2019 IEEE 43rd Annual Computer Software and Applications Conference (COMPSAC)

An Empirical Study on the Spreading of Fault Revealing Test Cases in Prioritized Suites

Wesley N. M. Torres
Software Practices Laboratory
Campina Grande, Brazil
wesley.torres@splab.ufcg.edu.br

Everton L.G. Alves
Software Practices Laboratory
Campina Grande, Brazil
everton@computacao.ufcg.edu.br

Patrícia D. L. Machado
Software Practices Laboratory
Campina Grande, Brazil
patricia@computacao.ufcg.edu.br

***Abstract*—Code edits are very common during software development. Specially for agile development, these edits need constant validation to avoid functionality regression. In this context, regression test suites are often used. However, regression testing can be very costly. Test case prioritization (TCP) techniques try to reduce this burden by reordering the tests of a given suite aiming at fastening the achievement of a certain testing goal. The literature presents a great number of TCP techniques. Most of the work related to prioritization evaluate the performance of TCP techniques by calculating the rate of test cases that fail per fault (the APFD metric). However, other aspects should be considered when evaluating prioritization results. For instance, the ability to reduce the spreading of failing test cases, since a better grouping often provides more information regarding faults. This paper presents an empirical investigation for evaluating the performance of a set of prioritization techniques comparing APFD and spreading results. Our results show that prioritization techniques generate different APFD and spreading results, being *total-statement* prioritization the one with the lowest spreading.**

***Index Terms*—prioritization, test case, metric, evaluation**

I. Introduction

Software testing plays an important role in the software development process. A test suite can provide more confidence to developers that a software comprises its expected behavior. In agile projects, tests are used as a safety net for preventing functionality regression [16]. For instance, Fowler recommends refactoring edits should be combined with regression testing for ensuring behavior preservation [4], [24]. However, studies show that the testing activity can be very costly, accounting for as much as half of a project's budget [13], [27]. In this context, Test Case Prioritization (TCP) is a well-known tool for reducing testing costs.

TCP techniques apply heuristics for proposing a new test case ordering. This new order often intends to fasten fault detection. A good prioritization leads to productivity gains and/or avoids bugs in production phases [31]. The prioritization problem was first defined by Rothermel et al. [25] as follows:

> Given: T, a test suite; PT, the set of permutations of T, and f, a function from PT to real numbers.
> Problem: Find $T' \in PT$ such as $\forall T'' \in PT \bullet T'' \neq T' \rightarrow (f(T') \geq f(T''))$, where PT represents the set of possible orderings of T, and f is a function that calculates the best results when applied to any ordering.

There are a number of TCP techniques (e.g., [6], [15], [16], [27]). Singh et al.'s literature review [29] discusses a total of 106 TCP techniques from eight categories: *coverage-based*, *modification-based*, *fault-based*, *requirement-based*, *history-based*, *genetic-based*, *composite approaches*, and *others*. The TCP techniques differ from each other by its prioritization heuristic and context. For instance, the *total* strategies advocate that test cases that cover more code elements (e.g., statements, branches) should be run first since they are more likely of detecting faults. While *additional* strategies try to vary the covered code elements by the test cases that should run first.

The great majority of the work regarding TCP focuses on evaluating the effectiveness of a giving technique or comparing techniques, by using the APFD (Average Percentage Faults Detected) metric [25]. This metric finds the weighted average of the percentage of faults detected for a given test suite execution. APFD is the most adopted metric for evaluating test case prioritizations. For instance, 17, out of the 20, most cited papers from a quick string-based search ("test case prioritization") at Google Scholar[1] at January 2019, use the APFD metric. Therefore, in this context, a tester may rely on studies that use the APFD when deciding which TCP technique to use. However, by considering only the APFD metric, other important aspects might be neglected (e.g., fault understanding, and fault localization).

In practice, a tester often runs the top test cases of a prioritized suite. The number of tests to run depends on practical constraints (e.g., shortage of time). However, if the top test cases gather only part of the failing test cases, a tester might have a hard time understanding and/or locating the faults. Yoo et al. [32] found that a tester often needs a set of failing test cases for the same type of fault since it helps fault localization. Thus, besides early fault detection, we believe that a good TCP technique should also reduce the spreading of related failing test cases.

Alves et al. [1] propose the *F-Spreading* metric for evaluating prioritized suites. It measures the spreading of test cases that reveal each fault. However, despite its possible applicability, no substantial study has yet investigated how the main prioritization techniques behave when considering *F-*

[1] http://scholar.google.com.br/

978-1-7281-2607-4/19/$31.00 ©2019 IEEE
DOI 10.1109/COMPSAC.2019.00027

Spreading, particularly regarding whether the results obtained are similar or different from the APFD ones.

In this work, we evaluate 12 of the most popular TCP techniques in an empirical study using 8 open source projects. Our goal is to investigate how these techniques behave when observing both the rate of fault detection (APFD) and the spreading of failing test cases. Our study comprises techniques that use different heuristics and different coverage granularity levels (statement, method and branch coverage). Moreover, we present different rankings for the TCP techniques by considering APFD and spreading results.

In summary, this paper presents the following contributions:

- An investigation on how TCP techniques behave considering a novel evaluation metric, the spreading of test cases.
- A ranking for TCP techniques based on APFD results.
- A ranking for TCP techniques based on spreading results.
- A guideline for helping testers to better choose a TCP technique depending on their practical goals.

The remainder of this paper is organized as follows. Section II presents the metrics investigated in this work. Section III presents an example for motivating the relevance of fault-revealing test cases. Section IV describes our empirical study and its results, along with guidelines for selecting a TCP technique based on the metrics investigated. Section V discusses related work regarding TCP metrics and empirical studies. Finally, Section VI presents concluding remarks and pointers to future work.

II. Metrics

Prioritized suites can be evaluated considering different aspects [23]. In this paper, we focus on two metrics: i) the rate of fault detection (APFD), and ii) the spreading of fault-revealing test cases. We base our analysis on the belief that a good prioritization not only detects faults faster but also better groups all failing test cases. The later is based on the assumption that test cases that fail due to a giving fault often provide good information for helping fault understanding and localization [32]. Here, we provide a detailed description of the evaluation metrics used in our study.

