2021

Dispersity-Based Test Case Prioritization

Chen Liu

UNIVERSITY
OF WOLLONGONG
AUSTRALIA

Dispersity-Based Test Case Prioritization

Chen Liu

Supervisors:
Associate Professor Zhiquan Zhou
Co-supervisor:
Senior Professor Willy Susilo

This thesis is presented as part of the requirement for the conferral of the degree:
Doctor of Philosophy

This research has been conducted with the support of the Australian Government Research Training Program Scholarship

University of Wollongong
School of Computing and Information Technology

March 2021

Abstract

With real-world projects, existing test case prioritization (TCP) techniques have limitations when applied to them, because these techniques require certain information to be made available before they can be applied. For example, the family of input-based TCP techniques are based on test case values or test script strings; other techniques use test coverage, test history, program structure, or requirements information. Existing techniques also cannot guarantee to always be more effective than random prioritization (RP) that does not have any precondition. As a result, RP remains the most applicable and most fundamental TCP technique.

In this thesis, we propose a new TCP technique, and mainly aim at studying the Effectiveness, Actual execution time for failure detection, Efficiency and Applicability of the new approach. The contributions of this thesis are summarized as follows:

- The first work, to the best of our knowledge, that presents the similarities of neighboring test cases are common in certain ways in real-world software projects.
- We introduce a novel concept of natural distance that can be used as a universal distance metric for measuring the dispersity among test cases in real-world test suites, which does not require information of the program under test, or execution information of the test cases.
- Base on the dispersity metric, we propose an extremely simple, effective, and efficient

way to prioritize test cases, and it is as applicable as RP.

- We conduct a series of large-scale empirical studies using real-world projects. Empirical results show that our technique is more effective than RP. With a linear computational complexity, our approach provides a practical solution to the problem of prioritizing very large test suites (such as those containing hundreds of thousands, or millions, of test cases).
- The results also show that, in terms of applicability, the execution time for failure detection, and efficiency, our lightweight approach outperforms one of the most frequently used heavyweight benchmark techniques, namely, the branch-coverage-based additional algorithm.
- A study of the naming convention of test cases has been conducted using real-world projects. The results show that our observation, namely, neighboring test cases often have similarities in certain ways while more dispersed test cases tend to be dissimilar, is valid.
- Our technique provides a practical solution to TCP when neither input-based nor execution-based techniques are applicable due to lack of information.

Acknowledgments

My PhD study has been a really long journey, and I could not have succeeded without the assistance and support of many individuals.

In particular, I would like to express my thanks, my supervisor Zhiquan Zhou for his excellent guidance and support, and Willy Susilo for his support as my co-supervisor. I wish to thank Prof. T. Y. Chen and Prof. T.H. Tse for their invaluable advice.

I would also especially thank my wife, Nan Zhou, for continuously supporting my research and the family during my long journey, my parents, Changrui Liu and Guilan Huang, for their encouragement and support, and give special thanks to my son Yitong Liu for being a sensible boy and trying to avoid interrupting my work.

Certification

I, Chen Liu, declare that this thesis submitted in fulfilment of the requirements for the conferral of the degree Doctor of Philosophy, from the University of Wollongong, is wholly my own work unless otherwise referenced or acknowledged. This document has not been submitted for qualifications at any other academic institution.

Chen Liu

24th March 2021

Publications

This thesis is based on the following publication.

Zhi Quan Zhou , **Chen Liu** , Tsong Yueh Chen , T. H. Tse , and Willy Susilo, “Beating Random Test Case Prioritization,” *IEEE Transactions on Reliability*, 10.1109/TR.2020.2979815, 2020

Contents

List of Figures

List of Tables

List of Notations

S_i	The set of branches covered by test case i, i = 1, 2, ..., 8.
T	A sequence of test cases t_1, t_2, ..., t_n, where n > 0.
$\|i - j\|$	The natural distance between test cases t_i and t_j, where $1 \leq i, j \leq n$.

Chapter 1

Introduction

1.1 Background

Nowadays, software continues to grow in size and complexity, and hence the demands of software testing are increasing continuously, thus increasing the cost of assuring the high quality of the software. Regression testing is one of the testing processes that validate software with newly modified code, and that there are no new faults introduced to the previously tested code. Regression testing could be expensive to execute and may delay the whole development process when the size of the test suite is large. For example, Rothermel et al. [1] reported that "Running all of the test cases in a test suite, however, can require a large amount of effort. For example, one of our industrial collaborators reports that for one of its products of about 20,000 lines of code, the entire test suite requires seven weeks to run". Test case prioritization (TCP), which is a major challenge in software testing, attempts to find an optimal ordering of test case execution, which can help to provide earlier fault detection and maximize benefit to the tester, by saving time and resources. Even if the testing is prematurely terminated, such as when the testing resources have been exhausted, the tester still can benefit from it [2-5].

To date, different methods of automated test case prioritization have been presented. The majority of automated TCP methods use the structural coverage information (such as control flow, data flow, call-tree paths, and relevant slices information) of the test cases [4], or an estimate of such information [6]. One of the intuitions is that early fulfillment of structural coverage should increase the chance of fault detection. Among the various prioritization methods, the additional algorithm (which is an instance of additional greedy algorithms) has been considered as one of the most frequently used benchmark methods [3, 4, 7, 8].

Other automated TCP approaches involve the use of requirements specifications [4, 9, 10], system models [11], test case execution history in previous runs [12-14], mutation testing [1, 15], cost-awareness information such as the severity of faults and the costs of test case executions [4, 13, 16], test case distance metrics based on their execution profiles [7, 17, 18], or differences between different versions of the software under test (SUT) [19]. More recently, Busjaeger and Xie reduced the TCP problem to that of learning to rank [20].

Under different assumptions and situations, the above approaches have their advantages. However, none of them or their combinations can be as applicable as, or can always be more cost-effective than, random prioritization (RP). This is because the required information (such as test case coverage, test case execution history, program structure, or requirements specifications) may not always be available in practical situations [21]. To address this problem, several input-based TCP techniques have been developed [21-23], making use of the test case values/test script strings rather than their code-coverage or other test performance information. Compared with most other approaches (which are execution-based), the input-based TCP strategy has better applicability; nevertheless, its application requires the tester to be able to

access the concrete input values of the test cases, and to design effective distance (or dis/similarity) metrics to measure the dissimilarity between the concrete values of two test cases. This means that the tester must have thorough knowledge of "the input structure and semantics of the application under test" [21]. Even with such knowledge, to design good distance metrics for any arbitrary input structure of any arbitrary program can be challenging. Therefore, input-based TCP techniques are still not as applicable as RP, as the latter does not even require the tester to know the concrete values of the test cases.

Furthermore, Zhou et al. [18] reported in 2012, traditional TCP research did not consider the fact that real-world test suites could become very large with "millions of test cases." In this situation, the computational overhead of TCP is a major concern. The additional statement (or branch) coverage algorithm [3], for example, has a time complexity of $O(n^2m)$, where n is the number of test cases and m is the number of statements (or branches) of the program under test. In our empirical studies with real-world large test suites, we find that their execution time can become prohibitive.

Miranda et al. [24] also reported the following: "The number of test cases to prioritize grows in size up to millions ... For real-world software, the size of a test suite can often exceed the size of the system under test. In contrast, the time available for test execution cycles decreases ... Every day at Google an amount of 800K builds and 150M test runs are performed on more than 13K code projects ... Most TCP approaches in the literature cannot handle such scale. Our experimental results show that some TCP approaches become soon inefficient even for small-medium size benchmarks."

Based on similarity, Miranda et al. [24] presented a FAST (recursive acronym for FAST

Approaches to Similarity-based Testing) family of scalable TCP techniques to address these problems, and compared FAST with other similarity-based TCP techniques (including adaptive random TCP techniques introduced by Jiang et al. [7] and Zhou et al. [17, 18]). FAST can be used either as a white-box prioritization technique based on code coverage information, or as a black-box prioritization technique based on the string representation of the actual test cases—in this aspect, the black-box FAST is essentially an input-based TCP strategy because it needs to know the input values/test scripts. As with the other TCP techniques discussed earlier in this section, therefore, FAST is still not as applicable as RP that requires neither white-box coverage information nor the values of the concrete test cases/scripts.

Therefore, we raise the practical research question: In real-world software testing, can there be a lightweight TCP method that is more effective, detects failures in a shorter execution time than RP, and is as efficient and readily applicable as RP?

In this thesis, to answer the above research question, we propose a concept of natural distance as a new dispersity metric, with a practical test case prioritization strategy, and conduct a series of empirical studies using 66 real-world projects, with 95 different versions.

1.2 Research Goals

In this research, we raise the following practical research goals:

RG1: To develop a lightweight TCP method that has the following properties:

1) It is more effective than RP.
2) It can more quickly detect failures, in terms of execution time, than RP.
3) It is as efficient as RP.

4) It is as readily applicable as RP.

RG2: To conduct empirical studies to evaluate the lightweight technique of RG1 with the heavyweight "additional" algorithm, with the respect to the same properties, namely:

1) Effectiveness.
2) Actual execution time for failure detection.
3) Efficiency.
4) Applicability.

1.3 Contribution of the Thesis

The contributions of this thesis are summarized as follows:

1) This is the first work, to the best of my knowledge, to point out that neighboring test cases in a real-world test suite often have similarities in certain ways while more dispersed test cases tend to be dissimilar.
2) Based on the above observation, a concept of natural distance is proposed, that can be used as a universal distance metric for measuring the dispersity among test cases in real-world test suites. The measurement requires neither the knowledge of the source code, requirements specifications, designs, input types, etc., of the program under test, nor the knowledge of coverage data, execution history, concrete input values, etc., of the test cases.
3) We propose a simple and practical test case selection/prioritization strategy, using the concept of natural distance, to address the RG1. The general form of the strategy is given as Hypothesis I in Chapter 3, and a specific implementation of the strategy

is given as an algorithm shown in Fig. 4 in Chapter 4, which has a linear time and space complexity. Our strategy, therefore, provides a practical solution to the problem of prioritizing very large test suites—for large test suites containing hundreds of thousands, or millions, of test cases, the execution time of conventional nonlinear prioritization algorithms can be prohibitive.

4) We conduct a series of empirical studies using 95 different versions of SUT from 66 real-world software projects. The empirical results show that the method significantly improves the effectiveness and reduces failure detection time of RP and that, even in the worst case, the performance of the method is still close to that of RP. We also conduct a case study with additional 50 real-world software projects on the naming conventions of their test cases, and the results show that the observation mentioned in contribution 1) commonly exists in real-world software projects.

5) The results also show that this lightweight approach outperforms the heavyweight branch-coverage-based additional algorithm with respect to applicability, execution time for failure detection, as well as efficiency, whereas the additional algorithm outperforms this approach in the majority of situations with respect to effectiveness. This finding addresses the RG2.

1.4 Organization of the Thesis

The reminder of this thesis is organized as follows:

In Chapter 2, the literature on test case prioritization and Adaptive Random Testing are

reviewed. Chapter 3 presents our observation of the nature of real-world test suites, and proposes a natural distance metric based on this observation. Chapter 4 describes the design of empirical evaluation and introduces the proposed TCP method, analyzes the empirical results, and further compares the efficiency of our method with that of RP. In Chapter 5, a case study is conducted on the naming conventions of real-world software projects, and Chapter 6 contains the design of an extended empirical study and an analysis of the empirical result on the aspect of the code coverage. Chapter 7 discusses practical challenges and unsuccessful attempts during the empirical study, and Chapter 8 presents the conclusion of this thesis and indicates some future research.

Chapter 2

Literature Review

2.1 Test Case Prioritization

Wong et al. first proposed test case prioritization in 1997 as a test suite modification, minimization and prioritization technique to be used for regression testing [25]. Test-case prioritization techniques schedule test cases for execution in an order that attempts to increase their effectiveness at meeting a chosen performance goal [1]. It attempts to find an optimal ordering of test case executions, which can maximize benefits of the tester's effort, e.g. increases in the rate of fault detection, discovering faults sooner and so on, even if the testing is prematurely terminated, such as when the testing resources have been exhausted [2-5].

Rothermel et al. [26] reported a real-world case in 1999, where the execution time of the test suite from a single product, which contained around 20 000 lines of code, could consume seven weeks. With the development of technology, software becomes larger and more complex. Under the current industry trends with the wide use of Agile software development, more and more frequent deployments become a common situation, which means the demand of testing and the benefit of test case prioritization increase rapidly. For example, recently, Miranda et al.

[24] reported that Google performs 800K builds and 150M test runs on daily basis across more than 13K projects.

Many test case prioritization techniques and strategies have been proposed in the past two decades. Most of the automated TCP methods require extra information, such as test case coverage, test case execution history, program structure, or requirements specifications. In addition, input-based TCP techniques have been developed by using test case values/test script strings [21-23], instead of the extra information that is used by the execution-based approaches, this gives them better applicability than the above methods.

2.2 Execution-based Test Case Prioritization

The execution-based test case prioritization techniques require extra information from the execution of test cases to prioritize the test cases. Yoo and Harman [4] reported that structural coverage information, such as control flow, data flow, call-tree paths or relevant slices information, is often used as the prioritization criterion. Rothermel et al. [26] proposed nine test case prioritization techniques, with the use of coverage of branches and statements information. Since then and until recently[27], many new test case prioritization techniques have been presented that rely on coverage data [25, 28-36]. In addition to the use of coverage data, Mei et al. [6] proposed another approach named JUnit test case Prioritization Techniques operating in the Absence of coverage information (JUPTA), which uses estimates of coverage information to address the problem that the coverage data is not always available. Recently, a new concept, *substate profiling*, has been presented by Assi et al. [37], which is an alternative profile of typical profile elements (functions, statements and branches), and later this technique has been

applied to TCP[38]. Among these coverage-based approaches, the additional algorithm, which prioritizes test cases according to the additional number of statements/branches covered by individual test cases [4], has been considered as one of the most frequently used benchmark methods [3, 4, 7, 8].

In addition to using coverage data, other execution information is required in other approaches. Korel et al. [11, 32, 39], Gökçe et al. [40] and Shin et al. [41] presented approaches that use system models. The requirement specifications have been used by Yoo and Harman [4], Cohen et al. [9], Krishnamoorthi and Sahaaya Arul Mary [42], and many others [10, 43-46]. Test case execution history data from previous runs has also been used in test case prioritization techniques [12-14, 47-51]. In addition, test case prioritization techniques make use of mutation testing [1, 15], cost-awareness information [4, 13, 16, 52], test case distance metrics based on their execution profiles [7, 17, 18], and the differences between software versions [19].

The execution-based test case prioritization techniques have their advantages under various situations and assumptions, but whether by single or combinations of approaches their applicability is not the same as, nor can they always be more cost-effective than, random prioritization. The reason behind this problem is that the information from the test case execution is not always available. Jiang and Chan [21] reported the poor availability of code coverage data, fault history, or test specifications, which are seldom well-maintained in real-world software development projects. To address this problem, the input-based test case prioritization techniques have been proposed.

2.3 Input-based Test Case Prioritization

Input-based test case prioritization techniques use the difference between test inputs to schedule the execution order of test cases [21, 53]. These approaches do not require any execution data of the test cases, such as code-coverage or other test performance information, but only the test cases themselves. It makes these approaches have better applicability than the execution-based test case prioritization techniques.

Ledru et al. proposed an input-based test case prioritization approach using string distance on the text of test cases, which could be applied to applications where their test cases can be treated as character strings. They discussed four classical string distances with this approach, including Hamming distance, Levenshtein or Edit distance, Cartesian and Manhattan distances, and Caveats associated with string distances [22, 54, 55].

After that, Thomas et al. presented another technique to prioritize test cases, which uses the linguistic data of the test cases, such as the identifier names, comments, and string literals that help to determine the functionality of the test cases [23, 56].

Jiang and Chan proposed a family of input-based test case prioritization techniques, which is “the first work that presents a family of novel input-based randomized test case prioritization techniques” [21]. Hao et al. reported that the test input has already been used in test generation, which is not a new adequacy criterion, but it is still novel to involve it in test case prioritization and avoid the cost in the collection of coverage information [53]. These approaches prioritize test cases based on the chosen distance, such as Euclidean distance for numerical applications and edit distance for command line applications, between already selected test cases and the unselected test cases in the candidate set [21, 57]. More details of the concept candidate set will

be discussed in the following section named Adaptive Random Testing.

For the above input-based test case prioritization techniques, when the test inputs are programs, e.g., when testing the compilers, Chen et al. reported that the cost of time will be large with these approaches. To address this issue and to prioritize the test cases of C compilers, they presented a technique that transforms test input into text-vector representing its fault-relevant characteristics. They have conducted experimental studies with two open source C compilers, and also indicated that their approach is "not specific to C compiler testing and can be extended to other testing scenarios" [58].

2.4 Similarity-based Test Case Prioritization

Similarity-based test case prioritization techniques using different measure of similarity between pairwise test cases to prioritize test cases. The purpose of the similarity-based approaches is to maximize the diversity of the selected test cases, in the other words, to minimize the similarity of these test cases. As a result, the chance to detect failures early will increase when the diversity of the test cases can be maximized [59]. Similarity-based techniques are widely discussed in test case selection and test case prioritization. According to the studies by Fang et al. [59], they can be mainly classified into distribution-based and adaptive random testing inspired. The former is usually involved in test case selection approaches, and the latter is applied to both test case selection and prioritization, e.g. Zhou [17] proposed a method for test case selection with an adaptive random testing technique, which can also be directly applied to test case prioritization.

As the similarity-based approaches use different measures and information, some of them

will also be classified into more than one type. For example, Jiang et al. [7] proposed a family of test case prioritization techniques, which use the coverage information to prioritize the test cases, has been classified as execution-based test case prioritization. But as they use the coverage information as the measure of the similarity between test cases and prioritize test cases base on their similarity, it can also be considered a similarity-based test case prioritization approach. There are other proposed similarity-based techniques that use coverage information, for example, Zhou et al. [18], Fang et al. [59] and Miranda et al. [24] proposed similarity-based test case prioritization approaches that use the coverage and other execution data, such as execution frequency information. Other execution data, in addition to coverage information, have also been used by different techniques, e.g., Leon and Podgursky [60] used the execution counts for three different granularities: functions, basic blocks, and control flow edges between basic blocks, and Noor et al. [61] used the historical failure data in their approach..

In addition to the execution-based test case prioritization, some similarity-based test case prioritization techniques may also be classified into other types. For example, the approaches proposed by Ledru et al. [22] and Miranda et al. [24] (when used as a black-box technique on the string representation of the actual test cases) can also be considered as input-based test case prioritization techniques.

2.5 Adaptive Random Testing

In software testing, two main approaches are used. One is named ‘white box testing’, and the other ‘black box testing’. In the black box testing techniques, random testing (RT) is one of the mostly used techniques. It is simple to implement, and it is an intuitively appealing technique

[57]. It is a useful and simple method for software testing, which has been used extensively, not only in the software testing, but also in other engineering subjects. However, in some situations, when the input domain of a program is significantly large, RT may require too much time to detect failures.

Adaptive Random Testing (ART) [62] is a variant of random testing, which was first proposed for test case generation. Later, many test case prioritization approaches have been proposed base on the key intuition of ART, which is to evenly spread test cases throughout the input domain in test case generation, or throughout the test suite in test case prioritization.

Chen et al. [57] introduced the enhanced form of RT named 'ART' ('Adaptive Random Testing'). They found that, for non-point patterns [57] [62], including the strip and block patterns, the capability of failure detection was improved by ART. In ART, instead of using the P-measure (the probability of detecting at least one failure) and the E-measure (the expected number of failures detected) as the effectiveness metrics, the F-measure (the expected number of test cases required to detect the first failure) is used. The F-measure is smaller when the testing strategy is more effective. It means that it uses fewer test cases to detect the first failure. The implementation of ART is as follows [57] [62]: it uses two sets of test cases, one is named the 'executed set' and the other is the 'candidate set'. The executed set records the test cases that have been executed without detecting failure, and the candidate set selects unexecuted test cases randomly. Then one element in the candidate set that is farthest away from all executed test cases will be selected as the next test case. The size of the candidate set is constant; this would be named the 'Fixed Size Candidate Set Version of the Adaptive Random Testing' (FSCS-ART). Chen et al.'s experiments [57] show that ART outperforms RT significantly for

the 12 programs used in their experiments. Although the improvement is significant, there is one weakness in ART: the cost of calculating distances is great. Given that it needs to calculate the distance from each element in the candidate set to each element in the executed set, the time complex of ART is $O(n^2)$ [63]. It will cost a great deal more time than RT, whose time complexity is O(n) [63].

In the high-dimensional input domain, for some ART methods, there is a problem that the fault-detection capability is compromised. Chen et al. [64] proposed a new method referred to as 'Adaptive Random Testing by Balancing' (ART by Balancing). This solves the fault-detection capability problem. It is a new version of ART, which improves the fault-detection capability in the high-dimensional input domain. It describes that the key objective of ART by Balancing is making the centroid of test cases in each partition of the input domain close to the centroid of the corresponding partition. From the simulation, the result shows that ART by Balancing has a better fault-detection capability in high-dimensional input domains, compared with Distance-based ART and ART by Bisection. It also shows that for the first n test cases, when n is small, there are some favourable regions and unfavourable regions. With the increase of n, the gap diminishes, and the distribution becomes even. Chen et al. [64] conclude that the fault-detection capability of ART by Balancing, in high-dimensional input domains, outperforms other ART methods greatly.

Computation costs may render ART less cost-effective than RT. Chen et al. [65] proposed a new technique known as 'Mirror Adaptive Random Testing', which reduces the computations compared to the original ART. This is an integration of mirroring and ART. For implementation, Mirror Adaptive Random Testing first divides the input domain into disjointed subdomains. It

then chooses one disjoint subdomain as a source subdomain, and the others are referred to as mirror subdomains. After that, it generates a test case from the source subdomain and executes it. It then generates a test case in each mirror subdomain and executes these. According to simulation, the result shows that the performance of mirror ART is as good as distance-based ART, but mirror ART requires less calculation. In most experiments, mirror ART can get best efficiency in block pattern, compared with strip pattern and point pattern.

Previous studies, used distribution metrics to, measure how evenly an ART method can spread its test cases. In these studies, it was observed that there is a correlation between the evenness of test-case distribution and the fault-detection capability. Chen et al. [66] proposed a new algorithm, namely ART based on distribution metrics (DM-ART), which uses distribution metrics (that measure the degree of even distribution of a set of points) as criteria for test case selection. It improves the evenness of test distribution and the fault-detection capability of ART. They introduced two test-case selection criteria. One is based on discrepancy, and the other is based on dispersion. They also proposed a new algorithm, which integrates discrepancy and dispersion with other criteria in test-case selection. From the simulation [66], the result shows that, with stand-alone metrics, the ART algorithms have poor fault-detection capabilities. However, with the integration of these metrics and the notion of far apart in FSCS-ART, the DM-ART spreads test cases more evenly, and has better fault-detection capabilities.

