Using Controlled Numbers of Real Faults and Mutants to Empirically Evaluate Coverage-Based Test Case Prioritization

David Paterson
University of Sheffield

Gregory M. Kapfhammer
Allegheny College

Gordon Fraser
University of Passau

Phil McMinn
University of Sheffield

ABSTRACT

Used to establish confidence in the correctness of evolving software, regression testing is an important, yet costly, task. Test case prioritization enables the rapid detection of faults during regression testing by reordering the test suite so that effective tests are run as early as is possible. However, a distinct lack of information about the regression faults found in complex real-world software forced prior experimental studies of these methods to use artificial faults called mutants. Using the DEFECTS4J database of real faults, this paper presents the results of experiments evaluating the effectiveness of four representative test prioritization techniques. Since this paper's results show that prioritization is susceptible to high amounts of variance when only one fault is present, our experiments also control the number of real faults and mutants in the program subject to regression testing. Our overall findings are that, in comparison to mutants, real faults are harder for reordered test suites to quickly detect, suggesting that mutants are not a surrogate for real faults.

ACM Reference Format:
David Paterson, Gregory M. Kapfhammer, Gordon Fraser, and Phil McMinn. 2018. Using Controlled Numbers of Real Faults and Mutants to Empirically Evaluate Coverage-Based Test Case Prioritization. In *AST'18: IEEE/ACM 13th International Workshop on Automation of Software Test , May 28–29, 2018, Gothenburg, Sweden.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3194733.3194735

1 INTRODUCTION

As shown by a recent study involving software engineering professionals, 37% of software patches are incorrect, of which, 75% either do not address the issue or introduce a regression [5]. Even though it is important to run tests to find regressions, large tests suites are often too costly to run on a regular basis. For instance, a recent build of Apache Geode required the tests to run for 14 hours [24].

In order to lessen the cost of effectively testing for regressions, a family of optimization techniques have been created [32], including test case prioritization, which aims to detect faults early by re-ordering the test cases according to a chosen heuristic. However, previous experimental evaluations of test case prioritization methods have relied heavily on the use of artificial faults, using either seeded faults, deliberately introduced by a human (e.g., [25]), or mutants, introduced programmatically by a tool (e.g., [9]).

We contend that, while artificial faults may support the empirical comparison of various prioritization techniques, it is important to show that, if prioritizers are effective for artificial faults, then they will also be effective for the real faults found in industry. To this end, this paper asks **RQ1: How does the effectiveness of test case prioritization compare between a single real fault and a single mutant?**, answering it by using five DEFECTS4J subjects [14] to study the detection of 200 faults by four prioritization techniques.

This paper shows that, when the program under test contains one real fault or mutant, the effectiveness of prioritization varies significantly. Since this is a validity threat, this paper also asks **RQ2: How does the effectiveness of test case prioritization compare between single faults and multiple faults?** It answers this question by creating DEFECTS4J program versions that contain 1, 5, and 10 defects, thus comparing prioritization's effectiveness at detecting a controlled number of real faults and mutants.

This paper's experiments find that the type of fault in a program has a significant influence on the effectiveness of a reordered test suite, as measured by the well-known average percentage of faults detected (*APFD*) metric [18]. This means that, for single faults, on average up to 659 extra tests need to be executed to detect a real fault compared to a mutant. The experiments also reveal that the number of faults in a program impacts the effectiveness of prioritization. For real faults, prioritization is similarly effective for single and multiple faults. However, for mutants, prioritization becomes more effective as their number increases. These results suggest that mutants are not a surrogate for real faults, demonstrating the need for subsequent evaluations of test prioritization to use both.

Given the evident need for further work in this area, it is important to note from the outset that it is our intention to promote robust empirical studies of test prioritization. To ensure that external researchers can verify our results, we have released the data collected from this paper's experiments and the source code that recreates its plots and tables [2]. Moreover, to facilitate the reproduction and/or extension of this paper's results, we used DEFECTS4J [14] and KANONIZO [3], two tools that are available in open-source repositories.

2 BACKGROUND AND MOTIVATION

Test Case Prioritization. Test prioritization aims to reduce the cost of finding regressions in software by reordering a test suite so that faults are revealed as early as is possible [18]. Given a program version V, a test suite $T = \langle t_1, \ldots, t_n \rangle$, and the set of permutations $P(T)$ of test suite T, prioritization aims to find $T' \in P(T)$ that maximizes the *APFD* of the reordered test suite [26]. Since *APFD* cannot

AST'18, May 2018, Gothenburg, Sweden

ACM ISBN 978-1-4503-5743-2/18/05…$15.00
https://doi.org/10.1145/3194733.3194735

Table 1: Test case outcomes for three program versions.

		t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}
Program Version	Fixed Version (V_1)	✓	✗	✓	✓	✓	✓	✓	✓	✓	✓
	Single-Fault Version (V_2)	✓	✗	✓	✓	✓	✓	✗	✓	✓	✓
	Multiple-Fault Version (V_3)	✓	✗	✓	✗	✓	✗	✓	✓	✓	✓

be calculated without knowing where faults exist in a system, prioritizers have to use surrogates to estimate *APFD*, including greedy approaches (e.g., [26]) that use test case coverage information to place tests within a prioritized test suite. Alternatively, search-based approaches to test case prioritization, such as genetic algorithms (e.g., [7, 20]), use a "fitness function" to assess the impact of small changes, keeping the modifications that improve fitness. Likewise, a random search repeatedly produces random orderings, keeping the best ordering according to the same fitness function. Finally, many evaluations of test prioritizers use, as an experimental control, a test suite resulting from a random shuffling [20, 26].