A. The Average Percentage of Faults Detected (APFD)

This metric was first proposed by Rothermel at al. [25] and measures how fast a prioritized suite can detect a set of faults. The earlier all faults are detected, higher the APFD. Formally, APFD is calculated according to Equation 1, let T be a test suite containing n test cases, F be a set of m faults revealed by T, *n* the number of test cases, and TF_i be the index of the first test case in ordering T' of T that reveals fault i.

$$APFD = 1 - \frac{(\sum_{i=1}^{m} TF_i)}{nm} + \frac{1}{2n} \quad (1)$$

Suppose a scenario where two faults (*f1* and *f2*) are detected by a suite with 10 test cases {TC1, TC2, TC3, TC4, TC5, TC6, TC7, TC8, TC9, TC10}. *f1* is detected by TC2 and TC8, and *f2* is detected only by TC9. Now, suppose that a tester applied two TCP techniques (T1 e T2) that generated the following prioritized suites S1 = {TC7, TC1, TC6, TC2, TC3, TC4, TC5, TC8, TC9, TC10} and S2 = {TC1, TC2, TC5 TC4, TC6, TC7, TC8, TC3, TC9, TC10}, respectively. By applying Equation 1 to S1 and S2, we have the following APFD values: 0.45 (45%) and 0.6 (60%), respectively. Thus, according to the APFD values, T2 would be the best TCP technique.

B. Spreading

The *F-Spreading* metric was defined by Alves et al. [1]. It measures how spread all fault-revealing test cases are in a prioritized suite. The intention behind *F-Spreading* is to figure out how clustered are all tests that could help a tester to find and located all faults. Equation 2 presents *F-Spreading*, where N is the number of test cases in a prioritized suite; m is the number of test cases that failed; TF is a sequence containing the positions of the test cases that failed; and TF_i is the position of the test case i in the prioritized set. The lower the *F-Spreading* the better.

$$\textit{F-Spreading} = \begin{cases} 0, & \text{if } m \text{ is } < 2 \\ (\sum_{i=2}^{m} TF_i - TF_{i-1}) * \frac{1}{N}, & \text{if } m \text{ is } >= 2 \end{cases} \quad (2)$$

Alves et al. gave the first step on the spreading analysis, however, we noticed that the *F-Spreading* alone might provide misleading information when dealing with groups of faults. For instance, suppose a prioritization that places in sequence positions failing test cases that detect different faults. In this scenario, as the suite would have a low spreading (*F-Spreading*), a tester might have the false impression that all failures refer to the same fault. Therefore, we propose a variation of the *F-Spreading* metric, which we call *M-Spreading* or *Mean-Spreading*. The M-Spreading calculates the average of the *F-Spreading* for each individual fault. Equation 3 formalizes *M-Spreading*, where N is the number of test cases; m is the number of faults; *F-spreading*$_i$ is the F-Spreading for i^o fault.

$$\textit{M-Spreading} = \frac{\sum_{i=1}^{m} \textit{F-Spreading}_i}{m} \quad (3)$$

For the example above mentioned, the *M-Spreading* values for S1 and S2 are 0.25 and 0.35, respectively. Therefore, the TCP technique that proposes the prioritization with the lowest spreading is T1, a conclusion that is opposed to the APFD.

III. Motivational Example

In this session, we present a motivating example that demonstrates how *spreading* can be an interesting way for evaluating prioritization results. For this example, we use a fault from the Lang project, collected from Defect4j [12], a database of real faults to enable controlled experiments in software engineering research. The code in in Figure 1 was adapted for the sake of simplicity. Figure 1 presents the methods `genHash` and `createNumber` that generate hash codes from key that can be represented in several numerical bases. `createNumber` calls `createInteger` (line 20), however, this method does

```
1  public static String genHash(String value, String
       algorithm){
2            Number key = NumberUtils.createNumber(value);
3            try{
4                 MessageDigest md = MessageDigest.
       getInstance(algorithm);
5                 md.update(key.toString().getBytes());
6                 return new BigInteger(1, md.digest()).
       toString(16);
7            }catch (NoSuchAlgorithmException e){
8                 return null;
9            }
10      }
```

(a) Method that generates hash codes.

```
1  public static Number createNumber(final String str)
       throws NumberFormatException{
2            if(str == null) return null;
3            if(str.length() == 0) throw new
       NumberFormatException("A blank string is not a
       valid number");
4            final String[] hex_prefixes = {"0x", "0X", "-0
       x", "-0X", "#", "-#"};
5            int pfxLen = 0;
6            ...
7            if (pfxLen > 0){
8                 final int hexDigits = str.length() -
       pfxLen;
9                 if(hexDigits > 16){
10                     return createBigInteger(str);
11                }
12                if(hexDigits > 8){
13                     return createLong(str);
14                }
15                return createInteger(str);
16           }
17           return createBigDecimal(str);
18     }
```

(b) Method that creates a valid number from a string.

```
1  public static Integer createInteger(final String str){
2    if(str == null){
3       return null
4    }
5    return Integer.decode(str);
6  }
```

(c) Method that decodes an integer value from a string. It contains a fault since it does not check whether the given string represents a valid integer.

Fig. 1. Code extract from Lang project that contains a fault.

not check whether the input value is within the range of valid numbers for the specific base (fault).

Suppose the project has a suite with 300 test cases, from which only two (Figure 2) reveal the fault (Figure 1). The asserts in line 4 (Figure 2(a)) and line 8 (Figure 2(b)) provide inputs that overflow the integer limits. Aiming at speeding fault detection, a tester decides to perform test case prioritization. The chosen TCP strategy places the two failing test cases in positions 3 and 200, respectively. For this scenario, the APFD value is 0.99. Now, suppose that, due to time constraints, a tester decides to run just the top 100 test cases of the prioritized suite. In this scenario, the fault would still be detected, but the single test case that could be used for understanding/locating it is `testGenHash`.

Although a high APFD was achieved, the first failing test cases might not be very helpful to understand/locate the fault. Studies have discussed that test cases that directly

```
1  @Test
2  public void testGenHash(){
3    assertEquals("bcabf6a179d32acfdd669938266c6b92", Auth.
       genHash("0x8000", "MD5"));
4    assertEquals("9ab98b1f1e647c65ebedf670363bc813", Auth.
       genHash("0x80000000", "MD5"));
5    assertEquals("d2f7d27c4d72f6d5b74d14b859e3c77d", Auth.
       genHash("0xFFFFFFF", "MD5"));
6    assertEquals("c4ca4238a0b923828dcc509a6f75849b", Auth.
       genHash("0x000001", "MD5"));
7  }
```

(a) Creating a Number Decimal from another numeric base

```
1  @Test
2  public void testCreateInteger(){
3    assertEquals(Integer.valueOf(0x8000), NumberUtils.
       createInteger("0x8000"));
4    assertEquals(Integer.valueOf(0x80000), NumberUtils.
       createInteger("0x80000"));
5    assertEquals(Integer.valueOf(0x800000), NumberUtils.
       createInteger("0x800000"));
6    assertEquals(Integer.valueOf(0x8000000), NumberUtils.
       createInteger("0x8000000"));
7    assertEquals(Integer.valueOf(0x7FFFFFF), NumberUtils.
       createInteger("0x7FFFFFF"));
8    assertEquals(Integer.valueOf(0x80000000L), NumberUtils
       .createInteger("0x80000000"));
9    assertEquals(Integer.valueOf(0xFFFFFFFFL), NumberUtils
       .createInteger("0xFFFFFFFF"));
10         }
```

(b) Generation hash code to user

Fig. 2. Test of creation of Integer

cover the faulty methods are more likely to both detect and help testers to understand and locate the faults [2]. In this case, `testGenHash` does not cover the faulty method (`createInteger`) directly, it ends up detecting the fault due to an indirect sequence of actions.