Recently, in February 2013, Shahbazi et al. [67] proposed a new version of ART, with the a time complex of O(n) [67], which is the same as the original RT. This version is named ‘Centroidal Voronoi Tessellations’. It has solved the problem of ART being time consuming.

First, it selects some test cases randomly, it then divides the input domain based on each point belonging to the nearest selected test case's partition. Then it calculates the centroid of each partition, and uses them to replace previous selected test cases. The result clearly shows improvement using this method.

Besides these approaches, various techniques have been proposed [68], some of which are summarized as follows: 1) Selection of the best candidate from a set of candidates, such as FSCS-ART. 2) Exclusion [69], which defines exclusion zones around previously executed test cases (that have not detected any failure) to restrict the regions from which the next test case is to be generated. Random inputs are generated one by one until one falls outside of all the exclusion zones and that input is taken as the next test case. 3) Partitioning [70, 71], which divides the input domain into partitions, and then selects a partition from which the next test case is to be generated. 4) Test profile [72], which achieves an even spread of test cases using a specially designed and dynamically adjusted test profile (which is different from the uniform test profile of RT). 5) Metric-driven [66], which uses distribution metrics (that measure the degree of even distribution of a set of points) as criteria for test case selection.

To compare the effectiveness of ART and RT, the F-measure has been the most widely used metric. Various studies have proven that ART is superior to RT in the F-measure and that this advantage of ART "is quite significant and is in no way diminished by any potential challenge to previous experiments' validity" [68].

It is clear that in different situations and environments, the ART algorithm has improved the effectiveness of test case prioritization. However, the improvements in effectiveness vary for different cases. Previous research has proposed many different versions of ART, but each one

has its advantages and disadvantages, or has a suitable type or a non-suitable type of target program, which is not as common in RT. All the research presented has strong evidence, but if the experiments can be applied to real-world programs, the evidence for the different methods would be a great deal stronger.

For solving the time-complex problem of ART, in another study, Chan et al. [73] introduced a method of reducing additional computational costs—Restricted Random Testing (RRT), named 'Forgetting'. Forgetting applies to a limited number of executed test cases, which reduces the computation costs of RRT. It sets a memory parameter, k, which is the maximum number of test cases in the executed set. Using this method, in the processing of ART, the size of the executed set will be controlled, which reduces the computation costs when calculating distances from the element in the candidate set to the element in the executed set. In this paper, Chan et al. [73] show three implementations of Forgetting: Random Forgetting, Consecutive Retention and Restarting. In their experiment, all Forgetting versions show similar results. From the results, we see clearly that the Forgetting method works with RRT. It is a version of ART that uses exclusion regions and restriction of test-case generation to outside these regions [73]. It is also stated that this Forgetting technique does not significantly reduce fault-detection effectiveness [73]. In another paper, Chen et al. [63] state that Forgetting is also suitable for FSCS-ART. With Forgetting, the time complex of ART can be reduced from $O(n^2)$ to O(n), which will have the same time complex as random testing. However, the experiment mentioned in that paper does not use the Forgetting method. As a result, the Forgetting technique is not examined with the FSCS-ART, although it works theoretically.

Chapter 3

A Natural Distance Metric for Real-world Test Suites

3.1 Observation I: Similarities of Neighboring Test Cases

First, observations of test suites from real-world projects, which shows a simple but important property that can support test case prioritization, are presented as follows:

Observation I: Real-world test suites have one important commonality, that is, to the best of our knowledge, never exploited for TCP: Neighboring test cases often have certain similarities while more dispersed test cases tend to be dissimilar.

Consider how real-world test cases would have been generated in their test suite. In white-box testing, for instance, test cases are generated to cover the statements, branches, functions, etc., of the source code. After testers have designed a test case to cover the true branch of an *if* statement S, they would normally consider a second test case to cover the false branch of the same statement. Therefore, these two consecutively designed test cases may execute similar paths or branches before statement S is reached. For real-world programs with complex loop

```
void testme(int a, int b, int c, int d, int e)
{
	if(a > b) /* Condition 1 */
	{
		printf("1T "); /* Condition 1, True Branch */
		if(b > c) /* Condition 2 */
		{
			printf("2T "); /* Condition 2, True Branch */
			if(c > d) /* Condition 3 */
			{
				printf("3T "); /* Condition 3, True Branch */
				if(d > e) /* Condition 4 */
					printf("4T "); /* Condition 4, True Branch */
				else
					printf("4F "); /* Condition 4, False Branch */
			}
			else
				printf("3F "); /* Condition 3, False Branch */
		}
		else
			printf("2F "); /* Condition 2, False Branch */
	}
	else
	{
		printf("1F "); /* Condition 1, False Branch */
		if(b == 1) /* Condition 5 */
		{
			printf("5T "); /* Condition 5, True Branch */
			if(c == 1) /* Condition 6 */
			{
				printf("6T "); /* Condition 6, True Branch */
				if(d == 1) /* Condition 7 */
					printf("7T "); /* Condition 7, True Branch */
				else
					printf("7F "); /* Condition 7, False Branch */
			}
			else
				printf("6F "); /* Condition 6, False Branch */
		}
		else
			printf("5F "); /* Condition 5, False Branch */
	}
	printf("\n");
	return;
}
```

Fig. 1. Illustrative example of a program under test, named “a”.

and branch structures, a basic technique of generating white-box test cases is to traverse the execution tree of the program under test [74, 75]. Sen et al. [75] observed, for instance, that “the feasible executions of a program can be represented as a tree, where the branch points in a program are internal nodes of the tree. The goal is to generate concrete values for inputs which would result in different paths being taken. The classic approach is to use depth first exploration of the paths by backtracking.” Consecutive test cases generated in this way will also exhibit

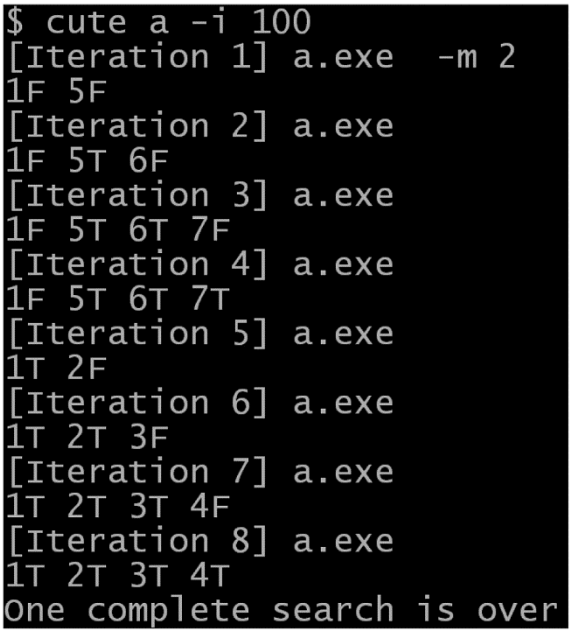


Fig. 2. Screenshot: CUTE generated and executed a total of eight consecutive test cases for a complete search of the execution tree of the program under test, whose source code is shown in Fig. 1. The executable code of the program under test is named a.exe.

similarities. Consider the example shown in Fig. 1 and Fig. 2. This simple illustration could help the readers to better understand Observation I.

The source code in Fig. 1 is from a C program, named "a.c". The program has seven *if* statements, marked as "Condition 1", "Condition 2", ..., "Condition 7". As a result, the program has a total of $7 \times 2 = 14$ branches. In each of these 14 branches, a printf statement is first executed to print the unique ID of the current branch. For example, the third line prints an ID "1T" to the console to indicate that a true branch of Condition 1 has been taken. The IDs "1F", "2T", "2F", ..., "7F" can be explained similarly.

Fig. 2 shows the output of using the Concolic Unit Testing Engine for C (CUTE), an automatic test case generator, for the above program. CUTE was developed by Sen et al. [75].

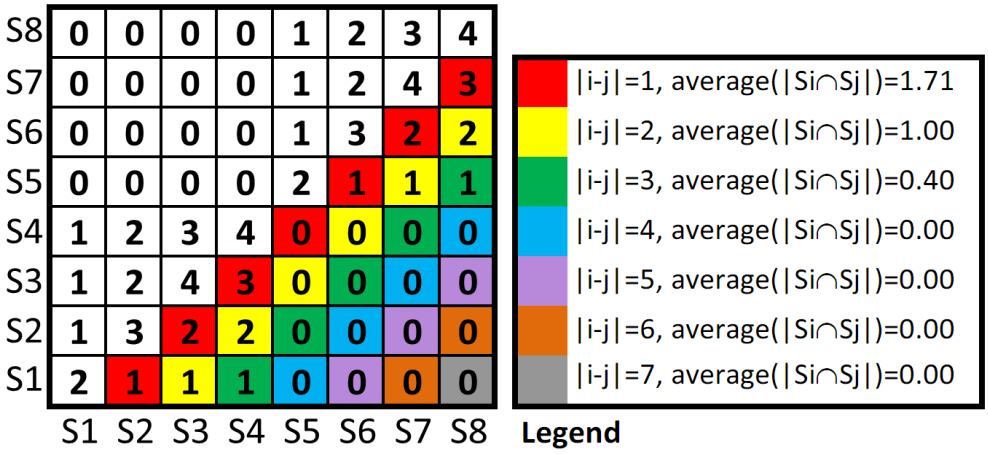


Fig. 3. $|S_i \cap S_j|, i, j = 1, 2, \ldots, 8.$

It combines concrete and symbolic execution techniques to automatically generate and execute white-box test cases. The first line "cute a −i 100" is a command entered by the user to run CUTE, where "a" indicates the name of the program under test, "−i 100" requests that the total number of test cases be no more than 100. The second line, starting with "[Iteration 1]", is an output line of CUTE, indicating that test case 1 is being generated and executed. The message "−m 2" indicates the work mode, which is not relevant to the present discussion. The output of the program under test is shown in the next line, that is, "1F 5F", which indicates that the execution path of test case 1 is branch 1F followed by 5F. The next line, starting with "[Iteration 2]", is again an output line of CUTE, meaning that test case 2 is being generated and executed. The output of the program under test against test case 2 is shown in the next line, which is "1F 5T 6F". This indicates that both test cases 1 and 2 took the false branch at Condition 1. In the end, a total of eight test cases have been generated and executed by CUTE. At the bottom of the screenshot, CUTE prints that the search in the execution tree of the program under test is complete. Let S_i denote the set of branches covered by test case i, i = 1, 2, ..., 8. It can be found that $|S_1 \cap S_2| = 1$, $|S_2 \cap S_3| = 2$, $|S_3 \cap S_4| = 3$, $|S_4 \cap S_5| = 0$, $|S_5 \cap S_6| = 1$, $|S_6 \cap S_7| = 2$, and $|S_7 \cap S_8| = 3$. In most situations, any two consecutively generated test cases have something in common in the branches that they cover (and in the paths that they execute), with the exception of test cases 4 and 5. For more dispersed test cases, they tend to be dissimilar (such as $|S_2 \cap S_8| = 0$ for test cases 2 and 8). This situation is further illustrated using Fig. 3, which gives all of the values of $|S_i \cap S_j|$, i, j = 1, 2, ..., 8. In the figure, because the matrix is

symmetric, only half of it is colored to show that, when the value of |i − j| increases from 1 to 7, the average of $|S_i \cap S_j|$ drops from 1.71 to 0.

For real-world complex and large programs, which have data structures and complicated control flow, a bounded depth-first search strategy can be used for test case generation [75], and the similarities between consecutively generated test cases can become more evident as the lengths of execution paths become large. Generally speaking, it is observed that test cases that are dispersed with respect to their positions in a real-world test suite often have a higher degree of dissimilarity than those that are close together.[1] This statement should not, of course, be absolute as there are always exceptions.

The above observation can also be made when test cases are generated using other techniques, e.g., black-box testing, model-based testing, fault-based testing, or in the context of regression testing [76]. This is because test designers or automated test case generators normally follow a logical or systematic approach when designing/generating test cases. Neighboring test cases, therefore, tend to have similarities in their logic or purpose. For example, test cases that are close together in their relative positions in a black-box test suite may have similarities in the functions that they exercise or the value combinations that they take, and those that are close together in a regression test suite may have been designed around the same period of time for a specific version of the SUT.

In a situation where all of the test cases are randomly generated, the above observation may

[1] This observation is made through participation in more than ten real-life software projects in collaboration with the Australian IT industry. The projects involve the development and testing of Web Application Programming Interfaces and Graphical User Interfaces, where most test cases are manually generated.

not hold true. However, in real-world software testing, random testing (RT) is often used in combination with other techniques such as partition testing and testing with special values. In these situations, nonrandom neighboring test cases may still have similarities. Even in situations where all the test cases are purely random, the testing method (which will be introduced shortly) will still do no harm to the effectiveness and efficiency of the original RT method.

Definition I: Let T = (t_1, t_2, ..., t_n), where n > 0, be a sequence of test cases. The natural distance between test cases t_i and t_j, where $1 \leq i, j \leq n$, is defined as $|i - j|$, the absolute value of $i - j$.

Definition I specifies a natural distance metric. In short, the natural distance between two test cases is the difference in their positions in the test suite.

Let T = (t_1, t_2, ..., t_{10000}) be test suite of a real-world program. Suppose that test case t_9 is selected and executed first, and then no failure is detected. According to Observation I, t_9 and t_{10} could be similar and, therefore, if t_9 does not reveal a failure, it is not wise to select t_{10} as the next test case. Instead, the next test case had better be farther apart from the previously executed test cases that have not yet detected a failure. Of course, there is also a chance that t_9 and t_{10} are very different. However, according to Observation I, these two neighboring test cases would be more likely to be similar.

We propose the following hypothesis.

Hypothesis I: Consider software testing in the real world where test cases are selected from, or prioritized using, a (possibly very large) suite of test cases. If the test cases are selected in such a way that they are evenly spread over the test suite in terms of natural distance, then the test effectiveness will be at least as good as RT.

Note that the concept of evenly spreading test cases is also the basic intuition of adaptive random testing (ART) [62, 68, 77]. ART has been proposed as an enhancement to RT based on the observation that failure-causing inputs tend to form contiguous failure regions. If certain test cases do not reveal any failure, ART recommends the selection of subsequent test cases that are far away from those already executed. In this way, ART generates test cases that are more evenly spread over the input domain than RT. There is a fundamental difference between ART and the present approach: The former generates concrete values of test cases by evenly spreading them across the input domain (and hence, needs to develop different distance calculation methods/metrics for different types of input domains). On the other hand, the latter does not consider the input domain—it generates an execution sequence of test cases (rather than actual test cases with concrete values) based on IDs assigned to them when they were first added to the test pool.

Additionally, the approach in this study is different from adaptive random TCP. Adaptive random TCP uses ART algorithms and coverage information to prioritize test cases [7, 17, 18]. The proposed approach does not need any coverage information.

For test case selection and prioritization, there are different effectiveness metrics, such as the P-measure, the F-measure, and the average percentage of faults detected (APFD) [1, 67, 68]. While Hypothesis I is not restricted to any specific effectiveness metric, APFD and the F-measure are adopted in these empirical studies, as will be explained later in the thesis.

3.2 Observation II: An Order of the Test Cases in Real-world Test Suite

The next question is: how can the order of the test cases in a test suite from a real-world program be identified? Observation II answers this question.

Observation II: An order of the test cases in a real-world test suite can often be decided easily.

Observation II states that deciding the position (ordinal rank) of a test case can often be easy in a real-world test suite. In automated testing, for instance, a test driver (such as a shell script file) is normally provided to run all the test cases. The order of test case executions can, therefore, be regarded as the order of the test cases in the test suite. The majority of the subject packages used in this empirical study are of this category. For instance, in the "mochitest-devtools" test suite of Firefox,[2] one of the test cases assigns a property "unchecked" to the "record snapshot" button; the immediate next test case assigns a property "enabled" to the same button; and the test case that follows makes the same button "visible", and so on. All these neighboring test cases involve setting certain properties of the same button.

When a test driver is not provided, the test case names, such as filenames, function names, and directory structures, can often provide hints on a potentially useful ordering of the test cases. For example, the alphanumeric order of the names can often be a useful indicator of a suitable order of the test cases. The following example shows typical test case names of the Apache Commons Text, a library of algorithms working on strings[3]:

testContains_char,
testContains_String,
testContains_StringMatcher,

[2] [Online]. Available: https://www.mozilla.org

[3] [Online]. Available: https://commons.apache.org/proper/commons-text/

testDeleteAll_char,
testDeleteAll_String,
testDeleteFirst_char,
testReplaceAll_char_char,
testReplaceAll_String_String,
testReplaceFirst_char_char,

and so on. See Section 4.1.4 for more discussion. The sequence of test cases may also be revealed by the dates and times that the test case files were first created.

More than one strategy to identify test case sequences may also be applied by the tester. For example, in a collaborative project (including open source ones), multiple contributors may be working on different parts of the code and adding test cases to the repository simultaneously. In this scenario, we may need to first partition the test cases into different groups according to the contributor ID, and then identify each group's test case sequence using the strategies discussed above.

Chapter 4

Empirical Evaluation of Dispersity-based Prioritization

4.1 Design of Empirical Evaluation

A series of empirical studies have been conducted with real-world software packages, to validate Hypothesis I in the context of TCP. This section describes the design of the empirical evaluation, including dependent and independent variables, the proposed TCP algorithm, subject packages, and how the orders of test cases in the test suites were decided.

4.1.1 Summary of Dependent and Independent Variables for the Empirical Studies

For the empirical studies, the independent variable is the TCP algorithm, namely: 1) RP, 2) dispersity-based prioritization (DBP), and 3) the additional algorithm based on the branch coverage information of the test cases collected from an earlier version of the program under test. Algorithm 2) is the proposed approach, based on the natural distance metric and will be elaborated in Section 4.1.2. Algorithm 3) is widely considered to be one of the best TCP

algorithms.

There are three dependent variables for the empirical studies, they are: 1) applicability of the TCP algorithm (that is, whether the algorithm can be applied to the object under study), 2) TCP effectiveness, evaluated using APFD and the F-measure, and 3) execution time for the detection of the first failure. These will be elaborated in Section 4.1.3.

By referring to the computational complexity of a TCP algorithm, the efficiency is evaluated. Because RP and DBP are both linear algorithms, their efficiency is further compared in Section 4.3 through an additional set of experiments.

The objects are composed of the programs under test and their test suites, which will be described in Section 4.1.3.

4.1.2 The DBP Algorithm

As described above, as the input domain of the program under test are not considered, the dispersity-based approach is not an ART technique. Nevertheless, for the purpose of empirical evaluation, an efficient ART algorithm is applied to generate an adaptive random sequence of integers in the range [1, n], where n is the number of test cases, to serve as a sequence of test case IDs. In this way, an even spread of test case IDs can be achieved, which enables validation of the Hypothesis I.

The algorithm applied in the empirical study is fixed size candidate set (FSCS)-ART enhanced with the “forgetting by consecutive retention” strategy [68, 73], as explained in the next paragraph.

In the ART family, FSCS-ART is a member of the algorithms that work as follows: Whenever a new test case is needed, c candidates are first generated randomly, where c is a

Purpose: This algorithm performs test case prioritization in linear time and space complexity. The FSCS-ART algorithm with the "consecutive retention" forgetting strategy is applied together with the natural distance metric to generate a sequence of test case IDs, where the size of candidate set is 10 and the Memory Parameter in "forgetting" is also 10 (that is, the distance calculation is applied to only the last 10 executed test cases).

Input: A positive integer n.

Precondition: The real-world test suite to be prioritized is a sequence of n test cases, denoted by (t_0, t_1, ..., t_{n-1}).

Output: Array A, which is a sequence of prioritized test case IDs. Therefore, the prioritized order of test cases will be ($t_{A[0]}$, $t_{A[1]}$, ..., $t_{A[n-1]}$).

Begin Algorithm

1. For i = 0, 1, ..., n-1, set A[i] to i;
   /*A[i] is initialized to store the ID of test case ti.*/
2. Randomly select an integer m in the range [0, n-1];
   /*Select the first test case ID, A[m], randomly.*/
3. Swap(A[0], A[m]);
   /*Let A[0] store the ID of the first selected test case.*/
4. Set nbOfSelectedTestCases to 1;
5. While(nbOfSelectedTestCases < n-1)
6. Set memoryParameter to minimum(nbOfSelectedTestCases, 10);
7. Apply FSCS-ART to select the next test case ID from the set {A[nbOfSelectedTestCases], A[nbOfSelectedTestCases+1], ... A[n-1]}, using {A[nbOfSelectedTestCases-memoryParameter], A[nbOfSelectedTestCases-memoryParameter+1], ..., A[nbOfSelectedTestCases-1]} as the set of selected test case IDs; The distance between A[x] and A[y] is given by |A[x]-A[y]|; Let A[r] be the finally selected test case ID, where nbOfSelectedTestCases <= r <= n-1;
8. Swap(A[nbOfSelectedTestCases], A[r]);
9. nbOfSelectedTestCases = nbOfSelectedTestCases + 1;
10. EndWhile;
11. Print("The IDs of the prioritized order of test cases are in the following sequence:");
12. For i from 0 to n-1
13. print(A[i]);
14. EndFor;

End of Algorithm

Fig. 4. Our dispersity-based algorithm for TCP, which is in the same order of time and space complexity as RP, namely, O(n) where n is the number of test cases.

constant. The distances between each candidate and all the already executed test cases are calculated, and the minimum distance is recorded. The candidate having the largest minimum distance is, then, chosen as the next test case, and all the other candidates are discarded. To generate n test cases, the time complexity of FSCS-ART is in $O(n^2)$. The "forgetting by consecutive retention" strategy improves the complexity to $O(n)$ [73]. To apply this strategy, instead of calculating the distances between each candidate and all the already executed test cases, the distance calculation is limited to the last k already executed test cases, where k is a constant known as the memory parameter.

In previous research, it was reported that as c increases up to about 10, the effectiveness of FSCS-ART improves, and then does not improve much further [57]. Another study found that the "forgetting by consecutive retention" strategy is more effective than RT even when k is as small as 10 [78]. Therefore, in the present study, when generating test case IDs, c and k are both given a constant value of 10. The enhanced algorithm in combination with the natural distance metric is shown in Fig. 4.

