A similarity-based approach for test case prioritization using historical failure data

Tanzeem Bin Noor and Hadi Hemmati
Department of Computer Science
University of Manitoba
Winnipeg, Canada
Email: {tanzeem, hemmati}@cs.umanitoba.ca

***Abstract*—Test case prioritization is a crucial element in software quality assurance in practice, specially, in the context of regression testing. Typically, test cases are prioritized in a way that they detect the potential faults earlier. The effectiveness of test cases, in terms of fault detection, is estimated using quality metrics, such as code coverage, size, and historical fault detection. Prior studies have shown that previously failing test cases are highly likely to fail again in the next releases, therefore, they are highly ranked, while prioritizing. However, in practice, a failing test case may not be exactly the same as a previously failed test case, but quite similar, e.g., when the new failing test is a slightly modified version of an old failing one to catch an undetected fault. In this paper, we define a class of metrics that estimate the test cases quality using their similarity to the previously failing test cases. We have conducted several experiments with five real world open source software systems, with real faults, to evaluate the effectiveness of these quality metrics. The results of our study show that our proposed similarity-based quality measure is significantly more effective for prioritizing test cases compared to existing test case quality measures.**

***Index Terms*—Test case prioritization; Test quality metric; Similarity; Execution trace; Distance function; Historical data; Code coverage; Test size.**

I. Introduction

Software testing is a crucial task in the software development process that ensures software quality. The goal of testing is to verify the software under development against the client's requirements and identify its faults. Generally, numerous test cases can be designed and executed to verify the software. Among these huge sets of test cases, only a few effective tests can detect faults (mismatches between actual and expected output). Test case prioritization are used to detect the faults faster by executing a prioritized list of effective test cases.

The ultimate effectiveness measure of a test case is its actual fault-detection power that indicates how many real faults the test case can detect. Unfortunately, this measure is not very practical because usually one needs to know about the effectiveness of test cases before execution. In addition, when a test case passes, it can not be simply inferred that there is no fault in the program because the test may pass due to its ineffectiveness in catching the fault. Therefore, we need other test quality metrics to estimate the effectiveness of the tests cases for test prioritization.

Code coverage is a widely used quality metric that measures how much of the code (e.g., number of lines, blocks, conditions etc.) from the program is exercised during the tests execution. As faulty code needs to be executed to reveal its fault, covering (executing) more code increases the probability of covering the faulty code, as well. However, covering the faulty code may not always result in detecting its faults [1]. Faults are only revealed when the faulty code is executed with special input values, which actually causes the tests to fail. Therefore, code coverage does not guarantee detecting faults and is simply a heuristic that estimates the test case quality.

Detecting previous faults is another important factor used to estimate the tests case quality. In the context of regression testing, test cases can be generated and or prioritized, based on the previous faults (history-based test prioritization) [2]. The rationale behind it is that if a test detects a fault in the past, it is probably touching a part of the code that used to be faulty. On the other hand, defect prediction studies have shown that if a file/method used to be faulty, it is highly likely to be faulty again, specially if it is being changed [3], [4]. So the test case that touches those faulty places might detect new faults, as well. The type of quality metric that quantifies this concept is called history-based quality metric, in this study.

The typical history-based quality metric goes through the history of the software and identifies test cases from the current release that used to fail in any of the previous releases [3], [4]. Those previously failed test cases will be ranked higher in the prioritized list of test cases. The problem with this approach is that in many situations (e.g., when a new test is added or when an old test is modified) the test case of the current release that detects a fault is not exactly the same as any of the previously failing test cases. However, it is quite similar to one (or more) old failing test case(s), in terms of the sequence of methods being called (these similar test cases are verifying different aspects of a risky scenario with minor differences).

In this paper, we define a set of test case quality metrics that assign historical faultiness values to the test cases, when they are similar to the failing tests from previous releases. Each metric defines similarity using a different similarity/distance function, but they all are applied on the test cases' execution traces. We look at an execution trace as a sequence of method

978-1-5090-0406-5/15/$31.00 ©2015 IEEE

```
@Test
public void testLang747() {
        assertEquals(Integer.valueOf(0x8000), NumberUtils.createNumber("0x8000"));
        assertEquals(new BigInteger("8000000000000000", 16), NumberUtils.createNumber("0x8000000000000000"));
        ....
        assertEquals(new BigInteger("FFFFFFFFFFFFFFFF", 16), NumberUtils.createNumber("0xFFFFFFFFFFFFFFFF"));
        assertEquals(Long.valueOf(0x80000000000000L), NumberUtils.createNumber("0x00080000000000000"));
        assertEquals(Long.valueOf(0x8000000000000000L), NumberUtils.createNumber("0x08000000000000000"));
        ...
        assertEquals(Long.valueOf(0x7FFFFFFFFFFFFFFFL), NumberUtils.createNumber("0x07FFFFFFFFFFFFFF"));
        assertEquals(new BigInteger("8000000000000000", 16), NumberUtils.createNumber("0x0008000000000000000"));
        assertEquals(new BigInteger("FFFFFFFFFFFFFFFF", 16), NumberUtils.createNumber("0x0FFFFFFFFFFFFFFF"));
}
```

Fig. 1. A failing test case in the Commons Lang project's latest version [5]

```
@Test
public void testStringCreateNumberEnsureNoPrecisionLoss() {
        String shouldBeFloat = "1.23";
        String shouldBeDouble = "3.40282354e+38";
        String shouldBeBigDecimal = "1.797693134862315759e+308";
        ... .. . .

        assertTrue(NumberUtils.createNumber(shouldBeFloat) instanceof Float);
        ..... . ..
        assertTrue(NumberUtils.createNumber(shouldBeDouble) instanceof Double);
        assertTrue(NumberUtils.createNumber(shouldBeBigDecimal) instanceof BigDecimal);
}
```

Fig. 2. A failing test case in the previous versions of the Commons Lang project that is similar to the test case in Fig. 1 [5]

calls from the program source code, when the test case is executed. We have conducted a series of empirical studies (using five open source java software systems) to compare the effectiveness of these metrics in the context of test case prioritization. The results of our study show that the similarity-based approach is more effective in prioritizing fault-revealing tests compared to the traditional history-based approach. Moreover, we have also found that the similarity-based quality metrics are also better than other test quality metrics, e.g., coverage and test size in prioritizing tests.