The second failing test case `testCreateInteger` was created for testing acceptable values for the `createInteger` method. Thus, it seems to provide more insightful information regard the fault. We believe that, by analyzing both failing test cases, a tester would have a better understanding and would be more likely to locate the fault. However, in the context of our example, since the chosen TCP technique placed the failing test cases in distant positions, the second test (`testCreateInteger`), which is the most informative one, might never be considered.

When we consider a spreading measure such as *M-spreading*, we observe that the prioritization result was not very effective (*M-spreading* = 0.66). This limitation could not be seen if the APFD was considered alone. Therefore, we believe that a good prioritization result should generate high APFD values combined with low spreading of failing test cases per fault (*M-spreading*). In this work, we investigate how typical TCP techniques behave regarding APFD and spreading.

IV. Empirical Study

In this section, we present the design of an empirical study to evaluate a set of prioritization techniques regarding results obtained by APFD and *M-Spreading* metrics.

A. Research Questions

The following questions guide our investigation:

- **RQ1:** How do TCP techniques behave according to APFD?
- **RQ2:** Which TCP technique produces the best results regarding fastening fault detection?
- **RQ3:** How do TCP techniques behaving according to *M-Spreading*?
- **RQ4:** Which TCP technique produces the best results regarding grouping fault revealing test cases?

B. Study Design, Objects, Faults and Variables

Our study replicates the design and reuses the implementation from Li et al.'s work [16][2]. Their work investigates how TCP techniques deal with fault detection during software and test evolution. We use 8 Java projects from Github[3] as objects of the study. These projects have also be used in other testing works [34]. From their repositories, we collect a set of stable versions for each project. Each version includes a JUnit test suite. Each pair of versions comprises a distance of at least 30 commits. For each version, we create different fault groups. Each fault group comprises five randomly injected mutant faults and each group can be detected by at least one test. All mutants are created by using the PIT tool [4]. Mutant faults as such have been widely used in other testing experiments and are found to be representative of real faults [3], [11], [33].

Table I summarizes the artifacts used in our study, where *Ver* is the number of versions, *MinS* and *MaxS* are the minimum and maximum number of lines of code (KLOC), *MinT* and *MaxT* are the minimum and maximum number of tests, and *Ft* the maximum number of faults present in all versions.

Proj	Ver	MinS	MaxS	minT	MatT	Ft
jasmine-maven	5	1.640	4.348	7	118	136
java-apns	8	1.362	3.839	15	87	82
jopt-simple	4	6.636	8.569	394	657	335
la4j	9	8.064	12.555	172	625	1013
scribe-java	8	2.497	5.957	38	99	112
vraptor-core	5	31.176	32.997	985	1.124	1463
assertj-core	8	55.443	67.282	4.055	5.269	1711
metrics-core	6	11.447	12.546	270	318	721

TABLE I
STUDY ARTIFACTS.

1) Independent Variables: Our study focuses on some of the most well-known TCP techniques [16]. Those are the independent variables in our empirical study. We prioritize the project's test suite according to four strategies: *total*, *additional*, *search-based*, and *adaptive random*. Here we describe the main idea for each strategy:

- **total**: This strategy reorders the test cases based on their descending code coverage rates;
- **additional**: This strategy builds a prioritized suite incrementally, where the next chosen test case is the one that covers most code elements that were not covered yet by the previously selected ones;
- **adaptive random (art)**: This strategy first selects a test case with the highest coverage. The remaining tests are reordered based on a function f that finds the longest minimum distance of a test compared to the previously selected ones [16].
- **search-based**: This strategy takes all permutations as candidate solutions and uses a genetic algorithm to guide the searching process. Then, it chooses the best permutation based on heuristics (e.g., the average percentage of statement coverage).

All mentioned strategies use code coverage as ordering factor. Thus, in the context of our study, we apply them considering three different coverage granularity levels: *method*, *statement*, and *branch*. Therefore, our study deals with twelve different TCP techniques: *total-method, total-statement, total-branch, additional-method, additional-statement, additional-branch, art-method, art-statement, art-branch, search-based-method, search-based-statement and search-based-branch*.

To guarantee statistical validity, and minimize the impact of random aspects of the TCP techniques, we ran each technique 1000 times. The results discussed in the following sections consider the average those executions.

2) Methodology and Dependent Variables: In order to evaluate the behavior of the TCP techniques, we calculate both the **APFD** and **M-Spreading** values for each fault group (1 fault group = 5 injected mutants). Figure 3 provides an overview of our study. For each Java project, a different number of versions are considered (Table I). For each version, we generate several faulty versions containing five faults (a mutant group) that are detected by the subject's test suite. Then, we run all twelve TCP techniques and collect both the APFD and *M-Spreading* metrics, for each fault group.

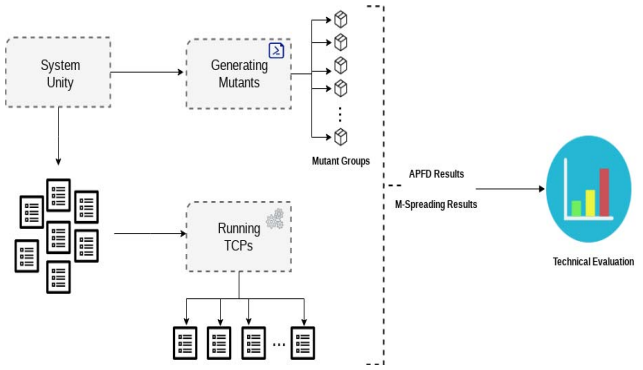

Fig. 3. Setup of our study.

C. Results and Analysis

In this section, we present the results of our experiment and discuss them. A detailed version of our results is provided in our website[5].