One single input parameter n is accepted by the algorithm, which is a positive integer. It is assumed that the original test suite is a sequence of test cases (t_0, t_1, ..., t_{n-1}), where i is the ID of test case t_i, i = 0, 1, ..., n − 1.

Statement 1 of the algorithm uses an array A to store the test case IDs. Statement 2 randomly selects the first test case ID, and statement 3 moves the selected test case ID to A[0] (which can be implemented by the following three statements: temp = A[0]; A[0] = A[m]; A[m] = temp;). In this way, the selected test case ID is stored in A[0], and the rest of the array is the set of not-yet-selected test case IDs, from which future test case IDs will be selected. Statement 4 sets

nbOfSelectedTestCases (for “number of selected test cases”) to 1, as one test case ID has been selected. In this way, the following invariant is created for the while loop starting from statement 5: {A[0], A[1], ..., A[nbOfSelectedTestCases−1]} is always the set of selected test case IDs, and {A[nbOfSelectedTestCases], A[nbOfSelectedTestCases+1], ..., A[n−1]} is always the set of not-yet-selected test case IDs.

Statement 6, means that the memory parameter to be used in “forgetting by consecutive retention” [73] is 10, that is, the distance calculation will be applied only to the last ten executed test cases. The variable memoryParameter is set to “minimum(nbOfSelectedTestCases, 10)” because in the beginning, the number of selected test case IDs is less than 10. Statement 7 applies FSCS-ART to select the next test case ID as follows: First, select ten candidates randomly from the set of not-yet-selected test case IDs. Let the ten candidates be A[c_1], A[c_2], ..., A[c_10]. (In the situation where the number of not-yet-selected test cases is smaller than 10, select all of them as candidates.) For each candidate A[c_i], calculate d_i, which is the minimum of

|A[c_i]−A[nbOfSelectedTestCases−memoryParameter]|

|A[c_i]−A[nbOfSelectedTestCases−memoryParameter+1]|

...,

|A[c_i]−A[nbOfSelectedTestCases−1]|.

Let d_j be the maximum value among {d_1, d_2, ..., d_10}. Then, A[c_j] will be selected to be the next test case ID. This A[c_j] is referred to as “A[r]” in statement 7. Because only up to ten candidates and up to ten executed test cases are involved in the distance calculation, the time complexity of statement 7 is constant. Statement 8 again moves the selected test case ID

TABLE I

SUMMARY OF SOFTWARE PACKAGES USED IN EMPIRICAL EVALUATION

row #	project name & version	project website	total size of package (SLOC)	main language(s)	size of test suite	test suite version	number of failing test cases
1	SQLite v3.7.10	http://sqlite.org	140,603	c	787,530	v3.7.15	13
2	g++, GCC v4.8.0	http://gcc.gnu.org	4,781,336	c, c++	51,829	GCC v4.8.0	143
3	gcc, GCC v4.8.0				92,603		93
4	gfortran, GCC v4.8.0			ada	43,032		16
5	libmudflap, GCC v4.8.0			java	1,436		3
6	libstdc++, GCC v4.8.0				8,474		4
7	commons-lang v3.0.1	http://commons.apache.org/proper/commons-lang	56,617	java	2,047	v3.1	2
8	commons-math v3.1	http://commons.apache.org/proper/commons-math	161,968	java	4,534	v3.1.1	4
9	jfreechart v1.0.14	http://jfree.org/jfreechart	147,590	java	2,203	v1.0.15	10
10	joda-time v2.0	http://www.joda.org/joda-time	86,337	java	3,888	v2.1	8
11	Firefox v31.0	https://www.mozilla.org	6,177,736	c++, c	480,575	v31.0	168

TABLE I Continued

SUMMARY OF SOFTWARE PACKAGES USED IN EMPIRICAL EVALUATION

row #	project name & version	project website	total size of package (SLOC)	main language(s)	size of test suite	test suite version	number of failing test cases
12	Autoconf v2.64	http://gnu.org/software/autoconf	7,896	sh perl lisp	503	v2.69	82
13	Autoconf v2.65		7,571				75
14	Autoconf v2.66		7,634				35
15	Autoconf v2.67		7,645				28
16	Autoconf v2.68		7,670				19
17	Automake v1.13	http://gnu.org/software/automake	67,840	sh perl c	1,313	v1.14	336
18	Automake v1.13.1		68,121				336
19	Automake v1.13.2		68,476				329
20	Automake v1.13.3		68,264				56
21	Automake v1.13.4		68,391				47
22	MySQL v5.6.7	http://dev.mysql.com/downloads/mysql	1,721,667	c++ c java	2,554	v5.6.12	361
23	MySQL v5.6.8		1,725,920				257
24	MySQL v5.6.9		1,727,633				195
25	MySQL v5.6.10		1,728,713				129
26	MySQL v5.6.11		1,731,524				34

TABLE I Continued

SUMMARY OF SOFTWARE PACKAGES USED IN EMPIRICAL EVALUATION

row #	project name & version	project website	total size of package (SLOC)	main language(s)	size of test suite	test suite version	number of failing test cases
27	Space v3	http://sir.unl.edu	about 6,199	c	13,551	n/a	645
28	Space v7		about 6,199				163
29	Space v8		about 6,199				95
30	Space v12		about 6,199				33
31	Space v16		about 6,199				503
32	Space v17		about 6,199				196
33	Space v18		about 6,199				33
34	Space v20		about 6,199				210
35	Space v21		about 6,199				210
36	Space v22		about 6,199				64
37	Space v23		about 6,199				274
38	Space v27		about 6,199				34
39	Space v33		about 6,199				32
40	Space v35		about 6,199				212
41	Space v36		about 6,199				90
42	Space v37		about 6,199				92
43	Space v38		about 6,199				36

to the left part of the array, and statement 9 moves the boundary between the prioritized and unprioritized sets rightward.

In statements 11–14, the IDs of test cases are printed in their prioritized order. In other words, the prioritized order of test cases is ($t_{A[0]}$, $t_{A[1]}$, ..., $t_{A[n-1]}$).

The time and space complexity of the above algorithm is a linear $O(n)$, which is in the same order of complexity as RP. This algorithm is also as applicable as RP because it demands as little information as the latter.

4.1.3 Subject Programs, Test Suites, and Evaluation Metrics

In this section, an overview of the subject programs is presented, followed by a discussion of the evaluation metrics. The challenges in designing the controlled experiments and the solutions are presented. Finally, more details of the subject packages are presented.

4.1.3.1 Overview

15 real-world software projects were investigated in the empirical evaluation, involving a total of 43 different versions of SUT. They are summarized in Table I. The SUTs are written in different programming languages and have various sizes and functionality. Their test suites also have various sizes ranging from a few hundred test cases to very large (which can be larger than the number of statements in the source code of the SUT). This set of projects, therefore, can be considered representative of real-world projects.

In Table I, the 15 projects listed are SQLite (row #1), g++ (row #2), gcc (row #3), gfortran (row #4), libmudflap (row #5), libstdc++ (row #6), commons-lang (row #7), commons-math (row #8), jfreechart (row #9), joda-time (row #10), Firefox (row #11), Autoconf (rows #12 to #16, five versions), Automake (rows #17 to #21, five versions), MySQL (rows #22 to #26, five versions), and Space (rows #27 to #43, seventeen versions). All the packages, including the SUTs and test suites, were downloaded from the project websites listed in the third column of Table I. Most packages contain programs, scripts, or other types of files, written in different programming, scripting, or markup languages. The fifth column of Table I lists only the main languages, which

are not exhaustive. Furthermore, a tool named SLOCCount[4] was used to count the source lines of code (SLOC) of the packages.

4.1.3.2 Evaluation Metrics

For the evaluation of the effectiveness of TCP approaches, because the F-measure [68, 79] and APFD [1] are the most common metrics, they are used in the empirical evaluation. The F-measure refers to the expected number of test case executions that need to be run in order to detect the first failure. APFD also measures how quickly failures can be detected, and takes multiple faulty versions into consideration.

The P-measure is another effectiveness metric, which is the probability of detecting at least one failure using a set of test cases [67]. By definition, the P-measure will increase when the number of selected test cases increases, and the difference in P-measure between two techniques also depends on the number of selected test cases. Hence, a range of sizes of test sets need to be used when comparing P-measures. Normally, the P-measure is used to evaluate test case selection (rather than prioritization) techniques where the size of the test set is supposed to be meaningful. Consider, for example, a scenario in branch coverage testing in which the test set achieves 100% branch coverage. Then, the same number of random test cases can be selected and the P-measure used to compare the effectiveness of the branch coverage and RT techniques. For the present study, which is on TCP, the P-measure is obviously not as suitable as the F-measure.

For every TCP technique, the total CPU time (including the time spent in test case selection/prioritization and in SUT execution) consumed to detect the first failure is recorded.

To make statistically meaningful comparisons, every time RP or DBP (which is an improved random technique) was performed, 10 000 trials were conducted, and the F-measure and the mean APFD of the two prioritization methods were computed.

The F-measure of RT by sampling with replacement is given by $\frac{1}{\theta}$, where θ is the failure rate.

[4] [Online]. Available: https://manpages.ubuntu.com/manpages/precise/man1/sloccount.1.html

In the context of TCP, however, test case sampling should be performed without replacement. In this situation, the F-measure of RP can be easily calculated using the approach developed by Zhou [17]. Despite the existence of this analytical solution, in this empirical study, for both the RP and DBP algorithms, the actual number of test cases executed to detect the first failure were recorded for each of the 10 000 trials (this treatment allowed the conduct of further statistical analysis on the results). This number of test case executions in a particular trial is referred to as the "F-count" [79] of that trial. In the rest of this article, the term "F-measure" is used less rigorously to refer to the mean F-count over all trials performed using a particular TCP algorithm for a particular version of the SUT.

Compared with RP and DBP, the additional algorithm is basically a deterministic algorithm. Therefore, any experiment with the additional algorithm involved only one trial.

For the evaluation of the applicability of a TCP technique, the subject packages are inspected to decide whether the technique can be applied. For the RP and DBP algorithms, this process is trivial: We just need to ensure that an ordering of test case executions can be generated. For the additional algorithm, we need to check whether test case coverage data can be collected.

4.1.3.3 Challenges

In controlled experiments with real-world software packages, the challenges are: first, in order to compare the fault-detection effectiveness of different TCP methods, the test suite must be able to detect at least one failure of the SUT. For some of the software packages investigated, it was found that if the test suite and the SUT were from the same version, no failure could be detected. This is because the SUT had already been thoroughly tested against the test suite and passed all of the tests before the package was released.

In a typical development project, regression testing is normally performed by running "old" test cases created for previous versions when changes are made to existing software. This is to ensure that the changes do not harm the existing functionality. Following this practice, the old test suites were applied to newer versions of the SUT to address the "no failure" problem described above. However, it was found that this approach could not detect any failure either.

This is because newer versions of the SUT must have already gone through regression testing and passed all of the test cases before they were released.

In software testing, researchers typically seed artificial defects into the SUT when no failures could be detected [4]. However, in this research, the objective is to investigate real-life projects with real-life test suites that can detect real failures. In empirical software engineering research, the subjects and objects must be representative of the population [80]. Therefore, instead of seeding artificial faults into the SUT, it was decided to use the following strategy: If no failure can be detected, then the test suite will be applied to an earlier version of the SUT. This strategy can produce failures and hence this is probably the best solution if one wants to experiment without seeding artificial faults. This treatment is valid as TCP techniques can be applied in many different scenarios that include but are not limited to regression testing—for instance, when the testing objective is program comprehension or change impact analysis by means of dynamic analysis with test cases [81], or to find behavioral differences between different versions of the SUT [82, 83].

In a general sense, there could be compatibility problems during test case executions when applying a test suite to a different version of the SUT. In this research, the compatibility issue has been carefully considered to ensure that the SUT and the test suite used for experimentation are compatible.

In order to measure APFD, it is also a basic technique and requirement to use the same test suite to test different versions of the SUT [1], which imposes another (and greater) challenge. The measurement of APFD requires that a number of different versions of the SUT be tested using the same test suite, but in some practical situations, this turned out to be impossible due to compatibility problems between the test suite and multiple versions of the SUT (such as abortion of execution caused by unrecognized parameters). Note that the problems here are more challenging than the compatibility problem stated in the preceding paragraph due to the involvement of multiple SUT versions. As a result, APFD was not applied to such programs, listed in rows #1 to #11 in Table I.

4.1.3.4 Subject Packages

In this section, a brief introduction of the projects listed in Table I is presented. Readers may refer to the project websites, as given in the table, for more information. The host machine runs Microsoft Windows 7 Ultimate, on top of which is a virtual machine (VMware Workstation) to run Ubuntu. The SUTs are installed on this platform.

In row #1, the first project is SQLite, which is claimed to be "the most widely deployed SQL database engine in the world." The test suites of SQLite meet many adequacy criteria such as "100% branch test coverage", "boundary value tests", and so on [84]. SQLite has three independent test harnesses. One of these could be downloaded, namely, the TCL Tests, which is in the public domain. The test suite consists of 787 530 test cases, many of which can run multiple times with different parameters to generate several million more test cases. In this thesis, the focus is on the original 787 530 test cases only (where the test oracle is embedded). Note that the size of the test suite (that is, the number of test cases) of SQLite is much larger than the size of the SUT measured in SLOC (140 603). Readers may refer to related literature [18, 24] for more discussions about the importance of using large test suites in empirical studies of TCP techniques.

From rows #2 to #6, five projects are listed, which are subprojects of the GNU Compiler Collection (GCC), where g++ and GCC have emphases on compiling C++ and C programs, respectively, gfortran is a Fortran compiler, libmudflap is a runtime library, and libstdc++ is a standard C++ Library. The total size of GCC (containing all of the above five projects) is 4 781 336 SLOC. For these five projects, the test suites and the SUTs are of the same version, namely, GCC v4.8.0.

A test driver is provided by GCC to run test cases. The test driver has an automated oracle to verify the output of each test case execution, in terms of "expected pass" (this is the result yielded by the majority of the test cases), "expected failure", "unexpected failure", "unexpected pass", "unresolved", and "unsupported". In this research, both "expected pass" and "expected failure" are treated as a passed test because they are both expected behavior, and "unexpected failure" and "unexpected pass" as a detected failure because they are both unexpected behavior. An

“unexpected failure” or “unexpected pass” may not necessarily indicate a fault of the SUT. Instead, they are more likely related to the environmental issues of the platform. Further discussion on the causes of the failures is beyond the scope of the article. Readers may refer to the GCC website for more information. The small number of “unresolved” and “unsupported” test cases were excluded from the study because the reasons behind were unknown according to the GCC test driver report.

From row #7 to row #26, the test oracles of listed projects are also embedded in their test suites. Rows #7 and #8 list the Apache software projects Commons Lang and Commons Math. The former provides extra methods for the manipulation of core Java classes. The latter is a library of lightweight, self-contained mathematics and statistics components. JFreeChart in row #9 is a Java chart library for developers to display professional quality charts in their applications. Joda-Time in row #10 provides a quality replacement for the Java date and time classes.

In row #11, Firefox is a popular web browser, which is also a free and open source software. The Firefox package includes several test suites, and the largest one was used, named “mochitest-plain”, to conduct experiments. Firefox provides a test driver to run all test cases in the test suite, together with an automated test oracle to verify each test result in terms of “pass”, “known failure”, “unexpected failure”, and “unexpected pass”. Similar to the case of GCC, both “pass” and “known failure” were treated as a passed test, and “unexpected failure” and “unexpected pass” as a detected failure.

From row #12 to row #16, Autoconf is “an extensible package of M4 macros that produce shell scripts to automatically configure software source code packages, according to its website. These scripts can adapt the packages to many kinds of UNIX-like systems without manual user intervention.” The same test suite was run on five versions of Autoconf to calculate APFD. Automake (rows #17 to #21) is a tool for “automatically generating Makefile.in files compliant with the GNU Coding Standards. Automake requires the use of Autoconf.” It was possible to run the same test suite on five versions of Automake. Rows #22 to #26 list five versions of MySQL (Community Server), which is claimed to be “a freely downloadable version of the world’s most

popular open source database that is supported by an active community of open source developers and enthusiasts."[5] We were able to run the same test suite on five versions of MySQL.

Space is the last project (rows #27 to #43), which is downloaded from SIR [85, 86]. The Space program was developed by Ingegneria Dei Sistemi, Pisa, Italy, for the European Space Agency [87]. The program consists of about 6199 SLOC written in C, and works as an interpreter for an array definition language (ADL). The downloaded package includes a base version and 38 faulty versions. Each faulty version contains a real fault discovered during the development of the program. In addition, the downloaded package contains a pool of 13 551 test cases. According to SIR [86], the test pool was constructed in two stages: An initial pool of 10 000 test cases was created using a test case generator, capable of generating "random" ADL input files [87]. Then, more test cases were added to the pool so that each executable statement of the base program or edge of its control flow graph would be exercised by at least 30 test cases. In this empirical study with Space, the testing objective is to detect failures of the faulty versions as quickly as possible by running test cases from the existing test suite. In order to detect failures, the base version (not shown in Table I) was used as a test oracle: Every time a test case is run, the output of the faulty version is compared with that of the base version, and any discrepancy means a failure.[6] It was found that, out of the 38 faulty versions, some were equivalent to the base version for all the test cases and, hence, they were excluded from the study. Furthermore, those faulty versions were excluded whose failure rate is higher than 5%. This is because, from the practicing testers' perspective, programs with small failure rates are more interesting than those with high failure rates, as failures of the latter can be detected easily by any testing technique. As a result, only 17 faulty Space versions with failure rates between 0 and 5% were included in this study. The failure rate threshold was set to 5% because "1 in 20" is normally considered a small probability, as

[5] [Online]. Available: https://www.mysql.com/products/community/

[6] In controlled experiments where different testing techniques are compared, it is a common practice to use the base version as an oracle to enable quick verification of large amounts of outputs of the faulty versions.

accepted in modern statistics [88].

4.1.4 Deciding the Order of Test Cases

Computation of the natural distance depends on the position of the test cases in the test suite, that is, the order of the test cases. Chapter 3 described several approaches to identifying such an order, and some of these approaches have been applied to the subject test suites.

For SQLite in the row #1 of Table I, a set of test scripts, which spread over 707 files, performed the test case executions. It was found that seven files do not contain their own test cases, 12 files do not print proper test results, and one file is unstable in the sense that it executes a different number of test cases every time. These files were, therefore, excluded from the study. As a result, a total of 687 files were used to run a total of 787 530 test cases. For test cases within a test script file, their order is defined by their execution order, whereas the order of test script files is defined by the alphanumeric order of their filenames. The alphanumeric order adopted is similar to alphabetical order except that the latter simply sorts the values left to right, character by character, whereas the former recognizes numeric values in filenames. For example, the alphanumeric order (a.test, a1.test, a2.test, a100.test) was used rather than the alphabetical order (a.test, a1.test, a100.test, a2.test). It is assumed that the alphanumeric order of filenames carries useful information about the similarity among test script files. For instance, “corrupt8.test”, “corrupt9.test”, “delete2.test”, and “delete3.test” are some of the filenames of real test scripts. Intuitively, “corrupt8.test” and “corrupt9.test” should have some similarities (such as the testing purpose); “delete2.test” and “delete3.test” should also be similar in certain ways (such as in the operations performed by the test cases).

The same treatment was also applied to the Space package (rows #27 to #43). Each test case of the Space program is an ADL file, and the alphanumeric order of the filenames was taken as the order of the test cases.

Purpose: To decide the order of test cases.
Begin Procedure
1. If (there is no test driver file) then
2. use the alphanumeric order of the test case files;
/*This is because, for our subject packages, the original file creation dates are unknown.*/
3. return;
4. Endif;
5. If (there is only one test driver file) then /*So, there is only one test suite.*/
6. use the test case execution order given by the test driver;
7. return;
8. Endif;

/*Now there are multiple test driver files (hence multiple test suites), each of which runs a number of test cases. This is the case of SQLite.*/

9. set the between-group order (the order of test suites) to be the alphanumeric order of the test driver filenames;
10. set the within-group order (the order of test cases within a test suite) to be the test case execution order given by the respective test driver;
11. return;

End of Procedure

Fig. 5. Deciding the order of test cases for each subject package used in these empirical studies.

To list the test case files for Automake (rows #17 to #21), the alphanumeric order of filenames was also used. Each of these files is a test script with a filename extension ".sh.". Note that a test driver might run multiple test scripts, and each test script might contain multiple test cases.

For the projects listed from row #2 to row #16 and from row #22 to row #26, the order of the test cases is straightforward to identify, because these projects have a test driver and, therefore, the sequence used by the driver to execute the test cases is directly taken as the order of the test cases.

The procedure of deciding the order of test cases for the above subject packages is summarized in Fig. 5.

4.2 Empirical Results

First, TCP approach (DBP) is compared with RP in Section 4.2.1, and then with the additional algorithm in Section 4.2.2.

4.2.1 Comparing DBP with RP

In section 1, RG1 was raised that identified four aspects of a TCP method. We first look at applicability. DBP and RP have the same applicability, and both these techniques are applicable to all 43 subject programs. This is because both DBP and RP can be applied whenever an original sequence of test cases—(t_0, t_1, ..., t_{n-1}), as shown in Fig. 4—is given.

In the next section, effectiveness, efficiency, and execution time for failure detection are discussed.

4.2.1.1 Effectiveness

First, the F-measure results are presented and, then, the APFD results.

4.2.1.1.1 F-measure Results

Table II summarizes the empirical results of testing effectiveness. Columns #3 and #4 show the F-measures of RP (F.RP) and DBP (F.DBP), respectively (out of 10,000 trials each). To compare these two scores, the ratio F.DBP ÷ F.RP is calculated and listed in column #5. When the ratio is smaller than 1, F.DBP is better. When it is greater than 1, F.RP is better. To conduct statistically meaningful comparisons, independent-samples t-tests at a significance level of 5% were performed and the (two-tailed) p-values are listed in column #6, together with the respective effect size (Cohen's d) [89]: d is the absolute value of the difference of the two means divided by the square root of the mean of the two variances. A p-value below 0.05 indicates that the difference between F.RP and F.DBP is statistically significant, and a p-value above 0.05 indicates that these two F-measures are equal (that is, there is no statistically significant difference). For ease of reading, the corresponding cells in columns #5, #6, #8, #9, and #10 are highlighted when the respective p-values are below 0.05—in this way, all statistically significant results are highlighted. While the p-value measures the statistical significance, the effect size (d) measures the practical significance and should be judged in context [90]. In the context of comparing TCP techniques, a d value of 0.1 or above can be considered nontrivial [44]. Therefore, in columns #6 and #10 of Table II, cells with d ≥ 0.10 are marked with an asterisk (*).