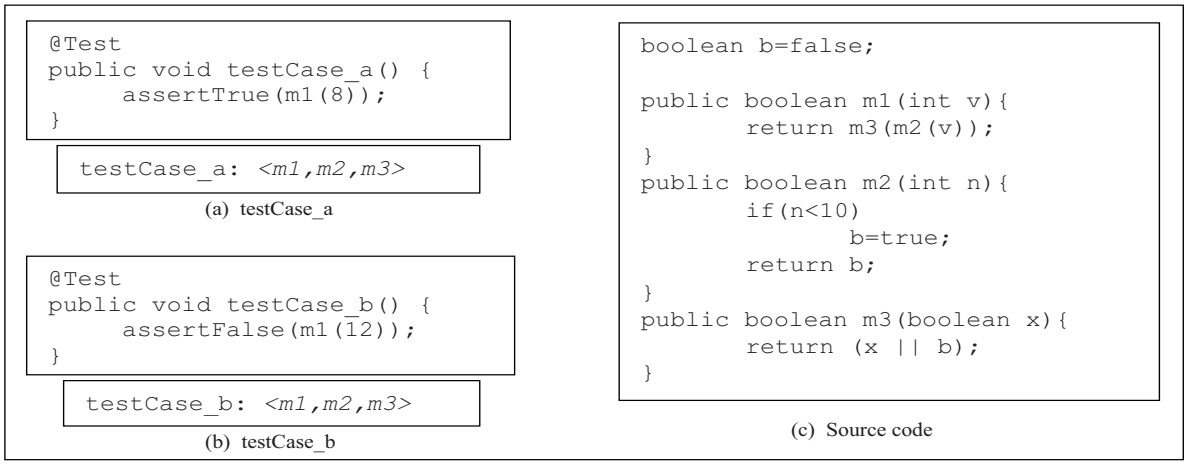
The rest of this paper is organized as follows: section II mentions our motivation behind this study; some existing traditional test quality metrics have been presented in section III. Our proposed similarity-based test prioritization has been explained in section IV. We have discussed our experiments and results in section V. Section VI states some of the related works. Finally, section VII concludes the paper and mentions our future work.

II. Motivation

Test case quality metrics are used in different applications; most commonly in evaluating existing test suites, to make sure enough testing has been done. An automatic test case generation tool also uses quality metrics to evaluate test case effectiveness in order to produce high quality tests. In addition, quality metrics are used in prioritizing test cases, when the resource (e.g., time, number of software testers) is limited. Test case prioritization ranks the test cases based on the quality metrics so that the more effective tests are being executed first and detect the software faults faster, within the limited testing budget. Test case prioritization is very important in practice for software companies, specially when continuous integration and rapid release demands fast development paces.

In continuous integration development environments, new or changed code are frequently integrated with the mainline codebase. Continuous integration processes require extensive testing to be performed prior to code submission. To make this process cost-effective, regression testing techniques must operate effectively within continuous integration development [6]. Recently, Elbaum *et al.* has shown that in the continuous integration process, test selection and prioritization techniques can be performed, cost-effectively, in the absence of coverage data by using readily available test suite execution history [6]. Test execution history provides the information regarding a test, i.e., whether it passed or failed (detected a fault, previously). In addition, the historical fault detection measure is also a guiding factor to detect faults in the current version [3], [4], [7], [8].

In the traditional history-based quality metrics, a test case from the current release would be considered as effective if the exact same test also failed in the previous releases [3], [4]. Now assume a test case, such as testLang747 (a test case from the latest version of project Commons Lang, explained in section V-B) in Fig. 1, that is just added to the current test suite and actually fails (detects a fault). This test case is not effective according to the traditional history-based quality metric, since it did not exist in the previous releases, to fail. However, there are some test cases in the past that are quite similar to this test case and they failed, e.g., the test case in Fig. 2. Therefore, it would be nice to have a history-based quality measure for test cases that do not only look at exact occurrences of the test case in the past, but look at its similar cases, as well. The key question here is "how do we identify



(a) testCase_a

(b) testCase_b

(c) Source code

Fig. 3. Example of test cases and their corresponding method calls

such similar test cases?"

Usually, along with the historical fault detection information, execution traces can also be collected from the history. In general, an execution trace of a test case is the sequence of method calls from the program source. For example, when a Junit test "testCase_a" in Fig. 3a is executed, it calls method $m1$ from the program source code in Fig. 3c and its execution trace would be *testCase_a*$< m1, m2, m3 >$ in terms of method invocation sequence. Now, assume another Junit test "testCase_b" in Fig. 3b. This test seems different from the test in Fig. 3a, since both its name and assert statement differ from "testCase_a". However, the execution trace of "testCase_b" is *testCase_b*$< m1, m2, m3 >$, which is very similar (in this scenario same) to the execution trace of "testCase_a", in terms of the method call sequence.

To quantify the similarity between a new/modified test case and a failing test case from history, we can represent them by their sequences of method calls as described in the previous example (Fig. 3). The sequence of method calls could be extracted from their execution traces. For instance, the sequences for the two test cases of Fig. 1 and Fig. 2 are shown in Fig. 4a and Fig. 4b. As it can be seen, $NumberUtils.createNumber(java.lang.String)$, $StringUtils.isBlank(java.lang.CharSequence)$ and *NumberUtils.isAllZeros(java.lang.String)* methods are the same in the two test case traces, which makes the two test cases similar.

This example, and other cases like this, which show the failing test case trace in the current release is very similar to the failing test case traces from previous releases, were the motivations behind this work and our previous work [5], where we proposed a new quality metric using test case similarity to historical failing test cases. In this work we expand that short paper by properly investigating different possibilities for such a quality metric, we also put the quality metric in the context of test case prioritization and compare it with other commonly used metrics in that domain.

III. Background

In the applications like test case prioritization and generation, different test case quality metrics are extensively used. The main goal of using test case quality metrics is to evaluate the tests and find the scope of improvement required in the test cases. The primary quality measure of a test case is its ability to detect software faults, i.e., whether the test fails on the program. Sometimes the severity of the revealed faults might be a crucial factor and therefore the tests that detect more severe faults be considered as higher quality.

As the ultimate goal of testing is detecting faults, any measure that directly quantifies the fault detection power is a perfect metric, in terms of effectiveness. However, the actual fault detection metric can not be practically used in test prioritization, since we don't know the actual number of faults before executing the test cases. Therefore, several heuristics are used to define test quality metrics with the hope that they have high correlations with real fault detection power of the test cases. A well-known set of such metrics are test adequacy criteria that is categorized mainly into coverage-based and fault-based test adequacy criteria [1], [9].