[2]http://utdallas.edu/~yxl131230/icse16support.html
[3]https://github.com
[4]http://pitest.org/

[5]https://sites.google.com/view/evaluating-tcp-spreading

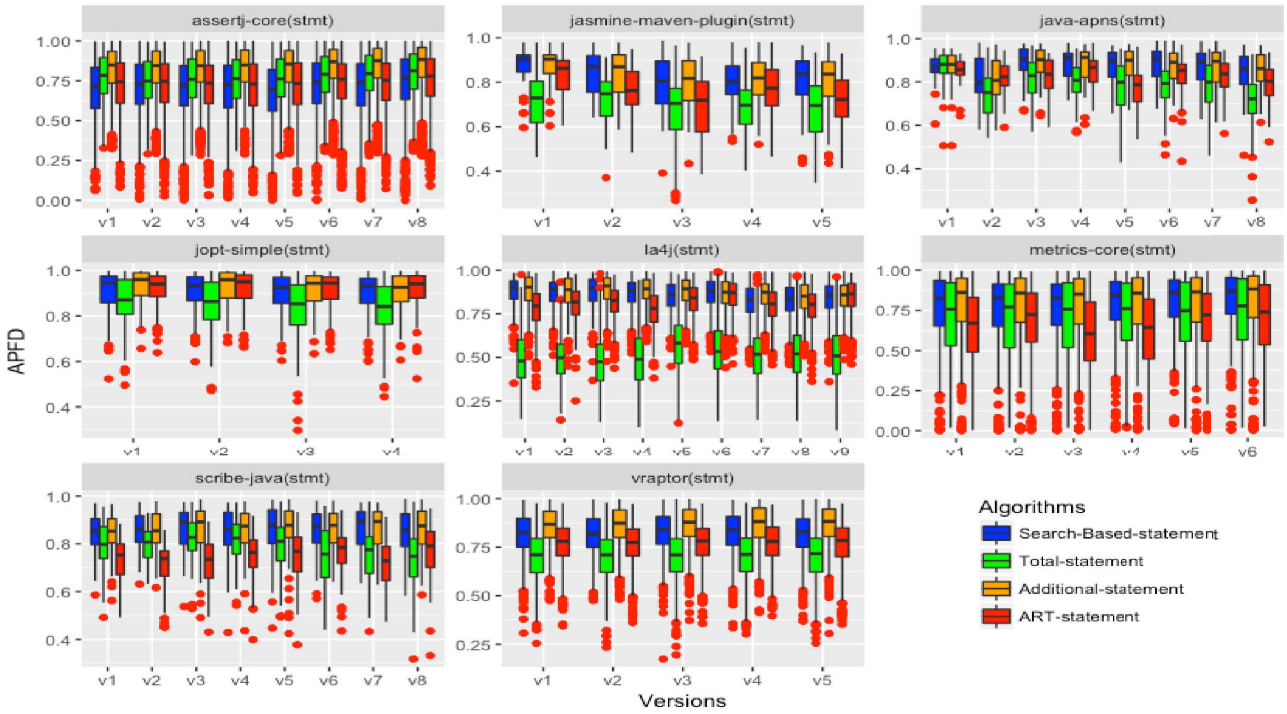

Fig. 4. *APFD* distribution per project.

1) RQ1: how do TCP techniques behave according to APFD?: For the sake of simplicity, Figure 4 presents the distribution of APFD values per fault groups considering only the TCP techniques that deal with statement coverage (red - *art-statement*, green - *total-statement*, orange - *additional-statement*, and blue - *search-based-statement*). Similar results were achieved when considering other coverage granularity levels. The full data can be found in our website.

In a visual analysis, we can see that, in most cases, the distribution boxes overlap, which may imply similar APFD distributions. This similarity is even more evident for some projects (e.g., AssertJ-Core), while in other cases, some techniques present results apart from the others (e.g., La4J). For instance, we can see that the *additional-statement* and *search-based-statement* techniques, in some cases, has the median above the others (Jasmine-Maven-Plugin project - v_5 version). We manually investigated the systems trying to reasoning those cases. Since the injected faults are similar for all systems, we understand that other aspects may have influenced the techniques' behavior (e.g., structural characteristics of the system and/or tests). For instance, in a coupled system, even small edits can impact several code elements. Thus, a TCP technique that favors test cases that covered different code elements may generate better APFD results. This show how volatile APFD results can be when evaluating the quality of TCP techniques.

To have a statistical understanding of the technique's behavior, we first run a normality test. Figure 5 shows the QQPlot graph. As we can see, the collected data does not follow a normal distribution. Thus, a non-parametric test is needed. To evaluate the variance of each group, we performed a *Levene* test [20]. This test verifies whether the groups have similar variance (null hypothesis). With a p-value $< 2.2e^{-16}$, we can state that we worked with variances that are not similar. To better understand how the techniques behave, we then applied *Welch's ANOVA* test [20], which shows whether there is a significant difference in at least one combination of the evaluated techniques. The *Welch's ANOVA Test* results found a significant difference between the distributions in at least one combination (p-value $< 2.2e^{-16}$, confidence of 95%). Thus, we can state that the TCP techniques do not have similar behaviors considering the APFD results. Since the techniques are not statistically similar, we can now rank them.

2) RQ2: Which TCP technique produces the best results regarding fastening fault detection?: Figure 6 shows the confidence intervals (95% of confidence) of the APFD results for each TCP technique, by project. To collect this data, we performed 10000 replications of each prioritization execution and we considered the APFD median for each fault group. This method is known as *Bootstrap* [28].

Since the TCP techniques are statistically not similar, we evaluated them pair-to-pair, and clustered them by type, using the *Tukey's(HSD) Post-Hoc* [16] test. By using this test, we were able to rank the techniques based on their APFD results (Table II). The second column presents the detailed Tukey HSD grouping results (each letter (e.g., "a") refers to a group

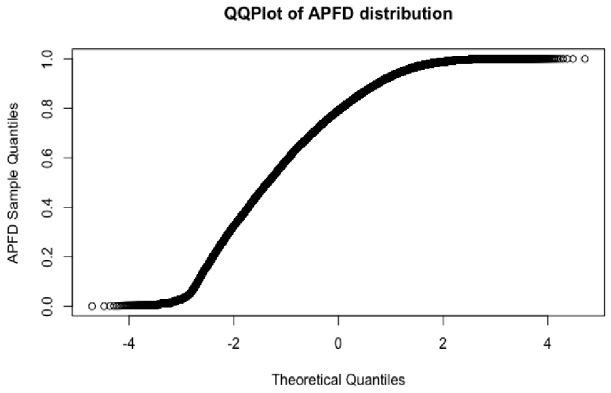

Fig. 5. QQPlot *APFD*

which is significantly different from each other (e.g., "j") comparing all possible pairs of means). We used this result to build a ranking of TCP techniques according to their APFD results (Table III), where $A >$ B means that technique A presented APFD results that are statistically better than B, and $A = B$ means no significant differences were found between A and B. As we can see, by considering only APFD results, the *additional* strategy seems to be more effective for fastening fault detection, followed by the *search-based*, *art*, and the worst results were observed by the *total* techniques.

Strategies	Distribution Groups
additional-branch	a
additional-method	a
additional-statement	b
search-branch	e
search-method	f
search-statement	g
art-branch	c
art-method	d
art-statement	d
total-branch	h
total-method	i
total-statement	j

TABLE II
Welch's AND *TukeyHSD post-hoc* TEST RESULTS FOR APFD

Additional-statement = Additional-branch >
Additional-method > Search-Based-statement >
Search-Based-branch > Search-Based-method >
ART-branch > ART-method =
ART-statement > Total-branch >
Total-method = Total-statement

TABLE III
RANKING OF TCP TECHNIQUES BY APFD.