Using Faults to Evaluate Test Prioritization. Since the evaluation of test prioritizers requires programs with test suites and known faults, and because large software repositories containing examples of real faults have not been readily available until recently, many prior studies of test prioritization have relied on artificial faults (e.g., [25]). Using a mutation testing tool is a common way to generate artificial faults. Mutation analysis makes many small syntactic changes to a program's source that are checked by its test suite. These syntactic changes, called mutants, are designed to be small in size, while still representing meaningful behavioral changes. The fault-detection capability of a test suite is assessed by checking how well it detects the small changes created by mutation.

Just et al. recently leveraged the Defects4J database of real faults to determine that mutant detection is positively correlated with fault detection [15], concluding that mutants are often a valid substitute for real faults in the evaluation of certain software testing methods. Yet, it is critical to note that the experiments conducted by Just et al. did not specifically reveal if mutants can serve as a surrogate for real faults in the context of evaluating test case prioritization techniques—which is the focus of this paper.

Evaluating Test Case Prioritization Techniques. To evaluate the effectiveness of prioritization, there is a need for a metric that characterizes the quality of a test suite's ordering. A well-established metric for evaluating test case prioritization techniques is the average percentage of faults detected, which represents the percentage of faults that have been detected after the execution of certain numbers of test cases [10]. Given a test suite $T = \langle t_1, \ldots, t_n \rangle$, a set of detectable faults $\Phi = \{\phi_1, \ldots, \phi_m\}$, a program version V containing a total of m faults, and the function $TF(\phi_j)$ returning the number of tests in T that must be executed before detecting ϕ_j, Equation 1 defines the higher-is-better *APFD* metric.

$$APFD = 1 - \frac{\sum_{j=1}^{m} TF(\phi_j)}{nm} + \frac{1}{2n} \qquad (1)$$

Table 1 illustrates both why test prioritization may be needed and how it can improve the *APFD* of a test suite. In this example, there are three versions of the same program, namely a "fixed" version (V_1) that does not contain a fault and two "buggy" versions, which contain one (V_2) and two faults (V_3). A *trigger test* $t_i \in T$ detects a fault in a program V when it passes on the fixed version and fails on the buggy version. It is important to note that trigger tests are not the same as failing tests — in Table 1 t_2 fails across *all* versions of the program, including the fixed version.

Table 2: Characteristics of the chosen Defects4J projects.

Identifier	Name	# Versions	KLOC	Test KLOC	Tests
Chart	JFreeChart	26	96	50	2,205
Closure	ClosureCompiler	133	90	83	7,927
Lang	ApacheLang	65	22	6	2,245
Math	ApacheMath	106	85	19	3,602
Time	JodaTime	27	28	53	4,130

We consider two orderings of the test suite T: a random ordering of the test suite $T' = \langle t_1, t_8, t_4, t_5, t_7, t_9, t_2, t_{10}, t_6, t_3 \rangle$ and an ordering produced by prioritization, $T'' = \langle t_4, t_7, t_5, t_6, t_1, t_{10}, t_8, t_9, t_3, t_2 \rangle$. In this example, the *APFD* for V_2 is $1 - \frac{5}{10} + \frac{1}{20} = 0.55$ for T' and $1 - \frac{2}{10} + \frac{1}{20} = 0.85$ for T'', while the *APFD* for V_3 is $1 - \frac{12}{20} + \frac{1}{20} = 0.45$ for T' and $1 - \frac{5}{20} + \frac{1}{20} = 0.8$ for T''. This example illustrates how prioritization can enhance the fault detection rate of test suites, since for both versions the *APFD* score is highest for the ordering produced by a prioritization technique. This example also highlights the risks associated with not systematically controlling the number of faults during an experiment: despite T'' finding *one* of the two faults in V_3 faster, it still results in a lower *APFD* than for V_2.

It is important to discuss the alternatives to *APFD*, including cost-cognizant *APFD* (*APFD$_C$*) [22] and Normalized *APFD* (*NAPFD*) [23]. Since our experiments neither used test case cost (*APFD$_C$*) nor removed tests (*NAPFD*), these alternatives do not differ from *APFD*.

Impact of Multiple Faults. The *APFD* of any prioritized test suite is directly connected to three factors: the position of trigger tests (i.e., $\sum_{j=1}^{m} TF(\phi_j)$), the number of test cases (i.e., n) and the number of faults (i.e., m). The primary focus of test prioritization research has been to maximize the *APFD* score, which involves minimizing $\sum_{j=1}^{m} TF(\phi_j)$. However, since the *APFD* equation is clearly designed to handle multiple faults, as evidenced by the inclusion of m, we must also consider the impact of different numbers of faults.

When only one fault is present in a program, the *APFD* is equivalent to the location of the first trigger test. As trigger tests can occur anywhere within the test suite, a random approximation of test prioritization would be expected to see an average *APFD* score of 0.5 as the number of trials increases, with a high variance since trigger tests can exist at any suite location. Yet, as the number of faults increases, the probability of an individual trial producing an *APFD* score close to 0.5 increases, assuming an even distribution of trigger tests. This results in a much lower *APFD* variance for multiple faults than for single faults. Since *APFD* score variance is likely for test prioritization, there is a risk associated with only using single faults, thereby motivating this paper's experiment design.

3 METHODOLOGY

Sampling of Real Faults. To obtain real faults, we used Defects4J, a repository of five open-source Java projects with 357 real faults that were mined from version control repositories. As shown in Table 2, these projects range between 22,000 and 96,000 lines of code and 2,205 and 7,927 tests. Under development for between five and 12 years, Defects4J's projects are mature. Along with containing a "pre-fix" and "post-fix" version for each fault, Defects4J has a developer-written JUnit test suite with at least one trigger test.

Both of the research questions that this paper answers require the use of the real faults from the Defects4J repository. For **RQ1**, we created and used a tool to randomly sample 25 single, real faults from each of the five projects, while for **RQ2** we created a modified version of Defects4J that supports the combination of real faults.