TABLE II
F-MEASURE AND APFD RESULTS

1	2	3	4	5	6	7	8	9	10	11
row #	project name & version	F.RP	F.DBP	F.DBP ÷ F.RP	p-value (2-tailed) for F-measure, and effect size (d)	F.Add	APFD.RP	APFD.DBP	p-value (2-tailed) for APFD, and effect size (d)	APFD.Add
1	SQLite v3.7.10	57,401.31	55,304.55	0.96	p=0.005, d=0.04	n/a	n/a	n/a	n/a	n/a
2	g++, GCC v4.8.0	363.36	151.51	0.42	p=0.000, d=0.76 *	n/a	n/a	n/a	n/a	n/a
3	gcc, GCC v4.8.0	1,001.68	979.18	0.98	p=0.104, d=0.02	n/a	n/a	n/a	n/a	n/a
4	gfortran, GCC v4.8.0	2,575.75	2,570.67	1.00	p=0.882, d=0.00	n/a	n/a	n/a	n/a	n/a
5	libmudflap, GCC v4.8.0	360.25	358.74	1.00	p=0.699, d=0.01	n/a	n/a	n/a	n/a	n/a
6	libstdc++, GCC v4.8.0	1,704.33	1,302.08	0.76	p=0.000, d=0.32 *	n/a	n/a	n/a	n/a	n/a
7	commons-lang v3.0.1	686.26	688.97	1.00	p=0.693, d=0.01	n/a	n/a	n/a	n/a	n/a
8	commons-math v3.1	911.13	920.81	1.01	p=0.360, d=0.01	n/a	n/a	n/a	n/a	n/a
9	jfreechart v1.0.14	201.43	205.57	1.02	p=0.115, d=0.02	n/a	n/a	n/a	n/a	n/a
10	joda-time v2.0	436.74	437.31	1.00	p=0.917, d=0.00	n/a	n/a	n/a	n/a	n/a
11	Firefox v31.0	2,876.61	1,663.14	0.58	p=0.000, d=0.52 *	n/a	n/a	n/a	n/a	n/a
12	Autoconf v2.64	6.19	5.93	0.96	p=0.000, d=0.05	n/a	0.972397	0.973838	p=0.000, d=0.07	n/a
13	Autoconf v2.65	6.72	6.48	0.96	p=0.003, d=0.04	n/a				
14	Autoconf v2.66	14.13	14.01	0.99	p=0.519, d=0.01	n/a				
15	Autoconf v2.67	17.29	17.47	1.01	p=0.424, d=0.01	n/a				
16	Autoconf v2.68	25.41	24.59	0.97	p=0.013, d=0.04	n/a				

TABLE II Continued
F-MEASURE AND APFD RESULTS

1	2	3	4	5	6	7	8	9	10	11
row #	project name & version	F.RP	F.DBP	F.DBP ÷ F.RP	p-value (2-tailed) for F-measure, and effect size (d)	F.Add	APFD.RP	APFD.DBP	p-value (2-tailed) for APFD, and effect size (d)	APFD.Add
17	Automake v1.13	3.94	3.75	0.95	p=0.000, d=0.06	1	0.990710	0.991223	p=0.000, d=0.07	0.999619
18	Automake v1.13.1	3.92	3.74	0.95	p=0.000, d=0.06	1				
19	Automake v1.13.2	4.01	3.79	0.94	p=0.000, d=0.07	1				
20	Automake v1.13.3	22.93	21.58	0.94	p=0.000, d=0.06	1				
21	Automake v1.13.4	27.88	27.32	0.98	p=0.132, d=0.02	1				
22	MySQL v5.6.7	7.03	7.02	1.00	p=0.929, d=0.00	n/a	0.990453	0.991448	p=0.000, d=0.15 *	n/a
23	MySQL v5.6.8	9.89	9.65	0.98	p=0.066, d=0.03	n/a				
24	MySQL v5.6.9	13.26	12.74	0.96	p=0.004, d=0.04	n/a				
25	MySQL v5.6.10	20.01	18.93	0.95	p=0.000, d=0.06	n/a				
26	MySQL v5.6.11	72.68	64.79	0.89	p=0.000, d=0.12 *	n/a				
27	Space v3	21.21	19.48	0.92	p=0.000, d=0.09	5				
28	Space v7	82.78	79.26	0.96	p=0.002, d=0.04	8				
29	Space v8	141.50	141.92	1.00	p=0.831, d=0.00	17				
30	Space v12	405.76	231.97	0.57	p=0.000, d=0.54 *	380				
31	Space v16	27.08	26.03	0.96	p=0.004, d=0.04	2				
32	Space v17	68.82	67.98	0.99	p=0.374, d=0.01	244				
33	Space v18	394.83	231.62	0.59	p=0.000, d=0.52 *	380				
34	Space v20	63.95	64.76	1.01	p=0.365, d=0.01	230				

TABLE II Continued
F-MEASURE AND APFD RESULTS

1	2	3	4	5	6	7	8	9	10	11
row #	project name & version	F.RP	F.DBP	F.DBP ÷ F.RP	p-value (2-tailed) for F-measure, and effect size (d)	F.Add	APFD.RP	APFD.DBP	p-value (2-tailed) for APFD, and effect size (d)	APFD.Add
35	Space v21	64.39	64.12	1.00	p=0.758, d=0.00	230	0.986551	0.990271	p=0.000, d=0.88 *	0.984683
36	Space v22	207.88	132.36	0.64	p=0.000, d=0.44 *	8				
37	Space v23	49.74	49.62	1.00	p=0.862, d=0.00	9				
38	Space v27	388.47	246.21	0.63	p=0.000, d=0.44 *	1,802				
39	Space v33	408.16	195.58	0.48	p=0.000, d=0.67 *	143				
40	Space v35	63.60	63.50	1.00	p=0.912, d=0.00	43				
41	Space v36	150.54	148.30	0.99	p=0.279, d=0.02	6				
42	Space v37	148.21	148.80	1.00	p=0.773, d=0.00	1				
43	Space v38	367.45	331.01	0.90	p=0.000, d=0.11 *	29				
			avg	0.90						
			max	1.02						
			min	0.42						

F.RP: F-measure of RP out of 10,000 trials; F.DBP: F-measure of DBP out of 10,000 trials; F.Add: F-measure of the additional algorithm; APFD.RP: mean APFD of RP out of 10,000 trials; APFD.DBP: mean APFD of DBP out of 10,000 trials; APFD.Add: APFD of the additional algorithm. Where there is a statistically significant difference between the RP and DBP means (with a p-value below 0.05), the corresponding cells are highlighted. “p = 0.000” means $p < 0.0005$. Where the effect size (Cohen’s d) is 0.10 or larger, the corresponding cells are starred (*). “d = 0.00” means $d < 0.005$.

In columns #5 of Table II, a total of 23 cells are highlighted, and the values of these 23 cells are all below 1. This means that DBP outperformed RP in the F-measure across all 23 statistically significant cases. Furthermore, all 10 starred cells in column #6 (indicating a practical significance) fall within these 23 cases.

The F-measures of RP and DBP of the remaining 20 subject programs can be considered equal. In the best case, the ratio F.DBP ÷ F.RP can be as low as 0.42 (row #2), which means that, on average, DBP used 58% fewer test cases than RP, or RP used 2.4 (= 363.36 ÷ 151.51) times as many test cases as DBP, to detect the first failure. This indicates that DBP achieved a saving that is both statistically significant and practically significant. In the worst case, the ratio F.DBP ÷ F.RP is 1.02 (row #9). This means that, even in the worst case, DBP used on average only 2% more test cases than RP to detect the first failure, and this difference was neither statistically nor practically significant (and hence can be ignored). The above results prove Hypothesis I.

Looking further into the different orders of test case executions would be useful. Consider SQLite, for example. First the 687 test script files were ordered according to their filenames, and then the test cases within each file according to their execution order. Do either or both of these strategies play an important role? It was found that both these strategies have contributed to the TCP. If we only apply DBP/RP to prioritize the 687 test script files, and then if the test cases within each file are still executed sequentially, then there is no significant difference between the F-measures of DBP and RP. This is because the 13 failure-causing test cases of SQLite are distributed over three test script files and these three files are not clustered (that is, they are not neighbors in the test file sequence). It is known that, for nonclustered failure patterns, ART and RT have similar F-measures [91]. Likewise, if we execute the 687 test script files sequentially and apply DBP/RP to the test cases within each test script file only, then there is no significant difference between the F-measures of DBP and RP either. This is because the first test script file in the sequence to fail SQLite includes only one failure-causing test case and, therefore, applying either DBP or RP to the test cases within this file makes no difference. Due to the sheer size and complexity of the real-world test suites involved in the experiments, this research will not attempt

to further investigate the inner workings of the different orderings of test case executions—such an investigation would require a separate study that is beyond the scope of this thesis.

4.2.1.1.2 APFD Results

As discussed above, for the programs listed in rows #1 to #11 of Table II, APFD was not applied, because of compatibility problems of the real-world test suites across multiple versions of the SUT. This observation indicates that the APFD metric may not be as applicable as the F-measure metric.

For four projects in rows #12 to #43 of Table II, available mean APFD scores of RP and DBP are given in columns #8 and #9. In all four projects, DBP outperformed RP with a constantly higher mean APFD score. Furthermore, as shown in column #10, the APFD differences between RP and DBP were statistically significant across all four projects and, therefore, the relevant cells are highlighted. Furthermore, two of these four projects are starred as they had a nontrivial d value (as shown in column #10).

Although it is found that DBP outperformed RP in APFD, across all four projects, with a statistical significance, and in two of these projects with a practical significance, it also should be pointed out that the differences between APFD.RP and APFD.DBP are very small, and all observed APFD scores are very large. This observation does not mean that RP and DBP have little difference in TCP effectiveness, because their F-measures differ a lot. This phenomenon of large APFD scores associated with large test suites was first reported by Zhou et al. [18], where it was suggested that "APFD may not necessarily be a suitable effectiveness measure or may need to be adjusted in certain situations, such as when the test suites are very large." Nevertheless, the outcomes of the APFD comparisons and those of the F-measure comparisons have been quite consistent: Both these metrics show that DBP outperformed RP with a statistical significance in testing effectiveness, and that 43 to 50% of these statistically significant cases were also practically significant.

4.2.1.2 Efficiency

In terms of computational complexity, DBP and RP have the same order of O(n), where n is the number of test cases prioritized. Further comparison of their efficiency is presented in Section 4.3.

4.2.1.3 Execution Time for Failure Detection

Next, the actual execution times for the detection of the first failure of DBP and RP are compared. The empirical results are shown in Table III. T.RP (column #3) and T.DBP (column #4) are the mean execution times (out of 10 000 trials) spent by RP and DBP, respectively, to detect the first failure, including the SUT execution time and the test case selection time.

In column #5, the ratio T.DBP ÷ T.RP is calculated and listed to compare T.RP and T.DBP. When the ratio is smaller than 1, T.DBP is better. When it is greater than 1, T.RP is better. To conduct statistically meaningful comparisons, independent-samples t-tests at a significance level of 5% are performed and the (two-tailed) p-values are listed in column #6. Cells in columns #5 and #6 are highlighted when their respective p-values are below 0.05, indicating statistically significant differences. The cells highlighted in light gray indicate that DBP outperformed RP with a statistical significance, and that highlighted in dark gray indicates that RP outperformed DBP with a statistical significance. Cells in column #6 are starred when their respective d values are 0.10 or larger, indicating a nontrivial practical significance.

In columns #5 of Table II, a total of 23 cells are highlighted, including only one in dark gray (in row #37) and 22 in light gray. A total of 9 cells are starred, all of which fall within the 22 light gray cases.

In terms of spending less time to detect the first failure, the above results indicate that DBP outperformed RP, and which is quite consistent with the F-measure results. For the remaining 20 subject programs, the mean execution times of RP and DBP can be considered equal. In the best case, the ratio T.DBP ÷ T.RP can be as low as 0.45 (row #2), hence, a saving of 55%. In the worst case, the ratio is 1.04 (row #37), indicating only 4% extra time.

TABLE III
EXECUTION TIME RESULTS (TIME TO THE FIRST FAILURE, IN SECONDS)

1	2	3	4	5	6	7	8
row #	project name & version	T.RP	T.DBP	T.DBP ÷ T.RP	p-value (2-tailed) and effect size (d)	Exe.Add	Preprocessing.Add
1	SQLite v3.7.10	71.346673	67.751497	0.950	p=0.000, d=0.05	n/a	n/a
2	g++, GCC v4.8.0	7.63275	3.433284	0.450	p=0.000, d=0.70 *	n/a	n/a
3	gcc, GCC v4.8.0	26.334649	26.229569	0.996	p=0.777, d=0.00	n/a	n/a
4	gfortran, GCC v4.8.0	81.503814	81.359644	0.998	p=0.894, d=0.00	n/a	n/a
5	libmudflap, GCC v4.8.0	14.379306	14.618975	1.017	p=0.129, d=0.02	n/a	n/a
6	libstdc++, GCC v4.8.0	304.885681	234.552912	0.769	p=0.000, d=0.32 *	n/a	n/a
7	commons-lang v3.0.1	32.739562	32.151085	0.982	p=0.074, d=0.03	n/a	n/a
8	commons-math v3.1	70.94557	70.646528	0.996	p=0.718, d=0.01	n/a	n/a
9	jfreechart v1.0.14	0.629425	0.616548	0.98	p=0.362, d=0.01	n/a	n/a
10	joda-me v2.0	1.060908	1.075443	1.014	p=0.506, d=0.01	n/a	n/a
11	Firefox v31.0	16.024432	10.076629	0.629	p=0.000, d=0.41 *	n/a	n/a
12	Autoconf v2.64	4.83672	4.564996	0.944	p=0.000, d=0.06	n/a	n/a
13	Autoconf v2.65	5.08985	4.812579	0.946	p=0.000, d=0.06	n/a	
14	Autoconf v2.66	10.907391	10.617755	0.973	p=0.047, d=0.03	n/a	
15	Autoconf v2.67	13.418242	13.312348	0.992	p=0.557, d=0.01	n/a	
16	Autoconf v2.68	19.816506	18.837227	0.951	p=0.000, d=0.05	n/a	

TABLE III Continued
EXECUTION TIME RESULTS (TIME TO THE FIRST FAILURE, IN SECONDS)

1	2	3	4	5	6	7	8
row #	project name & version	T.RP	T.DBP	T.DBP ÷ T.RP	p-value (2-tailed) and effect size (d)	Exe.Add	Preprocessing.Add
17	Automake v1.13	2.684771	2.432545	0.906	p=0.000, d=0.06	5.536	9269
18	Automake v1.13.1	2.721153	2.504413	0.92	p=0.000, d=0.05	5.58	
19	Automake v1.13.2	2.822524	2.551866	0.904	p=0.000, d=0.06	5.6	
20	Automake v1.13.3	16.530596	15.554517	0.941	p=0.000, d=0.06	**5.62**	
21	Automake v1.13.4	19.9923	19.456912	0.973	p=0.073, d=0.03	**5.564**	
22	MySQL v5.6.7	20.215466	20.10903	0.995	p=0.863, d=0.00	n/a	n/a
23	MySQL v5.6.8	21.489901	21.112889	0.982	p=0.260, d=0.02	n/a	
24	MySQL v5.6.9	29.073387	28.535064	0.981	p=0.222, d=0.02	n/a	
25	MySQL v5.6.10	37.554342	35.821509	0.954	p=0.002, d=0.04	n/a	
26	MySQL v5.6.11	132.062247	118.960734	0.901	p=0.000, d=0.10 *	n/a	
27	Space v3	0.004122	0.003869	0.939	p=0.000, d=0.07	1.537	
28	Space v7	0.014429	0.01407	0.975	p=0.070, d=0.03	1.845	
29	Space v8	0.023454	0.023929	1.02	p=0.151, d=0.02	2.806	
30	Space v12	0.087596	0.050711	0.579	p=0.000, d=0.53 *	36.713	
31	Space v16	0.005111	0.004974	0.973	p=0.0498, d=0.03	1.188	
32	Space v17	0.010594	0.01068	1.008	p=0.563, d=0.01	24.387	
33	Space v18	0.06977	0.041618	0.596	p=0.000, d=0.50 *	36.768	

TABLE III Continued
EXECUTION TIME RESULTS (TIME TO THE FIRST FAILURE, IN SECONDS)

1	2	3	4	5	6	7	8
row #	project name & version	T.RP	T.DBP	T.DBP ÷ T.RP	p-value (2-tailed) and effect size (d)	Exe.Add	Preprocessing.Add
34	Space v20	0.025857	0.025962	1.004	p=0.772, d=0.00	22.965	5
35	Space v21	0.012606	0.012803	1.016	p=0.270, d=0.02	22.9	
36	Space v22	0.045247	0.029189	0.645	p=0.000, d=0.43 *	1.854	
37	Space v23	0.014726	0.015314	1.04	p=0.006, d=0.04	1.955	
38	Space v27	0.077955	0.050109	0.643	p=0.000, d=0.43 *	160.595	
39	Space v33	0.074699	0.036271	0.486	p=0.000, d=0.66 *	14.863	
40	Space v35	0.011852	0.012003	1.013	p=0.361, d=0.01	5.376	
41	Space v36	0.025377	0.025727	1.014	p=0.319, d=0.01	1.633	
42	Space v37	0.045219	0.04565	1.01	p=0.493, d=0.01	1.088	
43	Space v38	0.063167	0.058255	0.922	p=0.000, d=0.08	4.037	
			avg	0.905			
			max	1.040			
			min	0.450			

T.RP: mean time to the first failure by RP, out of 10,000 trials (= test case selection time + test case execution time); T.DBP: mean time to the first failure by DBP, out of 10,000 trials (= test case selection time + test case execution time); Exe.Add: time to the first failure by the additional algorithm (= test case selection time + test case execution time); Preprocessing.Add: preprocessing time taken by the additional algorithm to collect test case coverage data. Where there is a statistically significant difference between the RP and DBP means (with a p-value below 0.05), the corresponding cells are highlighted (light gray: DBP outperformed RP; dark gray: RP outperformed DBP). “p = 0.000” means $p < 0.0005$. Where the effect size (Cohen’s d) is 0.10 or larger, the corresponding cells are starred (*). “d = 0.00” means $d < 0.005$.

4.2.1.4 Summary

In summary, in terms of applicability, DBP is the same as RP, and DBP has the same order of computational complexity as RP, but significantly outperformed RP in both effectiveness (in terms of the F-measure and APFD) and execution time for failure detection. DBP often achieved large savings in both the F-measure and the execution time. Overall, its F-measure performance (in terms of the F.DBP ÷ F.RP ratio) was slightly better than its execution time performance (in terms of the T.DBP ÷ T.RP ratio), and this was expected because the DBP test case selection algorithm involves more computations than RP although they are both linear algorithms. Nevertheless, it is interesting to observe that, for some programs, such as Autoconf and Automake, DBP's execution time performance was better than its F-measure performance. This was because DBP selected test cases that consumed less execution time than those selected by RP.

4.2.2 Comparing DBP with the Additional Algorithm

This section addresses the research goal RG2.

4.2.2.1 Applicability

We found that the additional algorithm has limitations in its applicability, while DBP was applicable to all the subject programs. This is because the additional algorithm requires that the test case coverage data be known prior to TCP. In the context of regression testing, if a test suite was used in the past on a previous version of the SUT, then the previous test coverage data could be used by the additional algorithm, in order to prioritize the test cases for the new version of the SUT [1]. It was found that such a strategy has its limitations when applied to our real-world subject programs. As explained previously, for packages listed from row #1 to row #11 in Table II, the same test suites could not run across multiple SUT versions due to compatibility problems. Therefore, the additional algorithm could not be applied to these packages. Furthermore, for the packages Autoconf and MySQL, test case coverage data could not be obtained (such as by using gcov, a standard utility used in concert with GCC). As a result, the additional algorithm could only be applied to the Automake programs (rows #17 to #21) and the Space programs (rows #27

to #43).

For collecting coverage data and in order to apply the additional algorithm, the test suite was run on a previous version of the SUT (v1.12.6) of Automake. For Space, the base version Space program was used to collect the test case coverage data so that the additional algorithm could be applied to the faulty Space versions.

4.2.2.2 Effectiveness

The F-measure and APFD were used to evaluate the effectiveness of the TCP methods. For the Automake package, the additional algorithm had a perfect F-measure of 1 across all five versions of the SUT (see the F.Add scores in column #7 of rows #17 to #21, Table II), as well as the best APFD (column #11).

The results are more involved in the Space package. On the one hand, out of the 17 faulty versions of the SUT, the additional algorithm outperformed DBP in the F-measure for 11 versions, whereas DBP outperformed the additional algorithm for only six versions; in the best case, the additional algorithm again achieved a perfect F-measure of 1 (row #42). On the other hand, DBP outperformed the additional algorithm in terms of APFD: the APFD of DBP (column #9) is 0.990271, which is the best among all three algorithms. As mentioned previously, it is observed that, when the test suites are large, the APFD scores of different TCP techniques are very close, and therefore APFD may not necessarily be a suitable effectiveness measure in this situation. In any case, in terms of effectiveness, the additional algorithm had better F-measures than DBP in the majority of cases. This is because the additional algorithm makes use of test case coverage information, whereas DBP does not use such information.

It should be noted that, with the additional algorithm, t-tests were not performed for comparisons. This is because, as a reminder, the additional algorithm is basically a deterministic algorithm and, therefore, only one trial was run for each version of the SUT; in contrast, 10 000 trials of DBP were run and a mean was calculated.

4.2.2.3 Efficiency

As mentioned above, in terms of time complexity for the (branch-coverage-based) additional algorithm, it is $O(n^2m)$, where n is the number of test cases in the test suite and m is the number of branches of the SUT. In comparison, DBP has a linear complexity $O(n)$, where n is the number of prioritized test cases.

4.2.2.4 Execution Time for Failure Detection

In column #7 of Table III, Exe.Add is the execution time spent by the additional algorithm to detect the first failure, which includes the SUT execution time and the test case selection time.