3) RQ3: How do TCP techniques behave according to M-Spreading?: As discussed in Section III, fault localization costs and time tend to increase when a tester/developer does not have enough information. Yoo et al. [32] found that it is important to have a set of failing test cases for the same type of fault since it helps fault localization. Up until now, we only evaluated a prioritization by considering early fault detection (APFD). From now on, we are interested in investigating a different point of view, by observing the spreading of test cases that reveal faults (the *M-Spreading* metric).

Figure 8 presents the found distribution of *M-Spreading* values considering each project and their versions. It is important to highlight that, while high APFD values may imply better results, with *M-Spreading*, low values are desired. A low *M-Spreading* means that a technique better grouped the fault revealing test cases for each fault. By analyzing Figure 8, we can observe that *total-statement* presented its median below all the other techniques, which infers a tendency to propose better-grouped results.

To determine whether the TCP techniques have similar statistical behavior, considering *M-Spreading*, we first performed a normality test, with a confidence level of 95% (α = 0.05). Again, our results are not normal (Figure 7). Then, we ran the *Levene* test to check whether the TCP techniques had similar variances. With a p-value $<$0.05, we rejected this hypothesis. Then, we ran the *Welch's ANOVA* test that enabled us to state that the techniques do not have similar behavior, with 95% confidence. Therefore, we can now rank them

4) RQ4: Which TCP technique produces the best results regarding grouping fault revealing test cases?: Again, to verify if this difference is significant among the techniques, we used the Bootstrap technique. With 10000 replications of each prioritization execution, we checked the median of the *M-Spreading* values. Figure 9 presents the confidence intervals, with 95% confidence. As we can see, at least one technique (*total-statement*) presented a significant difference, as there is no overlapping with other confidence intervals. This information was further reinforced by *Welch's ANOVA test*, showing that there is a significant difference among the distributions. This results already goes against the conclusions from the APFD analysis, where the *total-statement* TCPs ranked the worst.

Finally, to rank the TCP techniques according to their *M-Spreading* results, as performed in Section IV-C2, we conducted a *Tukey's(HSD) Post-Hoc*. Table IV presents the results. As we can see, our results show that *total* techniques better group related failing test cases, which goes against the APFD conclusions where the *total* techniques presented the worst results. Therefore, by trusting only on APFD results, a tester could choose a TCP technique that would not help her to understand and locate the faults (worst *M-Spreading* results). However, the *additional* strategy seems to present a better balance by combining best APFDs and second best *M-Spreading* results, followed by the *search-based* strategy.

In a side investigation, we also created rankings for the TCP techniques per project. The results in this analysis were, in general, similar to the general one. The techniques with better APFD results were the ones with the worst *M-Spreading*. Details about this analysis can be found in our website[6].

[6]https://sites.google.com/view/evaluating-tcp-spreading

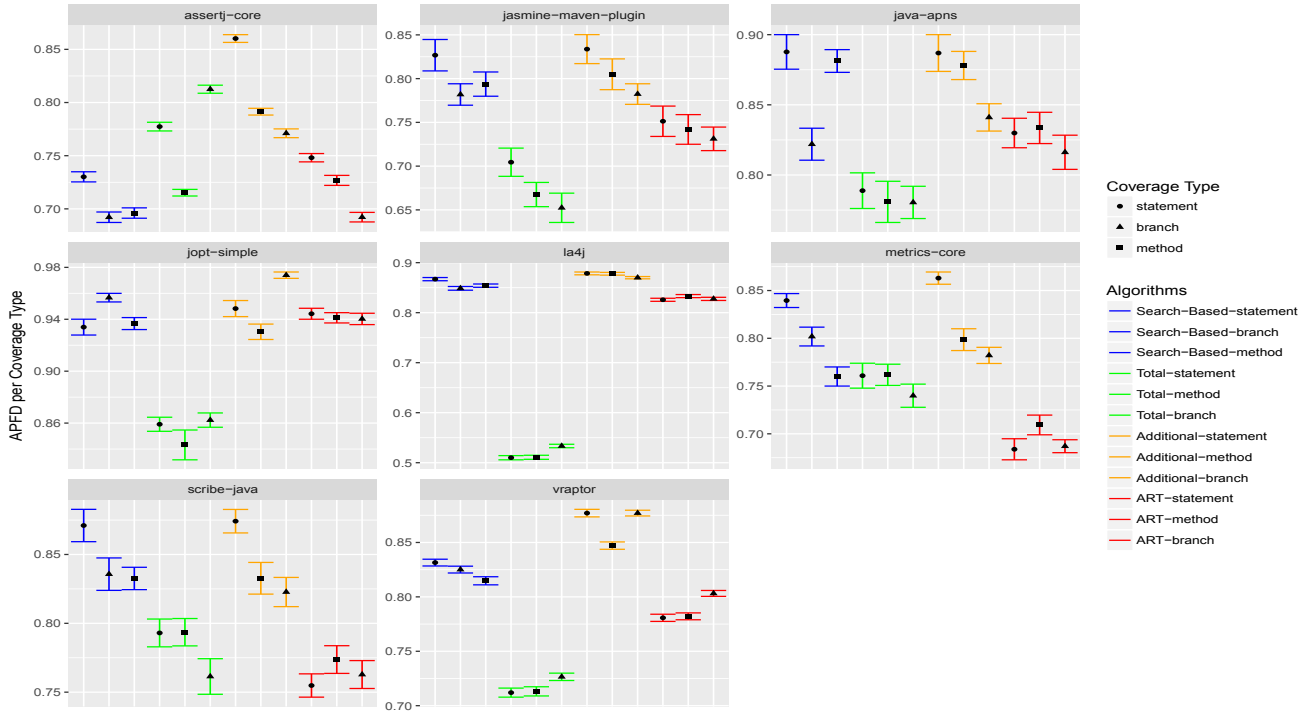

Fig. 6. Confidence intervals for the *APFD* values.

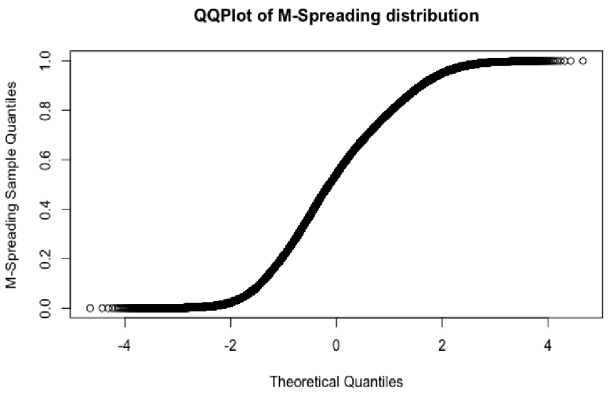

Fig. 7. QQPlot *M-Spreading*

Technique	Distribution Groups
additional-branch	a
additional-method	b
additional-statement	a
search-branch	c
search-method	d
search-statement	e
art-branch	f
art-method	g
art-statement	c
total-branch	h
total-method	i
total-statement	h

TABLE IV
Welch's AND *TukeyHSD post-hoc* RESULTS FOR *M-Spreading*.