Sampling of Mutants. Along with real faults, answering this paper's two research questions required the use of mutants. Since our experiments aim to evaluate whether prioritization techniques are effective on both real faults and mutants, we make no assumptions about the location of the mutants, since in practice the location of faults is not known [32]. To allow us to evaluate how effective test prioritizers are with mutants, and to eliminate the risk of using equivalent mutants [17], which are mutants that result in no *actual* difference in program execution, we ensured that each mutant is killed by at least one of the developer-written test cases.

To investigate the hypothesis that real faults and mutants are roughly equivalent for evaluating prioritization techniques, we used the Major mutation tool [16] to create and analyze large numbers of mutants. To produce the mutant versions of the Defects4J projects, we applied Major to all classes within the project and iteratively picked random mutants for mutation analysis, keeping those that were killed by at least one test until sufficient numbers had been selected (i.e., 1, 5, or 10). For the Closure project, there were numerous insurmountable out-of-memory issues, ultimately resulting in the generation of no mutants for this project.

Test Case Prioritization. To prioritize test suites, we developed the Kanonizo tool [3], which provides four techniques that reorder tests based on coverage information. Notably, test cases should, by nature, be independent of each other, with no test affecting the behavior of any other [33]. In particular, the version of JUnit used by Defects4J's projects does not specify the execution order of the tests [1], implying no default ordering in which to run the tests. Moreover, many prior studies of test case prioritization considered a "no prioritization" baseline (e.g., [25]). Thus, we used a randomly shuffled ordering of each project's test suite as a baseline.

We compared baseline orderings against the following four prioritizers, which we implemented into Kanonizo: *Total Statement* reorders test cases based on the number of lines covered by each, such that those that cover the most lines of code appear first. *Additional Statement* prioritizes test cases based on the number of unique lines of code covered, such that those that cover the most lines not covered by other tests appear first. Kanonizo further implements a *Genetic Algorithm (GA)* that uses a fitness function to evolve and evaluate candidate solutions (i.e., test orderings). The fitness function follows Li et al.'s approach, maximizing the average percentage of lines covered (*APLC*), a metric formulated like the *APFD*, but based on lines of code covered rather than faults [20].

Finally, we implemented a *Random Search* method that repeatedly re-orders the tests at random, returning the best ordering as determined by the same fitness function used for the GA (thereby differing from the random baseline method that returns the first random ordering produced). Although many other methods exist (e.g., [7, 31]), we chose these since prior work regularly studied them (e.g. [10, 20, 25]). To give an appropriate search budget to the GA and Random Search, we used the maximum runtime of the deterministic algorithms (i.e., Additional Statement and Total Statement), thereby ensuring that their search budgets are sensitive to the complexities of each project and thus enabling a fair comparison across the different test case prioritization techniques.

Statistical Analysis. Any algorithm that makes random choices during its execution may produce variable results across multiple runs. We therefore repeated the baseline, GA, and Random Search 30 times. As mentioned in the previous paragraph, the budget for each search-based technique is determined by the runtime of the Additional Statement and Total Statement prioritizers. Although they are deterministic, we repeated Additional Statement and Total Statement 10 times to account for any potential execution time variations due to load fluctuations on the computer cluster.

In addition to accounting for fluctuations caused by the random choices in algorithms, it is also important to consider the likelihood that any results observed can have occurred as a result of chance, rather than as a result of an improved algorithm. The Mann-Whitney U-test determines the probability that two samples have originated from the same distribution, without assuming sample normality or requiring identical sample sizes. If the probability p reported by the U-test is less than 0.01, then it is *statistically unlikely* that the differences could have occurred by chance.

Like the Mann-Whitney U-test, the Vargha-Delaney effect size, $\hat{A}$, takes two samples, but instead measures the probability that a prioritizer r_1 yields higher values (i.e., *APFD* scores) than another prioritizer r_2 [30]. Effect size is often used to gauge "practical significance" [11], since statistical significance has a tendency to yield smaller p-values as the size of the data sets increase. The Vargha-Delaney effect size is independent of sample size. For example, an effect size of 0.7 indicates that a prioritizer r_1 achieves higher *APFD* scores than r_2 70% of the time. Since an $\hat{A}$ value is equivalent to $1-\hat{A}$ if r_1 and r_2 are reversed, the order in which the two prioritizers are supplied to the $\hat{A}$ computation is important. With this in mind, Vargha and Delaney further quantified and categorized an effect size as **N**one ($|\hat{A} - 0.5| < 0.06$), **S**mall ($0.06 \leq |\hat{A} - 0.5| < 0.14$), **M**edium ($0.14 \leq |\hat{A} - 0.5| < 0.21$), or **L**arge ($|\hat{A} - 0.5| > 0.21$).

In Section 4's plots, the results of both statistical tests are displayed at the top of each plot in the following way. For **RQ1**, we represent the Mann-Whitney U-test results with a ✓ or a ✗, and the $\hat{A}$ with "N/S/M/L": these values represent whether the *APFD* distribution for mutants was significantly higher than the *APFD* distribution for real faults. For **RQ2**, we use the same notation to indicate whether the *APFD* for 5 or 10 faults, respectively, was significantly higher than the *APFD* score for a single fault.

Threats to Validity. Before discussing this paper's validity threats, it is important to note that all experimentation artifacts are publicly available [2, 3, 14], thus allowing external researchers to confirm that we correctly ran the experiments and analyzed the results.