Besides these test adequacy criteria, there are some other quality metrics that have high correlation to fault detection ability of the test cases, such as historical fault detection, coverage, size of the test, complexity, code churn, etc. [3]. In the rest of this section, we explain some of these test quality metrics that are being used in this paper.

1) Procedure/Method coverage: Code coverage is a commonly used test adequacy criteria that indicates how much of the program is executed when the test case runs. A test case reveals a fault when it executes a segment from the program that causes a failure. Most existing automated test generation

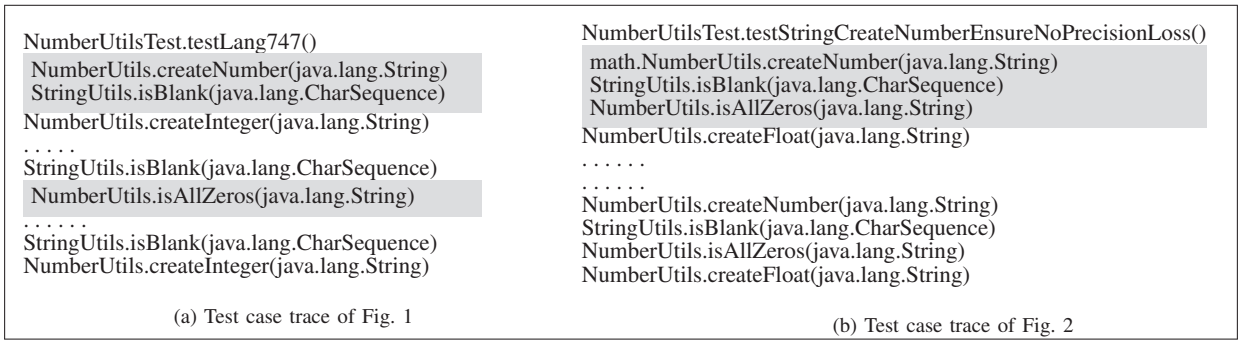


Fig. 4. Example of execution traces for Fig. 1 and Fig. 2 [5]

tools [10], [11] try to generate test cases that cover/execute 100% (or as close as possible to that) of the source code. Therefore, high coverage has always been an indicator of good quality for test cases.

Coverage based test adequacy criteria is further grouped into control-flow coverage and data-flow coverage. One of the simple control-flow coverage criteria is measuring procedure/method coverage of the tests. The key question here is "has each function (or method) in the program been called?" For each test case, the method coverage refers to the number of methods called from the test case (directly or indirectly) divided by the total number of methods in the program [9].

2) Code coverage of the changed parts: Since in regression testing, the source code is modified from the previous version, metrics that measure the coverage of the changed parts of the code are very important [3], because even if the total coverage of the test suite is not high, we still expect a high coverage in the change part to assure proper regression testing. This metric prioritizes the test cases that execute any change part of the code directly or indirectly.

3) Size of tests: The size of the test cases is also a commonly used test quality metric, where the large size indicates a more effective test. Generally, size of a test case refers to the LOC (Line of Codes) in the test method. However, sometimes the number of *assertions* in a test case is considered a better size measure, since it directly measures the amount of verifications applied by the test cases. This is to mention that size of test cases is not the same as its coverage [1]. For example, test t_1 can cover a method with one assertion and test t_2 do the same but try it with 5 more assertions. In this case, test t_2 has higher chances to detect faults than t_1.

4) Historical fault detection: Historical fault detection metrics assign higher priority to test cases of the current release that failed historically, i.e., on any previous release [3], [4], [7], [8]. The reason is that studies have shown that the tests with higher historical fault detection rate are more likely to fail in the current version, as well.

IV. Proposed approach

To deal with problems such as those explained in the motivation (example in section II), we propose to prioritize test cases using an improved history-based quality metric. In the rest of this section, we explain our proposed approach.

A. Similarity-based test quality metric

Our proposed similarity-based metric considers a test as effective, if it is similar to any of the failed test cases from the previous versions. The similarity between test cases is defined based on their sequences of method calls, extracted from execution traces. Therefore, first, execution traces containing the sequence of method calls need to be retrieved for all of the previously failed tests. Then, the sequences of method calls (i.e., execution traces) are also collected for all the modified or newly added tests in the current version. Finally, a similarity function is used to determine the similarity between the execution traces of the modified/new tests in the current version and the previously failed tests. So, the input of the similarity function is the execution traces of both the current releases' modified/new tests and the previously failed tests. The similarity function, at the end, returns a value indicating how similar is a test case to the previously failed test cases.

There are several similarity functions that can be used to determine similarity among test cases. In this paper, we study three functions and also try to improve them by combining the two best candidates.

1) Basic Counting (BC): This measure is a very basic function, which does not account for the method call orders nor for their position in the sequence. The function simply looks at the past failing sequences of unique method calls and counts the overlap with the unique method calls of the test case under study. The total similarity value of a test case is the sum of all these occurrences, which indicates how many unique method calls from the current test case also appeared in the previous failing tests traces. However, instead of using the actual summation value, a normalized value between 0 and 1 is used. The normalization is performed by diving the total summation value by the total number of unique method calls

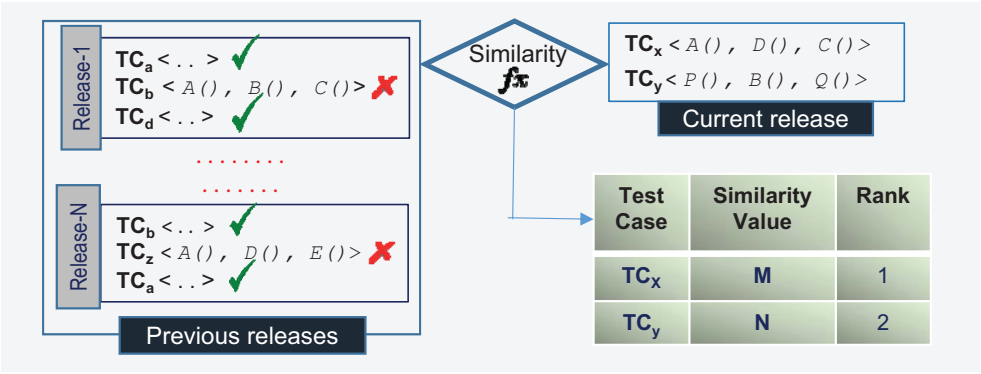


Fig. 5. An overview of the proposed similarity-based test case prioritization, using historical failure data.

from the history. The higher normalized value of a test case means the test has also higher similarity with the previous failing tests.

