2017

Improving testing performance by dynamic prioritization of tests based on method invocation orders

Sriram Balasubramanian
Iowa State University

Improving testing performance by dynamic prioritization of tests based on method invocation orders

by

Sriram Balasubramanian

A thesis submitted to the graduate faculty

in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Major: Computer Science

Program of Study Committee:

Samik Basu, Major Professor

Pavan Aduri

Carl K Chang

Iowa State University

Ames, Iowa

2017

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ACKNOWLEDGEMENTS

I would like to acknowledge the Dr.Samik Basu for his guidance in with this work, his patience with me and his excellent deductive process, without neither, no part of this work would have been possible. I would also like to thank Dr.Ganesh Ram Santhanam for his valuable insight in analyzing results and creative ideas that propelled this work in the right direction.

ABSTRACT

We present a dynamic test prioritization technique with the objective to speed up uncovering updates to existing software and therefore, increase the rate at which faulty software can be debugged. Our technique utilizes two types of data—the results of executing tests on prior version of the software; and the results of executing tests on the new version which determines the next test to be executed.

The contributions of the thesis are two-fold: understanding what constitutes an effective ordering of tests and developing an algorithm that can and efficiently generate such order.

At its cores, the proposed dynamic ordering technique relies on two basic conjectures. Firstly, tests that are closely related are likely to uncover similar updates/faults and tests that are not related are likely to widen the search for updates/faults. In other words, if a test uncovers updates in a software, i.e., its execution behavior (in terms coverage) differs considerably between prior and current version of the software, then selecting a test closely related to it is likely to be beneficial. Similarly, if a test does not uncover updates in a software, it would be good to select an unrelated test to execute next to increase the chances of better coverage. The relationship between tests are determined from the execution of tests while testing prior versions of the software. The second conjecture is that selecting tests in the above order will speed up uncovering bugs in the software.

We develop a baseline ordering using complete knowledge about the results of executing tests in two different versions of the software. The baseline ordering arranges the tests in descending order in terms of amount of changes the tests uncover between the prior and new version of the software. We evaluate the effectiveness of this ordering (i.e., the validity of the conjectures) by computing the rate at which the order can identify (seeded) bugs in a software– the measurement is referred to as `APFD`. The baseline order produces high `APFD` values indicating that the order is indeed effective. However, note that the baseline ordering can be only obtained

if the tests are already executed in two versions of the software; the challenge is to identify the ordering before executing the tests on the version being tested.

We have developed an algorithm that estimates the baseline ordering. We evaluate the quality of the estimates using a rank relationship measure refer to as Order-Relationship Measure (`ORM`). We find that the `ORM` is high when call-sequences resulting from executing tests are used for estimation. We also find that low `ORM` implies low `APFD` values for the estimate. We have evaluated our algorithm on two non-trivial software repositories. We have investigated the role of two important parameters (thresholds capturing the closeness relationship between tests) in identifying high quality (high `APFD`) ordering and outlines how these parameters can be statically determined based on executing tests on the prior versions of the software. Finally, we have showed that the application of our algorithm in generating the test orders dynamically has close to 3% overhead.

CHAPTER 1. INTRODUCTION

1.1 Background

Most software are built iteratively where it starts off as a basic working model and additional features are added over time. As the software undergoes change, it is vital to ensure that the changes do not introduce any negative consequences ranging from unintended modified behavior of existing functionality to out right execution breaking bugs [1] that causes the program to crash. For this purpose, a suite of tests grouped into test cases are maintained and these tests are executed upon every change to ensure sanity of the software repository. Each test would validate a certain behavior and collective execution of all of them would ensure that the software is free of any predictable bugs.

There are several unit testing frameworks, and each of them have their own behavior when it comes to ordering the execution of the tests. Eg. MSTest for Visual Studio have no guarantee when it comes to execution order [2], JUnit 3 [3] for Java runs tests in the order in which JVM Reflections API returns the test. Some software repositories have a very rigorous series of tests, or tests that are complicated or intensive and the end result would be they involve a lot of time or effort to execute the entire test suite. In some cases, it is impractical to run the entire regression test suite for every change. In such cases, prioritizing the execution of tests such that discovery of bugs sooner than later is extremely beneficial. Even in cases where all tests can be run in a reasonable time, it would save time in finding bugs sooner.

In an iterative development model, it is a reasonable assumption that any new errors introduced would be rooted in or at least associated to newly added changes. Using that assumption as a base, our work attempts to order tests such that sections of the software that are more

[1] The terms bug, fault, exception and error are will be interchangeably throughout the course of this paper to indicate negative unintended behaviour of software

likely to have changed are higher in the execution order than sections that are less likely to change. The ordering takes into consideration the set of tests, the measure of their impact of applying them on some prior version of the software and the result of applying the tests in the version being tested. For instance, let the measure of impact of tests on a version be defined in terms of sequence of function-calls executed by the tests. Tests with similar measures are similar or are closely related–indicating that they *test* similar artifacts or feature of the software. Next, we start with some test, say t, and deploy it in the new version, the version being tested. We take into consideration two aspects, the measure of impact of t in the new version and the result, whether or not the test passes (produces output as expected). If the measure of t in the new version is markedly different from its measure in prior version or it does not pass, then it is likely that t has been able to identify parts of the new version that resulted from *considerable* updates to the prior version of the software. In this case, the next test we consider is one that is most closely related to t; otherwise, we may consider a test that is least closely related to t (as is typically done to increase coverage measures [4]). In short, the ordering in which the tests are executed are decided dynamically based on the past information about the tests and new information obtained as the tests are executed on the version being tests. Two challenges need to be addressed carefully before this method can be deployed in practice: (a) identify the appropriate way to measure impacts, and (b) compute the closeness relation based on the measures efficiently to keep the overhead of finding the next best test execute low. We address these challenges by considering simple but effective heuristics to measure the impact and closeness relation.

1.2 Motivating Example

In the rest of the thesis, we will rely on Apache XML Security software repository [5] to motivate and illustrate the necessity and effectiveness of our method. For instance, when a few faults were seeded by the Software-artifact Infrastructure Repository in Apache XML Security repository Version 1.0.71, running the tests without any change in the default defined order resulted in only 0.18 `APFD` [1] detected. This means that, if we measure the amount of undetected faults discovered against the fraction of the tests run and take an average of the

values, each percentage of the tests only cover 18% of the total faults on average. In contrast, when we apply our method, we were able to obtain APFD value between 0.69-0.80 depending on the first test being used.

1.3 Problem Statement

Given a Repository, a set of tests and the call sequence of each of those tests from the previous version of that repository, we would like to output a sequence of test order, which is dynamically generated based on difference between call sequences among various tests in the previous version and the difference between the call sequences of the same tests between the two versions. The objective to find and prioritize the execution of tests, testing sections of the repository that has changed. The end goal of this ordering is to speed up the detection of faults by forcing tests that cover new or modified sections of the repository to run earlier.

A fault that cannot be discovered by any test is outside the scope of Test Prioritization.

1.4 Contribution

Our contributions include

- A method for correlating (closeness measure) tests based on the call sequences realized by the test execution. The correlation can be measured by treating call sequences as sets to find set difference, or by weighted matching among call sequences.
- An algorithm to dynamically order tests based on the similarity between them and the result of the execution on the test subject.
- A modular framework that allows pluggable interface with regards to similarity measures and dynamic ordering logic. This is an important aspect of the thesis as it allows for future extension and comparison between newly developed methods for test case ordering.
- An evaluation of the test correlation and the dynamic ordering on different versions of software. Note that, the better test correlation is likely to have better dynamic ordering, which, in turn, will lead to better `APFD` values. However, for such a dynamic ordering to

yield practical value, it must be done without prohibitively high overhead in correlation measure and dynamic ordering.

1.5 Organization of Thesis

The rest of the paper is divided in the following sections. In Section 2, we look at the work done so far and list their motivations and shortcomings. In Section 3, we explain our heuristic to determine test similarity and algorithm to prioritize tests using similarity values. Section 4 describes the test repositories and presents the results of empirical evaluation on different repositories. Section 5 summarizes the entire work and talks about any possible future work.

CHAPTER 2. RELATED WORKS

Improving of testing performance is typically done through three ways - *minimization*, *selection* and *prioritization* [6]. Test Minimization aims to identify and remove tests that are redundant. Test Selection focuses on selecting the smallest subset of tests that fulfill certain testing requirements. Test Prioritization aims to reorder the test execution to speed up fault detection. Among the three optimization methods, our work would focus on *Test Prioritization*.

Rothermal et al. [4] formally defined Test prioritization problem as

Given the set of permutations PT of a test suite T and a function $f : PT \to \mathbb{R}$.

Problem: Find $T' \in PT$ such that $\forall T''$ where $T'' \in PT$ and $T'' \neq T'$, $f(T') \geq f(T'')$.