While the subjects in Defects4J vary in terms of their total lines of code, total number of tests, and number of years under development, we cannot guarantee that the results observed for these subjects will generalize to, for instance, programs and tests with different characteristics. Thus, we intend to replicate this paper's experiments with additional subjects that are more complex and contain long-running tests that detect real faults with varying degrees of severity. Finally, since the sampling of Defects4J's defects may lead to variance in results when experiments are repeated, in future work we will run experiments with more sampled defects.

This paper's experiments consider four test case prioritization methods, which we judge to be representative of previous work due to the frequency with which others have studied them (e.g., [25]).

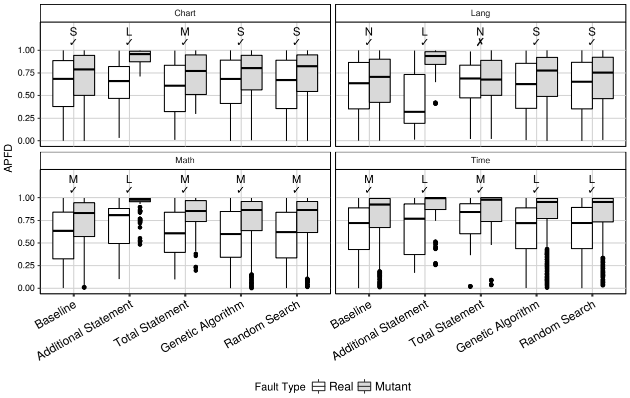


Figure 1: ***APFD*** **scores for programs containing one real fault (white) and one mutant fault (gray). See Section 3 for an explanation of the symbols (e.g., the ✓ and ✗) in the legend.**

However, many other prioritizers exist, such as those employing alternative genetic algorithms (e.g., [20]), clustering algorithms (e.g., [6]), or other static code-based approaches (e.g., [29]). Thus, the results of this paper's experiments may have been different if other algorithms had been used. Since the Kanonizo tool is extensible, we will control this validity threat by implementing new algorithms and experimentally evaluating them in future work.

This paper empirically compares prioritization methods that use statement coverage. Yet, it is also possible to prioritize tests by analyzing how well they cover program branches [25] or kill mutants [9]. Importantly, Kanonizo supports prioritization with reports about branch coverage or mutant killing. Thus, although testers often use statement coverage [32], we will control this threat by later studying methods that use other test quality measurements.

Like many past studies of test prioritization (e.g., [31]), this paper assumes that tests are independent, meaning that their reordering will not influence test outcomes [18]. While this assumption does not always hold true in practice [33], Defects4J's programs come with JUnit test suites that we found to be independent.

The final validity threat for this paper's results is defects in the tools used to run the experiments. We mitigated this threat by implementing JUnit tests when developing Kanonizo and by hand checking the empirical results to ensure correctness. It is also possible that problems with Defects4J compromised the results. Yet, this well-tested database has been used in prior studies (e.g., [14]) without any concerns. We also handled defects in the statistical analysis and graphing routines by extensive testing and checking.

4 EXPERIMENTAL RESULTS

RQ1: How does the effectiveness of test case prioritization compare between a single real fault and a single mutant? Figure 1 presents boxplots of the *APFD* scores observed from the experiments involving single faults and single mutants. The figure reveals a clear difference between the effectiveness of test case prioritization with single real faults and mutants, with significant differences in all but one case. Effect sizes range from small to large. Large effect sizes are obtained for the Additional Statement prioritizer for every project. For only one case (i.e., the Lang project and Total Statement) is the median *APFD* for real faults below that for mutants, yet investigation of the means (not shown in the figure) indicated the reverse, and overall there is no significant difference. Furthermore, Table 3 gives the number of extra tests required to detect a real fault compared to a single mutant. For all projects, at least 10% more test cases must be executed in order to detect real faults, a phenomenon that we investigate further in Section 5.

Table 3: Mean number of test cases required to detect each type of fault, and differences relative to test suite size.

Project	Real	Mutant	Test Cases	Difference
Chart	703.4	498.5	1826.0	11.2%
Lang	818.9	611.4	1960.8	10.6%
Math	1461.7	815.8	3566.9	18.1%
Time	1341.9	683.4	3929.1	16.8%

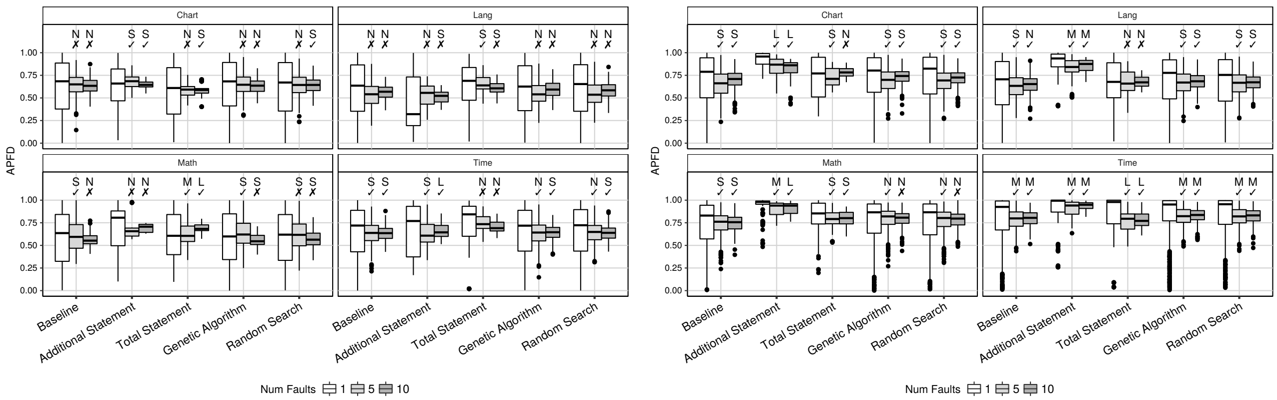
Table 4: Effect sizes for different fault types. Significant values are bold and effect sizes are labelled as given in Section 3.