Total-statement > Total-branch = Total-method > Additional-branch > Additional-statement = Search-Based-statement > Search-Based-method = ART-branch > Search-Based-branch > ART-statement > Additional-method = ART-method

TABLE V
RANKING OF TCP TECHNIQUES BY M-SPREADING.

D. Guidelines on Selecting a TCP Technique

The results from our empirical study enable us to set a few guidelines for helping testers on selecting appropriated prioritization technique. Although there is no consensus on a technique that can always provide high APFD and low spreading, we can state the following:

- When a tester's goal is to fasten fault detection, we recommend choosing a TCP technique that performs the *additional* strategy, since they often generate better APFD results.
- When a tester's goal is to favor fault localization, we recommend using a TCP technique that runs the *total* strategy. Since they better group fault revealing test cases. We believe a prioritization in this sense would, in general, group test cases that provide more information about the faults.
- When a tester's goal is to work with a more balanced

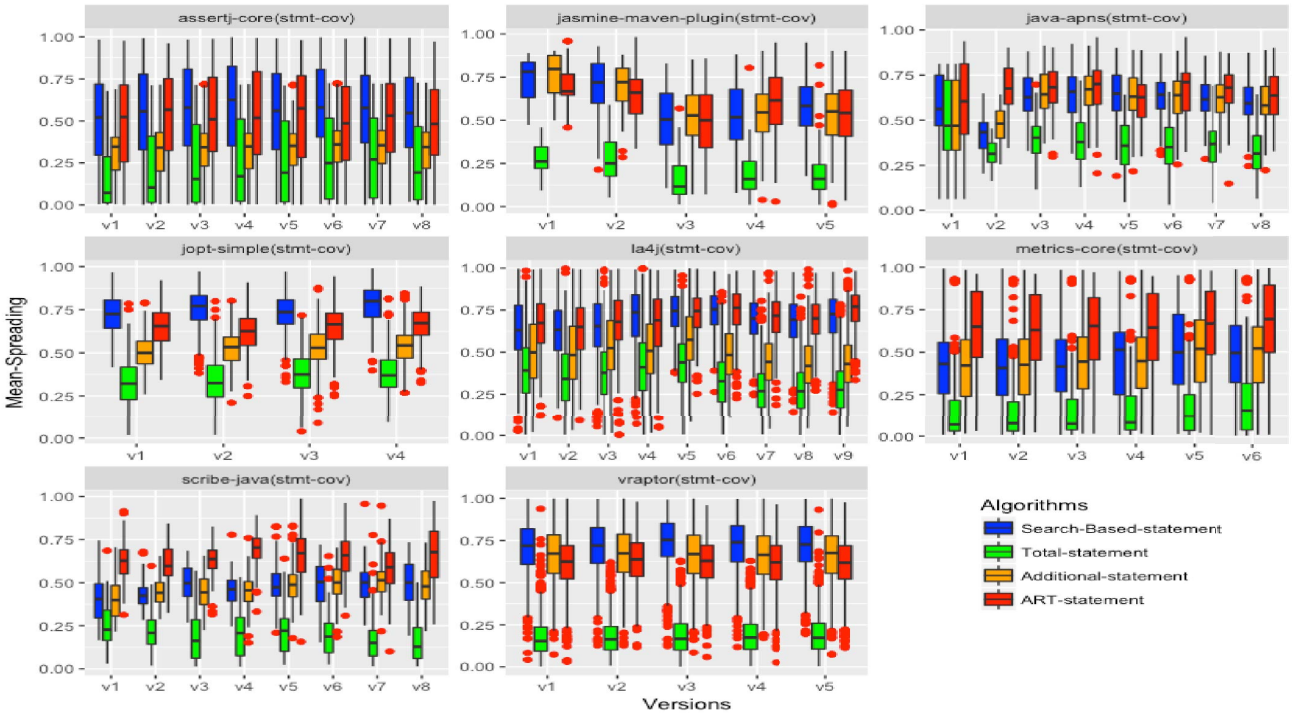

Fig. 8. Distribution of *M-Spreading* by version of project

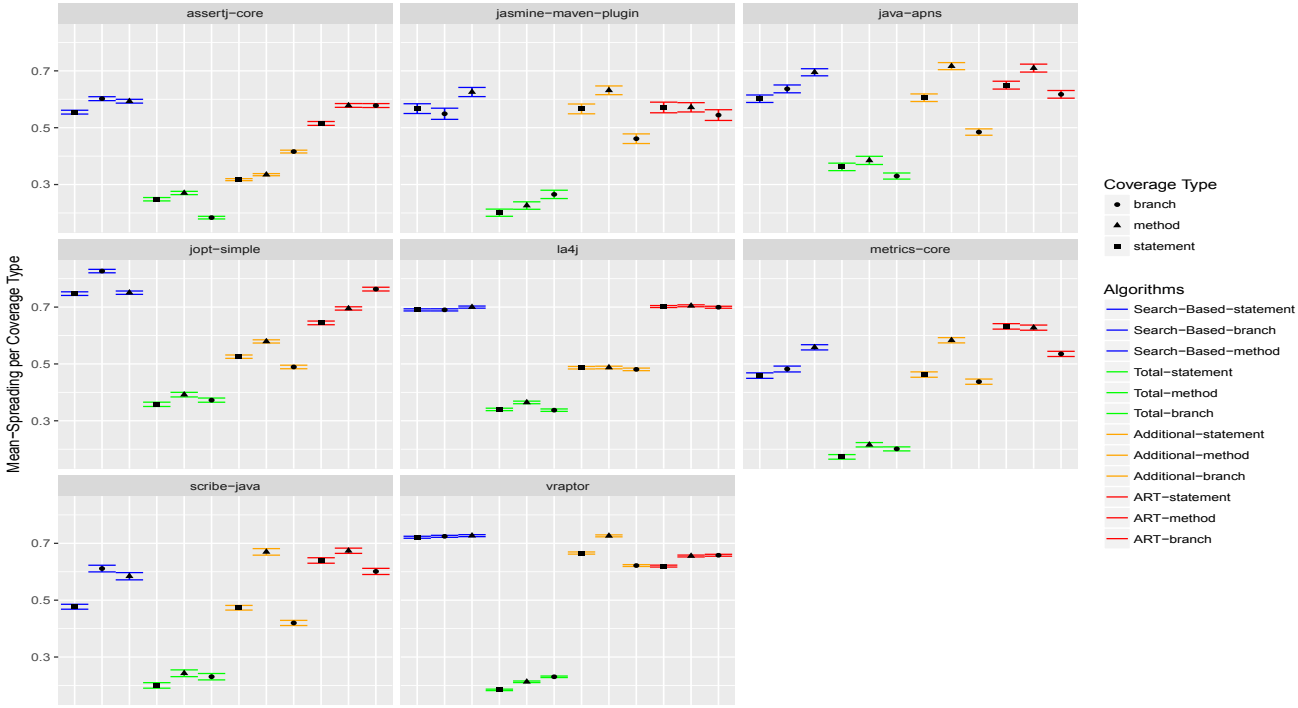

Fig. 9. *M-Spreading* confidence intervals.

prioritization, i.e., a test order that favors both early detection and fault localization, we also recommend a technique that follows the *additional* strategy, since, according to our analysis, they generate the best APFD

and second best *M-Spreading* results. Techniques from the *search-based* group performed second in this context.