*2) **Hamming Distance (HD):*** Hamming Distance is a widely used distance functions used in the literature, which is a basic edit-distance. The edit-distance between two sequences is defined as the minimum number of edit operations (insertions, deletions, and substitutions) needed to transform the first sequence into the second [12], [13]. Hamming is only applicable on identical length inputs and is equal to the number of substitutions required in one input to become the second one [12]. If all inputs are originally of identical length, the function can be used as a sequence-aware measure. However, in most of the applications, test inputs have different lengths. Therefore, to force them to have an identical length, a binary vector is made per input that indicates which elements from the set of all possible elements of the encoding exist in the input. As a result, the function does not preserve the original order of elements in the input anymore and it becomes a sequence ignorant (or set-based) similarity function [14], [15].

In our case, the total hamming distance of a modified/new test, e.g., $T1$ in the current release, is calculated by summing up all hamming distances, between $T1$ and each of the failing tests from the previous releases. For example, assume, two previous failed test traces are $T3 < A(), B(), B(), C(), B(), D() >$ and $T2 < A(), E(), D(), B(), A() >$, and the modified/new test trace in the current release is $T1 < F(), A(), C(), M(), N(), X(), A() >$. Therefore, the total hamming distance of $T1$ would be the summation of hamming distances between $T1$ and $T3$ (i.e., *Hamm*$(T1, T3)$), and hamming distances between $T1$ and $T2$ (i.e., *Hamm*$(T1, T2)$).

To calculate the hamming distance between $T1$ and $T2$ (*Hamm*$(T1, T2)$), first, all unique method calls from $T1$ and $T2$ form a set of all possible elements, i.e., $V < A(), E(), D(), B(), F(), C(), M(), N(), X() >$. Now, both the $T1$ and $T2$ are encoded as binary vector of identical length, where a bit is true only if the encoded test case contains the corresponding element from V. So, the test $T1$ is encoded as $< 1, 0, 0, 0, 1, 1, 1, 1, 1 >$ and $T2$ is encoded as $< 1, 1, 1, 1, 0, 0, 0, 0, 0 >$, with respect to the set of all possible elements V.

Then *Hamm*$(T1, T2)$ is measured by applying XOR operation between their binary encoded representations and then normalized between 0 and 1 by dividing the sum of XOR values by the length of V. Therefore, the normalized *Hamm*$(T1, T2)$ is $< 0, 1, 1, 1, 1, 1, 1, 1, 1 >= 8/9$. Similarly, the normalized hamming distance between $T1$ and $T3$ (*Hamm*$(T1, T3)$) is also calculated.

Finally, all of these normalized hamming distances are summed up and normalized between 0 and 1 again by dividing the summation value by the total number of failed test cases from previous versions. The low hamming distance value of a test case means the test has high similarity with the previous failing tests. So, we convert this distance to similarity by subtracting the total normalized hamming distance value from 1.

3) Edit Distance (ED) The general edit distance function is a sequence-aware function, where the order and position of method calls in the traces would matter. One of the most well-known algorithms implementing edit-distance which is not limited to identical length sequences is Levenshtein [12] where each mismatch (substitutions) or gap (insertion/deletion) increases the distance by one unit. To change distances into similarities, we need to reward each match and penalize each mismatch and gap. The relative scores assigned to matches, mismatches, and gaps can be different. A basic setting for the function would be implemented in a way where matches are rewarded by one point and mismatch and gap are treated the same by giving no reward [14], [15].

B. Similarity-based test prioritization

In our proposed prioritization approach, the test cases are sorted based on their descending similarity values, which are calculated using different approaches mentioned in the previous sub-section. The overall test prioritization process is shown in Fig. 5 where the previous failed tests are TC_b and

TABLE I
PROJECTS UNDER STUDY

Projects	#Faults (#Versions)	#Test Cases	Median number of Test cases per version
JFreeChart	26	2,205	1,751
Closure Compiler	133	7,927	7,066
Commons Math	106	3,602	1,976
Commons Lang	27	2,245	1,757
Joda Time	65	4,130	3,748

TC_z, and the modified/added tests in the current release are TC_x and TC_y. The higher rank of TC_x indicates that it should be executed earlier than TC_y to detect the faults earlier.

V. EMPIRICAL STUDY

A. Research Questions

In this study, we have investigated the following research questions:

RQ1: Can a similarity-based test quality metric improve the traditional history-based metric, in the context of test case prioritization?

RQ2: Which similarity/distance function works best for the similarity-based test quality metric?

RQ3: Can the best similarity-based test quality metric improve existing quality metrics, such as code coverage, test size and change-related metrics?

B. Subjects under study

In our experiment, we have used five different Java projects from the *defects4j* database [16]. The database provides 357 faults and 20,109 Junit tests from five different open-source Java projects as mentioned in Table I. All the faults are real, reproducible and have been isolated in different versions [17]. There is a faulty version and a fixed version of the program source code, for each fault. The faulty source code is modified in the fixed version to remove the fault. The test cases are the same in both of the faulty and the fixed versions. However, there is at least one test case (a Junit test method) in each version that fails on the faulty version but passes on the fixed version.

C. Experiment design

As our proposed similarity functions require test case traces containing method call sequences, we need to use a tool for trace generation. We have used daikon [18] tool to produce the execution traces from three projects (Commons Lang, Joda Time and JFreeChart). Daikon is a tool to dynamically detect likely program invariants and it allows to detect properties in C, C++, C#, Eiffel, F#, Java, Perl, and Visual Basic programs [18]. The daikon front end (instrumenters) for Java, named Chicory, executes the target Java programs and creates the *.dtrace* file that contains the program execution flow along with the variable values in each program point. The method sequence calls have been extracted from the *.dtrace* file.

However, for the other two projects (Commons Math and Closure Compiler), we could not use daikon for trace collection, as the tests from these projects generated very large *.dtrace* files. So, instead of using daikon, we have used AspectJ [19] to produce the trace of method sequence calls directly. Although the trace extraction process is different, the format of the extracted method sequence calls is same. We have collected the method sequences for all modified tests in the current release and also all the failed tests traces from the previous releases. Finally, we have ranked the tests in each version by using different approaches.