Additionally, for the additional algorithm, as explained above, the test case coverage data is required to be available prior to TCP, as thus such data were collected by running all the test cases on a previous (instrumented) version of the SUT, and the time taken is shown as Preprocessing.Add in column #8 (it may not necessarily be counted as a cost of the additional algorithm as it is assumed that such information is available before the start of the TCP process). For Automake, each of its test case files allowed the user to either enable or disable the test coverage generation feature. To collect T.RP and T.DBP, we disabled this feature in order to run the test cases as fast as possible; to collect Preprocessing.Add, we enabled this feature and, subsequently, the system generated a total of 1313 HTML files corresponding to the coverage profiles of the 1313 individual test cases, at a cost of 9269 s as shown in Table III. Note that this does not include the execution time of my own code that read the 1313 HTML files to extract the branch coverage data. For the Space program, the gcov tool was used to collect the test coverage data. The time (5s) shown in Table III does not include the execution time of the code that processed the intermediate system files to extract the branch coverage data after each test case execution. Even if we do not consider this Preprocessing.Add, DBP still outperformed the additional algorithm very obviously across all subject programs except on Automake v1.13.3 and v1.13.4 (rows #20 and #21), where the two Exe.Add scores are printed in bold.

4.2.2.5 Summary

In short, from the empirical results, it shows that in, addition to efficiency, DBP strongly outperformed the additional algorithm with respect to applicability and execution time for failure detection. On the other hand, it is hardly surprising from the results that the heavyweight additional algorithm was generally more effective than the proposed lightweight approach. In any case, it is pleasing to find that DBP was more effective than the additional algorithm in 27.3% (= 6 ÷ (5 + 17)) of the situations where the latter was applicable.

4.3 Further Comparison of TCP Efficiency Between RP and DBP

In terms of computational complexity, although DBP has the same order as RP, for DBP more steps are involved in selecting a test case. In this section, therefore, we compare the execution times of the DBP and RP algorithms for prioritizing the same number of test cases, without involving actual test suites, SUTs, or failure detections. The following test suite sizes were considered: 1000, 10 000, 100 000, 1 000 000, and 10 000 000. For each test suite size, the RP and DBP algorithms were run to prioritize the entire test suite[7] and the times taken recorded. For each test suite size and each algorithm, 1000 trials were conducted. The mean values of the execution times are compared as shown in Table IV. They are named “driver time” because the results do not involve any actual SUT execution.

In Table IV, for prioritizing the same number of test cases, the results indicate that RP can be up to 63 times faster than DBP. However, if we compare Tables II and III, we can see that this advantage of RP does not help much with its overall failure-detection time as compared with DBP. This is because, for real-world large and complex software, the time consumed by the linear-

[7] It should be noted that no actual test suites were involved in the experiment, because to apply the DBP and RP algorithms does not need to access the actual test cases.

complexity RP and DBP test drivers for test case selection is trivial when compared with the time spent for the actual test case execution.

TABLE IV

COMPARING THE DBP AND RP TEST DRIVER EXECUTION TIMES FOR PRIORITIZING THE SAME NUMBER OF TEST CASES (1000 TRIALS PER ALGORITHM WITH RESPECT TO VARIOUS TEST SUITE SIZES)

size of (virtual) test suite	**mean DBP driver time ÷ mean RP driver time**
1000	40.37
10 000	60.37
100 000	62.65
1 000 000	58.17
10 000 000	32.65

Actual SUTs and failure detections are not involved.

More formally, for the RP, let x be the average time consumed by the test driver to select a test case, then, as stated in Table IV, that of DBP can be up to 63x. Let y and z be the average time to execute a test case and to verify a test result, respectively. The total time consumed by RP and DBP to process a test case is, therefore, $x + y + z$ and $63x + y + z$, respectively. Let k be the F-measure of RP. According to the "avg" result shown in Table II, the F-measure of DBP can be estimated as 0.9k. For RP to more quickly detect a failure than DBP, the following relation must be satisfied: $k(x + y + z) < 0.9k(63x + y + z)$, which gives $y + z < 557x$. This means that, if, on average, the total time to execute a test case and verify a test result is smaller than 557x (which is mainly the time of generating 557 random numbers), then RP will detect a failure more quickly; if larger, DBP will detect a failure more quickly. For real-world large and complex programs, the test case execution time plus the result verification time is generally far longer than the time of generating 557 random numbers. In these situations, therefore, DBP can be used to replace RP.

For instance, in the platform used, it takes 0.0000088 s to generate 557 random numbers. The mean time of executing a test case for each of the 43 programs under test were also calculated. Of these 43 mean results, the minimum, maximum, and average are 0.0001507 (= 0.0000088 × 17.125) s, 2.7974794 s, and 0.4333023 s, respectively.

4.4 Threats to Validity

In this thesis internal validity refers to whether causal relations are properly examined, it is mainly concerned with the correctness of the software tools that we developed to conduct the experiments. To avoid faults in these software tools, their code has been carefully reviewed and tested. Furthermore, all the resulting F-measures of RP have been checked against the mathematical expectations of F-measures (for random sampling without replacement) calculated using the approach developed by Zhou [17]. The discrepancies are very small and can be ignored. All the intermediate and final empirical data have also been carefully checked for correctness and consistency.

In this research, the p-value and the effect size when performing a t-test were calculated, but the power was not analyzed [89]. This is because, when the sample size is large, the observed effect size is a good estimator of the true effect size, but there is no guarantee that the observed power is a good estimator of the true power—Yuan and Maxwell [92] provided a detailed examination of the post hoc power (the observed power) and concluded that the observed power typically provides little useful information about the true power of a single study.

There are some basic assumptions in independent-samples t-tests. First, the observations must be independent. By examining the designs of the experiments, it was confirmed that this assumption is satisfied. Second, the populations from which the samples are taken should be normally distributed [93]. Note that "it is not in fact necessary for the distribution of the observed data to be normal, but rather the sample values should be compatible with the population (which they represent) having a normal distribution" [94]. If the sample data are approximately normal, the sampling distribution will be normal too [93]. Furthermore, according to the central limit

theorem, the sampling distribution will tend to be normal regardless of the population distribution if the sample size is above 30; in other words, in big samples, the sampling distribution tends to be normal anyway regardless of the shape of the data that are actually collected [88]. This means that “if we have samples consisting of hundreds of observations, we can ignore the distribution of the data” [93]. As the sample size used in the experiments (Tables II and III) was 10 000, the assumption of normality is satisfied. The third assumption is homogeneity of variance, that is, variances in groups should be approximately equal. It was found that the experimental data did not perfectly satisfy this assumption. Nevertheless, for a large number of different situations, the t-test is robust enough even if the assumption of homogeneity of variance is untenable [95]. This includes situations, for example, where sample sizes are large (above 25 or 30) and equal or nearly equal, and two-tailed hypothesis is considered [95-97]. In this research, all sample sizes are equal and large, and two-tailed hypothesis has been used. Furthermore, in situations where equal variances are not assumed in a t-test, the SPSS software package provides users with an alternative t-value, which compensates for the fact that the variances are not the same [98].

Regarding external validity, the main concern is the ability to generalize the findings. It should be noted that all the subject programs used in the empirical studies have been taken from the public domain. To enhance external validity, further studies with programs from other sources will be needed. This will require collaboration work with the industry.

Chapter 5

Case Study on Naming Convention of Real-world Projects

In this chapter, we use a new set of real-world programs to study the naming convention of their test cases, supporting Observation I.

5.1 Objectives of The Case Study

A case study was conducted to support the universality of the observation: “Neighboring test cases often have similarities, while more dispersed test cases tend to be dissimilar”; the scope of the study was extended to examine the naming convention of test cases for a new set of programs with different coding languages.

5.2 Experimental Design

The previous empirical study used 15 real-world programs with 43 different versions of SUT. In this study, the empirical study was extended to include more real-world projects, and study the naming conventions of the test cases. Test cases from 50 projects were collected for analysis of their test case naming convention. Two methods were used to collect the test cases: 1) if the test

driver exists in the program, the order of the test case execution could be regarded as the order of the test cases in the test suite. 2). When the test driver is not provided, the alphanumeric order of the test case names will be used to indicate a suitable order of the test cases. In this case study, we focus on studying the similarity of the testing function/target of neighboring test cases.

5.3 Subject Packages

This section provides a brief introduction of the 50 real-world projects used in the study. All projects are listed in Table V, and more details can be found on the project websites. The host machine for the SUTs runs Microsoft Windows 10 Enterprise and uses a virtual machine (VMware Workstation) to run Ubuntu.

When selecting these projects, the projects must have test cases in their source code package, either executable by their test driver, or clearly indicated as test cases from the package file structure or readme/introduction. Some investigated projects were not included in the experiment, because there were no test cases in their downloaded packages, that is, no test driver or test files in their package.

Gawk v5.0.1

Gawk is the GNU implementation of awk, which is used for making changes in various text files, which according to its website, "interprets a special-purpose programming language that makes it possible to handle simple data-reformatting jobs with just a few lines of code"[8]. The test driver exists in the test suite, so the order of the test cases is the same as the executed order in the test driver. There are 492 test cases in the test suite, and the test case names were extracted from the output of the test driver execution.

Gnuastro v0.10

The Gnuastro is short for GNU Astronomy Utilities, according to the project homepage; it is

[8] [Online]. Available: https://www.gnu.org/software/gawk/

"an official GNU package consisting of various programs and library functions for the manipulation and analysis of astronomical data"[9]. There are 59 shell scripts in the testing folder, which are separated by different sub-folders. We consider the folder_name/script_file_name as the test case names, and compare the names of neighboring test cases.

Gnurl v7.66.0

Gnurl, which is also known as libgnurl, is a fork of libcurl. It supports HTTP and HTTPS with a single crypto backend[10]. The test driver exists in the test suite, so the execution order was used as the order of test cases. The test cases name is simple with "test" and a four digit number, such as "test 0001". There are 488 test cases in the execution output, although the output said there are 1245 test cases in total. After checking the details of the output, some of the test cases are not executed by the test driver and the four digital number in the test case names is not continuous, e.g., "test 0098", "test 0099", "test 0151" and "test 0152" is a set of four sequentially executed test cases in the test driver.

Grep3.3

Grep is a popular command-line utility widely used in Unix, Linux, and other Unix-like systems that searches input files for lines that contain a given pattern[11]. There are two folders that contain test cases, named "tests" and "gnulib-tests". The test driver will execute test cases from both of the two folders, and in total, there are 283 test cases extracted from the execution output of the test driver. There are three statuses of the test cases in the output, which are PASS, SKIP and XFAIL. As per the previous empirical study, there is another status FAIL but as no test case failed in this experiment, the FAIL status is not shown in this study.

---

[9] [Online]. Available: https://www.gnu.org/software/gnuastro/

[10] [Online]. Available: https://gnunet.org/en/gnurl.html

[11] [Online]. Available: https://www.gnu.org/software/grep/

TABLE V
SUMMARY OF SOFTWARE PACKAGES USED IN CASE STUDY

Row #	project name and version	project website	size of test suite
1	gawk v5.0.1	https://www.gnu.org/software/gawk/	492
2	gnuastro v0.10	https://www.gnu.org/software/gnuastro/	59
3	gnurl v7.66.0	https://gnunet.org/en/gnurl.html	488
4	grep v3.3	https://www.gnu.org/software/grep/	283
5	grub v2.04	https://www.gnu.org/software/grub/	82
6	libredwg v0.8	https://www.gnu.org/software/libredwg/	45
7	rush v2.1	https://savannah.gnu.org/projects/rush/	68
8	tar v1.32	https://www.gnu.org/software/tar/	234
9	texinfo v6.6	https://www.gnu.org/software/texinfo/	161
10	wget2 v1.99.2	https://www.gnu.org/software/wget/	24
11	bison v3.4.2	https://www.gnu.org/software/bison/	584
12	datamash v1.5	https://www.gnu.org/software/datamash/	20
13	libmicrohttpd v0.9.67	https://www.gnu.org/software/libmicrohttpd/	16
14	gdb v8.3.1	https://www.gnu.org/software/gdb/	28
15	direvent v5.2	https://www.gnu.org.ua/software/direvent/	138
16	gsl v2.6	https://www.gnu.org/software/gsl/	58
17	gettext v0.20	https://www.gnu.org/software/gettext/	680
18	libidn2 v2.2.0	https://www.gnu.org/software/libidn/	9
19	libtasn1 v4.14	https://www.gnu.org/software/libtasn1/	29
20	mailutils v3.7	https://mailutils.org/	1238
21	guile v2.2.6	https://www.gnu.org/software/guile/	251
22	nettle v3.5	https://www.lysator.liu.se/~nisse/nettle/	102
23	pies v1.4	https://www.gnu.org.ua/software/pies/	165
24	cflow v1.6	https://www.gnu.org/software/cflow/	42
25	dico v2.9	https://puszcza.gnu.org.ua/software/dico/	238
26	cssc v1.4.1	https://www.gnu.org/software/cssc/	1314
27	coreutils v8.31	https://www.gnu.org/software/coreutils/	948
28	Gzip v1.10	https://www.gnu.org/software/gzip/	22
29	bash v5.0	https://www.gnu.org/software/bash/	81
30	MPFR v4.0.2	https://www.mpfr.org/	180
31	Commons-BCEL v6.4.1	https://commons.apache.org/proper/commons-bcel/	33
32	commons-net v3.6	https://commons.apache.org/proper/commons-net/	43
33	commons-beanutils v1.9.4	http://commons.apache.org/proper/commons-beanutils/	97
34	commons-chain v1.2	https://commons.apache.org/proper/commons-chain/	20
35	commons-codec v1.13	https://commons.apache.org/proper/commons-codec/	57

TABLE V CONTINUED
SUMMARY OF SOFTWARE PACKAGES USED IN CASE STUDY

36	commons-collections4 v4.4	https://commons.apache.org/proper/commons-collections/	171
37	commons-compress v1.19	https://commons.apache.org/proper/commons-compress/	139
38	commons-configuration2 v2.6	https://commons.apache.org/proper/commons-configuration/	81
39	commons-csv v1.7	https://commons.apache.org/proper/commons-csv/	15
40	commons-fileupload v1.4	https://commons.apache.org/proper/commons-fileupload/	15
41	commons-dbcp2 v2.7.0	https://commons.apache.org/proper/commons-dbcp/	46
42	commons-text v1.8	https://commons.apache.org/proper/commons-text/	80
43	commons-pool2 v2.7.0	https://commons.apache.org/proper/commons-pool/	22
44	commons-vfs v2.4.1	https://commons.apache.org/proper/commons-vfs/	89
45	commons-imaging v1.0	https://commons.apache.org/proper/commons-imaging/	114
46	openhab-core v2.5.0	https://github.com/openhab/	161
47	tomcat 10 build 5092	http://tomcat.apache.org/	1194
48	ofbiz v18.12	https://ofbiz.apache.org/	592
49	spark v2.4.5	https://spark.apache.org/	159
50	ant v1.10.7	https://ant.apache.org/index.html	1185

GRUB 2.04

GRUB is a Multiboot boot loader (boot loader is the first software program that runs when a computer starts)[12]. The test case names and order are from the execution order of the test driver and the output of the execution. There are 82 test cases according to the output of the execution of the test suite, and there are three states for all of the test cases: PASS, FAIL and SKIP. As the focus in this study is on the naming convention, the cause of the failed test cases were not investigated.

[12] [Online]. Available: https://www.gnu.org/software/grub/

LibreDWG 0.8

LibreDWG is a C library that handles AutoCAD format files, named DWG files[13]. The names of test cases are extracted from the output of the test driver, and the order of test cases is the same as the execution order. There are three parts in the test, containing: 2 test cases, 1 test case and 42 test cases, which is 45 test cases in total. As they all come in the single execution of the test suite, we extract all the test cases and put them together as a single test suite, and the execution order is considered as the order of the test cases.

Rush v2.1

Rush is the abbreviation of Restricted User Shell, which is "intended for use with ssh, rsh and similar remote access programs"[14]. There are 68 test cases according to the output of the test suite. All test cases have been separated to different aspect of the testing target, such as "Lexical structure", "Base", "Simple Conditions" and so on, but all test cases have the number from 1 to 68 at the beginning of each line of the output of the test driver. From the sections we can learn the functions/targets that the test cases are testing for.

Tar v1.32

"Tar" originally stands for tape archiver, which also the name of a popular utility that creates tar archives and support various other manipulations[15]. 234 test cases were collected from the execution output of the test driver. Some of the test cases have been separated into different sections, such as "Options", "Option compatibility", "Checkpoints" and so on. These sections show clearly the function or target being tested. Some of the test cases are "skipped" according to the output, but it will not affect this study as they are still named in the output of the test driver.

Texinfo v6.6

Texinfo can generate various output formats using a single source file, and it is the official

[13] [Online]. Available: https://www.gnu.org/software/libredwg/

[14] [Online]. Available: https://savannah.gnu.org/projects/rush/

[15] [Online]. Available: https://www.gnu.org/software/tar/

documentation format of the GNU project[16]. According to the output of the test driver, the test cases are in different folders, such as ""install-info/tests", "tp/tests" and so on. The test driver runs all test cases in different folders as a single part of the entire execution. There are 161 test cases in total, and the naming of the test cases is different for each folder. Some test cases are named with number as a variable, such as "ii-0001-test", "ii-0002-test", …, "ii-0057-test". Others are named by the function/target that the test cases are testing, e.g. "contents_double_contents_chapter.sh" and "contents_double_contents_section.sh". There are two different results of the test cases, PASS and SKIP, but all test cases' names are contained in the output of the execution.

Wget2 v1.99.2

Wget is a non-interactive command line tool for retrieving files using the most widely used Internet protocols, including HTTP, HTTPS, FTP and FTPS[17]. 24 test cases were collected from the output of the test driver. According to the output, there are two folders that contain test cases, named "fuzz" and "unit-tests". The folder "fuzz" has 20 test cases, and 14 of them come with a status of SKIP and 6 were PASS. The folder "unit-tests" has the other 4 test cases, and the status of test cases is PASS.

Bison v3.4.2

Bison is a parser generator, which can be used to "develop a wide range of language parsers, from those used in simple desk calculators to complex programming languages"[18]. The order of the test cases is extracted from the execution of the test driver from the test suite. There are two sections in the output, and the total number of test cases is 584. The first section contains 9 test cases, and all these test cases are in different folders and they are .test file. The second section of the output contains 575 test cases, and these test cases have been separated into different parts,

[16] [Online]. Available: https://www.gnu.org/software/texinfo/

[17] [Online]. Available: https://www.gnu.org/software/wget/

[18] [Online]. Available: https://www.gnu.org/software/bison/

such as “Input Processing”, “Named references tests”, “Output file names” and so on. Both of the two sections are in a single execution of the test driver, and they are use different naming methods, the first section uses the folder name/file name as the test case name, and the second section names the test cases following testing target/aspect and test case name.

Datamash v1.5

Datamash is a command-line utility that “performs basic numeric, textual and statistical operations on input textual data files”[19]. 20 test cases were extracted from the output of the test driver. 18 test cases were PASS, and 2 of them were SKIP. From the “Test suite Summary” section we can know that there are 6 different statuses for executed test cases, they are PASS, SKIP, XFAIL, FAIL, XPASS and ERROR. The name of the test cases is folder name / file name.

Libmicrohttpd v0.9.67

Libmicrohttpd is a C library that provides an easy way to run an HTTP server as part of another application[20]. There is one test driver in the test suite, and the test cases are extracted from the output of the execution of the test driver. There are 16 test cases, and they are all in PASS status. The names of these test cases are start with “test_”, followed by the target/function that the test case is testing for, e.g. test_str_compare.

GDB v8.3.1

GDB is the GNU Project debugger, which allows user to “see what is going on `inside' another program while it executes -- or what another program was doing at the moment it crashed”[21]. There is a test driver in the test suite, so the order of the test cases is the same as the executed order in the test driver. Some parts of the output indicate that hundreds of test cases have been executed but without details of the test case, such as “./test-demangle: 331 tests, 0 failures”. It was not possible to find out their names from the test suite, but most of them are in a single file

[19] [Online]. Available: https://www.gnu.org/software/datamash/

[20] [Online]. Available: https://www.gnu.org/software/libmicrohttpd/

[21] [Online]. Available: https://www.gnu.org/software/gdb/

with no name for each test case, and only some of them have comments in the script, such as "This is from gcc PR 8861", which is more like a description or note, instead of a test case name, so these test cases were ignored. From the output of the test driver, there are 28 test cases. The name starts with "test-", followed by the test target/function and number, e.g. test-expandargv-0, test-expandargv-1.

Direvent v5.2

According to its website, "Direvent monitors events in the file system directories. For each event that occurs in a set of pre-configured directories, the program calls an external program associated with it, supplying it with the information about the event and the location within the file system where it occurred"[22]. There are 3 sections in the output of the test driver, and each section contains a different number of test cases, which are 81, 36 and 21 test cases, respectively. In the first section, all test cases are in the same part, but in the second and third sections, most test cases have been separated into different sub-sections with the name of the target/function that the test cases are testing for, such as "Formats", "Options" and so on.

GSL v2.6

GSL is the abbreviation of GNU Scientific Library. It is a numerical C and C++ library, which provides "a wide range of mathematical routines such as random number generators, special functions and least-squares fitting"[23]. There are over 1000 functions in total. There are 55 sections in the output of the execution of the test driver of the GSL, and most of them contain 1 test case; several sections contain 2 test cases. Each section is testing a single sub-folder of the program, which means it is testing single function of the project. The number of the test cases extracted from the output is 58. For each section, there is a summary part that indicates there are 6 different statuses for executed test case, they are PASS, SKIP, XFAIL, FAIL, XPASS and ERROR. Since in most of the sections, there is only one test case and the name of the test case is "test", the folder

[22] [Online]. Available: https://www.gnu.org.ua/software/direvent/

[23] [Online]. Available: https://www.gnu.org/software/gsl/

name was used as their test case name. When there is more than one test case in single section, the [folder name].[test case name], e.g. vector.test, vector.test_static was used. In the last section of the output, there are two test cases that are executed in the root folder of the program, so I used their test cases' name directly, which is test_gsl_histogram.sh and pkgconfig.test.

Gettext v0.20

Gettext is a well-integrated set of tools and documentation that help to produce multi-lingual messages[24]. 680 test cases were collected from 5 sections of the output of the test driver, and the order of the test cases is the same as they are executed in the test driver. In the first section, there is no test case executed, and the second section only contains 1 test case. There are 433 test cases in the third section, and for the other two sections, there are 14 and 232 test cases, respectively. For each section, the summary part indicates the total number of executed test cases, and their status, which are PASS, SKIP, XFAIL, FAIL, XPASS and ERROR.

Libidn2 v2.2.0

Libidn2 is a utility that encodes and decodes internationalized domain names.[25] 2 sections were found in the output of the test driver with 9 test cases. The first section contains 6 test cases and the second one has 3 test cases.