E. Threats to validity

Any controlled experiment is subject to some threat to validity. Here, we list some:

External Validity: Our results certainly do not generalize beyond the projects used in our study. However, due to the variability of the projects in size, characteristics, and test suites, we believe they are good representatives of the universe of open source Java programs. Moreover, our dataset of projects was extracted from other works [16]. To reduce this treat and improve the generalization we intend to include more projects with different programming languages and context. One can argue that seeded faults do not represent typical real faults introduced by developers. However, seeded faults have been widely used in prioritization studies [7], [18]. Moreover, we refer to several studies that found that mutants are representative of real faults [3], [11], [33]. Moreover, to reduce this threat, we plan to rerun the study only real faults and compare the results considering the injected ones.

Internal Validity: In terms of internal validity, we reused the implementation of the TCP techniques provided by [16]. Moreover, the authors created several tests for those implementations including suites with different sizes and complexities.

Construction Validity: Other metrics besides APFD and *M-Spreading* could have to be used. However, we choose these two because the first is the most well-known and most used in different prioritization works [22]; and *M-Spreading* due to its novel approach for evaluating prioritization.

V. Related Work

Prioritization Techniques. Most of the work related to test case prioritization focus on defining new prioritization strategies/algorithms or improving existing ones [5], [8], [19], [21], [29]. Among the most traditional TCP techniques, we can list the *Total* and the *Additional* [27] strategies. The *Total* strategy reorders the test cases based on how much code they cover. The *Additional* strategy, on the other hand, performs the prioritization considering test cases that cover different code elements. Zhang et al. [33] propose a unified model that combines elements from both (Total and Additional) strategies. Li at al. [15] apply the prioritization as a search problem and propose a prioritization technique that uses a genetic search algorithm. Lu et al. [16] combine random prioritization with a heuristic based on selecting test cases that are distant from each other. Overall, an impressive number of TCP techniques have been proposed, as described in the literature reviews performed by Singh et al [29] and Khatibsyarbini et al [14]. However, the latter provides evidence that TCP techniques are still subject to improvement.

Metrics. Regarding metrics for evaluating prioritization, the most well-known is the APFD (Average Percentage Faults Detected), which measures how fast a suite detects all fault [31]. However, other metrics are important to mention. For instance, the APCC (Average Percentage of λ-wise Combinations Covered) [9], which measures the interaction rate of strength λ, where the strength is the level of interaction between a test case and a code unit. The PTR (Problem Tracking Reports) metric [23] calculates the percentage of prioritized cases that need to be executed for detecting all faults. Moreover, Alves et al. [1] propose the *F-Spreading*, metric investigated in our study. It calculates how spread all test cases that detect faults in a prioritized suite are. The idea is that test cases that detect faults should be placed near in order to provide more information about them, and, consequently, better help a tester to understand and locate the faults.

Empirical Studies. As far as we know, most of the work regarding empirical studies using TCP techniques focus on providing a comparative analysis of the techniques. For instance, Rothermel et al. [26] investigate the influence of different coverage criteria (e.g., statement, method) for prioritization. Moreover, Rothermel et al. [27] investigate how TCP works using different faults (e.g., real faults, mutants). Furthermore, Do et al. [3] investigate prioritization using different time constraints [30]. They found that time constraints can affect TCPs differently. Jiang et al [10] investigate how TCP techniques may impact fault localization. They found that coverage-based and random techniques can be more effective to support statistical fault localization. Recently, Luo et al [17] conducted a study to compare TCP techniques applied to real-world and mutation faults. They observed that results may not strongly correlate, particularly chosen operators may not reflect typical faults of the program.

In this work, we focused on performing an empirical study for evaluating a set of the most well-known TCP considering two different metrics APFD, and a variation of *F-Spreading* - metric related to investigate the impact of the prioritization ordering defined on fault localization by assuming that the low spreading of failing test cases can improve fault localization.

VI. Conclusion

Costs and time restrictions often prevent testers to run the whole regression test suite. To cope with this problem, TCP techniques can be used to reorder the tests aiming at achieving some testing goal. There are several TCP techniques. Testers often choose a TCP technique based on APFD results. However, other aspects may influence the effectiveness of a prioritization. In this paper, we presented an empirical study that used 8 real open-source projects for evaluating a series of TCP techniques regarding two different aspects: how fast they detect the faults (APFD metric), and how well they group fault revealing test cases (M-Spreading metric). Our results show that TCP techniques generate different results for APFD and *M-Spreading*. While techniques that follow the *additional* and *total* strategies obtained better and worst results for APFD, respectively; techniques that follow the *total* strategy performed better considering *M-Spreading*. Moreover, a set of guidelines were stated to help testers to better choosing a TCP technique, based on their goals.

As future work, we plan to expand our study by including new metrics and projects. We also plan to develop a configurable TCP technique that would be adaptive to a tester's needs (faster fault detection or aid for fault understanding/localization). Finally, we intend to run a study to understand how the system and test suite's characteristics may influence the spreading of failing test cases for each prioritization technique.

VII. Acknowledgments

This work was supported by the National Council for Scientific and Technological Development (CNPq)/Brazil (Process 437029/2018-2 and 429250/2018-5). The first author was supported by Brazilian Federal Agency for Support and Evaluation of Graduate Education (CAPES)/Brazil. The third author was supported by CNPq/Brazil (Process 311239/2017-0). We also thank the ePol-PF project, for the sponsorship.

References

[1] Everton LG Alves, Patrícia DL Machado, Tiago Massoni, and Miryung Kim. Prioritizing test cases for early detection of refactoring faults. *Software Testing, Verification and Reliability*, 26(5):402–426, 2016.

[2] Everton L.G. Alves, Tiago Massoni, and Patrícia Duarte de Lima Machado. Test coverage of impacted code elements for detecting refactoring faults: An exploratory study. *Journal of Systems and Software*, 123:223 – 238, 2017.