To answer RQ1, we compare the prioritization using our Hamming Distance (HD) metric with the traditional history-based approach (TM). To answer RQ2, we compare HD, Basic Counting (BC), and Edit Distance (ED) functions to prioritize test cases. We also propose an improved basic counting (IBC) metric. In the improved BC, at first, the normalized BC and HD values are rounded into two decimal places. This will help avoiding ranking differences where the similarities are very close. The second improvement is to combine the BC and HD. To do that for cases where BC values are lower than 0.5, we first look at HD values if they are higher than 0.5. We rank them using HD, otherwise using BC. This will help getting the most out of both BC and HD.

To answer RQ3, we have compared the Size of Testcase (ST), Method Coverage (MC) and Changed Method Coverage (CMC) metrics against our similarity-based metric. ST is the number of uncommented statements in a test method that has been derived from Java Abstract Syntax Tree (AST) parser using Eclipse JDT API [20]. MC of a test case is the number of unique methods called during the test execution. CMC of a test case is the number of unique method calls that are called by the test case and have been changed since the last version. Both the MC and CMC has been derived from the execution traces.

We have implemented the the similarity functions using Java and we have used R [21] for our evaluation purposes.

D. Results and Discussion

The defects4j dataset mentions which test case actually fails in the current version [16]. So, we compare the rank of the first failing test provided by different approaches in each version (note that there is only one fault per version, but there might be more than one test that detect it). In other words, we compare the percentage of the test cases that need to be executed in

TABLE II
NUMBER OF STUDIED VERSIONS

Projects	#Studied Versions	#Versions (TM works)
Commons Lang	60	22
JFreeChart	24	2
Commons Math	43	12
Joda Time	25	0
Closure Compiler	95	0

order to catch the fault in each version, separately. We do this by dividing the rank of the first failing test by the total number of modified/new tests in each version. In case of ties in the similarity values, we have ranked the tied test cases randomly, by applying the *rank* function in R with a parameter *ties.method="random"* [22]. To deal with this randomness, we have calculated the ranks 30 times for each version of the projects.

Whenever we compare to distributions of the results and we say one outperforms the other, we have first conducted a non-parametric statistical significance test (U-test) [23] to make sure the differences are not due to the randomness of the algorithms. In addition to that, it is also crucial to assess the magnitude of the differences [24]. Effect size measures are used to analyze such a property [24], [25]. In our study, we have used a non-parametric effect size measure, called Vargha and Delaneys $\hat{A}_{12}$ statistics [24]–[26]. Given a performance measure M, the $\hat{A}_{12}$ statistics measures the probability that running algorithm X yields higher M values than running another algorithm Y. When the two algorithms are equivalent, then $\hat{A}_{12}$ is 0.5. Therefore, while comparing algorithm X and Y, $\hat{A}_{12} = 0.7$ represents that we would obtain higher results in 70% of the time with algorithm X compared to the algorithm Y [24].

In our context, the given performance measure M is the percentage of the tests need to be executed in order to catch the first fault in a version. Therefore, we get a $\hat{A}_{12}$ value for each version of the project while comparing two prioritization approaches. So, in our case, $\hat{A}_{12} = 0.8$ indicates that algorithm X ranks the first failing test higher than algorithm Y in 80% of the time. In other words, algorithm X can detect the fault faster than algorithm Y in 80% of time. We have used this $\hat{A}_{12}$ measure to evaluate all of our results.

In the rest of this section we answer research questions mainly by comparing the $\hat{A}_{12}$ measure. We also make sure that the differences of the results are statistically significant.

1) Experimental results for RQ1: To evaluate RQ1, we compare the first failing test's rank assigned by our proposed similarity-based approach (in this case, by using Hamming Distance (HD)) and the traditional history-based approach. The traditional approach ranks the previous failing tests higher in the current release. Therefore, the traditional approach works only when the set of all modified/new tests in the current release contains at least one test that failed in any of the earlier versions.

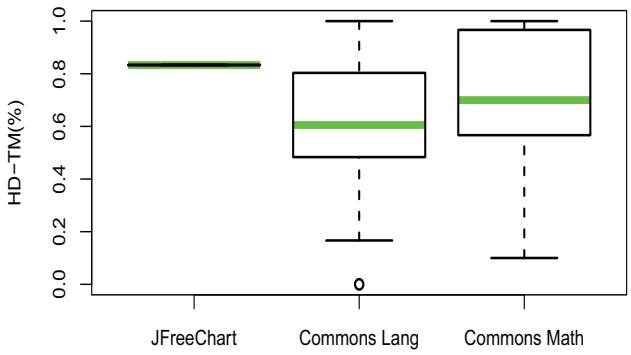


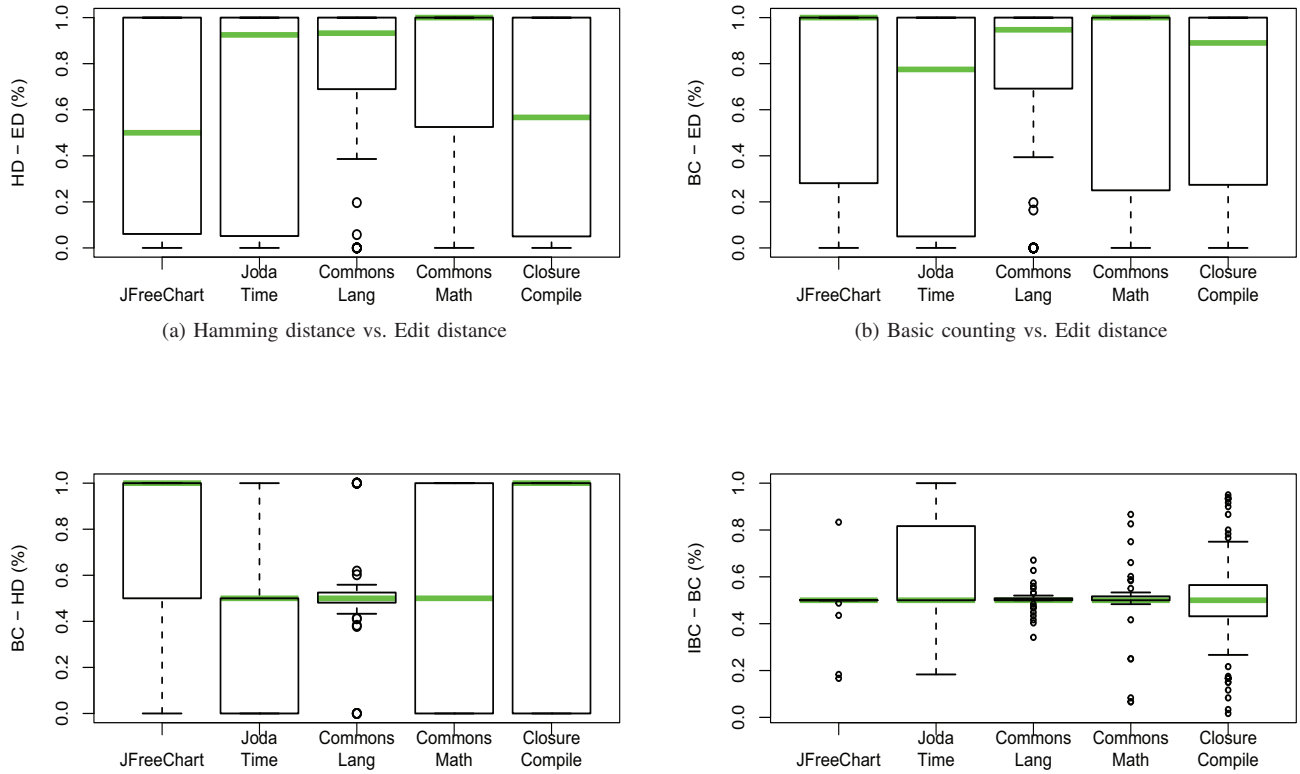
Fig. 6. The boxplots of the effect size measures for finding the first fault using HD and TM, when comparing 30 runs of each versions of each project.

