{A}_{AB}$	A	B	+	=	-	$\hat{A}_{AB}$
GRD	SCV	238	36	78	0.7012	HYB-060	SCV	266	30	56	0.7813
HYB-005	SCV	257	29	66	0.7514	HYB-065	SCV	267	29	56	0.7820
HYB-010	SCV	267	25	60	0.7687	HYB-070	SCV	267	30	**55**	0.7822
HYB-015	SCV	**268**	28	56	0.7791	HYB-075	SCV	267	29	56	0.7820
HYB-020	SCV	264	32	56	0.7756	HYB-080	SCV	266	31	**55**	0.7822
HYB-025	SCV	262	31	59	0.7736	HYB-085	SCV	266	30	56	0.7836
HYB-030	SCV	263	31	58	0.7738	HYB-090	SCV	265	32	**55**	**0.7838**
HYB-035	SCV	263	30	59	0.7739	HYB-095	SCV	266	30	56	0.7830
HYB-040	SCV	264	32	56	0.7772	GRK	SCV	236	58	58	0.7501
HYB-045	SCV	267	27	58	0.7793	MOK	SCV	253	39	60	0.7444
HYB-050	SCV	267	28	57	0.7806	MOD	SCV	250	38	64	0.7367
HYB-055	SCV	267	29	56	0.7809	Average	SCV	261.6	31.9	58.5	0.7699

DOI: 10.1002/stvr

Table V. Comparison of prioritization effectiveness of all mutation-based techniques. For every pair (A, B), there are the number of cases where the effectiveness of A is statistically superior (+), equal (=), or inferior (-) to B based on the Mann-Whitney U-tests with $\alpha = 0.001$. The average $\hat{A}_{AB}$ value is given to represent the effect size.

Pair		Superiority			Effect size	Pair		Superiority			Effect size
A	B	+	=	-	$\hat{A}_{AB}$	A	B	+	=	-	$\hat{A}_{AB}$
HYB-010	GRD	218	74	60	0.6780	HYB-090	HYB-050	81	226	45	0.5415
HYB-050	GRD	212	62	78	0.6633	GRK	HYB-050	83	188	81	0.5078
HYB-090	GRD	215	55	82	0.6637	MOK	HYB-050	40	170	142	0.3913
GRK	GRD	214	50	88	0.6441	MOD	HYB-050	75	113	164	0.4029
MOK	GRD	122	116	114	0.5049	GRK	HYB-090	47	225	80	0.4562
MOD	GRD	138	127	87	0.5409	MOK	HYB-090	38	163	151	0.3830
HYB-050	HYB-010	102	169	81	0.5285	MOD	HYB-090	79	100	173	0.3951
HYB-090	HYB-010	109	163	80	0.5411	MOK	GRK	71	137	144	0.4274
GRK	HYB-010	108	147	97	0.5200	MOD	GRK	99	84	169	0.4230
MOK	HYB-010	62	139	151	0.4003	MOD	MOK	47	259	46	0.5077
MOD	HYB-010	70	144	138	0.4277						

Table VI. Execution time for each prioritization technique. The multi-objective techniques require the most execution time, which is approximately 37 minutes. The greedy and hybrid techniques require less than 8 seconds.

Technique	Time (ms)
RND	21.2
SCV	150.3
GRK	2253.9
HYB-050	7245.2
GRD	7651.7
MOK (or MOD)	2198981.8

DOI: 10.1002/stvr