Libtasn1 v4.14

Libtasn1 is a library for "Abstract Syntax Notation One (ASN.1) and Distinguished Encoding Rules (DER) manipulation"[26]. There is one existing test driver in the test suite, so the names of test cases are collected from the output of the test driver, and the order of the test cases is the same as their execution order. There are 29 test cases in this test suite, and there are 6 statuses for executed test cases, which are PASS, SKIP, XFAIL, FAIL, XPASS and ERROR.

Mailutils v3.7

---

[24] [Online]. Available: https://www.gnu.org/software/gettext/

[25] [Online]. Available: https://www.gnu.org/software/libidn/

[26] [Online]. Available: https://www.gnu.org/software/libtasn1/

Mailutils is a set of utilities and daemons for processing e-mail, which are able to work on mailboxes of any existing format, ranging from standard UNIX maildrops to remote mailboxes[27]. 13 sections were found in the output of the test drive in the test suite, and each section contains a different number of test cases. The number of test cases in each section is: 670, 17, 19, 10, 101, 6, 11, 97, 4, 7, 6, 269, and 21, and the total number of test cases is 1238. There is a summary part in each section, which shows the test result of executed test cases and the total number of the test cases in that section. Some sections have sub-sections for different test targets or functions under test, for example, in the first section there are 670 test cases, and these test cases are in different sub-sections, such as "Conversions", "Word wrapper", "Command line parser", and so on. For each line of output that contains a test case, there is an "OK" at the end of the line to indicate the result of that test case.

Guile v2.2.6

Guile is a programming language that help programmers create flexible applications for desktop, Web, or command-line, which can be extended by others with "plug-ins, modules, or scripts"[28]. 4 sections were noted in the output of the test driver; for the first three sections there are 39, 20 and 19 test cases, respectively, and all the names of the test cases are listed in the output of the test driver. But for the last section, instead of listing the test case name, the test script names are listed ending with .test. These test scripts were investigated and there are no test case names. This may be because of the large number of test cases in this section, which according to the summary part of this section from the output of the test driver, is 42454. This makes it hard to give names to each single test case. As a result, the file names of the test scripts were taken as their test case name in this section, which all end with ".test", giving 173 test case in section 4. These test scripts can also be found from the test suite folder in the downloaded package, but it is slightly different from the execution output of the test driver, as one of the test scripts is not

[27] [Online]. Available: https://mailutils.org/

[28] [Online]. Available: https://www.gnu.org/software/guile/

included in the test driver. The list of test scripts from the test driver was used, as the remaining single test script could be disabled or excluded from the test suite of the version used.

Nettle v3.5

Nettle is a cryptographic library that can fit easily in crypto toolkits, applications, or even in kernel space[29]. There are 2 sections in the output of the test driver from the test suite. The first section contains 99 test cases, and the second one contains 3 test cases, which makes the total number of test cases 102. For each test case in the output of the test driver, each line lists its execution status and the name of the test case, such as "PASS: aes". Every section has a summary that lists the total number of test cases.

Pies v1.4

Pies stands for Program Invocation and Execution Supervisor. It can run and manage the execution of external stand-alone programs, which are executed in the foreground[30]. There is an existing test driver in the test suite, so this was executed to collect the names of the test cases from the output. There are 2 sections in the output of the test driver, and each section holds the test cases in a separate folder. The first section contains 121 test cases, and the second one has 44 test cases. The total number of test cases in this test suite is 165. Both sections have sub-sections, which put the test cases into different sub-sections by testing target or function under test, such as "Wordsplit", "Formats", "Options" and so on.

Cflow v1.6

Cflow is a utility that "analyzes a collection of C source files and prints a graph, charting control flow within the program, and it is able to produce both direct and inverted flowgraphs for C sources"[31]. The existing test driver from the test suite was used to collect the names of the test cases. There are 42 test cases extracted from the output of the test driver. Each test case has been

[29] [Online]. Available: https://www.lysator.liu.se/~nisse/nettle/

[30] [Online]. Available: https://www.gnu.org.ua/software/pies/

[31] [Online]. Available: https://www.gnu.org/software/cflow/

given a number at the beginning of the line in the output, indicating the number of that test case, followed by the name of the test case, and then end by the execution status in the same line. E.g. "1: cflow version ok". These test cases have also been put into different sections by their test target or the function under test, such as "struct", "typedefs" and so on.

Dico v2.9

Dico is a flexible modular implementation of DICT server that has no dependence on a specific database format[32]. One test driver exists in the test suite, so this was used to collect test cases. From the output of the execution of the test driver, 11 sections that contain test case data were found in one single execution. Every section contains a different number of test cases, from 1 test case to 111 test cases, the total number of test cases is 235. In some sections, the test cases have also been classified into different sub-sections, such as "HELP output", "SHOW commands" and so on.

CSSC v1.4.1

CSSC is a free replacement for SCCS, which is source code control utility on various versions of Unix[33]. The existing test driver from the test suite is used for collecting test case data. There are three parts in the output of the test driver. The first and second parts have 59 and 11 test cases, respectively. The names of these test cases are listed in the output and each section has a summary with the details of the number of test cases and status. The third section contains 83 test scripts, and each test script contains multiple test cases, there are 1244 test cases in this section. The name of each test case is composed of the folder name, the test script name, and the test case number, e.g. "rmdel/basic.sh:a1" and "rmdel/basic.sh:a2". At the same time, there are some test cases from several test scripts that do not follow this naming method, and they only have a simple name, such as "t1" and "t2". This could because these test cases were not created by the same person under the same standard.

---

[32] [Online]. Available: https://puszcza.gnu.org.ua/software/dico/

[33] [Online]. Available: https://www.gnu.org/software/cssc/

Coreutils v8.31

Coreutils include “the basic file, shell and text manipulation utilities of the GNU operating system”, which are “expected to exist on every operating system”[34]. The output of the existing test driver in the test suite is used to collect test case data. There are two sections in the output of a single execution of the test driver. The first section has 616 test cases, and the second contains 332 test cases. The total number of the test cases is 948. In each section, after all test cases are executed, the test case execution status is summarized; status includes PASS, SKIP, XFAIL, FAIL, XPASS and ERROR. The same order was used in this study as executed in the test driver for both sections.

Gzip v1.10

Gzip is a widely used data compression tool with an enhanced compression ratio. It is a replacement for *compress*, as the latter has patents claimed by Unisys and IBM[35]. According to the output of the existing test driver in the test suite, there are 22 test cases in this test suite. The order of the test cases is the same as they executed by the test driver. The summary section of the output shows the total number of test cases, and the number of test cases in each status, including PASS, SKIP, XFAIL, FAIL, XPASS and ERROR.

Bash v5.0

According to its website, “Bash is the GNU Project's shell—the Bourne Again SHell. This is a sh-compatible shell that incorporates useful features from the Korn shell (ksh) and the C shell (csh). It offers functional improvements over sh for both programming and interactive use. In addition, most sh scripts can be run by Bash without modification.”[36] 81 test cases were collected from the output of the existing test driver. At the same time, in some sections of the output, it was found that multiple tests exist in a single test case. But after looking into these test cases in the

[34] [Online]. Available: https://www.gnu.org/software/coreutils/

[35] [Online]. Available: https://www.gnu.org/software/gzip/

[36] [Online]. Available: https://www.gnu.org/software/bash/

test suite, it was noted that these tests are not named, so the named test cases are used in the study. The order of these test cases will be their execution order from the test driver.

MPFR v4.0.2

The MPFR is a "C library for multiple-precision floating-point computations with correct rounding", and it is "efficient and has a well-defined semantics"[37]. There is an existing test driver in the test suite of the downloaded package. After execution of the test driver, it lists all executed test cases and a summary part with the total number of test cases and the number in each execution status. 180 test cases were collected from the output, and the order of the test cases is the same as the executed order in the test driver.

Commons-BCEL v6.4.1

Commons-BCEL stands for Byte Code Engineering Library, which provides an easy way to "analyze, create, and manipulate (binary) Java class files"[38]. The test driver from the downloaded package was executed to collecting the test case data. There are 33 test classes are listed in the output of the test driver, and the class names are following Java naming convention, e.g. org.apache.bcel.AnnotationDefaultAttributeTestCase. These test classes were used as the test cases in the following study.

For the following commons programs, the test driver exists in the test suite, so the output of the test drivers was used to collect test case data. The names of test classes listed in the output of the test driver are used as the names of the test cases. The order of the test cases is the same as their execution order in the test driver:

Commons-net v3.6

Commons-net is a library of basic Internet protocol implementations on the client side. As presented on its website, it includes the following protocols[39]:

---

[37] [Online]. Available: https://www.mpfr.org/

[38] [Online]. Available: https://commons.apache.org/proper/commons-bcel/

[39] [Online]. Available: https://commons.apache.org/proper/commons-net/

FTP/FTPS	*Finger*
FTP over HTTP (experimental)	*Whois*
NNTP	*rexec/rcmd/rlogin*
SMTP(S)	*Time (rdate) and Daytime*
POP3(S)	*Echo*
IMAP(S)	*Discard*
Telnet	*NTP/SNTP*
TFTP	

Commons-beanutils v1.9.4

Commons-beanUtils provides an easy-to-use and flexible wrapper around the reflection and introspection APIs of Java[40]. There is one test driver in the test suite, and the output of the test driver was used to collect test cases. There are 97 test classes from the output used as test cases in the study.

Commons-chain v1.2

Commons-chain is a library that uses the Chain of Responsibility pattern – a widely used technique for "organizing the execution of complex processing flows"[41].

Commons-codec v1.13

Commons-codec is a set of encoders and decoders, including Base64, Hex, Phonetic and URLs[42].

Commons-collections4 v4.4

Commons-collections is a Framework added to JKD 1.2, which provides powerful data structures and helps to accelerate the development of Java applications[43].

---

[40] [Online]. Available: http://commons.apache.org/proper/commons-beanutils/

[41] [Online]. Available: https://commons.apache.org/proper/commons-chain/

[42] [Online]. Available: https://commons.apache.org/proper/commons-codec/

[43] [Online]. Available: https://commons.apache.org/proper/commons-collections/

Commons-compress v1.19

Commons-compress is a library that defines an API for working with many compression files formats, including "ar, cpio, Unix dump, tar, zip, gzip, XZ, Pack200, bzip2, 7z, arj, lzma, snappy, DEFLATE, lz4, Brotli, Zstandard, DEFLATE64 and Z files"[44].

Commons-configuration2 v2.6

Commons-configuration is a library that allows Java applications to read configuration data from various sources with a generic interface[45].

Commons-csv v1.7

Commons CSV can read and write Comma Separated Value (CSV) files in various formats[46].

Commons-fileupload v1.4

Commons-fileupload provides a convenient way to add "robust, high-performance, file upload capability to your servlets and web applications"[47].

Commons-dbcp2 v2.7.0

DBCP stands for Database Connection Pools, which provides an easy way for Java applications to interaction with a relational database[48].

Commons-text v1.8

Commons-text is a Java library that provides a set of tools for processing strings, "from computing distance between strings to being able to efficiently do string escaping of various types"[49].

Commons-pool2 v2.7.0

---

[44] [Online]. Available: https://commons.apache.org/proper/commons-compress/

[45] [Online]. Available: https://commons.apache.org/proper/commons-configuration/

[46] [Online]. Available: https://commons.apache.org/proper/commons-csv/

[47] [Online]. Available: https://commons.apache.org/proper/commons-fileupload/

[48] [Online]. Available: https://commons.apache.org/proper/commons-dbcp/

[49] [Online]. Available: https://commons.apache.org/proper/commons-text/

Commons-pool is an open source Java library, which provides “an object-pooling API and a number of object pool implementations”[50].

Commons-vfs v2.4.1

VFS stands for Virtual File System, which provides a uniform view of files, with a single API, from various systems, including “the files on local disk, on an HTTP server, or inside a Zip archive”[51].

Commons-imaging v1.0

Commons-imaging was known as Commons-Sanselan. It is a Java library that provides an easy way to read and write various image formats, including parsing of image information and metadata[52].

OpenHAB-core v2.5.0

OpenHAB stands for open Home Automation Bus, which is a project that intends to provide “a universal integration platform for all things around home automation" and “brings together different bus systems, hardware devices and interface protocols by dedicated bindings”. The openHAB-core contains core bundles of the openHAB runtime, and it is a framework to build solutions on top instead of a product itself[53]. There is an existing test driver in the downloaded package, and test cases data were collected from the output of this test driver. There are 161 test classes in the output of the test driver, and each test class contains several tests, which is also called test method. But as the details of the test method are not displayed in the output of the test driver, e.g. [INFO] Tests run: 4, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.089 s - in org.eclipse.smarthome.core.auth.client.oauth2.AccessTokenResponseTest, each test class is considered as a single test case in this study.

---

[50] [Online]. Available: https://commons.apache.org/proper/commons-pool/

[51] [Online]. Available: https://commons.apache.org/proper/commons-vfs/

[52] [Online]. Available: https://commons.apache.org/proper/commons-imaging/

[53] [Online]. Available: https://www.openhab.org/

Tomcat 10 build 5092

According to its website, "Tomcat software is an open source implementation of the Java Servlet, JavaServer Pages, Java Expression Language and Java WebSocket technologies", that "powers numerous large-scale, mission-critical web applications across a diverse range of industries and organizations"[54]. The test data were collected from the execution log of the test driver. 1194 test classes were extracted from the log and these test classes were used as the test cases in this study. The names of these test cases follow the Java class naming convention, e.g.,

TEST-org.apache.catalina.ant.TestDeployTask.APR and

TEST-org.apache.catalina.core.TestStandardContext.APR.

The order of the test cases is the same as they displayed in the log, which means it is the same as the execution order of the test driver.

OFBiz v18.12

OFBiz is a free tool that used for the automation of processes in an enterprise environment, which includes "framework components and business applications for ERP (Enterprise Resource Planning), CRM (Customer Relationship Management), E-Business / E-Commerce, SCM (Supply Chain Management), MRP (Manufacturing Resource Planning), MMS/EAM (Maintenance Management System/Enterprise Asset Management)" [55]. 35 packages were obtained as a result of the unit test, and each package contains from several to hundreds of test cases. The total number of test cases extracted from the output of the test driver is 592. The order of the test cases is the same as the execution order from the test driver.

Spark v2.4.5

Spark is an analytics engine targeting large-scale data processing. It provides "high-level APIs in Java, Scala, Python and R, and an optimized engine that supports general execution graphs", and it "supports a rich set of higher-level tools including Spark SQL for SQL and structured data

[54] [Online]. Available: http://tomcat.apache.org/

[55] [Online]. Available: https://ofbiz.apache.org/

processing, MLlib for machine learning, GraphX for graph processing, and Structured Streaming for incremental computation and stream processing"[56]. There is an existing test driver from the downloaded package, and the output of the test driver was used to collect test data. 159 test classes were extracted from the output and used as test cases. These test classes come from 11 sections of the output; there are another 4 sections but without any test cases executed. The order of the test cases is the same as the order they are executed by the test driver.

Ant v1.10.7

Ant is an open source library and a command-line tool, which aims to "drive processes described in build files as targets and extension points dependent upon each other"[57]. There are two sections in the output of the existing test driver, junit testing and au:antunit testing. In the junit testing section, the test suite names are listed in the output, where the test suite name is the test class name. These have been collected as the test case names in this study; the number of test cases of this section is 257. In the second part, the au:antunit testing section, is the unit testing from the test suite. The output of the test driver lists the test Target, which is considered as the test case names in this study. There are 928 test cases in this section, and the total number of test cases is 1185. The order of the test cases will be the same as the execution order in the test driver.

5.4 Experimental Results

In studying the test cases in the above 50 programs, there are two types that clearly show the test cases have a similar testing function or testing target:

Type 1: The letters in the name of the test cases are the same, and are then followed by different numbers.

Type 2: The name of the test case can be divided into several meaningful sections and most

[56] [Online]. Available: https://spark.apache.org/

[57] [Online]. Available: https://ant.apache.org/index.html

of the sections are the same; or when test cases have been divided into different sections by the test driver, the test cases in the same section, which indicate that they are testing similar function/component/target, are considered similar to each other.

For example, arrayind1, arrayind2 and arrayind3 are Type 1; Type 2 has case-fold-char-class, case-fold-char-range and case-fold-char-type, or the following test cases:

Lexical structure

1: identifier

2: unquoted string

3: number

For both Type 1 and Type 2, only neighboring test cases will be considered as similar test cases. For example, the four test cases listed below are in their execution order:

test-augment-element

test-illegal-attribute

test-illegal-element

test-augment-attribute

We consider test-illegal-attribute and test-illegal-element as Type 2 test cases, but test-augment-element and test-augment-attribute are neither Type 2 nor Type 1 test cases, as they are not neighboring test cases.

Table VI summarizes the results of all 50 programs under test. Column #2 lists the name of programs. Column #3 and #5 show the number of test cases that are classified as Type 1 and Type 2, respectively. Column #4 is the percentage of the Type 1 out of the total number of the test cases in each program, and column #6 is the percentage for Type 2. Column #7 lists the total number of test cases of Type 1 and Type 2, and the percentages of this figure out of all test cases in each program are listed in column #8.

In the Table VI, we can see that similar test cases have been found in all the 50 programs, and in some programs, all test cases are similar to their neighboring test cases and make the result reaches 100%. For example, gnurl-7.66.0 has 488 test cases, and the naming of the test cases is

TABLE VI
NAMING CONVENTION OF TEST CASES RESULTS

Row #	project name and version	Type 1	Type 1 ÷ total	Type 2	Type 2 ÷ total	Type 1 + Type 2	(Type 1 + Type 2) ÷ total
1	gawk v5.0.1	223	0.45	92	0.19	315	0.64
2	gnuastro v0.10	4	0.07	46	0.78	50	0.85
3	gnurl v7.66.0	488	1.00	0	0.00	488	1.00
4	grep v3.3	27	0.10	119	0.42	146	0.52
5	grub v2.04	0	0.00	30	0.37	30	0.37
6	libredwg v0.8	0	0.00	15	0.33	15	0.33
7	rush v2.1	0	0.00	68	1.00	68	1.00
8	tar v1.32	0	0.00	232	0.99	232	0.99
9	texinfo v6.6	57	0.35	103	0.64	160	0.99
10	wget2 v1.99.2	0	0.00	24	1.00	24	1.00
11	bison v3.4.2	5	0.01	578	0.99	583	1.00
12	datamash v1.5	0	0.00	20	1.00	20	1.00
13	libmicrohttpd v0.9.67	0	0.00	16	1.00	16	1.00
14	gdb v8.3.1	28	1.00	0	0.00	28	1.00
15	direvent v5.2	4	0.03	101	0.73	105	0.76
16	gsl v2.6	2	0.03	8	0.14	10	0.17
17	gettext v0.20	461	0.68	88	0.13	549	0.81
18	libidn2 v2.2.0	0	0.00	9	1.00	9	1.00
19	libtasn1 v4.14	0	0.00	12	0.41	12	0.41
20	mailutils v3.7	59	0.05	1139	0.92	1198	0.97
21	guile v2.2.6	53	0.21	74	0.29	127	0.51
22	nettle v3.5	20	0.20	33	0.32	53	0.52
23	pies v1.4	4	0.02	89	0.54	93	0.56
24	cflow v1.6	0	0.00	23	0.55	23	0.55
25	dico v2.9	4	0.02	120	0.50	124	0.52
26	cssc v1.4.1	1222	0.93	47	0.04	1269	0.97
27	coreutils v8.31	164	0.17	669	0.71	833	0.88
28	Gzip v1.10	0	0.00	5	0.23	5	0.23
29	bash v5.0	17	0.21	15	0.19	32	0.40
30	MPFR v4.0.2	18	0.10	79	0.44	97	0.54
31	Commons-BCEL v6.4.1	0	0.00	21	0.64	21	0.64
32	commons-net v3.6	0	0.00	36	0.84	36	0.84
33	commons-beanutils v1.9.4	28	0.29	46	0.47	74	0.76
34	commons-chain v1.2	0	0.00	9	0.45	9	0.45
35	commons-codec v1.13	0	0.00	52	0.91	52	0.91
36	commons-collections4 v4.4	0	0.00	149	0.87	149	0.87

TABLE VI CONTINUED
NAMING CONVENTION OF TEST CASES RESULTS

37	commons-compress v1.19	0	0.00	133	0.96	133	0.96
38	commons-configuration2 v2.6	0	0.00	75	0.93	75	0.93
39	commons-csv v1.7	5	0.33	5	0.33	10	0.67
40	commons-fileupload v1.4	0	0.00	3	0.20	3	0.20
41	commons-dbcp2 v2.7.0	0	0.00	44	0.96	44	0.96
42	commons-text v1.8	0	0.00	62	0.78	62	0.78
43	commons-pool2 v2.7.0	0	0.00	22	1.00	22	1.00
44	commons-vfs v2.4.1	0	0.00	79	0.89	79	0.89
45	commons-imaging v1.0	0	0.00	112	0.98	112	0.98
46	openhab-core v2.5.0	0	0.00	156	0.97	156	0.97
47	tomcat 10 build 5092	796	0.67	398	0.33	1194	1.00
48	ofbiz v18.12	7	0.01	583	0.98	590	1.00
49	spark v2.4.5	0	0.00	154	0.97	154	0.97
50	ant v1.10.7	62	0.05	464	0.39	526	0.44
		avg	0.14		0.61		
		max	1.00		1.00		
		min	0.00		0.00		

“test” and 4 digitals, such as test0001, test0002 and test0003. The minimum percentage is 17.24%, for the program named gsl-2.6. The reason is that there are only 58 test cases in its test suite, and most of the test cases are testing different aspect of the program. But even in this situation, we still find 8 similar test cases in 4 groups, which are vector.test and vector.test_static, matrix.test and matrix.test_static, multifit and multifit_nlinear, and multilarge and multilarge_nlinear.

There are 12392 test cases across the 50 programs, and we find 3758 Type 1 test cases, and 6457 test cases for Type 2, which makes the total number of similar test cases 10215, which is 82.43% of the total number of the test cases. It shows that the similarity of neighboring test cases is very common in real-world projects, which supports the Observation I that reported above.

Chapter 6

Empirical Evaluation of Dispersity-based Prioritization in Code Coverage Rate

6.1 Background

The code coverage is a widely used concept in software testing, it indicates the statements, branches, functions, etc., of the source code that have been executed during the testing. After the above empirical studies of the proposed algorithm, DBP, have been conducted in F-measure and execution time, the studies were extended to include code coverage rate with a new set of open source software packages. This chapter describes the Objectives of the Experiment, the Experimental Design, and the Experimental Results.