	Additional Statement		Total Statement		Genetic Algorithm		Random Search	
	Real	Mutant	Real	Mutant	Real	Mutant	Real	Mutant
Chart	(N) 0.46	**(L) 0.75**	**(N) 0.44**	(N) 0.53	(N) 0.51	(N) 0.52	(N) 0.49	(N) 0.53
Lang	**(S) 0.37**	**(L) 0.77**	(N) 0.53	(N) 0.50	(N) 0.50	**(N) 0.55**	(N) 0.51	(N) 0.53
Math	**(S) 0.58**	**(L) 0.80**	(N) 0.53	**(S) 0.57**	(N) 0.50	(N) 0.54	(N) 0.49	**(N) 0.54**
Time	**(N) 0.55**	**(M) 0.66**	**(S) 0.61**	**(S) 0.59**	(N) 0.51	**(N) 0.54**	(N) 0.50	**(N) 0.54**

Overall, these results indicate that the use of mutants may overestimate the possible improvements achievable with test case prioritization compared to the use of real faults. Although this over inflation of effectiveness may set wrong expectations about the possible benefits of using test case prioritization in practice, it is not necessarily an issue prohibiting the use of mutants for test case prioritization experiments. That is, as long as the inflation is consistent across software projects and prioritizers and does not change the decision as to whether test case prioritization achieves a benefit or not, mutants could still be used for comparative studies of test case prioritization. However, our experiments do not suggest that this is the case. We checked whether the different test prioritizers improve over the baseline of a random ordering of tests, and whether this is consistent between mutants and real faults.

Table 4 shows the effect sizes comparing different prioritizers with the baseline; values increasingly over 0.5 indicates the prioritizer is more likely to return a higher *APFD* than the single random ordering, while values decreasingly below 0.5 means that the random ordering is more likely to be better. The table clearly shows that effect sizes are not consistent between the two types of faults. For mutants, all effect sizes are greater than (or equal to) 0.5, and more than half of the cases are statistically significant, suggesting that test prioritization was beneficial. For real faults, however, most cases show no statistical difference, with an effect size generally close to 0.5 (i.e., no practical difference). In fact, there are several cases where the random ordering is (significantly) better than the prioritized tests, with an effect size less than 0.5. In Section 5, we discuss reasons for why prioritization may be better for mutants, also investigating cases where it works well for real faults.

Conclusion: Single real faults vs single mutants. These results show that *APFD* scores are inflated when prioritization is run with one mutant, compared to one real fault. If this inflation were predictable, mutants could still usefully substitute for real faults. But, our results indicate that it is not. When compared to the baseline, the difference in *APFD* scores depends on the test case prioritizer



(a) Real Faults **(b) Mutant Faults**

Figure 2: Distribution of *APFD* scores for different numbers of faults. See Section 3 for an explanation of the legend.

and the project. In Section 5, we further discuss the differences between real faults from Defects4J and mutants, showing some of the syntactic and semantic differences between the two fault types.

RQ2: How does the effectiveness of test case prioritization compare between single faults and multiple faults? Figure 2 shows the boxplots of the *APFD* scores obtained from running the experiments with different numbers of real faults and mutants, respectively. For one real or mutant fault, the variance in *APFD* scores is relatively high, reducing for five faults, and then further reducing for ten. This is to be expected, as discussed in Section 2.

Figure 2a shows that, in general, median *APFD* scores decrease for 5 and 10 faults compared to 1 fault. Yet, these decreases are not always accompanied by statistical significance. Effect sizes tend to be negligible or small, with the exception being Time-Additional Statement and the distributions for 1 and 10 real faults. The distributions for Math-Total Statement and 1–5 and 1–10 faults are also medium or large, but for these project-prioritizer configurations the median *APFD* score increases, rather than decreasing.

Figure 2b plots the distributions of *APFD* scores for mutants. Again, the median *APFD* scores decrease. In contrast to real faults, however, we found that the reduction is significant for 35 out of 40 project-prioritizer pairings. These results tend to be accompanied by non-negligible effect sizes. Additional Statement is the most successful test prioritizer for mutants, achieving the highest median *APFD* scores across all projects. For this prioritizer, increasing the number of mutants always results in a decrease of the median *APFD*, a result that is always significant and accompanied by a medium or large effect size. These results for Additional Statement and mutants are contrasted with those evident when this prioritizer handles real faults: Figure 2a shows that those scores are often not significantly different and the effect sizes are normally negligible or small.

While the evidence for decreases in *APFD* values as the number of faults increases is stronger for mutants compared to real faults, the use of mutants over real faults is not necessarily prohibited if test case prioritization techniques see consistent decreases in the *APFD* scores when their test suite orderings are compared to the baseline random ordering. That is, if the decreases are consistent across software projects and test prioritization methods, then mutants could still be used as surrogates for real faults. As such, we checked whether the different prioritizers improve over the random baseline.

Table 5 shows the effect sizes of multiple real faults and mutants for each prioritizer, compared to the random ordering baseline for the same number and type of fault. From Table 5a, for real faults, it is clear that there are only a few cases where test prioritization is practically better for 5 and 10 faults than for single faults. In most cases the $\hat{A}$ values are similar and do not significantly increase as more faults are introduced. However, in Table 5b, for mutants, all but one of the $\hat{A}$ values increase when moving from single to multiple faults. Therefore, as for RQ1, we conclude that there is an inconsistent difference between real faults and mutants as more of a fault type is introduced. That is, one fault type cannot be reliably substituted for the other. In comparison to the results for single faults, this contrast is greater when comparing test case prioritization with multiple faults to the random baseline.