The above definition could be applied to both Test Prioritization and Test Case Prioritization (a test case would be a grouping of tests that test a common functionality).

2.1 Average Percentage of Faults Detected (APFD)

As the goal is to maximize the rate of fault detection, Rothermal [4] proposed a metric called `APFD` to quantify the speed at which faults are detected. The *APFD* would be a value between 0 and 1, and a higher value indicates a higher rate of fault detection.

There could be any number of faults, and every test could discover a number of faults ranging from none to all of them. While there can also be faults that are not discoverable by any test, such faults are outside the scope of Test Prioritization.

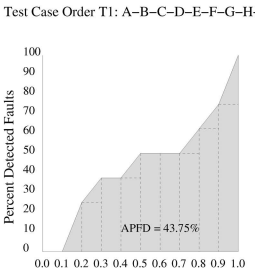
If there are n tests and m faults, Equation 2.1 gives the formula to calculate `APFD`

$$\texttt{APFD} = 1 - \frac{(TF_1 + TF_2 + \cdots + TF_M)}{mn} + \frac{1}{2n} \tag{2.1}$$

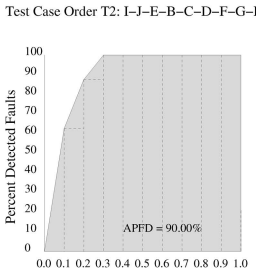
where TF_i is the position in which fault i was discovered.

test	fault							
	1	2	3	4	5	6	7	8
A								
B	x	x						
C	x	x	x					
D		x	x					
E	x							x
F								x
G		x						
H				x				x
I	x	x	x	x	x			
J					x	x	x	

(a) Sample repository



(b) APFD for test order A-B-C-D-E-F-G-H-I-J



(c) APFD for test order I-J-E-B-C-D-F-G-H-A

Figure 2.2: APFD example from [1]

Figure 2.2 [1] illustrates how `APFD` is calculated. Figure 2.1a gives a sample repository with 8 faults (1 to 8) and 10 tests (A to J). Each fault can be discovered by one for more tests. For instance, Fault 1 can be discovered by Tests B,C,E and I. If the execution order is A-B-C-D-E-F-G-H-I-J, then $APFD = 1 - (2+2+3+8+9+10+10+5)/(10*8) + 1/(2*10) = 0.4375$ (Figure 2.1c). Figure 2.1b shows us that the execution order I-J-E-B-C-D-F-G-H-A has a higher `APFD` value of 0.9 as all faults are discovered with 30% of the tests are executed.

2.2 Coverage Based Methods

When Rothermal et al. [4] proposed `APFD`, it was to perform a comparison of various coverage based prioritization methods [7]. Coverage metrics used are statement coverage where the goal is to prioritize tests in an order that maximizes the rate at which statements are covered, and function coverage where the prioritized order maximizes the rate at which functions are covered. For these coverage metrics, the techniques used are *total* [8], which orders tests from highest coverage to lowest, *additional* [8], which orders tests such that the incremental coverage is maximized, *fep*, where mutation testing is used to come up with a fault-exposing-potential for each test, *fi*, where Principal Component Analysis is used to come up with a probability of fault occurring in each test based on their history of fault-proneness. *fep* and *fi* are methods introduced by the authors here. The authors evaluate the methods empirically, and present an analysis of the results which suggest that *additional* techniques provide the best performance and fi methods provide the best results.

Elbaum et al. [1] expand on the previous work by providing a categorization of the previously mentioned techniques into *general* and *version-specific* prioritization. General Prioritization aims to improve the rate of fault detection multiple versions, whereas Version Specific techniques aim to maximize fault detection for a specific version. The authors also introduce a new Prioritization technique *DIFF*. In DIFF, a delta difference of the changes in each file is generated and is used as the criteria to perform prioritization on. The authors also expand upon the empirical comparison by using new case studies.

2.3 Improvement Over Random Testing

Random testing [9] involves picking test inputs randomly and independently from the input domain of the software. Chan et al. [10] showed that failures occur in patterns across the input domains. These patterns in n-dimensional space could be *point*, where failure causing inputs are points that are equally distributed across the domain, *block*, where the failures are located in a block around a single point, and *strip*, where the failure causing inputs form a contiguous strip. Chen et al. proposed *Adaptive Random Testing (ART)* in [11] for generating test cases, which aimed to improve upon random testing by using the failure pattern information. Each test case input generated would be such that they are distributed equally from a random starting point so as to maximize the chance of finding faults.

Jiag et al. [12] applied the *Adaptive Random Testing* to Test Case Prioritization. Execution would begin from a random test. A candidate set would be drawn by randomly selecting tests from all unexecuted tests, and next test to be executed would be the test in the candidate set that is farthest away from all executed tests. A variation of this testing would be that, if a fault is found, the next test would be the test closest to the failed test in the candidate set. The similarity between tests would be measured by string similarity measure between the test codes. For instance, if *testA* and *testB* are compared, then, the measure of how similar they are would be some string difference measure between the source code of the two tests.

2.4 Use of Method Call Information

When a test suite for a program is executed, each test would call one to any number of methods in the Program. Each of those methods would further call more methods. A compilation of all such calls can give a good idea about the regions of the program executed. If this method call information is obtained statically without running the program, all calls in loops and conditional statements will have be considered as there is no way to determine the status of a condition or a loop until the program is executed. This would lead to a *Call Graph* which would have information on all possible methods that can be called by a certain method. Zhang et al. [13] uses such static call graphs to perform prioritization. The authors explore the call graphs of each tests to generate a *Testing Ability (TA)* score for each test. The tests are finally run in the descending order of their TA.

If such a method call information is obtained dynamically when executing the tests, we would get *Call Sequences*. The call sequences would give accurate information on exact methods called including arguments used and depth of recursion. However, to use call sequences for prioritization, it has to done during run time.

2.5 Other Prioritization Techniques

Various search algorithms have been used to perform Test Prioritization. The algorithms used include Genetic Algorithm [14, 15], Hill-Climbing [15] and Mutation [15]. In case of these techniques, a fitness function would be defined that evaluates the rate at which faults are detected and these algorithms would work towards maximizing the fitness function.

2.6 Classification Based on Information Used

Luo et al. [16] classified various existing Test case prioritization techniques into *Static Techniques* and *Dynamic Techniques*. Static Techniques perform prioritization solely using information about the tests and repository that can be obtained without executing the tests. These includes using static call graphs of tests obtained through static code analysis, calculating similarity between various test cases using string-edit distances such as Hamming or Levenshtein

distance and prioritizing execution of dissimilar test cases, or assigning topic-identifiers to each tests and distributing testing evenly across topics.

Dynamic Techniques require information that can only be obtained by executing the tests. These information can be used in addition to information that has been obtained statically. The various Greedy Coverage Based Methods from Section 2.2 would fall under Dynamic Techniques as the tests have to be run in order to obtain the coverage information. Other dynamic techniques would be *Adaptive Random Testing*, where the next test to be executed would be the closest or farthest test from a set of randomly selected tests, and tests that uses search algorithms on tests to maximize fault detection.

The authors perform empirical evaluation using 30 programs on selected techniques from both categories to compare efficiency and performance. They also compare the effect on granularity by evaluating all techniques on both test-case level (where test cases are prioritized and every test inside a test case is executed when a test case is executed) and test level (individual tests are prioritized independently).

The comparison between static and dynamic techniques show that each technique performed differently in different situations with respect to APFD and no technique consistently outperformed the other. The authors conclude that more work is needed to establish exactly when each of those techniques are effective. On the other hand, the authors seem to agree that the finer test level granularity performed better than test-case level granularity. The results suggest that Static techniques perform better at test-case level granularity whereas Dynamic techniques perform better at test level prioritization.

2.7 Our Work

Call sequences provide a very detailed description on the execution of a method. While call graphs [13] have been used before for test prioritization, they have only been used with static call information. This would not give the complete picture as not all method calls in the call graph would be executed and some methods could be called more number of times than others. The exact efficiency of Dynamically Obtained Method Call Sequence has not been explored and we believe that this is a perfect area of research. As the prioritization

can be done on the fly, given an efficient means to obtain the call sequence of a test, a quick analysis of such call sequence can help identify the next best test to execute with reasonable degree of confidence. Furthermore, if call sequence information from previous version of the repository can be processed ahead of type, it could be combined with other statically obtainable information that can be used during prioritization, so that the amount of information available when selecting the next test to execute is high.

While works in Prioritization operate on both Test and Test-case level, there is usually no clear reason for the choice of granularity and there are comparisons [16] that show that the finer granularity of Test level performs better. In our work, we would operate on both levels of granularity. As test cases are functional groupings of tests, we would use tests to identify test cases to execute next, and inside a test case, we would perform test-level comparisons to decide the next test to be executed.