6.2 Objectives of the Experiment

In the previous study, we have completed the empirical evaluation of the new proposed DBP algorithm, in terms of the F-measure and execution time, compared with the RP and Additional algorithms. We can see that the DPB has the better or same applicability, effectiveness, and execution time for failure detection than RP, and better applicability, execution time for failure detection and efficiency than the Additional algorithm. In this study, the focus was on the code

coverage rate when applying the two algorithms, DBP and RP, on a new set of open source programs, and compare the two test case prioritization techniques in terms of the code coverage.

6.3 Experimental Design

The two TCP algorithms applied in this study are RP and DBP. The focus is on the code coverage rate results of RP and DBP, and the comparison provides a clear understanding of the proposed new algorithm in the terms of code coverage.

6.3.1 The Proposed DBP Algorithm

As mentioned above, our dispersity-based approach does not consider the input domain of the program under test and it is not an ART technique. However, the same efficient ART algorithm from the previous empirical study has been applied to generate an adaptive random sequence of integers in the range [1, n], where n is the number of test cases, to serve as a sequence of test case IDs. This helps to achieve an even spread of test case IDs. The fixed size candidate set (FSCS)-ART algorithm was applied, and another repeat that enhanced the "forgetting by consecutive retention" strategy [68, 73], as explained in the Chapter 4.

6.3.2 Subject Programs

In this empirical evaluation, 10 real-world software projects were investigated, which are summarized in Table VII. The SUTs have various sizes and functionality, and they are written in different programming languages. Their test suites also have various sizes ranging from tens to thousands. Therefore, this set of programs can be considered as representative of real-world projects.

All the packages listed in Table VII are downloaded from the project websites that listed in the column #3 of Table VII, including the SUTs and test suites. Most packages contain programs, scripts, or other type of files, written in different programming, scripting, or markup languages. The fourth column of Table VII lists the size of test suite, that is, the total number of test cases in the test suite.

There are challenges when selecting these projects. First, the test suite must exist in the project, with at least 10 test cases. In many open source projects studied, it was noted that some of them do not have any test cases in their source code. Next, code coverage information must be available in the test driver, or can be collected through test coverage tools, such as gcov. Finally, the test cases can be executed separately, so we can collect code coverage data for each test case. In some of these test suites, the test driver provides the coverage information, for the whole test suite. In this case, coverage data is not available for single test cases, so it was not possible to calculate the average coverage rate in the experiment.

TABLE VII
SUMMARY OF SOFTWARE PACKAGES USED IN EMPIRICAL EVALUATION IN CODE COVERAGE RATE

1	2	3	4
Row #	project name and version	project website	size of test suite
1	commons-bcel v6.4.1	https://commons.apache.org/proper/commons-bcel/	33
2	commons-codec v1.13	https://commons.apache.org/proper/commons-codec/	57
3	commons-collections4 v4.4	https://commons.apache.org/proper/commons-collections/	171
4	commons-compress v1.19	https://commons.apache.org/proper/commons-compress/	139
5	commons-csv v1.7	https://commons.apache.org/proper/commons-csv/	15
6	commons-dbcp2 v2.7.0	https://commons.apache.org/proper/commons-dbcp/	46
7	commons-text v1.8	https://commons.apache.org/proper/commons-text/	80
8	commons-imaging v1.0	https://commons.apache.org/proper/commons-imaging/	114
9	replace	http://sir.unl.edu	5542
10	space	http://sir.unl.edu	13551

6.3.3 Evaluation Metrics

To compare the accumulated code coverage rate of the algorithms, that is from t_1 to t_n (there are n test cases in the test suite), for each number of test cases we will have an average coverage rate for 1000 trials. For example, with 1 test case, the test is executed, and the coverage rate is recorded. It will repeat 1000 times and then the average of the coverage rate will be the result for 1 test case. Then with 2 test cases, the two test cases are executed, and the coverage rate of the two test cases is recorded and which will repeat 1000 times, and the average of the 1000 recorded coverage rate is the second result; …; with n test cases, the average coverage rate of 1000 trials would be the result for n test cases. For easy comparison of the coverage data of the 10 projects, if a branch or statement is not covered by any of the test cases from the test suite, it will be removed from the list, so that the coverage rate of the programs will reach 100% by executing all test cases from their test suite. As we compare the ratio of average coverage rates between different algorithms instead of the coverage rate itself, this makes the comparison clearer and more intuitive.

6.3.4 Subject Packages

This section is a brief introduction of the projects that are included in the Table VII. The project websites are listed in the Table VII, and more information can be found there. The SUTs platform is: host machine running Microsoft Windows 10 Enterprise and a virtual machine (VMware Workstation) to run Ubuntu.

The first project, Commons BCEL (row #1), is a Byte Code Engineering Library that is "intended to give users a convenient way to analyze, create, and manipulate (binary) Java class files (those ending with .class)." The test suite has 33 test files and each of them contains 1 or several tests. Each test file is considered a test case, and the code coverage data is collected for each test case. Commons Codec (row #2) provides "implementations of common encoders and decoders such as Base64, Hex, Phonetic and URLs", comes with 57 test files.

After that, they are Commons Collections4 (row #3) and Commons Compress (row #4), which

have 66,936 Source Lines of Code with 171 test files in the test suite, and 50,920 SLOC with 139 test files in the test suite, respectively. “The Java Collections Framework was a major addition in JDK 1.2. It added many powerful data structures that accelerate the development of most significant Java applications and has become the recognized standard for collection handling in Java.” “The Apache Commons Compress library defines an API for working with ar, cpio, Unix dump, tar, zip, gzip, XZ, Pack200, bzip2, 7z, arj, lzma, snappy, DEFLATE, lz4, Brotli, Zstandard, DEFLATE64 and Z files.”

Commons CSV (row #5) reads and writes files in variations of the Comma Separated Value (CSV) format. It comes with the smallest size of test suite in this set of programs, which is only 15. The next is Commons DBCP2 (row #6), according to its website, “This Database Connection Pools package provides an opportunity to coordinate the efforts required to create and maintain an efficient, feature-rich package under the ASF license.”

In row #7, Commons Imaging has 41,350 SLOC with 114 test cases in the test suite, which is a library that “reads and writes a variety of image formats, including fast parsing of image info (size, color space, ICC profile, etc.) and metadata.” Commons Text in row #8 is a “library focused on algorithms working on strings.”, which has 80 test cases and 24,621 SLOC.

Replace (row #9) and Space (row #10) are downloaded from SIR [85, 86]. Replace performs pattern matching and substitution, and it is one of the "Siemens" programs, according to SIR documentation, which were assembled by Tom Ostrand and colleagues at Siemens Corporate Research for a study of the fault detection capabilities of control-flow and data-flow coverage criteria [99] and were made available to SIR by Tom Ostrand, and then modified by other researchers for further studies [100]. There are 5542 test cases in the downloaded package. The Space program was developed by Ingegneria Dei Sistemi, Pisa, Italy, for the European Space Agency [87]. The program consists of about 6199 SLOC written in C, and works as an interpreter for an array definition language (ADL). The downloaded package includes a base version and 38 faulty versions. Each faulty version contains a real fault discovered during the development of the program. In addition, the downloaded package contains a pool of 13 551 test cases. According

to SIR [86], the test pool was constructed in two stages: An initial pool of 10 000 test cases was created using a test case generator, capable of generating “random” ADL input files [87]. Then, more test cases were added to the pool so that each executable statement of the base program or edge of its control flow graph would be exercised by at least 30 test cases. In this empirical study with Replace and Space, the focus was on comparing the coverage data of the two algorithms, RP and DBP. As a result, only the original version from the downloaded packages was included in this study, and all other faulty versions are excluded as the test suites are identical.

6.3.5 Deciding the Order of Test Cases

For row #1 to row #8, there are test drivers in these projects, so the sequence in the driver was used as the order to directly execute the test cases. For Replace project (row #9), there is one file named “runall.sh” in the downloaded package, which contains all test cases. The execution order of test cases follows the order in this file. The last one is the Space project (row #10), and each test case of the Space program is an ADL file. The order of the test cases is defined by the alphanumeric order of the filenames.

6.4 Experimental Results

We first compare RP with the proposed TCP approach (DBP) without the “forgetting by consecutive retention” strategy [68, 73], and then compare the DBP with the forgetting strategy.

6.4.1 Comparison of DBP without Forgetting Strategy with RP

6.4.1.1 Results of DBP without Forgetting Strategy and RP

Commons BCEL 6.4.1

For the Commons BCEL, there are 33 test cases in the test suite. Comparing the results of RP and DBP in Table VIII, 84.8% of the values of DBP/RP are greater than 1, which means the DBP has a higher coverage rate. For 12.1% of the results DBP/RP is less than 1, which happened with test cases 22, 24, 27 and 30, and all the differences are less than 0.001 in terms of DBP/RP (0.999980, 0.999852, 0.999933, 0.999876). The remaining 3.0%, that is the one with 33 test cases,

has equal average coverage rates for both RP and DBP, where the coverage rate reaches 100% when all test cases are executed.

Commons Codec 1.13

Among 57 test cases, Table VIII shows that 91.2% of the results get higher coverage rate for the DBP, and RP get better results in only 7% of them, when the other 1.8% get equal outcomes. In Table IX, the minimum DBP/RP is 0.96 and the maximum is 1.09. According to Fig. 6 Commons Codec 1.13 DBP-without-Forgetting/RP, there are only 4 results less than 1.

Commons Collections4 4.4

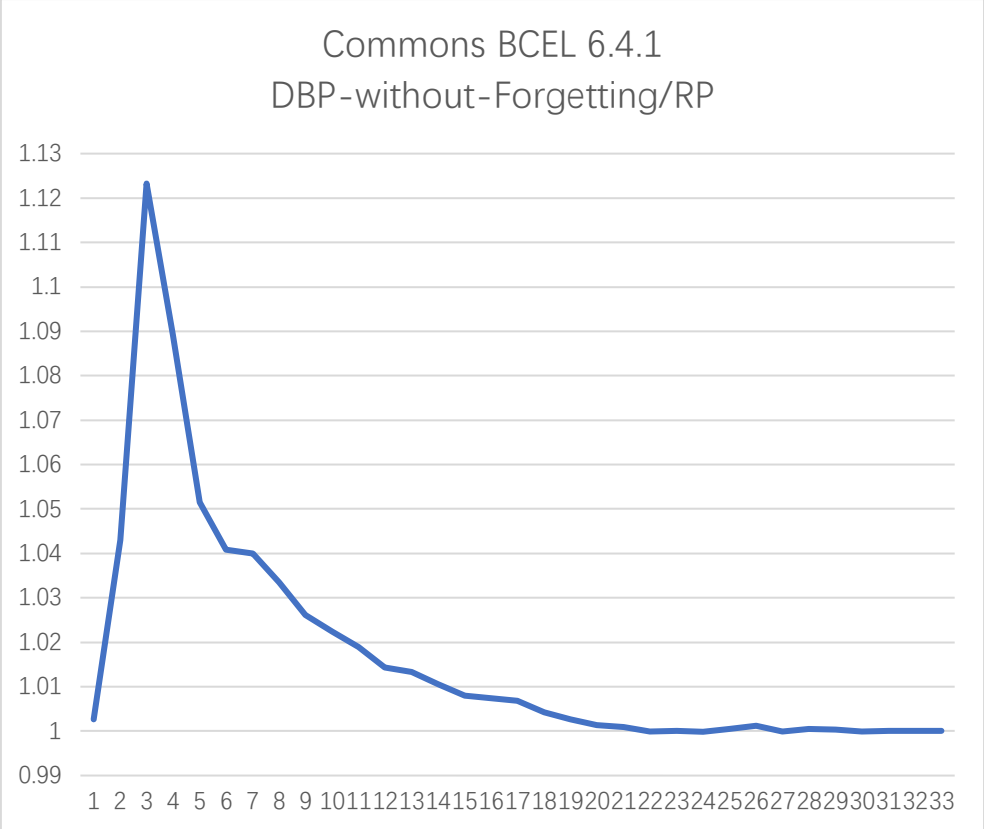
This program has 171 test cases in the test suite. In 165 cases the coverage data of this program gives DBP/RP > 1, which is 95.9% of all the test cases, and the maximum reaches 1.1 (Table IX). There are only 6 results that come with DBP/RP < 1, which are all above 0.999 with the minimum of 0.9994. From Fig. 6 Commons Collections4 4.4 DBP-without-Forgetting/RP, we can see that the DBP/RP is above 1 in most cases, and only get drops slightly when the coverage rate reaching 1.

Commons Compress 1.19

There are 139 test cases in the test suite of the Compress. At the beginning of the experiment, according to Fig.6 Commons Compress 1.19 DBP-without-Forgetting/RP, from one test case, two test cases, until six test cases, the DBP/RP is less than 1, but from the 7^{th} test case, the DBP/RP has increased above 1 and reached 1.02. After that, the DBP/BP reached the maximum of 1.03, and remained >1 until 91 test cases. After that, the result stabilized around 1, with all DBP/BP between 0.999 and 1.

Commons CSV 1.7

This is the program that has the smallest size of test suite, which has only 15 test cases. But even in this case, there are still 60% of the results that come with DBP/RP > 1, and only 26.7% of the result has DBP/RP > 1. From the Fig. 6 Commons CSV 1.7 DBP-without-Forgetting/RP, we can see that it reaches the maximum of DBP/RP with one test case, which is 1.06, and after that it drops to 0.84, which is the minimum result. Then the results start to increase and reach



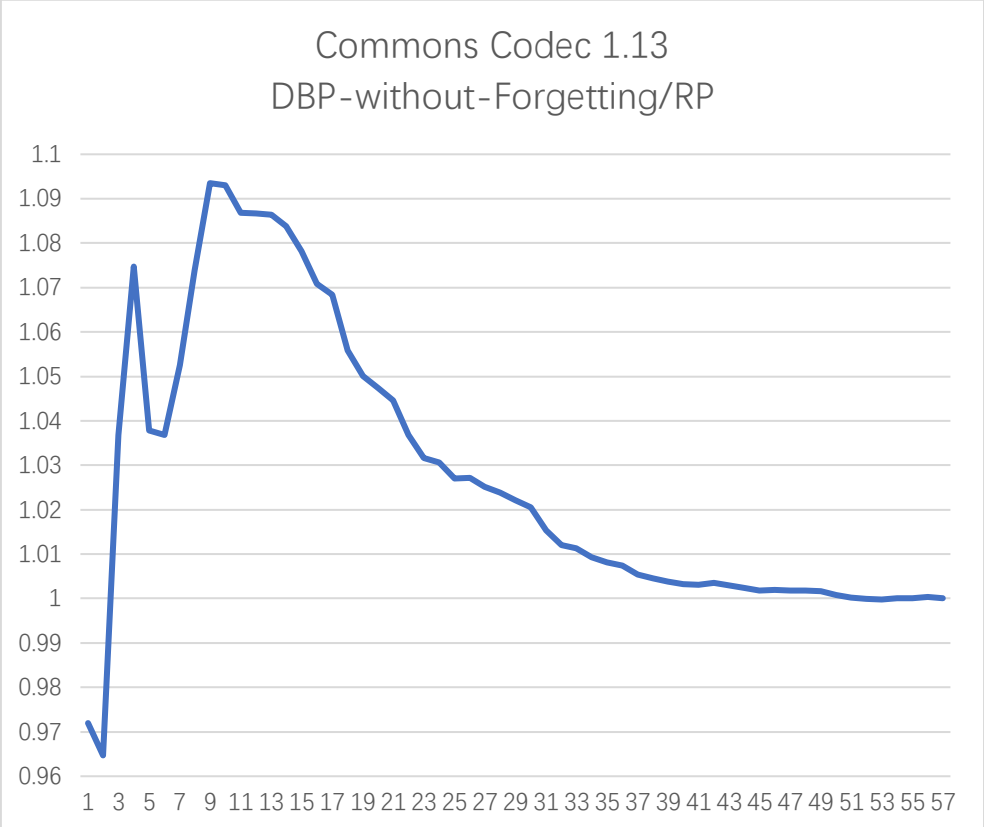
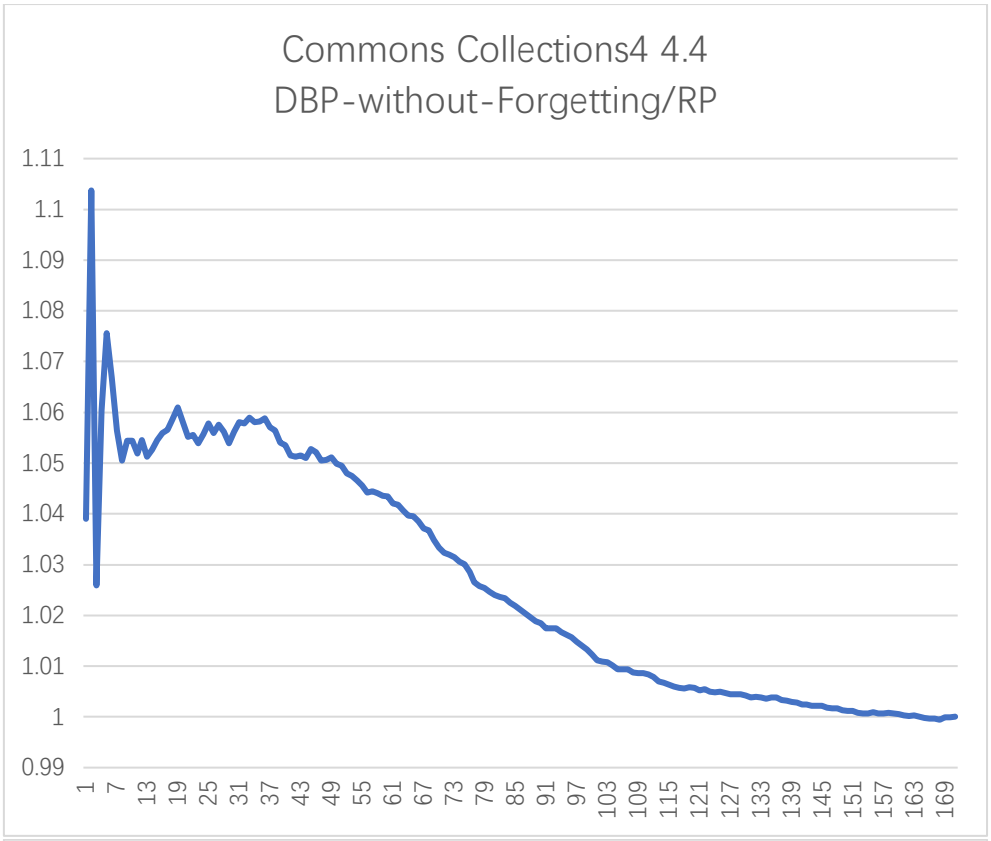


Fig. 6. DBP/RP Results Continued



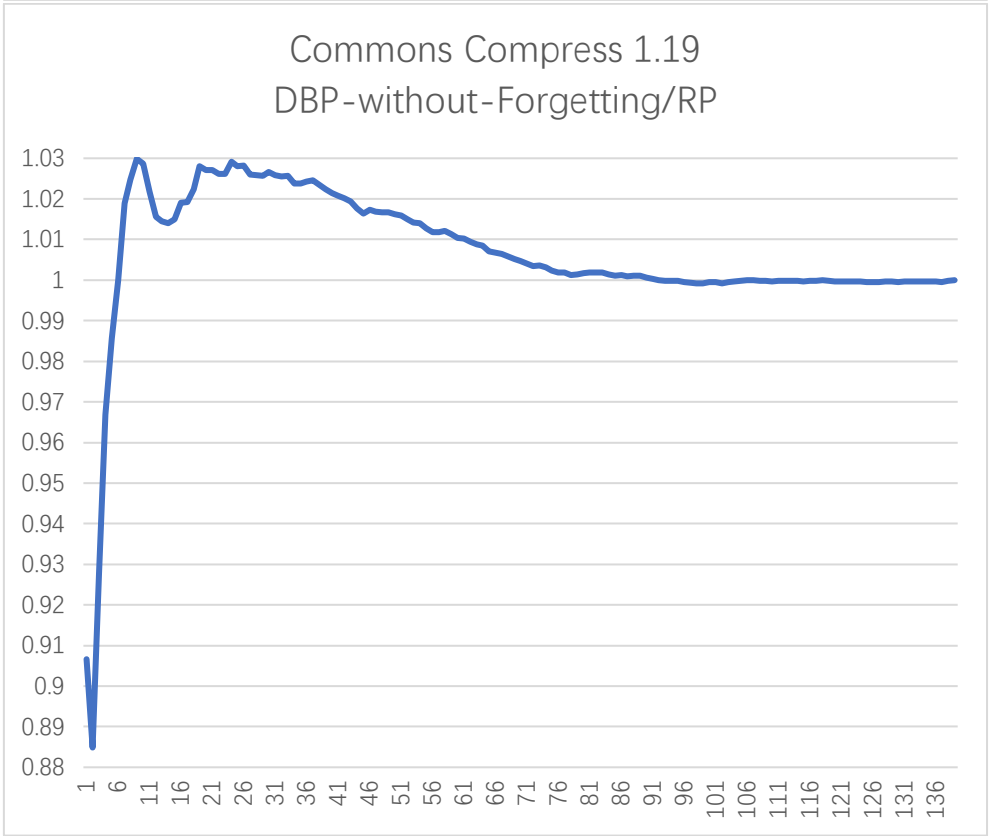
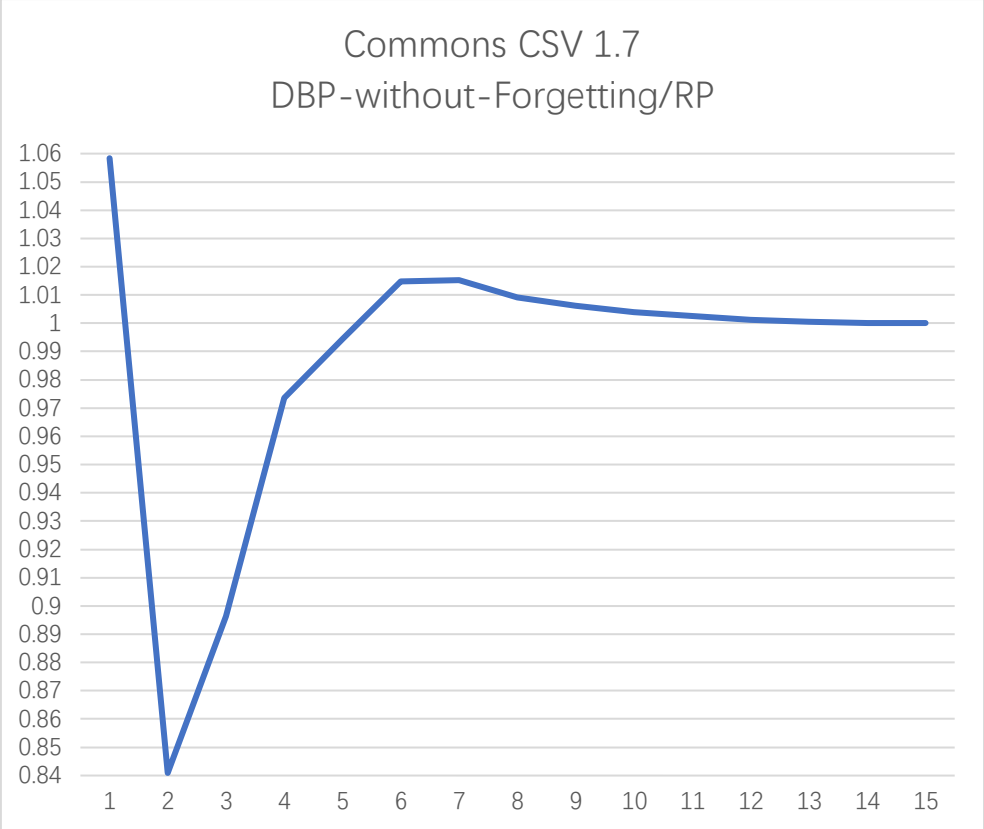