Conclusion: Single vs multiple faults. From the evidence presented in this empirical evaluation, we conclude that the use of multiple faults can aid evaluations of test case prioritization techniques by reducing variance caused by randomness. Median *APFD* scores tend to decrease for both fault types as the number of faults increases. However, the effects are inconsistent when we compare prioritization with real faults and mutants against the random baseline. Prioritization with mutants is more likely to significantly outperform the baseline with a non-negligible effect size, while prioritization with real faults tends to show little difference. The contrast observed between real faults and mutants and the baseline is more marked than for single faults, as observed in RQ1. Considering the answers to both research questions, we conclude that future studies of test prioritization's effectiveness should, whenever possible, use real faults in conjunction with, or as a substitute for, mutants.

5 DISCUSSION

Real Faults vs. Mutants in Test Prioritization. While Section 4 reveals that there are clear differences between real faults and mutants when it comes to how effective test prioritization is, these results do not develop an understanding concerning the syntactic and semantic differences between fault types that may be the root cause. For the 125 real faults used in this study, we found that on average 7.2 lines were removed (max. removed lines 49) and 1.98 lines were added (max. added lines 22) to fix a real fault. When generating mutants, the code change always involved a maximum of one new line of code and at most one removed statement.

Table 5: Effect sizes for controlled numbers of faults. Significant values are bold and effect sizes are labelled as in Section 3.

(a) Real Faults

	Additional Statement			Total Statement			GA			Random		
	1	5	10	1	5	10	1	5	10	1	5	10
Chart	(N) 0.46	**(S) 0.60**	(N) 0.52	**(N) 0.44**	**(M) 0.33**	**(M) 0.30**	(N) 0.51	(N) 0.51	(N) 0.49	(N) 0.49	(N) 0.50	(N) 0.51
Lang	**(S) 0.37**	(N) 0.53	**(S) 0.36**	(N) 0.53	**(M) 0.70**	**(M) 0.69**	(N) 0.50	(N) 0.53	**(S) 0.58**	(N) 0.51	**(N) 0.55**	**(S) 0.58**
Math	**(S) 0.58**	**(S) 0.60**	**(M) 0.69**	(N) 0.53	**(S) 0.57**	**(L) 0.72**	(N) 0.50	**(N) 0.55**	(N) 0.53	(N) 0.49	(N) 0.52	(N) 0.51
Time	**(N) 0.55**	(N) 0.50	(S) 0.56	**(S) 0.61**	**(L) 0.75**	**(L) 0.74**	(N) 0.51	(N) 0.51	(N) 0.54	(N) 0.50	(N) 0.51	(N) 0.51

(b) Mutant Faults

	Additional Statement			Total Statement			GA			Random		
	1	5	10	1	5	10	1	5	10	1	5	10
Chart	**(L) 0.75**	**(L) 0.86**	**(L) 0.85**	(N) 0.53	**(M) 0.65**	**(L) 0.76**	(N) 0.52	**(S) 0.57**	**(S) 0.59**	(N) 0.53	**(S) 0.56**	**(N) 0.55**
Lang	**(L) 0.77**	**(L) 0.90**	**(L) 0.96**	(N) 0.50	**(S) 0.60**	**(S) 0.59**	**(N) 0.55**	**(S) 0.58**	**(S) 0.61**	(N) 0.53	**(S) 0.56**	**(S) 0.57**
Math	**(L) 0.80**	**(L) 0.89**	**(L) 0.92**	**(S) 0.57**	**(S) 0.63**	**(M) 0.66**	(N) 0.54	**(S) 0.64**	**(M) 0.66**	**(N) 0.54**	**(S) 0.60**	**(S) 0.62**
Time	**(M) 0.66**	**(L) 0.81**	**(L) 0.94**	**(S) 0.59**	(N) 0.46	**(S) 0.44**	**(N) 0.54**	**(S) 0.58**	**(S) 0.61**	**(N) 0.54**	**(S) 0.56**	**(S) 0.59**

The relatively high volume of code changes that are required to fix a real fault outline the complexity associated with them—there are domain- and context-specific changes required to fix the fault. In contrast, mutants are independent of domain and context. This indicates that real faults require more targeted tests than mutants, since they not only need to execute the line, but also create the correct system state in order to verify that the fault no longer occurs. This phenomenon can also be observed in the relative number of trigger tests, where we observed an average of 3.18 trigger tests for single real faults, compared to 57.38 for single mutants. The implication that mutants are easier to find than real faults suggests the need for future work in creating more realistic mutants.

Where Prioritization was Successful with Real Faults. While the results in Section 4 indicate that test case prioritization was, on the whole, not successful at improving the fault detection rates for test suites finding real faults, we found a small number of exceptions to this where it led to noteworthy increases in fault detection. We inspected the ten versions from the single real fault experiments where the highest *APFD* scores were achieved. In five of these cases, the *APFD* is within .005 of 1, indicating that the prioritizer placed a trigger test prominently within the prioritized suite. In most of these circumstances, the trigger tests executed many more program source code lines than expected, with 9 of the 10 trigger tests executing more than 2% of the *total program lines*.

There is, however, a notable exception to this in Lang v51, for which only 0.35% of the total lines in the version are executed by the trigger test. This example exhibits one of the greatest differences between the Total Statement and Additional Statement prioritizers, which achieved *APFD* scores of 0.68 and 0.97, respectively. This result indicates that, despite the trigger test having a lower overall coverage, it is still possible for it to contribute to a test suite with a high *APFD* score. This is an example of a situation where the Total Statement prioritizer leads to a low-quality test ordering, thus further confirming that the maximization of test coverage is often not correlated with improved fault detection [12, 13].