Table II shows the number of versions where the traditional approach (TM) works. The large number of versions that the traditional approach is not working at all, is indeed our main motivation to propose an improved history-based metric. Since the traditional approach can not rank the tests in many versions, it is already falling behind our new metric, but to be fair we also compare the results for only the working versions.

For these versions of the projects, we compare the rank of the first failing test using the traditional approach against the similarity-based approach (in this case, by using HD). We have used $\hat{A}_{12}$ measure to compare these prioritization approaches. Fig. 6 shows the boxplots of the $\hat{A}_{12}$ measures distribution for the versions of three projects where the traditional prioritization approach works. The higher than 0.5 median lines in the boxplots represent that in general prioritization using the hamming distance-based similarity function is better than using traditional history-based prioritization, even for cases where the traditional approach is working (note that the differences are also statistically significant with p-values<0.05).

2) Experimental results for RQ2: To answer RQ2, we compare Basic Counting (BC), Hamming Distance (HD), Edit Distance (ED) and Improved BC (IBC) similarity functions. We look at the rank of the failing test provided by the prioritization approaches using these similarity functions and we also use the $\hat{A}_{12}$ measure for this evaluation. For this evaluation, we consider only the versions with at least 4 modified/new test cases. We assume that in the versions having less than 4 modified/new test cases, executing all test cases is not costly and hence the prioritization is not actually beneficial.

Fig. 7 shows the boxplots of the $\hat{A}_{12}$ measures distribution for the versions for each project where there are at least



(a) Hamming distance vs. Edit distance

(b) Basic counting vs. Edit distance

(c) Basic counting vs. Hamming distance

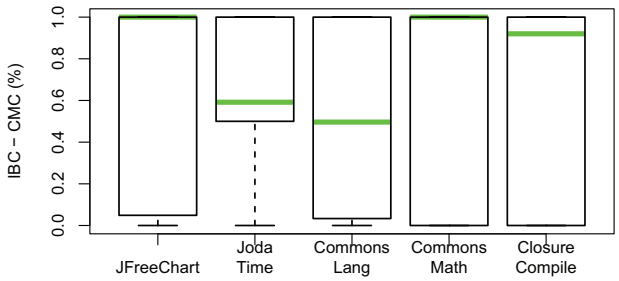
(d) Improved basic counting vs. Basic counting

Fig. 7. The boxplots of the effect size measures for finding the first fault using BC, HD, ED, and IBC, when comparing 30 runs of each versions of each project.

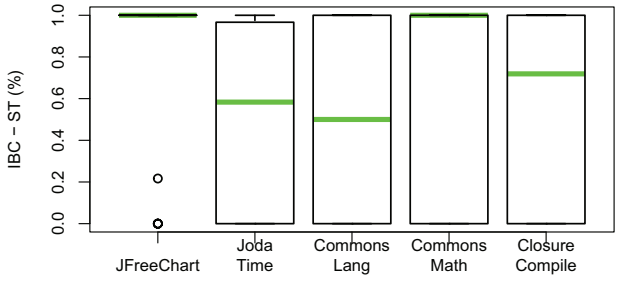
4 test cases. Fig. 7a compares HD and ED, and Fig. 7b compares BC and ED. It can be observed that the similarity-based prioritization using HD provides significantly better result than the use of ED in Joda Time, Commons Lang and Commons Math projects, as the median lines for these box plots are closer to 1.0. However, for the remaining two projects, the improvements are not that much significant (p-values are still lower than 0.05 but the median effect size is around 50%). On the other hand, BC is much better than the ED for all of the projects. Therefore, the results shown in Fig. 7a and Fig. 7b suggests that it is better to use the BC as a similarity function compared to the ED. The results also represent that the sequence-aware similarity function (ED) is falling behind the sequence-ignorant (or set based) similarity function (BC) in our test case prioritization. This result is actually inline with the recent studies [15] in the context of test case selection/prioritization. One plausible explanation is that the sequence-aware matching is too restrictive, which ignores potential weak similarities

Next, we compare the similarity function BC with HD and the results are shown in Fig. 7c. Here we can see that BC performs significantly better than the HD in JFreeChart and Closure Compiler project. However, the performance of BC and HD is quite similar for the Commons Lang and Commons Math projects, where neither of these functions outperforms the another (again p-values are low but the median effect size is around 50%). On the other hand, HD performs better than the BC for the Joda Time project. This result actually motivated us to propose the improved similarity function, IBC (defined in the section V-C), which uses both the BC and HD values. However, we give more weight to the BC values than HD values to calculate the IBC, as the BC ranks are slightly better than the HD ones in identifying the fault in the current version.