The concept of closeness among tests have been used by Jiang et al. [11] in *Adaptive Random Testing*. However, only closeness between test code is considered. As a same routine can be written in two different ways, there is a possibility that the distance measure is not accurate, especially in a large software, written by several people. For instance, if two tests simply differ by the value of one variable, they would have a very high similarity value, but, it is possible that that variable may cause an entirely different section of the code to be executed. We would try to compare the way in which tests operate by comparing two tests using their call sequences, so that we get an accurate comparison in terms of code sections covered by the test.

CHAPTER 3. DYNAMIC TEST ORDERING

We define a Test Ordering as the sequence in which all tests in a test suite are to be executed. The Default Ordering is the sequence in which the tests would be executed if the ordering is to be done by the testing framework. The Prioritized Ordering would be a sequence of execution, rearranged toward to achieve some goal (e.g., to maximize APFD [1]).

Recall that, we are considering the ordering of tests in the context where the same set of tests are being used to evaluate the (predicable) behavior of two different versions V_1 and V_2 of the same software (V_2 being the newer version of the software obtained from or extended from V_1). Broadly speaking, the (prioritized) order is of two types: *static* and *dynamic*. In case of static ordering, the information used in the ordering is solely obtained from the properties of V_1 and the result executing the tests on it; for instance, execution paths/call-graphs of the tests executed on V_1 can be used to order the tests to be executed on V_2. As the name suggests, static ordering remains unchanged as the tests are executed on V_2. Dynamic ordering, on the other hand, uses dynamically derived information (result of executing tests on V_2) in tandem with statically available information available prior to test execution (result of executing tests on V_1). For instance, success or failure of running the tests on V_2, coverage information can be used to find the ordering that is best suited for V_2. Dynamic ordering, therefore, is likely to change the ordering in which tests are deployed on-the-fly.

We believe that dynamically obtainable information on test runs is highly valuable, and present a method for dynamic ordering technique. In this context, we also understand that any operation that involve gathering these dynamic information must be reasonably fast, involve negligible overhead and must be improve upon the static ordering techniques. We focus on a dynamic test ordering technique that is efficient and is effective in exploring the "new sections"

of code in new version V_2 (new code are likely to house the bugs that may have been introduced in V_2). In the following section, we present such a dynamic test ordering technique.

3.1 Method

Our ultimate goal is to identify an ordering that maximizes the speed in which faults are discovered. The input to our method is the version of a repository (*current version*: V_2) for which we would like to maximize the speed of fault detection, the *previous version* (V_1) of the same repository where there are no faults discover-able by the test suite, the call sequences of all tests from the previous version and a means to obtain the call sequence of each test in the current version right after the test is run. As the tests t_i are executed on V_2, our method computes the next best test t_{i+1} to execute using the above information—thus generating a order of tests being executed dynamically. The efficiency of the prioritized ordering will be measured in terms of maximizing the rate at which faults are detected.

We will use the difference between tests to determine the next test in the prioritized order. In particular, if a test t_i, when executed on version V_2, passes, then the tests that are "farthest" from t_i are likely candidates for t_{i+1}. The degree of being farthest can be quantified based on coverage metrics or line numbers being executed by the tests. The intuition behind such a choice is to maximize the chance of faults by choosing tests that will widen the region being tested at each step. This idea is exploited effectively by Chen et al [11] as *adaptive random testing*, and has been adopted widely [17, 18, 19, 20]. In this thesis, we take this one step further: if a test t_i, when executed on version V_2, covers new code fragment of V_2, then the tests that are "closest" to t_i are considered as likely candidates for t_{i+1}. The intuition is that tests that are similar are likely to expose similar behavioral changes in V_2 (in comparison to V_1). We will show that closeness consideration improves the detection of faults (if one exists).

We operate on the call sequences of each tests as a way to measure the closeness/difference between tests. The difference between two tests thus reduce to quantifying the difference between two lists of Strings (i.e sequence of method calls). This gives us different possible ways of quantifying the method calls, such as, considering or ignoring repetition, considering or ignoring if each position of the list has the same method call. It is important to note here that

the closeness/difference between tests are measured from the result of their executions in V_1 of the software (as V_2 is being tested and closeness/difference between two tests can be quantified only after they are both executed). Our conjecture is that this measure is a good estimate of their measure in V_2 as well.

It is important to note that as change information is unavailable on newly added tests, such tests cannot be effectively prioritized by our technique until a later version of the repository.

3.1.1 Measuring Test Difference

The difference between tests are measured in terms of the method calls executed by the tests. It is important that such measures can be efficiently measured to minimize the overhead in dynamic ordering (see Section 3). We present two methods of measurement.

Set Difference. For call sequences S_1 and S_2, Set Difference between them is defined in Equation 3.1

$$d_{set}(S_1, S_2) = |S_1 - S_2| + |S_2 - S_1| \tag{3.1}$$

where $|S_i - S_j|$ is the cardinality of the set difference between the set containing the calls in S_i and the set containing the calls in S_j. Note that, the ordering between the method calls are discarded in this measure.

Positional Weighted Difference. For call sequences S_1 and S_2, if at a position i, the method call in S_1 is different from that in the same position in S_2, then the measure of this difference is quantified to be inversely proportional to i. In essence, differences towards the beginning of the sequences has more impact than the differences towards the end.

The positional weighted difference between S_1 and S_2 is defined by Equation 3.2.

$$d_{pos}(S_1, S_2) = \sum_{i=1}^{n} f(S_1(i), S_2(i)) \times (n - i) \tag{3.2}$$

Where the function f is

$$f(l_1, l_2) = \begin{cases} 0 & \text{if } l_1 = l_2 \\ 1 & \text{otherwise} \end{cases} \tag{3.3}$$

and n is the length of the longer sequence.

We will use d_* to indicate either d_{set} or d_{pos}.

3.1.2 Outline for Prioritized Ordering

Given a set of tests $T = \{t_1, t_2, \ldots, t_k\}$ and two versions V_1 and V_2 (V_2 being the current test subject). We measure the closeness/difference between all pairs of tests. The complexity is $O(k^2 \times n)$, where n is the longest sequence associated with the tests; however, this does not contribute to runtime overhead as it will be computed statically. Then, we start with some test, say t_i. Consider that the S_i is the call sequence associated with executing t_i in V_1. On executing t_i on V_2, the call sequence obtained in S_i'. If V_2 does not pass t_i or $d_*(S_i, S_i')$ is greater than some pre-specified threshold `STT` (closeness-threshold), then we conclude that t_i has been able to explore some code fragment/dependencies that are new in V_2. In that case, the next test t_j to consider is such that $d_*(S_i, S_j)$ is less than some pre-specified threshold `DTT` (difference-threshold), where S_j is the call sequence in V_1 associated with test t_j. This will help in identifying the tests that are likely to explore the parts of code that have been discovered by t_i to be new in V_2. If, on the other hand, $d_*(S_i, S_i')$ is less than the `STT`, the the next test test t_j to consider is such that $d_*(S_i, S_j)$ is maximum among the tests still to be executed in V_2. This is likely to widen the search for new pieces of code in V_2. In addition to base the method on closeness/difference measures, the candidates for choosing the next tests also depend on the test case to which they belong as tests cases naturally classify the tests in terms of the test-objectives (features being tested). We will discuss the details of the algorithm in Section 3.3. The computation (of the order $O(n)$) of $d_*(S_i, S_i')$ contributes to the runtime overhead of the method.

3.2 Data Structure

We use a difference matrix to store the difference between all the tests in a version in order to facilitate quick lookup of $d_*(S_i, S_j)$ for $i \neq j$. For n tests, the matrix is a $n \times n$ matrix. The entry a_{ij} corresponding to the i-th row and j-th column denotes $d_*(S_i, S_j)$

$$a_{ij} = \begin{cases} 0 & \text{if } i = j \\ d(S_1, S_2) & \text{otherwise} \end{cases} \tag{3.4}$$

As the difference property is symmetric, $d_*(S_j, S_i) = d_*(S_i, S_j)$—the matrix is upper-triagular. For instance, Table 3.1 represents the organization of a Difference Matrix, and in order to find the tests closest to $test01$, we would traverse the first column and find the test with the smallest difference value other than the test itself.

Table 3.1: Organization of section of Difference Matrix for Positional Weighted Difference for XML Security Repository

Tests	test01	test02	test03	...	testBad01	testBad02	testBad03
test01	0	53	75	...	233	124	563
test02	0	0	88	...	12	69	91
test03	0	0	0	...	62	58	800
...	...	...	...	...	...	...	...
testBad01	0	0	0	...	0	4	576
testBad02	0	0	0	...	0	0	758
testBad03	0	0	0	...	0	0	0

3.3 Algorithm for generating a prioritized test order

Recall that in Section 3.1.2, we have outlined the basic intuition for generating prioritized order of tests—tests that are *closely* related are likely to uncover similar updates to the software and tests that are *not closely* related are likely to widen the search for updates to the software.