While it is possible for prioritization techniques to produce test orderings that improve fault detection, in many cases this is, in fact, indicative of poor unit test design. For Math v74, the trigger test covers a total of 1069 lines of code, which may partly be due to it testing a small method that uses a lot of core code during its execution. Yet, for this project, this behavior is likely symptomatic of a test that is going beyond the purpose of unit testing as it checks, for instance, sequences of complex method interactions.

Results With the Genetic Algorithm. One of the more prominent trends in Section 4 was that Random Search and the GA had poor performance for both real faults and mutants, barely outperforming the baseline random orderings in many cases. Given that the GA is initialized with a population of random orderings and only makes changes that result in an increased coverage score for the test suite, it seems unlikely that the baseline's test suites would outperform the GA's orderings. Notably, the fitness function used by the GA was *APLC*, also adopted by Li et al. [20] and designed to maximize line coverage. Yet, Hao et al. conducted a study in which they discovered that, in most cases, the Additional Statement prioritizer results in optimal or near-optimal levels of code coverage [12]. Thus, in its current configuration, even if the GA was given an extended search budget, its final test ordering would likely not be capable of outperforming the ones created by Additional Statement.

Additionally, calculating *APLC* in the GA evaluates the entire test suite even when only two test cases change position. This fitness calculation is both frequently called and expensive, particularly when there are large programs and/or test suites involved. As a result, the GA only makes a small number of changes to the population of test orderings before exhausting its search budget, thereby limiting its exploration of the search space. Future work should investigate potentially more competitive GAs (e.g., [7, 31]).

6 RELATED WORK

This paper's primary purpose is to use a controlled number of real faults and mutants to compare coverage-based test prioritization techniques. Complementing this paper's goal, several prior studies broadly compared fault types within the field of software testing. For instance, Andrews et al. investigated the use of mutants in software testing experiments [4] and Just et al. examined the differences between mutants and real faults [15]. In the context of regression testing, Luo et al. studied the effectiveness of prioritization methods at detecting mutants [21] and Do and Rothermel evaluated test prioritization with both mutants and seeded faults [9]. Finally, there are two papers that, like this one, use Defects4J to empirically evaluate test case prioritization. Yet, Lu et al. only furnish a preliminary study of one Defects4J program [21] and Shin et al. concentrate on a prioritization type different from this paper's methods [27].

Like this paper, Leon and Podgurski also studied test case prioritization using real faults [19]. Yet, they did not empirically compare prioritization with both mutants and real faults. Several other papers studied test case prioritization with real faults in industrial systems. For instance, Srivastava and Thiagarajan assessed whether or not prioritization methods could detect real faults in Microsoft software [28] and Di Nardo et al. studied how well coverage-based techniques detect the real faults in another industrial system [8]. While these aforementioned papers add to the body of knowledge about the effectiveness of test prioritization, they also present studies that are difficult for external researchers to replicate. To better support future studies, this paper provides its supporting code and data [2, 3], leveraging Defects4J, a public database of real faults.

Finally, there is an extensive literature on test prioritization. For instance, Rothermel et al. introduced coverage- and mutation-based test case prioritization methods [25] and Elbaum et al. showed how to reorder tests based on the likelihood of faults occurring in certain code regions [10]. For an overview of other approaches, we refer the reader to Yoo and Harman's survey of regression testing [32].

7 CONCLUSIONS AND FUTURE WORK

Regression testing is important since software engineering professionals often introduce defects when they modify software [5]. By reordering a test suite so that the faults in a program can be detected as early as is possible, test case prioritization may lessen the cost of testing for regressions [32]. However, since many previous studies of prioritization's effectiveness often used seeded faults or mutants there is limited evidence that the method works for real faults. To ascertain whether or not the effectiveness of prioritization differs when programs contain either a real fault or a mutant, this paper uses five Defects4J subjects [14] to study the detection of 125 real faults by four representative test case prioritization techniques.

This paper's results suggest that, in comparison to mutants, the real faults in Defects4J are harder for prioritized test suites to detect. Moreover, mutants lead to *APFD* scores that are inflated in unpredictable ways. Given the usefulness of these results, we will incorporate more of Defects4J's faults and new subjects in future experiments. We will also add more test case prioritizers (e.g., [29]) to Kanonizo. Furthermore, future work should develop new operators for generating mutants that better mirror the behavior of real faults. The combination of this paper's contributions and the suggested future work will yield a comprehensive framework for studying prioritization methods with both real faults and mutants.

REFERENCES

[1] 2017. JUnit test execution order. (2017). Retrieved 02/03/2018 from https://github.com/junit-team/junit4/wiki/Test-execution-order

[2] 2018. Experimental data from this paper's evaluation. (2018). https://www.bitbucket.com/testprioritisation/ast2018_data

[3] 2018. Kanonizo. (2018). https://github.com/kanonizo/kanonizo

[4] J. H. Andrews, L. C. Briand, and Y. Labiche. 2005. Is Mutation an Appropriate Tool for Testing Experiments?. In *Proceedings of the 27th International Conference on Software Engineering.*

[5] Marcel Böhme, Ezekiel O. Soremekun, Sudipta Chattopadhyay, Emamurho Ugherughe, and Andreas Zeller. 2017. Where is the Bug and How is It Fixed? An Experiment with Practitioners. In *Proceedings of the 11th Joint Meeting on Foundations of Software Engineering.*

[6] Ryan Carlson, Hyunsook Do, and Anne Denton. 2011. A Clustering Approach to Improving Test Case Prioritization: An Industrial Case Study. In *Proceedings of the 27th International Conference on Software Maintenance.*

[7] Alexander P. Conrad, Robert S. Roos, and Gregory M. Kapfhammer. 2010. Empirically studying the role of selection operators during search-based test suite prioritization. In *Proceedings of the 12th International Conference on Genetic and Evolutionary Computation.*

[8] Daniel Di Nardo, Nadia Alshahwan, Lionel Briand, and Yvan Labiche. 2015. Coverage-based regression test case selection, minimization and prioritization: a case study on an industrial system. *Journal of Software Testing, Verification and Reliability* 25, 4 (2015).