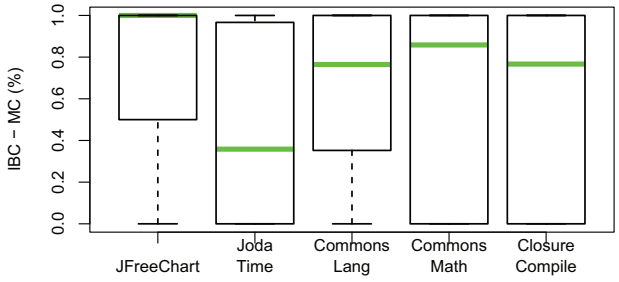
Fig. 7d shows the comparison between IBC and BC. As can be seen from the figure, IBC improves BC in the Joda Time project (p-values$<$0.05 and the distribution of effect sizes leaning more toward higher than 50%) and is as good as BC in the others. Therefore, we consider IBC as the best similarity function to be used for the similarity-based test case prioritization. Note that we did not report all combinations of comparisons between TM, HD, BC, and IBC, for the sake of brevity. However, given the results of RQ1 (HD performing better than TM) and RQ2 (HD performing worse than BC and



(a) Improved basic counting vs. Changed method coverage



(b) Improved basic counting vs. Size of tests



(c) Improved basic counting vs. Method coverage

Fig. 8. The boxplots of the effect size measures for finding the first fault using IBC, CMC, ST, and MC, when comparing 30 runs of each versions of each project.

IBC), one can conclude that BC and IBC perform better than TM, as well.

3) Experimental results for RQ3: In RQ3, we compare prioritization using the best similarity function (in this case IBC, based on RQ2 results) with the other traditional measures, e.g., Method Coverage (MC), Size of Testcase (ST) and Changed Method Coverage (CMC). These other traditional metrics have been explained in section III. To evaluate the results of RQ3, we also use the $\hat{A}_{12}$ measure distribution in a boxplot to compare two metrics and again we have considered the versions having at least 4 new/modified test cases for all projects.

The performance of the best similarity metric (IBC) against CMC, ST and MC is shown in Fig. 8. In general, IBC ranks the failing test higher than any of these metrics in more than 80% of the cases for three projects (JFreeChart, Commons Math and Closure Compiler). Also in Commons Lang project, IBC is significantly better than the MC, in 80% of the cases. However, the differences are not very significant (median effect size around 55-60% with p-values <0.05) when comparing IBC with ST or CMC in Commons Lang and Joda Time projects. Nonetheless, there is only one case that IBC falls behind and that is in Joda time when comparing IBC and MC, where the median effect size is around 40%.

The results also show that neither of these traditional metrics completely outperforms others for all the projects. However, the proposed IBC is much more consistent in all the projects.

E. Threats to validity

In terms of conclusion validity, we have conducted solid experiments to ensure that the results are statistically significant and the magnitude of differences are significant, as well (effect size).

In terms of internal validity, we have used existing libraries and tools as much as possible (e.g., Daikon [18], AspectJ [19]). However, one potential threat would be the impact of "method name change" in the new releases, when the body of the method does not change. In such cases, our method sequence-based similarity function is weak. In general, if one observes significant cases of "method name change" in the history, a finer grain representation of test cases (e.g., sequences of covered statements) is recommended. However, in our case studies, we did not experience any case where only method name changes (the total "method name change" statistics is 4 versions out of 247 versions=1.6%).

In terms of construct validity, we use a pretty well-known evaluation metric, which is the rank of first test that catches the fault. The other alternative would be APFD (Average Percentage of Faults Detected) [27], which is commonly used for test prioritization evaluation. However, in our study, each version contains only one fault. So, using APFD would not make sense.

In terms of external validity, we have conducted our empirical study based on five real-world open source java libraries from *defects4j* [16] database with several versions and faults. However, generalizing the results to different types of systems may still require further experiments.

VI. Related work

Test case/suite quality has been studied in different domains such as test case prioritization, test case generation, bug prediction etc. A number of quality metrics have been proposed so far in order to evaluate test case quality from these

perspectives. In the rest of this section, we mention some of the related works.

Nagappan *et al.* has proposed a set of 9 test quantification and complexity metrics and Object-Orientation (OO) metrics to evaluate Junit tests in terms of early estimation of software defects [28]. Test quantification metrics evaluate the tests by the amount of the tests (e.g., number of assertions or LOC) written to check the program thoroughly. We have also used the size of tests (i.e., LOC) and coverage (i.e., method coverage) for our comparison.

Shihab *et al.* studied the impact of change-metrics in terms of risk management of a software [29]. A change is considered risky when they might result in some faults in the future. The authors found the number of bug reports linked to a change and the faultiness of the files being changed as the best indicators of change risk. Moser also mentioned the number of bug fixes as one of the powerful fault-predictor process metrics [30]. He showed that the previous bug-fixing activities are likely to introduce new faults in the later releases. Similar to this, Zimmermann *et al.* [4] also showed that the number of past bug fixes extracted from the repository is also correlated with the number of future fixes. Anderson *et al.* [8] also showed the most frequent failures from the history is a good predictor for the future failures, however, the recent failures increased the predicting ability in their study compared to using the older failure history.

We consider two main findings from these studies. 1) The previously faulty source code are likely to have bugs again in the new release. This is the main motivation behind the historical fault detection metrics that we have explored in this study. 2) Changing a source code increases its chance to fail. So we have used this in terms of the "Changed Method Coverage (CMC)" metric for our comparison.

The most relevant studies to this work are those that use historical fault detection as a quality metric, in the context of regression test case selection, prioritization and minimization. Kim proposed to prioritize tests based on historical test execution that also improves the overall regression testing [31]. They considered the number of previous faults exposed by a test as the key prioritizing factor.

Elbaum *et al.* proposed a regression testing technique that use time windows to track from the history how recently test suites have been executed and revealed failures, to select and prioritize tests [6]. Park *et al.* considered the test case execution costs and the severity of detected faults to prioritize tests in regression testing [32]. However, they considered the total history of the test execution assuming that the costs of the test cases execution and the fault severity of detected faults can significantly change from one release and therefore, the complete history can further improve test case prioritization. Similar to this, we have also considered the complete history for the historical test execution data.

All these traditional historical fault detection measures prioritize the test cases that detect faults in previous releases. However, in our study, instead of looking into the exact matches with failing tests, we prioritize tests that are similar to the previous failing tests from the history.

VII. Conclusion and Future work

In the applications such as test prioritization, generation, selection etc., test case quality metrics are extensively used. Previous failing information of a test case is one of the existing quality metrics. The metric provides higher ranking to the test cases that failed in any of the previous releases. Higher ranking suggests that the test case has higher probability to detect more faults in the current release, as well. However, in practice, the fault revealing test in the current release may not be exactly the same as the previous failed tests, they might be similar though; specially when new tests are added to the existing test suite or old tests are modified. Therefore, we have proposed a similarity-based test quality metric that uses historical failure data. We have conducted a large empirical study (247 versions from five real-world java projects with real faults) that shows the proposed similarity-based metric is more effective in test prioritization compared to the other traditional metrics.