[3] Hyunsook Do, Siavash Mirarab, Ladan Tahvildari, and Gregg Rothermel. The effects of time constraints on test case prioritization: A series of controlled experiments. *IEEE Transactions on Software Engineering*, 36(5):593–617, 2010.

[4] Martin Fowler. *Refactoring - Improving the Design of Existing Code*. Addison-Wesley, 2 edition, 2018.

[5] Avinash Gupta, Nayneesh Mishra, Aprna Tripathi, Manu Vardhan, and Dharmender Singh Kushwaha. An improved history-based test prioritization technique technique using code coverage. In *Advanced Computer and Communication Engineering Technology*, pages 437–448. Springer, 2015.

[6] Dan Hao, Lu Zhang, and Hong Mei. Test-case prioritization: achievements and challenges. *Frontiers of Computer Science*, 10(5):769–777, 2016.

[7] Dan Hao, Lu Zhang, Lei Zang, Yanbo Wang, Xingxia Wu, and Tao Xie. To be optimal or not in test-case prioritization. *IEEE Transactions on Software Engineering*, 42(5):490–505, 2016.

[8] Charitha Hettiarachchi, Hyunsook Do, and Byoungju Choi. Risk-based test case prioritization using a fuzzy expert system. *Information and Software Technology*, 69:1–15, 2016.

[9] Rubing Huang, Yunan Zhou, Weiwen Zong, Dave Towey, and Jinfu Chen. An empirical comparison of similarity measures for abstract test case prioritization. In *Computer Software and Applications Conference (COMPSAC), 2017 IEEE 41st Annual*, volume 1, pages 3–12. IEEE, 2017.

[10] B. Jiang, Z. Zhang, T. H. Tse, and T. Y. Chen. How well do test case prioritization techniques support statistical fault localization. In *2009 33rd Annual IEEE International Computer Software and Applications Conference*, volume 1, pages 99–106, July 2009.

[11] Bo Jiang, Zhenyu Zhang, Wing Kwong Chan, and TH Tse. Adaptive random test case prioritization. In *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, pages 233–244. IEEE Computer Society, 2009.

[12] René Just, Darioush Jalali, and Michael D Ernst. Defects4j: A database of existing faults to enable controlled testing studies for java programs. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, pages 437–440. ACM, 2014.

[13] G Kapfhammer. The computer science handbook, chapter software testing, 2004.

[14] Muhammad Khatibsyarbini, Mohd Adham Isa, Dayang N.A. Jawawi, and Rooster Tumeng. Test case prioritization approaches in regression testing: A systematic literature review. *Information and Software Technology*, 93:74 – 93, 2018.

[15] Zheng Li, Mark Harman, and Robert M Hierons. Search algorithms for regression test case prioritization. *IEEE Transactions on software engineering*, 33(4), 2007.

[16] Yafeng Lu, Yiling Lou, Shiyang Cheng, Lingming Zhang, Dan Hao, Yangfan Zhou, and Lu Zhang. How does regression test prioritization perform in real-world software evolution? In *Software Engineering (ICSE), 2016 IEEE/ACM 38th International Conference on*, pages 535–546. IEEE, 2016.

[17] Q. Luo, K. Moran, D. Poshyvanyk, and M. Di Penta. Assessing test case prioritization on real faults and mutants. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 240–251, Sep. 2018.

[18] Qi Luo, Kevin Moran, and Denys Poshyvanyk. A large-scale empirical comparison of static and dynamic test case prioritization techniques. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 559–570. ACM, 2016.

[19] R Uma Maheswari and D Jeya Mala. Combined genetic and simulated annealing approach for test case prioritization. *Indian Journal of Science and Technology*, 8(35), 2015.

[20] John H McDonald. *Handbook of biological statistics*, volume 2. sparky house publishing Baltimore, MD, 2009.

[21] Subhas Misra, Vinod Kumar, Uma Kumar, Kamel Fantazy, and Mahmud Akhter. Agile software development practices: evolution, principles, and criticisms. *International Journal of Quality & Reliability Management*, 29(9):972–980, 2012.

[22] David Paterson, Gregory M Kapfhammer, Gordon Fraser, and Phil McMinn. Using controlled numbers of real faults and mutants to empirically evaluate coverage-based test case prioritization. In *Proceedings of the International Workshop on Automation of Software Test (AST 2018)*. IEEE, 2018.

[23] R Pradeepa and Vimala K Devi. Effectiveness of testcase prioritization using apfd metric: Survey. In *IJCA Proceedings on International Conference on Research Trends in Computer Technologies*, pages 1–4, 2013.

[24] Napol Rachatasumrit and Miryung Kim. An empirical investigation into the impact of refactoring on regression testing. In *Software Maintenance (ICSM), 2012 28th IEEE International Conference on*, pages 357–366. IEEE, 2012.

[25] Gregg Rothermel and Mary Jean Harrold. A safe, efficient regression test selection technique. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 6(2):173–210, 1997.

[26] Gregg Rothermel, Roland H Untch, Chengyun Chu, and Mary Jean Harrold. Test case prioritization: An empirical study. In *Software Maintenance, 1999.(ICSM'99) Proceedings. IEEE International Conference on*, pages 179–188. IEEE, 1999.

[27] Gregg Rothermel, Roland H. Untch, Chengyun Chu, and Mary Jean Harrold. Prioritizing test cases for regression testing. *IEEE Transactions on software engineering*, 27(10):929–948, 2001.

[28] Kesar Singh and Minge Xie. Bootstrap: a statistical method. *Unpublished manuscript, Rutgers University, USA. Retrieved from http://www. stat. rutgers. edu/home/mxie/RCPapers/bootstrap. pdf*, 2008.

[29] Yogesh Singh, Arvinder Kaur, Bharti Suri, and Shweta Singhal. Systematic literature review on regression test prioritization techniques. *Informatica*, 36(4), 2012.

[30] Xiaolin Wang and Hongwei Zeng. History-based dynamic test case prioritization for requirement properties in regression testing. In *Proceedings of the International Workshop on Continuous Software Evolution and Delivery*, pages 41–47. ACM, 2016.

[31] Shin Yoo and Mark Harman. Regression testing minimization, selection and prioritization: a survey. *Software Testing, Verification and Reliability*, 22(2):67–120, 2012.

[32] Shin Yoo, Mark Harman, and David Clark. Fault localization prioritization: Comparing information-theoretic and coverage-based approaches. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 22(3):19, 2013.

[33] Lingming Zhang, Dan Hao, Lu Zhang, Gregg Rothermel, and Hong Mei. Bridging the gap between the total and additional test-case prioritization strategies. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 192–201. IEEE Press, 2013.

[34] Lingming Zhang, Miryung Kim, and Sarfraz Khurshid. Faulttracer: a change impact and regression fault analysis tool for evolving java programs. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, page 40. ACM, 2012.