It is worth noting that in all practical scenarios, tests are organized in the form of test cases, and each test case is categorized based on the functionality of the software that is being tested by the tests in the test case. For instance, there can be n test cases: $TC_1, TC_2, \ldots, TC_n$, and each TC_i corresponds to some functionality F_i of the software and contains tests $t_{i1}, t_{i2}, \ldots, t_{im}$. Therefore, in our algorithm it is important to also take into consideration the test case information to which the tests belong—tests in the same test case are likely to be closely related. The following steps constitutes our algorithm:

1. Start with test t_i in test case TC_i

2. If the difference between the result of execution of t_i between the current version and old version is above `STT` then

(a) add t_i to the core set of tests that are able to find differences between software versions so far

(b) find t_i' in TC_i that is closest to the core set of tests, if no such test exists, look for some t_j in a different test case TC_j that is closest to the core set, and repeat from step 1.

3. Otherwise,

   (a) if core set of tests is empty, i.e., t_i is the first test that is executed on the current version, then

      i. find t_j in some other TC_j that is furthest from t_i and repeat step 1

   (b) Otherwise,

      i. find t_j in some other test case TC_j that is closest to the core set of tests and repeat from step 1.

4. If no such t_i' or t_j can be obtained, then a core set of tests are reset to empty set, and the Step 1 is re-started with another (randomly selected) test t_k from a test case TC_k.

Figure 3.1 illustrates the above steps. The dotted lines represents nearest tests whereas the solid lines represent farthest tests. The numbers on the lines indicate the point in time in which the nearest test or the farthest tests are calculated. At time 1, the algorithm finds the starting test to have not changed significantly and looks for the farthest test. At time 2, having still not discovered any tests with enough change, the algorithm looks for the farthest test from both tests discovered. At time 3, the farthest test search yields a test with enough change. More tests with enough change are discovered by looking for closest tests at time 4 and 5. The nearest test search at time 6 yields a test without enough change, so at the next step, the algorithm looks for a test with enough change in a different test case.

Note that, a test is added to core set if and only if it uncovers enough changes[1] to the software. One can consider different variants of conditions that indicates that a test is a good candidate for the core. The simplest variant is presented above: one that produces changes

[1]Uncovering a bug will also constitute enough change.

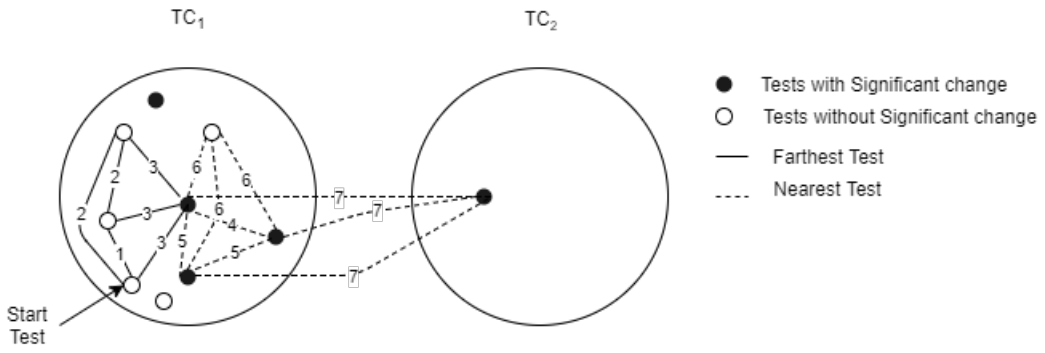


Figure 3.1: Illustration of the Algorithm 3.1

that can be quantified to some value above a pre-specified threshold `STT`. Another variant can be a conjunction, where one conjunct is the same as above, and the other conjunct captures whether there exists a different test t' that is close to the current test such that that t' also uncovers enough changes in the software. We refer to this variant as the *strict* variant, as it is likely to generate relatively small core sets.

The core set is generated incrementally till no new tests can be added to it. That happens when tests from all test cases that are close to the core set has been selected and executed. We refer to this phenomenon is *core-saturation*. Formally,

$$\neg \left[\exists TC_i.\ \exists t_i \in TC_i.\ \sum_{t_j \in \texttt{Core}} d_*(S_i, S_j) \leq \texttt{DTT} \right]$$

where S_i, S_j are the execution sequences resulting from tests t_i and t_j in the previous version, and `Core` is the core set that is saturated. The above presents the condition under which no new tests can be added to an existing core set. If the above condition is violated, then there exists some candidate tests that can be considered to be added to the core. Specifically, when the candidate test uncovers enough changes.

Algorithm 3.1 presents the details of our algorithm. It takes as input the difference matrix capturing the $d_*(S_i, S_j)$ (see Section 3.2), the test suite containing information about all the test cases and tests, the thresholds `STT` and `DTT`, a test to start execution and a parameter that defines the strictness for considering a test to be relevant.

Algorithm 3.1: Prioritizer

Input: Difference Matrix `D`, Test Suite `S`, Start test `st`, Integer `STT`, Integer `DTT`, Boolean `isStrict`

Output: Prioritized Order O

```
1  O = ""
2  currentTest = st
3  coreTests = {st}
4  testsDiffered = false
5  while Not all tests in S has been executed do
6      O = O · currentTest
7      c = execute(currentTest)
8      if c ≥ STT then
9          if testsDiffered = false then
10             testsDiffered = true
11             coreTests = currentTest
12         end
13         closestTest = D.getClosestToSetInTestCase
           (coreTests, currentTest.testCase, (DTT * coreTests.size))
14         if isStrict ≠ true then
15             coreTests = coreTests ∪ currentTest
16         else
17             if execute(closestTest) > STT then
18                 coreTests = coreTests ∪ currentTest
19             end
20         end
21         if closestTest is not empty then
22             currentTest = closestTest
23             continue
24         else
25             currentTest = findNextTest(coreTests,S,testsDiffered)
26         end
27     else
28         currentTest = findNextTest(coreTests,S,testsDiffered)
29     end
30     if currentTest is empty then
31         coreTests = ∅
32         testsDiffered = false
33         currentTest = D.getRandomUnexecutedTest()
34         coreTests = coreTests ∪ currentTest
35     end
36 end
37 return O
```

Algorithm 3.1: Prioritizer - Continued

```
38 Function findNextTest(coreTests,S,testsDiffered):
39     candidates = D.allUnexecutedTests()
40     for each test t in coreTests do
41         suite = D.getTestsInSuiteOfTest(t)
42         candidates = candidates - suite
43     end
44     if testsDiffered then
45         test = getClosestInSetFromTests(candidates,coreTest)
46     else
47         test = getFarthestInSetFromTests(candidates,coreTest)
48     end
49     return test
50 End
```

We will assume that we have a method *execute*(), that takes a test as a parameter, runs the test and returns amount that test has changed compared to its run in the previous version, a method *sort*() that takes a sequence of tests, a criteria to sort and a target test, and returns a reordered sequence that is sorted on the criteria provided based on their distance to the target test.

The Difference Matrix d, will also have helper methods and it can return all tests, all tests in the order in which they were executed in the previous version(default order), and tests by certain test cases.

Lines 1 to 4 is the initialization of the Prioritizer. The `currentTest` is set to the input st and `coreTests` is initialized with the same. A flag `testsDiffered` is initialized to $false$. This flag would be used to determine if the next test to be found is to be the farthest test or nearest test to the `coreTests`. We would look for the farthest test until a test that has changed more that `STT` is found, after which we would look for the nearest test. O is the execution sequence that would be returned.

The while loop at line 5 is the core loop of the Prioritizer and would execute until all tests have been run. In line 6, we add the `currentTest` to O and execute the test in line 7. The *execute*() method would run the test, obtain the call sequence of the test, and return

the difference between the call sequence based on the difference measure used(Set difference or Positional weighted).

The if loop at line 8 checks if the test has changed more than the threshold `STT`. If the test has not changed, we find the next test to execute at line 28 by looking for either the closest test or the farthest test from the `coreTests` depending on if the flag `testsDiffered`. If the test has changed, the algorithm would proceed with using the test information to identify the next test to be run. The if loop at line 9 would set the `testsDiffered` flag. This would signal the `findNextTest` method that a valid starting point with enough change has been found, and further tests could be identified by looking for those tests closer to the `coreTests`.