[9] Hyunsook Do and Gregg Rothermel. 2006. On the Use of Mutation Faults in Empirical Assessments of Test Case Prioritization Techniques. *Transactions on Software Engineering* 32, 9 (2006).

[10] Sebastian Elbaum, Alexey G. Malishevsky, and Gregg Rothermel. 2000. Prioritizing Test Cases for Regression Testing. In *Proceedings of the International Symposium on Software Testing and Analysis.*

[11] S. M. Ellis and H. S. Steyn. 2003. Practical significance (effect sizes) versus or in combination with statistical significance (p-values). *Management Dynamics* 12, 4 (2003).

[12] D. Hao, L. Zhang, L. Zang, Y. Wang, X. Wu, and T. Xie. 2016. To Be Optimal or Not in Test-Case Prioritization. *Transactions on Software Engineering* 42, 5 (2016).

[13] Laura Inozemtseva and Reid Holmes. 2014. Coverage is Not Strongly Correlated with Test Suite Effectiveness. In *Proceedings of the 36th International Conference on Software Engineering.*

[14] René Just, Darioush Jalali, and Michael D. Ernst. 2014. Defects4J: A Database of Existing Faults to Enable Controlled Testing Studies for Java Programs. In *Proceedings of the International Symposium on Software Testing and Analysis.*

[15] René Just, Darioush Jalali, Laura Inozemtseva, Michael D. Ernst, Reid Holmes, and Gordon Fraser. 2014. Are Mutants a Valid Substitute for Real Faults in Software Testing?. In *Proceedings of the 22nd International Symposium on Foundations of Software Engineering.*

[16] René Just, Gregory M. Kapfhammer, and Franz Schweiggert. 2012. Using non-redundant mutation operators and test suite prioritization to achieve efficient and scalable mutation analysis. In *Proceedings of the 23rd International Symposium on Software Reliability Engineering.*

[17] Gregory M. Kapfhammer. 2004. Software testing. In *The Computer Science Handbook.*

[18] Gregory M. Kapfhammer. 2010. Regression testing. In *The Encyclopedia of Software Engineering.*

[19] David Leon and Andy Podgurski. 2003. A Comparison of Coverage-Based and Distribution-Based Techniques for Filtering and Prioritizing Test Cases. In *Proceedings of the 14th International Symposium on Software Reliability Engineering.*

[20] Z. Li, M. Harman, and R. M. Hierons. 2007. Search Algorithms for Regression Test Case Prioritization. *Transactions on Software Engineering* 33, 4 (2007).

[21] Qi Luo, Kevin Moran, and Denys Poshyvanyk. 2016. A Large-scale Empirical Comparison of Static and Dynamic Test Case Prioritization Techniques. In *Proceedings of the 24th International Symposium on Foundations of Software Engineering.*

[22] Alexey G. Malishevsky, Joseph R. Ruthruff, Gregg Rothermel, and Sebastian Elbaum. 2006. *Cost-cognizant test case prioritization.* Technical Report TR-UNL-CSE-2006-0004. Department of Computer Science and Engineering, University of Nebraska, Lincoln, Nebraska, USA.

[23] X. Qu, M. B. Cohen, and K. M. Woolf. 2007. Combinatorial Interaction Regression Testing: A Study of Test Case Generation and Prioritization. In *Proceedings of the 23rd International Conference on Software Maintenance.*

[24] Apache Geode Nightly Test Report. 2018. Apache Geode Nightly Test Report. (2018). https://builds.apache.org/view/E-G/view/Geode/job/Geode-release/lastCompletedBuild/testReport/

[25] G. Rothermel, R. H. Untch, Chengyun Chu, and M. J. Harrold. 1999. Test case prioritization: an empirical study. In *International Conference on Software Maintenance.*

[26] G. Rothermel, R. H. Untch, Chengyun Chu, and M. J. Harrold. 2001. Prioritizing test cases for regression testing. *Transactions on Software Engineering* 27, 10 (2001).

[27] D. Shin, S. Yoo, M. Papadakis, and D.-H. Bae. 2017. Empirical Evaluation of Mutation-based Test Prioritization Techniques. *ArXiv e-prints* (2017). arXiv:cs.SE/1709.04631

[28] Amitabh Srivastava and Jay Thiagarajan. 2002. Effectively Prioritizing Tests in Development Environment. In *Proceedings of the International Symposium on Software Testing and Analysis.*

[29] S. W. Thomas, H. Hemmati, A. E. Hassan, and D. Blostein. 2014. Static test case prioritization using topic models. *Journal of Empirical Software Engineering* 19(1) (2014).

[30] András Vargha and Harold D. Delaney. 2000. A Critique and Improvement of the CL Common Language Effect Size Statistics of McGraw and Wong. *Journal of Education and Behavioral Statistics* 25, 2 (2000).

[31] Kristen R. Walcott, Mary Lou Soffa, Gregory M. Kapfhammer, and Robert S. Roos. 2006. Time-aware Test Suite Prioritization. In *Proceedings of the International Symposium on Software Testing and Analysis.*

[32] S. Yoo and M. Harman. 2012. Regression Testing Minimization, Selection and Prioritization: A Survey. *Journal of Software Testing, Verification and Reliability* 22, 2 (2012).

[33] Sai Zhang, Darioush Jalali, Jochen Wuttke, Kıvanç Muşlu, Wing Lam, Michael D. Ernst, and David Notkin. 2014. Empirically Revisiting the Test Independence Assumption. In *Proceedings of the International Symposium on Software Testing and Analysis.*