In the future, we will try to improve the similarity function by abstracting the method sequence calls into a state model. Moreover, we will also assess the impact of run-time parametrization on the test case inputs. We are also interested to build an automatic test generation tool that can generate high-quality tests, using this new quality metric.

Acknowledgment

The research of the first author is supported in part by a University of Manitoba Graduate Fellowship (UMGF), and the research of the second author is supported in part by the Natural Sciences and Engineering Research Council of Canada (NSERC).

References

[1] M. Pezzè and M. Young, *Software testing and analysis: process, principles, and techniques*. Wiley, 2008.

[2] P. Ammann and J. Offutt, *Introduction to Software Testing*. Cambridge University Press, 2008.

[3] M. D'Ambros, M. Lanza, and R. Robbes, "An extensive comparison of bug prediction approaches," in *7th IEEE Working Conference on Mining Software Repositories (MSR), 2010*. IEEE, 2010, pp. 31–41.

[4] T. Zimmermann, R. Premraj, and A. Zeller, "Predicting defects for eclipse," in *International Workshop on Predictor Models in Software Engineering, PROMISE'07: ICSE Workshops 2007*. IEEE, 2007, pp. 9–9.

[5] T. Noor and H. Hemmati, "Test case analytics: Mining test case traces to improve risk-driven testing," in *Software Analytics (SWAN), 2015 IEEE 1st International Workshop on*. IEEE, 2015, pp. 13–16.

[6] S. Elbaum, G. Rothermel, and J. Penix, "Techniques for improving regression testing in continuous integration development environments," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2014, pp. 235–245.

[7] S. Kim, T. Zimmermann, E. J. Whitehead Jr, and A. Zeller, "Predicting faults from cached history," in *Proceedings of the 29th International Conference on Software Engineering (ICSE), 2007*. IEEE Computer Society, 2007, pp. 489–498.

[8] J. Anderson, S. Salem, and H. Do, "Improving the effectiveness of test suite through mining historical data," in *Proceedings of the 11th Working Conference on Mining Software Repositories*. ACM, 2014, pp. 142–151.

[9] M. P, *Foundations of Software Testing*. Pearson Education, 2008. [Online]. Available: https://books.google.ca/books?id=yU-rTcurys8C

[10] G. Fraser and A. Arcuri, "A large-scale evaluation of automated unit test generation using evosuite," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 24, no. 2, p. 8, 2014.

[11] K. Sen and G. Agha, "Cute and jcute: Concolic unit testing and explicit path model-checking tools," in *Computer Aided Verification*. Springer, 2006, pp. 419–423.

[12] G. Dong and J. Pei, *Sequence Data Mining*, ser. Advances in Database Systems. Springer US, 2007. [Online]. Available: https://books.google.ca/books?id=GESJmZpkePIC

[13] D. Gusfield, *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology*. Cambridge University Press, 1997. [Online]. Available: https://books.google.ca/books?id=Ofw5w1yuD8kC

[14] H. Hemmati and L. Briand, "An industrial investigation of similarity measures for model-based test case selection," in *21st International Symposium on Software Reliability Engineering (ISSRE), 2010*. IEEE, 2010, pp. 141–150.

[15] H. Hemmati, A. Arcuri, and L. Briand, "Achieving scalable model-based testing through test case diversity," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 22, no. 1, p. 6, 2013.

[16] http://homes.cs.washington.edu/~rjust/defects4j/, [Online; last accessed 2-June-2015].

[17] R. Just, D. Jalali, and M. D. Ernst, "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 (ISSTA), 2014*. ACM, 2014, pp. 437–440.

[18] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao, "The daikon system for dynamic detection of likely invariants," *Science of Computer Programming*, vol. 69, no. 1, pp. 35–45, 2007.

[19] "The AspectJ Project," https://eclipse.org/aspectj/, [Online; last accessed 02-June-2015].

[20] "Eclipse java Development Tool (JDT)," https://eclipse.org/jdt//, [Online; last accessed 02-June-2015].

[21] "The R Project for Statistical Computing," http://www.r-project.org/, [Online; last accessed 02-June-2015].

[22] "R:Sample Ranks," https://stat.ethz.ch/R-manual/R-devel/library/base/html/rank.html, [Online; last accessed 02-June-2015].

[23] H. B. Mann and D. R. Whitney, "On a test of whether one of two random variables is stochastically larger than the other," *The annals of mathematical statistics*, pp. 50–60, 1947.

[24] A. Arcuri and L. Briand, "A practical guide for using statistical tests to assess randomized algorithms in software engineering," in *Software Engineering (ICSE), 2011 33rd International Conference on*. IEEE, 2011, pp. 1–10.

[25] K. J. Goulden, "Effect sizes for research: A broad practical approach," 2006.

[26] A. Vargha and H. D. Delaney, "A critique and improvement of the cl common language effect size statistics of mcgraw and wong," *Journal of Educational and Behavioral Statistics*, vol. 25, no. 2, pp. 101–132, 2000.

[27] G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold, "Test case prioritization: An empirical study," in *Software Maintenance, 1999.(ICSM'99) Proceedings. IEEE International Conference on*. IEEE, 1999, pp. 179–188.

[28] N. Nagappan, L. Williams, M. Vouk, and J. Osborne, "Using in-process testing metrics to estimate post-release field quality," in *18th International Symposium on Software Reliability (ISSRE), 2007*. IEEE, 2007, pp. 209–214.

[29] E. Shihab, A. E. Hassan, B. Adams, and Z. M. Jiang, "An industrial study on the risk of software changes," in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*. ACM, 2012, p. 62.

[30] R. Moser, W. Pedrycz, and G. Succi, "Analysis of the reliability of a subset of change metrics for defect prediction," in *Proceedings of the Second ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*. ACM, 2008, pp. 309–311.

[31] J.-M. Kim and A. Porter, "A history-based test prioritization technique for regression testing in resource constrained environments," in *Proceedings of the 24rd International Conference on Software Engineering (ICSE), 2002*. IEEE, 2002, pp. 119–129.

[32] H. Park, H. Ryu, and J. Baik, "Historical value-based approach for cost-cognizant test case prioritization to improve the effectiveness of regression testing," in *Second International Conference on Secure System Integration and Reliability Improvement (SSIRI), 2008*. IEEE, 2008, pp. 39–46.