`getClosestToSetInTestCase()` would give us the next closest test to `coreTests`, that belong to the same test case as `currentTest`. The test must also be close enough to `coreTests` by a factor that is `DTT` times the size of `coreTests`. It is necessary for the threshold to scale here as the size of the `coreTests` can vary between 1 and the size of the test suite. If the prioritizer is run in *strict* mode, then the `closestTest` must also have changed enough for the `currentTest` to be considered a part of `coreTests`. The stricter requirement is that, in addition for the test to have changed significantly, it must also lead to more tests that have changed significantly to be considered important. This check is done at line 17. If a `closestTest` was discovered, then `closestTest` becomes the `currentTest` and the while loop at line 5 is repeated. If not test was found, then there is either no test that is close enough in the test case or all tests in the test case have already been executed. In that case, we find a test in a test case that has not been covered at line 25.

If the `currentTest` at line 30 is empty, that would mean that at least one test from all test cases have been executed and there are no new test case left. In this case, we reset the `coreTests` and begin anew by selecting a random unexecuted test at line 33 and setting it to be the current and core test. The randomization would help distribute the execution by selecting a different starting point for the next iteration.

The method `findNextTest` would be used to find a test that is closest to the `coreTests` from a whole new test case. We obtain all unexecuted tests at line 39, and remove all tests from test cases covered in lines 41 and 42. Now, depending on if a valid starting test has been

located, we find either the closest test or the farthest test from the `coreTests` as shown in the if block at line 44.

CHAPTER 4. EXPERIMENTAL EVALUATION

In this section, we explain our choice of test data, list the metrics we used to evaluate the ordering and the motivations behind the metrics, and we also show how the algorithms stack up against the ultimate goal of speeding up fault detection. The evaluation includes two parts, firstly we define a baseline ordering which is the ideal ordering for maximizing the prioritization of executing changed sections of the repository first and secondly, we measure the algorithm's `APFD` which shows how quickly the prioritized order detects faults. We also present a Correlation measure between Test Orderings that only penalizes later occurrences and not earlier occurrence of a test.

4.1 Test Data

As mentioned in Section 1, Ordering of tests to speed up error detection is extremely desirable in cases where software grows over iterations and requires validation in each step to ensure no unintended behaviors are introduced. In order to evaluate the algorithms on a representative sample, we used two repositories, `ant` and $\texttt{xml} - \texttt{security}$ from Software-Artifact Infrastructure Repository [21].

Software-Artifact Infrastructure Repository holds a curated set of Programs for use in experimentation with testing and analysis techniques, and provides materials facilitating that use. The Repository has Programs in several versions, with those having faults that are from real world, faults that are seeded and faults that deal with concurrency. These Programs may have unit tests built in that may be used to detect faults.

`ant` repository is a Java-based build tool supplied by the open source Apache project. The repository has 8 versions with 80500 LOC. The faults in the repository are seeded and has unit tests built into it using the JUnit Framework.

`xml − security` is a component library implementing XML signature and encryption standards, supplied by the XML subproject of the open source Apache project. The repository has 4 versions with 16800 LOC. The faults in this repository too are seeded and has unit tests built into it using the JUnit Framework.

Both `ant` and `xml − security` are actual open source software built collaboratively and are used in real world. This makes them an ideal test bed for the prioritizing algorithms.

4.2 Objectives

Our end goal is to maximize the rate of fault detection by Dynamically Prioritizing Test execution. We will detail our progress towards that goal by showing how we measure the rate of fault detection, what our algorithms work towards, how well our algorithm achieves it's goal and how that translates towards speeding up fault detection.

1. To show the relation between our intermediate goal of prioritizing testing of changed code and maximizing fault detection, we would evaluate the `APFD` of the Baseline Ordering.

2. The Prioritizer algorithm attempts to order tests such that any test that has changed is to be run before those tests that have not changed, and tests that have changed more must be executed before tests that have changed less. To evaluate this, we would be comparing the result of the Prioritizer (Prioritized Order) against the best possible ordering of tests (Baseline Ordering) where section of code that has changed more are tested before sections of code that have changed less.

3. As our end goal is to speed up fault detection, the ultimate objective would be to maximize the `APFD` of the Prioritized Order. To that end, we would be comparing the `APFD` of Prioritized Order against the Default Order.

4.3 Objective 1 : Validation of Reference Ordering

Our Algorithm attempts to prioritize testing of sections of the program that has changed. In order to show that such a goal is worthwhile to purse, we will evaluate if an ordering where tests are executed in the descending order of change measured as either set difference or positional weighted difference provides a good rate of fault detection. Such an ordering would be called the Baseline Ordering.

Table 4.1 lists the `APFD` values for both the `xml − security` and `ant` repositories, against the Default Order (the unprioritized order in which the tests are normally run), and the Baseline Order (the ideal order that the prioritizer algorithm hopes to achieve) for both set difference and positional weighted difference measure.

Table 4.1: `APFD` values

Order		`APFD`	
		xml-sec	ant
Default Order		0.1872	0.353
Baseline Order	set-diff	0.9216	0.4201
	pos-w	0.8949	0.7155

We can observe that the Default Order has a very low rate of fault discovery of 0.1872 and 0.353 respectively. The best achievable change prioritizing order for both `set − diff` and `pos − w` improves on this value. While `set − diff` does improve the `APFD` value significantly for `xml − security`, the improvement is only marginal for `ant`. However, `pos − w` has a much more consistent and significant improvement in `APFD` across both repositories.

4.4 Objective 2 : Evaluation of Prioritizing Algorithm

We have seen that the targeted change metrics provide promising results. Now, we evaluate how well our Prioritizer achieves this targeted metrics. The output of the Prioritizer is a sequence in which tests can be executed. In order to see how effective the Prioritizer result is, we would need a measure to evaluate how well the Prioritized Order matches to the Baseline

Order. In this section, we define such a measure and then proceed to evaluate the output of the prioritizer using this measure.

4.4.1 Order Relationship Measure (ORM)

Table 4.2: Sample Change Matrix

test1	test2	test3
10	30	20

Based on the distance measure between the versions of tests, there will be one or more ideal ordering, where the tests are executed in the decreasing order of their change. For instance, if tests $test1$, $test2$ and $test3$ are the tests of some repository with Change Matrix given by Table 4.2, then the ideal order of execution where tests that have changed the most is executed earlier than tests that have changed less would be $test2, test3, test1$. While such an ordering cannot be derived until all tests have been executed, they are still a perfect candidate to compare the results against as they form one of possibly many ideal result. We call such an ordering of tests a *Baseline Ordering*, b.

To evaluate the result of each algorithm using a certain test difference measure, it would make sense to compare the result against the Baseline Ordering for that difference measure.

The ordering of tests can be represented as ranks, where each test has a rank between $1 \dots n$, where n is the number of tests. This would indicate the position where the test is executed. And if the tests are sorted by their ranks, the result would be order in which the tests are executed. This way, comparing the two execution orders becomes a comparison between ranked variables [22]. As we have a reference order (Baseline Order) and we would like to know how well we achieve that order, Ranked Correlation [22] performs such a comparison but does so in a bi-directional way.

Spearman's Ranked Correlation coefficient [23] for two ranked variables X and Y is defined by

$$r_s = 1 - \frac{6 * \sum d_i^2(t)}{n * (n^2 - 1)}$$

where, $d_i = rg(X_i) - rg(Y_i)$ is the difference between the two ranks of each observation and n is the number of observations.

While comparing the two Test Orderings, it is important to note that when comparing a Prioritized Order to a Baseline, a test in the Prioritized Order occurring earlier in the sequence compared to its position in the Baseline, means that some other test that has to occur early has in fact occurred later in the sequence. Thus, in order to avoid doubling penalizing a changed order, it is enough to just count the tests that occur later in the Prioritized Order when compared to the Baseline.

In other words, we are only interested in how one Ordering (Prioritized Order) correlates to the other Ordering (Baseline Order) and not vice versa.

Using the above penalizing mechanism, we propose a *Order Relationship Measure (*`ORM`*)* which is a modification to Spearman's Ranked Correlation Coefficient.

$$r_{test} = 1 - \frac{6 * \sum_{t \in S} d_t^2}{n * (n^2 - 1)}$$

where, n is the number of tests, and

$$d_t = \begin{cases} 0 & \text{if } X_t > Y_t \\ X_t - Y_t & \text{otherwise} \end{cases}$$

Here the X would be the Baseline Ordering and Y would be our Prioritized Order.

Let $1, \ldots, 5$ be a set of tests. If the Baseline Ordering is "12345", then Table 4.3 shows the r_{test} for a few examples.

Table 4.3: Sample `ORM` Values

Sequence	r_{test}
12345	1.0
54321	0.0
51234	0.8
21453	0.75
12354	0.95
12543	0.8
14325	0.8

The d_t value for "12345" is 0 as each test has the same rank in both sequences. Thus, r_{test} is 1. Whereas for "54321", $d_1 = 0$, $d_2 = 0$, $d_3 = 0$, $d_4 = 2$ and $d_5 = 4$. Thus, $r_{test} = 1 - \frac{120}{120} = 0$. Similarly, for "12543", the d_t values that are different would be $d_3 = 0$, $d_4 = 0$ and $d_5 = 2$. This gives an r_{test} value of 0.8.

4.4.2 `ORM` of Prioritized Orders

In this section, we show how our algorithms perform for the two test difference measure, on both the repositories.

As observed in Table 4.3, both "12543" and "14325" has the same r_{test} value of 0.8 which shows that change in different sections of the execution sequence can lead to the same r_{test} value. Thus, a Prioritized Order can have a bad sequence in the early, middle or later part of the execution sequence. As the program may begin execution at any test, an ideal operation of the algorithm would be a good ordering after the first test with significant change has been detected. In order to highlight the performance of the ordering as it progresses, we would calculate r_{test} values for the first 10%, and keep calculating it in bands of 10% increments. Table 4.4 gives us the Banded Correlation Score for $\texttt{xml} - \texttt{security}$ and Table 4.5 gives us the Banded Correlation Score for `ant`. Both tables shows the `ORM` value for each band of given size. Each row in the table is the `ORM` value of a certain percentage of the results, where the Prioritized Order begins execution at a different test. The number at the left most column denotes the index of such a test in the Baseline Ordering. For instance, the first row gives us the `ORM` values of the Prioritized Ordering build using the $\texttt{set} - \texttt{diff}$ measure where the Ordering begins with the test at the 0th index in the Baseline Ordering for $\texttt{set} - \texttt{diff}$. This arrangement is to show how the algorithm would perform when starting on various different test and to evaluate how resilient the algorithm is in moving towards Baseline ordering regardless of where the execution begins, as in real world, we would not know the test that has changed the most until it is executed.

For each starting point, the `ORM` values in Tables 4.4 and 4.5 are an average of 5 runs. This is to account for the randomness in selecting tests when all test cases have been exhausted.

Table 4.4: Banded ORM Values for xml − security

Band		1	2	3	4	5	6	7	8	9	10	11
Size		8	16	24	32	40	48	56	64	72	80	83
set − diff	0	0.87757	0.72132	0.63317	0.58034	0.54757	0.53691	0.53314	0.53109	0.53109	0.53109	0.53109
	5	0.84250	0.74588	0.66145	0.60537	0.56246	0.55335	0.55001	0.54828	0.54828	0.54828	0.54828
	13	0.84715	0.74756	0.65432	0.60212	0.56009	0.54620	0.54293	0.54102	0.54102	0.54102	0.54102
	17	0.81976	0.69954	0.61678	0.56430	0.54385	0.52017	0.51798	0.51632	0.51632	0.51632	0.51632
	25	0.84390	0.73057	0.62672	0.57705	0.54327	0.52351	0.52208	0.51944	0.51943	0.51943	0.51943
	36	0.81702	0.66094	0.58285	0.54535	0.51447	0.49815	0.49749	0.49666	0.49666	0.49666	0.49666
	44	0.80918	0.68678	0.59299	0.54335	0.50289	0.48826	0.48802	0.48744	0.48744	0.48744	0.48744
	58	0.80334	0.68531	0.58685	0.54814	0.52126	0.49610	0.49507	0.49446	0.49446	0.49446	0.49446
	62	0.82504	0.70139	0.60858	0.55285	0.52141	0.50664	0.50390	0.50322	0.50322	0.50322	0.50322
	80	0.85853	0.76541	0.67555	0.61269	0.57531	0.55398	0.55132	0.55038	0.55038	0.55038	0.55038
pos − w	0	0.85514	0.79831	0.79726	0.79302	0.78317	0.78298	0.78244	0.78193	0.78193	0.78193	0.78193
	5	0.98583	0.98248	0.97991	0.97315	0.95582	0.95470	0.95274	0.95138	0.95132	0.95015	0.95015
	13	0.98561	0.98227	0.97970	0.97294	0.95561	0.95449	0.95253	0.95117	0.95111	0.94994	0.94994
	17	0.98561	0.98227	0.97970	0.97294	0.95561	0.95449	0.95253	0.95117	0.95111	0.94994	0.94994
	25	0.98561	0.98227	0.97970	0.97294	0.95561	0.95449	0.95253	0.95117	0.95111	0.94994	0.94994
	36	0.92951	0.87676	0.83695	0.82126	0.81618	0.81609	0.81412	0.81276	0.81271	0.81153	0.81153
	44	0.92951	0.87676	0.83695	0.82126	0.81618	0.81609	0.81412	0.81276	0.81271	0.81153	0.81153
	58	0.92951	0.87676	0.83695	0.82126	0.81618	0.81609	0.81412	0.81276	0.81271	0.81153	0.81153
	62	0.97617	0.96162	0.95148	0.93566	0.89548	0.88787	0.88702	0.88575	0.88570	0.88452	0.88452
	80	0.98656	0.97936	0.97454	0.96501	0.94037	0.93774	0.93405	0.93161	0.93105	0.92987	0.92987

In Table 4.4, the Prioritized Order of xml − security is divided into 11 bands. The initial bands for set − diff have a worse ORM value than pos − w and as the bands progress, the performance of set − diff seems to degrade faster than pos − w. Finally, pos − w results end with a much better ORM value than set − diff. This seems to suggest that our algorithm performs better at estimating the pos − w values than set − diff. It can be seen that with both difference measures, regardless of the starting test, the algorithm tends to have results with very close ORM values. This suggests that the Prioritizer is resilient in identifying the Ordering and the starting test does not have a significant impact. This is important, as the test with the most change cannot be used as the starting test as it cannot be identified without running all the tests.

In Table 4.5, the Prioritized Order of ant is divided into 10 bands. Similar to xml − security, pos − w performs much better when compared to set − diff.

These ORM scores show that when our Prioritizer algorithm operates on Positional Weighted Difference between call sequences, they seem to have good accuracy in estimating the Baseline Ordering.

Table 4.5: Banded ORM Values for ant

Band		1	2	3	4	5	6	7	8	9	10
Size		88	176	264	352	440	528	616	704	792	877
set − diff	0	0.88066	0.75886	0.66664	0.61008	0.58341	0.57065	0.56144	0.55739	0.55632	0.55631
	100	0.88029	0.76664	0.67570	0.62106	0.59396	0.58007	0.56949	0.56579	0.56476	0.56473
	200	0.87548	0.75940	0.67131	0.61554	0.58929	0.57670	0.56643	0.56255	0.56143	0.56139
	300	0.87146	0.75272	0.65893	0.60676	0.58184	0.56836	0.55789	0.55432	0.55326	0.55324
	400	0.87235	0.75229	0.66349	0.60771	0.58445	0.57138	0.56052	0.55687	0.55567	0.55566
	500	0.87394	0.75497	0.66669	0.61292	0.58615	0.57487	0.56420	0.56056	0.55943	0.55939
	600	0.87290	0.75024	0.65959	0.60479	0.57727	0.56505	0.55523	0.55140	0.55032	0.55030
	700	0.87991	0.76260	0.67030	0.61517	0.58846	0.57529	0.56536	0.56156	0.56051	0.56049
	800	0.87393	0.75722	0.66816	0.61544	0.58889	0.57635	0.56567	0.56238	0.56118	0.56116
	876	0.87347	0.75848	0.66798	0.61360	0.58815	0.57552	0.56470	0.56084	0.55968	0.55966
pos − w	0	0.98428	0.95120	0.93509	0.92804	0.92039	0.90961	0.90271	0.89486	0.89296	0.89287
	100	0.98375	0.94915	0.93302	0.92672	0.91931	0.90717	0.89977	0.89227	0.89032	0.89024
	200	0.98335	0.95016	0.93590	0.92907	0.92138	0.91020	0.90298	0.89522	0.89332	0.89326
	300	0.98202	0.94657	0.92955	0.92351	0.91679	0.90522	0.89805	0.89038	0.88852	0.88842
	400	0.98286	0.95249	0.93622	0.93077	0.92313	0.91259	0.90580	0.89812	0.89607	0.89598
	500	0.98349	0.94920	0.93204	0.92505	0.91670	0.90543	0.89822	0.89048	0.88853	0.88846
	600	0.98333	0.95135	0.93488	0.92887	0.92089	0.90937	0.90284	0.89495	0.89294	0.89285
	700	0.97539	0.93485	0.91009	0.90123	0.89070	0.87821	0.87082	0.86300	0.86091	0.86086
	800	0.98486	0.95235	0.93574	0.92904	0.92124	0.90953	0.90212	0.89412	0.89220	0.89212
	876	0.97520	0.93187	0.90756	0.89836	0.88882	0.87592	0.86865	0.86044	0.85843	0.85833

4.5 Objective 3 : APFD of Prioritized Order

We have already shown that the ideal orderings based on descending order or change using either Set Difference or Positional Weighted Difference yield a high rate of fault detection. We have also shown that our algorithm performs well in estimating these ideal ordering. Now, we will empirically show that by estimating the Baseline Ordering well, we can achieve a good APFD for the Prioritized Order.

Table 4.4 and Table 4.5 shows us that our algorithm is not very effective in estimating set − diff measure and hence, any improvement in APFD values by using the said measure is not expected to be very high. However, the prioritizer algorithm is able to efficiently estimate the pos − w measure and our conjecture is that this would give us a good improvement in APFD. This is shown in Table 4.6 which gives us the Average, Standard Deviation, Minimum and Maximum values of APFD and ORM values for both xml − security and ant repositories, where a Prioritized Order from every possible starting test is considered.

For both repositories, when pos − w distance measure is used, we can observe significant improvements to APFD values over the APFD of default execution order (See Table 4.1). On

Table 4.6: ORM and APFD Values

Repository		APFD				ORM			
		Avg	**SD**	**Min**	**Max**	**Avg**	**SD**	**Min**	**Max**
xml-security$_{(DTT = 10^3, STT = 10^2)}$	set − diff	0.5285	0.0568	0.4408	0.7077	0.5217	0.0232	0.47	0.5924
$_{(DTT = 10^3, STT = 10^{10})}$	pos − w	0.7144	0.0989	0.5548	0.8266	0.8976	0.0572	0.8115	0.9539
ant$_{(DTT = 10^3, STT = 10^2)}$	set − diff	0.564	0.0231	0.5039	0.6484	0.5596	0.0043	0.5462	0.5742
$_{(DTT = 10^3, STT = 10^9)}$	pos − w	0.7293	0.0404	0.5719	0.8387	0.6985	0.0156	0.6587	0.7462

average, set − diff improves APFD in xml − security by 0.3413 and in ant by 0.211, whereas pos − w has a much higher improvement of 0.5272 in xml − security and 0.3763. We will now consider pos − w distance measure to understand the impact of the different thresholds on our algorithm.

The outcome of Prioritization depends on multiple controllable parameters and also the random selection of test performed by the Algorithm when no significantly close test (see Section 3.3). The Controllable Parameters are the Starting Tests and the thresholds, STT and DTT. It is not possible to know the best starting test ahead of time as that would require change information to be collected ahead of test run. If STT is set to a very high value, then no change in a test would be significant enough and each iteration of the Prioritizer would cover one test of a different test case at a time. On the other hand, if STT is set to 0, then every discovered test would be considered to have changed enough leading to covering an entire test case in each iteration. Similarly, if DTT is a too low value, then no test would be considered significantly close enough, leading to the same effect as STT set to a very high value. Whereas when DTT is too high, then closeness would be unconstrained, and the next closest test would be picked up each time until a test that has not changed more than STT is discovered.

4.5.1 ant : Effect of Thresholds

Table 4.7 and Table 4.8 shows us how the thresholds, STT and DTT affect the APFD and ORM values of the Prioritized Order in ant. In Table 4.7, we fix the STT to 10^{10} and vary DTT value. It can be observed that APFD and ORM values peak at different times. The maximum average APFD of 0.7209 is achieved at DTT = 10^8, whereas the maximum average ORM is 0.7841

Table 4.7: ORM and APFD Values for pos − w in ant for different DTT values with STT = 10^5

DTT	APFD				ORM			
	Avg	**SD**	**Min**	**Max**	**Avg**	**SD**	**Min**	**Max**
10^7	0.4539	0.0261	0.42	0.5164	0.5243	0.0081	0.5091	0.5373
10^8	0.7209	0.0547	0.6095	0.7959	0.7035	0.0117	0.7183	0.6845
10^9	0.6997	0.0351	0.6416	0.7515	0.7841	0.012	0.7628	0.801
No Threshold	0.6674	0.0332	0.6226	0.722	0.7772	0.0133	0.752	0.7955

and is obtained only at DTT = 10^9. As our ultimate goal is to maximize the APFD value, we use DTT = 10^8 and observe APFD and ORM values for different STT in Table 4.8.

Table 4.8: ORM and APFD Values for pos − w in ant for different STT values with DTT = 10^8

STT	APFD				ORM			
	Avg	**SD**	**Min**	**Max**	**Avg**	**SD**	**Min**	**Max**
No Threshold	0.6355	0.0379	0.58355	0.7204	0.7691	0.0194	0.7339	0.7901
10^3	0.6329	0.0389	0.6121	0.748	0.771	0.0184	0.7346	0.7854
10^4	0.6235	0.0326	0.5923	0.7103	0.7728	0.0185	0.7362	0.7916
10^5	0.7273	0.0471	0.7831	0.6277	0.7015	0.0091	0.7169	0.6891
10^6	0.5619	0.0179	0.5275	0.5867	0.5894	0.0218	0.5575	0.6214

Just like with different DTT values, maximum average ORM and APFD values are obtained at different threshold values. The maximum APFD value of 0.7273 is obtained at STT = 10^5, whereas the maximum ORM value of 0.7728 is obtained at STT = 10^4. STT values of 10^2 and 10^3 have notably similar ORM and APFD values. This is because, at this point, the threshold is low enough to consider that every test has changed enough.

4.5.2 xml − security : Effect of Thresholds

Now, we go through the same process with xml − security repository where we vary both DTT and STT values in Table 4.9 and Table 4.10.

Unlike the effect of DTT on ORM and APFD in ant, in xml − security, both values obtain their average maximum value at the same time when no threshold is used. The same applies for STT in Table 4.10 where the maximum is at a STT of 10^2 and when no threshold is used.

Table 4.9: ORM and APFD Values for pos − w in xml − security for different DTT values with STT = 10^3

DTT	APFD				ORM			
	Avg	**SD**	**Min**	**Max**	**Avg**	**SD**	**Min**	**Max**
10^7	0.6206	0.0909	0.4671	0.7564	0.4487	0.1493	0.2553	0.7222
10^8	0.5477	0.1148	0.4543	0.8194	0.5011	0.23	0.202	0.9063
10^9	0.7204	0.0882	0.5548	0.7998	0.8882	0.062	0.782	0.9499
No Threshold	0.7343	0.095	0.5548	0.8078	0.9096	0.0561	0.8115	0.9539

Table 4.10: ORM and APFD Values for pos − w in xml − security for different STT values with DTT = 10^{10}

STT	APFD				ORM			
	Avg	**SD**	**Min**	**Max**	**Avg**	**SD**	**Min**	**Max**
No Threshold	0.7343	0.095	0.5548	0.8078	0.9096	0.0561	0.8115	0.9539
10^3	0.7343	0.09501	0.5548	0.8078	0.9096	0.0561	0.8115	0.9539
10^4	0.6762	0.1382	0.5066	0.8078	0.8645	0.0924	0.731	0.9464
10^5	0.7061	0.0812	0.6432	0.8132	0.8461	0.0345	0.8146	0.8943
10^6	0.5635	0.0565	0.505	0.6534	0.5925	0.0163	0.558	0.6116

4.5.3 Estimating DTT

DTT defines how close tests should be to be picked up by the Prioritizer when it looks for tests similar to a test of interest. As it operates on the closeness between tests, the ideal indicator for DTT would be the difference between the tests. Specifically, the measure of how close faulty tests are to each other, and the median becomes the natural candidate to set the cut off for difference.

Table 4.11: Statistics for pos − w Difference Values

Statistics	xml − security		ant	
	Difference Matrix	Faulty Tests	Difference Matrix	Faulty Tests
Median	163254	812866	13981405	32203508
Row-wise Median	20995	677119	13208850	40547112

In order to compare how the faulty tests compare against the set of all tests, we calculate the Median values of the matrix of difference between all tests, and the matrix of difference between faulty tests. Table 4.11 gives us the median of both matrices. We also calculate a

Row-wise Median, were we calculate the median of the differences of each test with other tests, and then the median of all such median values.

As we can observe, the Median values of Faulty Tests differ significantly from the Median values of the collection of all tests. There could be no discernible relation between the two as faults could occur in sections tested by any test. Hence, it becomes infeasible to predict `DTT` with reasonable confidence. Thus, we would recommend that no threshold be used with regards to `DTT` and Tables 4.7 and 4.9 show that using no threshold still gives us a very good improvement to `APFD`.

4.5.4 Finding the right `STT`

`STT` is the threshold that defines how much any test must have changed in order for that test to be considered significant. As seen in Section 3.1.1, Positional Weighted Difference assigns difference values to tests based directly on the size of the Call Sequence. The difference value has a potential to grow high depending on the length of the call sequence. If l is the length of the longer call sequence, then the maximum possible size would be $l+(l-1)+(l-2)+\cdots+(l-(l-1))$. Thus, the size is $O(l^2)$.

If we measure significance relative to the size of the call sequence, we could say that a good candidate for change would be to have a sequence size greater than the bottom 10% range. In this effect, we calculated the length of all call sequences and measured the 10th Percentile of the lengths to be 174 and 24 for $\texttt{xml}-\texttt{security}$ and `ant` respectively. The lengths are in the order 10^2 and 10^1. By translating the call sequence size to their maximum possible Positional Weighted Difference value, we get a threshold of 10^4 and 10^2 for $\texttt{xml}-\texttt{security}$ and `ant` respectively. 10^2 for `ant` is low enough for the threshold to be considered No Threshold. Tables 4.8 and 4.10 shows us that the thresholds does yield us good `APFD`.

4.6 Performance of Prioritizer

The performance measures were collected on a computer running Fedora Linux Version 25, with an Intel(R) Core(TM)2 Duo P7350@2.00GHz CPU and 2GB of RAM. In order to evaluate the performance of the Prioritizer, we calculate the time taken to run all tests, the time taken

for the prioritizer to come up with a Prioritized Order so that we may get the percentage of extra time that would be added to run all tests with prioritization. Table 4.12 gives us the time measurements. Here, the time taken to prioritize includes the time taken to load previously calculated Difference Matrix to memory. As the Difference Matrix can be calculated any time after the previous version of the software completes execution, the time taken to build the Difference Matrix is not considered here.

Table 4.12: Execution times

	xml-security		**ant**	
	set-diff	**pos-w**	**set-diff**	**pos-w**
Test Execution time	14930ms		180211ms	
Time taken to prioritize	46ms	44ms	1511ms	6158ms
Total time taken	14976ms	14974ms	181722ms	186369ms
% of time taken by prioritizer	3%	2.9%	0.8%	3.3%

As we can see, the Prioritizer only adds at most 3.3% to the execution time of the tests.

CHAPTER 5. CONCLUSION

Summary. In our work, we have provided a novel approach where the call sequence of tests obtained during their execution is used to quantify relationship between different tests and different versions of the same test. We have defined a data structure, Difference Matrix, that can be used to efficiently hold and retrieve correlation values among different tests. We have built an algorithm that operates on this data structure to consume test correlation information and dynamically select tests in an order that prioritizes testing of sections of programs that have changed over sections that have not changed. We have introduced a new rank relationship measure that penalizes only those ranks that occur later than they should. This prevents doubly penalizing a change in the execution order.

Finally, we have shown empirically that our goal of prioritizing testing of modified sections improves fault detection by showing that our baseline orderings have significantly improved APFD values compared to the default execution order. We also show that the output of the prioritizer improves fault detection compared to unprioritized execution and the extend of improvement is comparable to the extent to which the goal of prioritizing execution of changed section is achieved.

Future Work. There is scope of improvement and further research in areas pertaining to Test Correlation Measures and Prioritizing algorithm.

If a mapping of each test to the files it tests is maintained, by looking at how much each file has changed between the two versions, one could identify tests that could possibly cover sections of code that has changed the most. These tests could be used as starting tests to maximize the chances of early discovery of changed sections of the code.

Our test correlation measures treat call sequences as either a set where order and repetitions does not matter, or as a strict sequence where the position of the method call determines

how much the tests differ. There is scope for experimenting with various ideas here such as, ignoring repetitions, considering only calls to classes and not methods, considering calls to only methods and not classes, considering the actual arguments used in the calls and so on. The call sequences can also be treated as strings to derive test correlation values that can be obtained by calculating the Jaccard similarity between the sequences or by calculating the longest common subsequences between them. By treating the call sequences differently in each of the above mentioned methods, we could learn the measure that best explains the difference between tests. But, it is possible that different repositories could have different ideal difference measures. Thus, the study must be extensive in observing how the call sequence difference measures perform in various kinds of software.

Currently, the same difference measure is used to see if a test has varied significantly and how different two tests are. It would be interesting to observe the impact of each difference measure in both parts of the algorithm. When considering how much a test has changed, one could ignore repetitions in call sequences and simply look for any new method calls. To do this, one has to deal with the cardinality as the more difference measures there are, the more combinations that becomes possible.

With enough case studies, learning algorithms could be used to clearly define the relation between various difference measures, the thresholds and APFD.

In a large software, the call sequences obtained from tests could easily run up to millions of methods calls. When comparing such call sequences, one could use Information Retrieval techniques to improve the speed at which such comparisons can be made. For instance, indexing the method calls in call sequences could lead to faster look ups. If all the method names could be collected, a word-frequency matrix can be built where the number of times each method call occurs in each call sequence can be calculated. This could enable estimating Jaccard similarity values very efficiently. However, for the overhead of such techniques to be justified, the call sequences must be sufficiently long, and the test difference measure used must not consider the exact position of each method call as such information would not be held in Information Retrieval techniques.

BIBLIOGRAPHY

[1] S. Elbaum, A. G. Malishevsky, and G. Rothermel. Test case prioritization: a family of empirical studies. *IEEE Transactions on Software Engineering*, 28(2):159–182, Feb 2002.

[2] Mstest manual, 2017.

[3] Junit manual, 2017.

[4] G. Rothermel, R. H. Untch, Chengyun Chu, and M. J. Harrold. Prioritizing test cases for regression testing. *IEEE Transactions on Software Engineering*, 27(10):929–948, Oct 2001.

[5] Apache santuario project - xml security platform. http://santuario.apache.org/.

[6] S. Yoo and M. Harman. Regression testing minimization, selection and prioritization: A survey. *Softw. Test. Verif. Reliab.*, 22(2):67–120, March 2012.

[7] R. Gupta, M. J. Harrold, and M. L. Soffa. An approach to regression testing using slicing. In *Proceedings Conference on Software Maintenance 1992*, pages 299–308, Nov 1992.

[8] Hiralal Agrawal, Joseph R. Horgan, Edward W. Krauser, and Saul London. Incremental regression testing. In *Proceedings of the Conference on Software Maintenance*, ICSM '93, pages 348–357, Washington, DC, USA, 1993. IEEE Computer Society.

[9] Richard Hamlet. Random testing. In *Encyclopedia of Software Engineering*, pages 970–978. Wiley, 1994.

[10] F.T. Chan, T.Y. Chen, I.K. Mak, and Y.T. Yu. Proportional sampling strategy: guidelines for software testing practitioners. *Information and Software Technology*, 38(12):775 – 782, 1996.

[11] T. Y. Chen, H. Leung, and I. K. Mak. *Adaptive Random Testing*, pages 320–329. Springer Berlin Heidelberg, Berlin, Heidelberg, 2005.

[12] B. Jiang, Z. Zhang, W. K. Chan, and T. H. Tse. Adaptive random test case prioritization. In *2009 IEEE/ACM International Conference on Automated Software Engineering*, pages 233–244, Nov 2009.

[13] L. Zhang, J. Zhou, D. Hao, L. Zhang, and H. Mei. Prioritizing junit test cases in absence of coverage information. In *2009 IEEE International Conference on Software Maintenance*, pages 19–28, Sept 2009.

[14] W. Jun, Z. Yan, and J. Chen. Test case prioritization technique based on genetic algorithm. In *2011 International Conference on Internet Computing and Information Services*, pages 173–175, Sept 2011.

[15] Z. Li, M. Harman, and R. M. Hierons. Search algorithms for regression test case prioritization. *IEEE Transactions on Software Engineering*, 33(4):225–237, April 2007.

[16] 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*, FSE 2016, pages 559–570, New York, NY, USA, 2016. ACM.

[17] Tsong Yueh Chen, Fei-Ching Kuo, and Huai Liu. Adaptive random testing based on distribution metrics. *J. Syst. Softw.*, 82(9):1419–1433, September 2009.

[18] T. Y. Chen, De Hao Huang, F.-C. Kuo, R. G. Merkel, and Johannes Mayer. Enhanced lattice-based adaptive random testing. In *Proceedings of the 2009 ACM Symposium on Applied Computing*, SAC '09, pages 422–429, New York, NY, USA, 2009. ACM.

[19] Tsong Yueh Chen, Fei-Ching Kuo, Robert G. Merkel, and T. H. Tse. Adaptive random testing: The art of test case diversity. *J. Syst. Softw.*, 83(1):60–66, January 2010.

[20] Ilinca Ciupa, Andreas Leitner, Manuel Oriol, and Bertrand Meyer. Artoo: Adaptive random testing for object-oriented software. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08, pages 71–80, New York, NY, USA, 2008. ACM.

[21] Hyunsook Do, Sebastian G. Elbaum, and Gregg Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Software Engineering: An International Journal*, 10(4):405–435, 2005.

[22] William H. Kruskal. Ordinal measures of association. *Journal of the American Statistical Association*, 53(284):814–861, 1958.

[23] W.W. Daniel. *Applied nonparametric statistics*. The Duxbury advanced series in statistics and decision sciences. PWS-Kent Publ., 1990.