Fig. 6. DBP/RP Results Continued



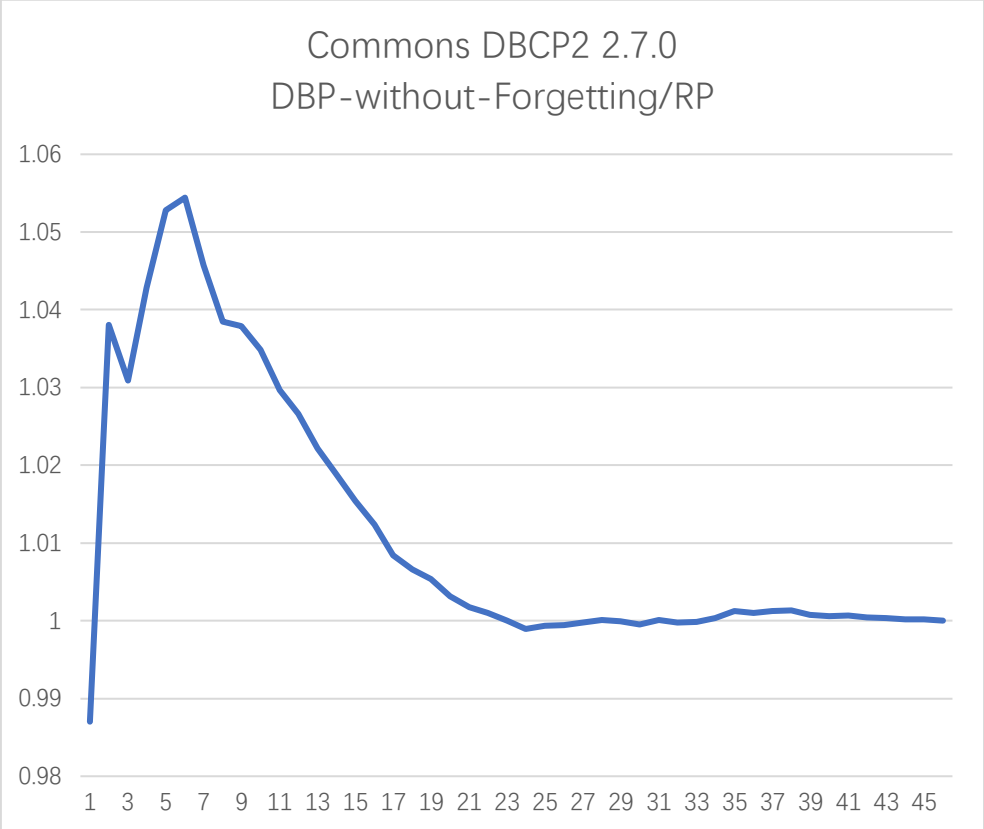
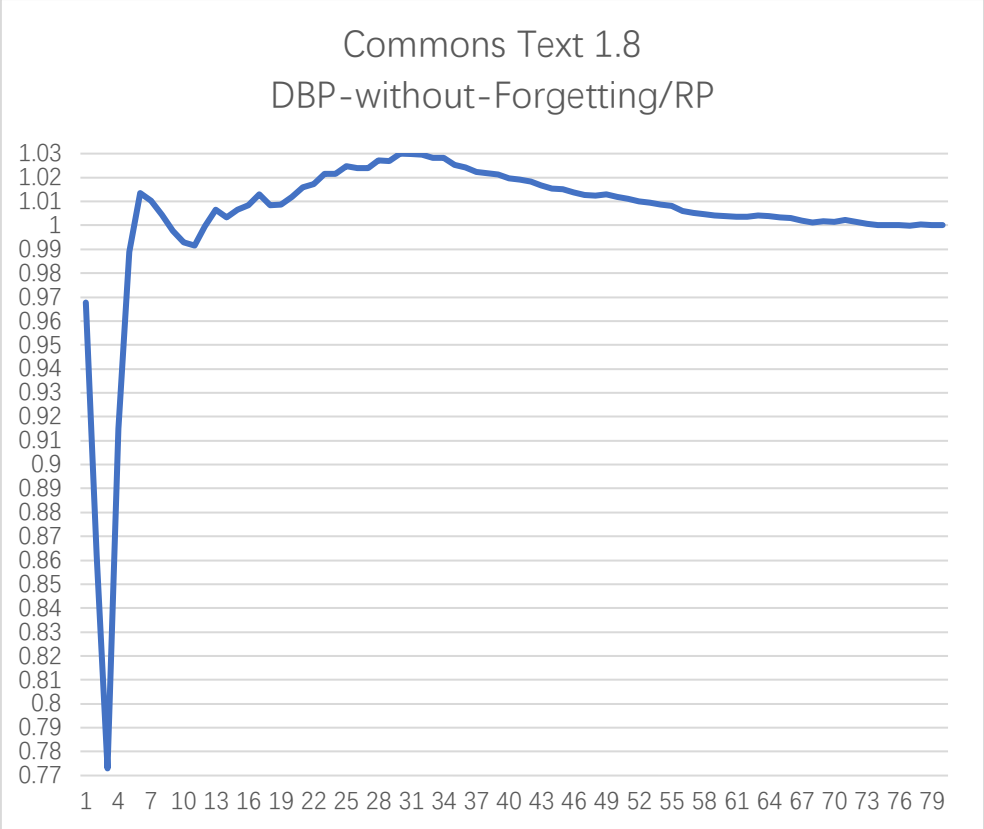


Fig. 6. DBP/RP Results Continued



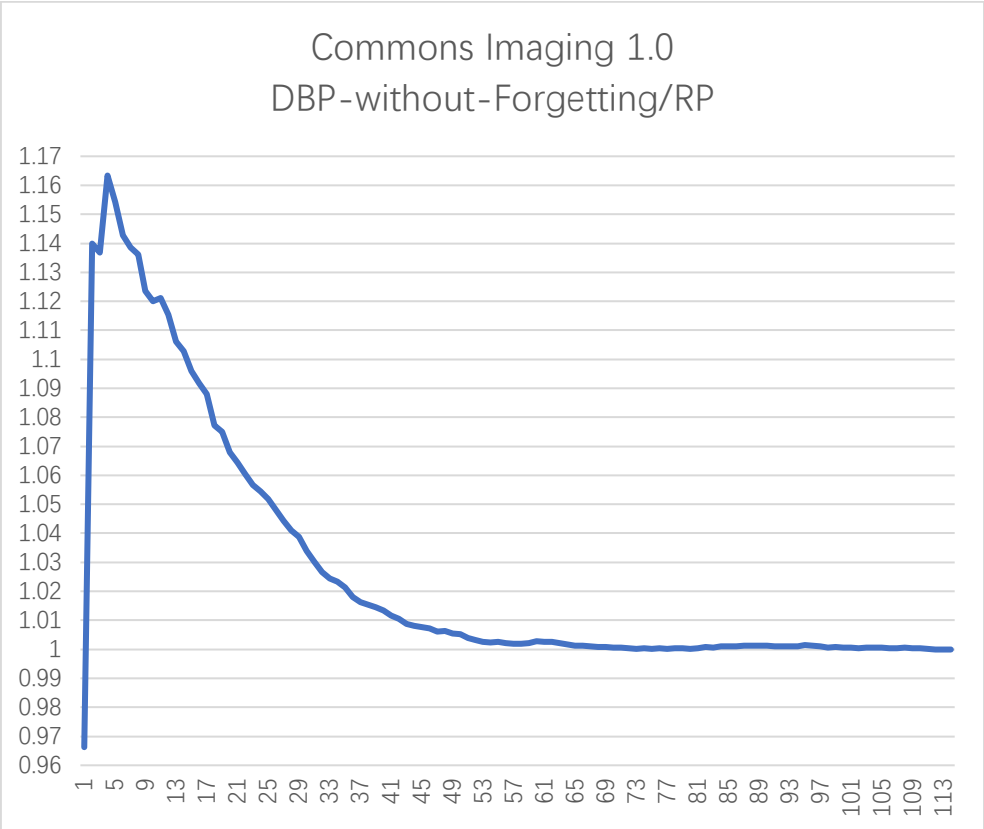


Fig. 6. DBP/RP Results Continued

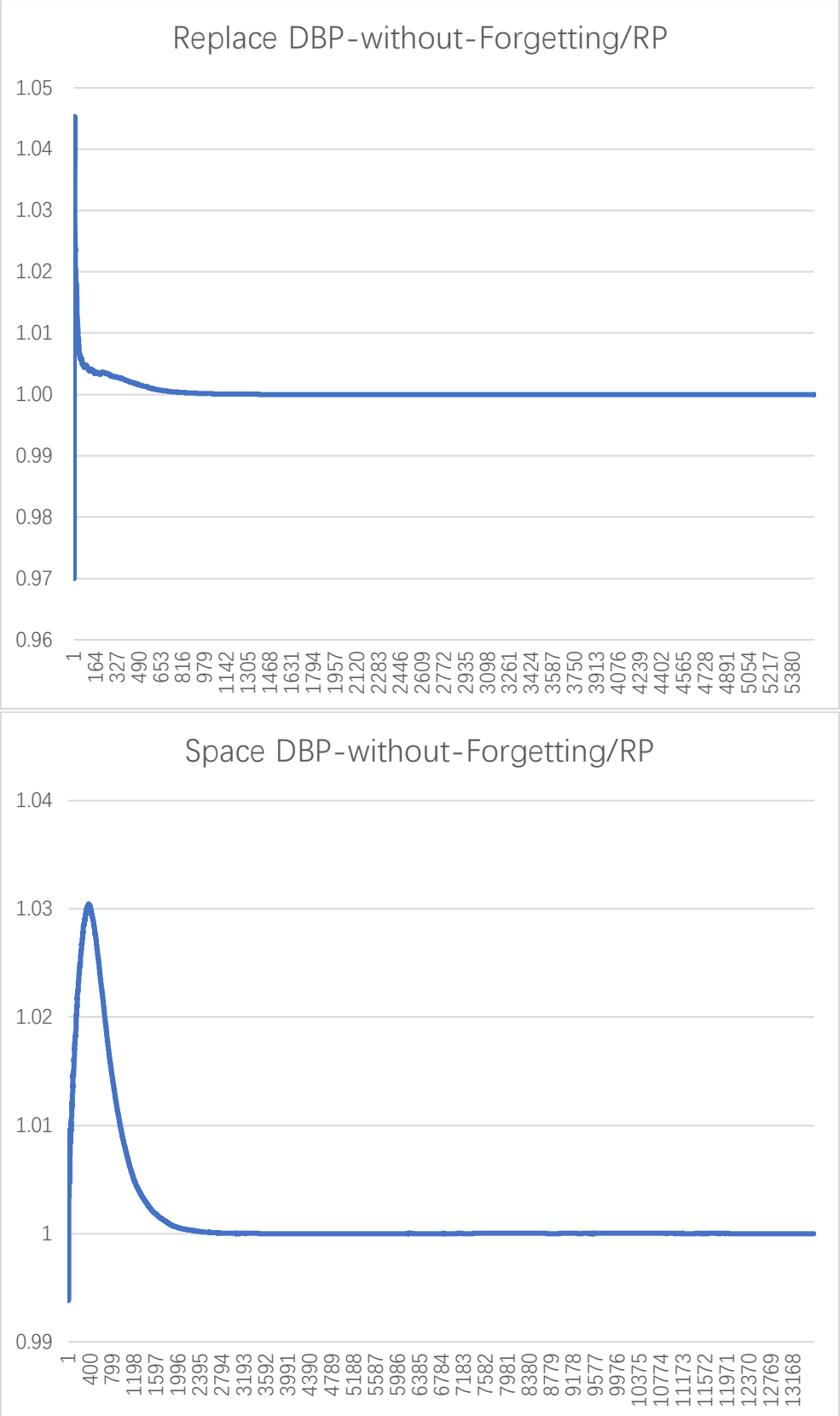


Fig. 6. DBP/RP Results Continued

around 1 from five test cases and keep around 1 after that.

Commons DBCP2 2.7.0

Among the 46 test cases in the test suite of this program, there is only one result of DBP/RP that is below 0.99, which is 0.987 with one test case according to Fig. 6 Commons DBCP2 2.7.0 DBP-without-Forgetting/RP. Besides that, all of the other results are around or above 1. From two test cases, three test cases, until twenty test cases, the DBP/RP is above 1, and reaches the maximum of 1.05 with 6 test cases. After that, the ratio is around 1, which means the result of DBP and RP are very close to each other, with the difference less than 0.001.

Commons Text 1.8

In this program we observe the minimum result of DBP/RP across all the ten programs, is 0.773 with 3 test cases. Apart from this, Fig. 6 Commons Text 1.8 DBP-without-Forgetting/RP shows that the DBP/RP is 0.968, 0.862 and 0.915 with 1, 2, and 4 test cases, respectively. After that, the result increases and reaches the maximum of 1.03 with 30 test cases, and then decreases and fluctuates around 1. There are 85.0% of the DBP/RP is larger than 1, and only 13.8% is less than 1, 1.3% is equal to 1 in Table VIII.

Commons Imaging 1.0

Commons Imaging 1.0 has 114 test cases in the test suite, and only one of them has a result of DBP/RP that less than 1, which is 0.966 with 1 test case in Fig. 6 Commons Imaging 1.0 DBP-without-Forgetting/RP. All the other results are larger or equal to 1. From 2 test cases, the DBP/RP was 1.14 and then increased to the maximum of 1.16 with 4 test cases. The result stays above 1.01 until 42 test cases, and then fluctuates between 1.01 and 1. From Table VIII, 96.5% of the results are larger than 1.

Replace

There are 5542 test cases in the test suite of the Replace There are similar result to the previous program – Commons Imaging 1.0: the first result of DBP/RP with 1 test case is the only one that below 1, which is 0.97 according to Fig. 6 Replace DBP-without-Forgetting/RP. All the other results are equal to or above 1. It reaches the maximum with 3 test cases at 1.045. After that the

ratio starts to drop and reaches 1 with 709 test cases. With 1113 test cases, the DBP got 100% coverage rate, whereas the RP reaches 100% coverage rate with 1839 test cases.

Space

The Space has the largest test suite in this experiment, which is 13551. From Fig. 6 Space DBP-without-Forgetting/RP, the minimum of the DBP/RP is 0.994 with 2 test cases, and the maximum is 1.030 at 367 test cases. We can see that the ratio of DBP/RP increases from the beginning, and then start to decrease after reaching the maximum, and drop below 1.01 with 938 test cases. Table VIII shows that there are 75.6% of results of the DBP/RP is larger than 1, and 21.8% are less than 1, when the left 2.6% are equal to 1. After that, the DBP reach 100% coverage rate with 13409 test cases, and the RP reaches the same coverage rate with 109 less test cases, which is 13300 test cases.

6.4.1.2 Summary

Let us consider Table IX. DBP/RP > 1 indicates DBP is better, and DBP/RP < 1 indicates RP is better. When DBP/RP = 1, they have an equal performance. Column 3 ("Highest DBP/RP") indicates situations where DBP achieved the best relative performance. Columns 4 and 5 show that 9 out of 10 such results are significant. Column 6 shows situations where RP achieved the best relative performance. Columns 7 and 8 show that only 3 out of 10 such results are significant.

Table VIII lists the percentage of the results (DBP/RP) by three categories, DBP/RP > 1, DBP/RP = 1 and DBP/RP < 1. For all of the 10 programs, the percentages of DBP/RP > 1 are larger than the percentages of DBP/RP < 1. For the values of DBP/RP > 1, three of them are more than 90%, two results are between 80% and 90%, 4 values are between 60% and 80%, and there is only one below these ranges, which is 33.1% for replace. But we also note that for this program, the DBP/RP = 1 is 66.8% and the DBP/RP < 1 is 0%, so for this program the DBP is still outperformed the RP.

In addition, for the highest and the lowest value of the DBP/RP in each program, statistically meaningful comparisons have been conducted by performing independent-samples t-test at a significance level of 5%, the results for the two-tailed p-values and the effect size (Cohen's d)

TABLE VIII
RESULTS COMPARISON: DBP WITHOUT FORGETTING WITH RP

		DBP without Forgetting		
Row #	Programs	DBP/RP>1	DBP/RP=1	DBP/RP<1
1	commons-bcel v6.4.1	84.8%	3.0%	12.1%
2	commons-codec v1.13	91.2%	1.8%	7.0%
3	commons-collections4 v4.4	95.9%	0.6%	3.5%
4	commons-compress v1.19	61.2%	0.7%	38.1%
5	commons-csv v1.7	60.0%	13.3%	26.7%
6	commons-dbcp2 v2.7.0	78.3%	2.2%	19.6%
7	commons-text v1.8	85.0%	1.3%	13.8%
8	commons-imaging v1.0	96.5%	0.9%	2.6%
9	replace	33.1%	66.8%	0.0%
10	space	75.6%	2.6%	21.8%

TABLE IX
RESULTS COMPARISON: DBP WITHOUT FORGETTING: HIGHEST AND LOWEST DBP/RP

Row #	Programs	Highest DBP/RP	p-value	effect size	Lowest DBP/RP	p-value	effect size
1	commons-bcel v6.4.1	1.12	0.000	0.34	1.00	0.928	0.00
2	commons-codec v1.13	1.09	0.000	0.26	0.96	0.269	0.05
3	commons-collections4 v4.4	1.10	0.001	0.16	1.00	0.081	0.08
4	commons-compress v1.19	1.03	0.008	0.12	0.88	0.000	0.16
5	commons-csv v1.7	1.06	0.066	0.08	0.84	0.000	0.42
6	commons-dbcp2 v2.7.0	1.05	0.000	0.25	0.99	0.712	0.02
7	commons-text v1.8	1.03	0.000	0.26	0.77	0.000	0.36
8	commons-imaging v1.0	1.16	0.000	0.31	0.97	0.541	0.03
9	replace	1.05	0.000	0.35	0.97	0.129	0.07
10	space	1.03	0.000	3.62	0.99	0.379	0.04

[89] are listed in Table IX.

For the highest DBP/RP, 9 out of 10 programs have a p-value below 0.05, which indicates that the difference between DBP and RP is statistically significant (highlighted). For the 9 programs,

all of them, with a nontrivial d value of 0.1 or above [44] are highlighted. For the lowest DBP/RP, there are only 3 programs that have a p-value below 0.05, and the d values in these programs are above 1.0. These are also highlighted.

6.4.2 Comparison of DBP with Forgetting Strategy, with RP

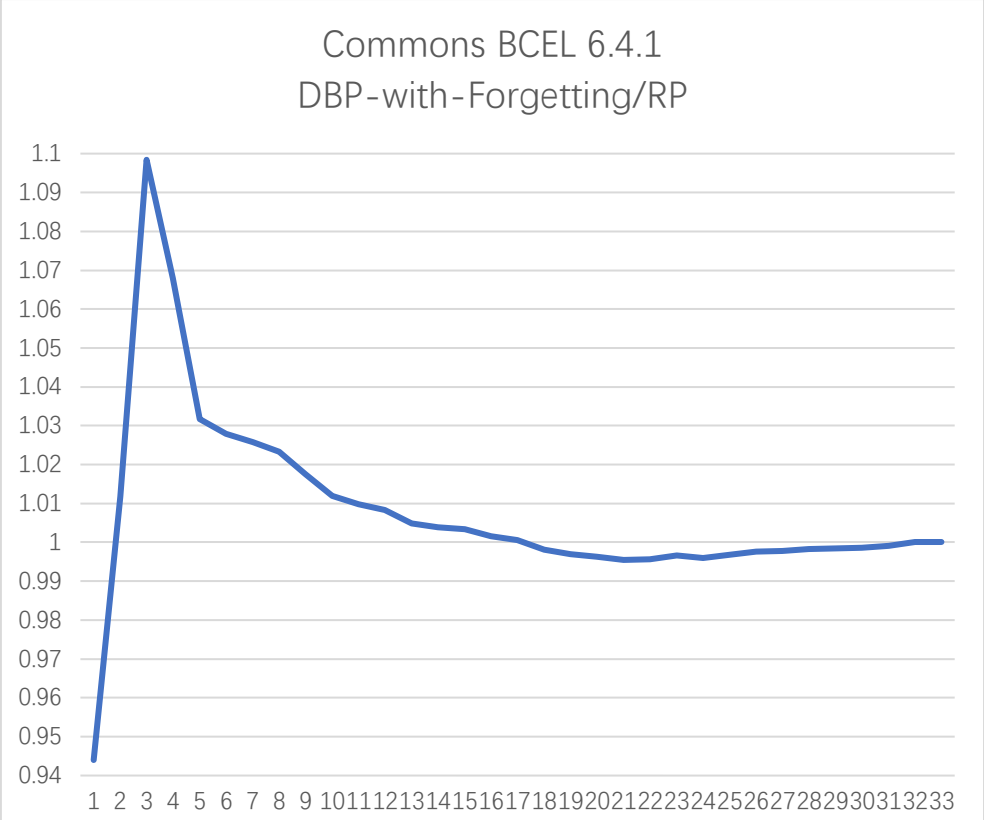
For the DBP with forgetting strategy, the Fig. 7 (commons-bcel v6.4.1, commons-codec v1.13, commons-collections4 v4.4, commons-compress v1.19, commons-csv v1.7, commons-dbcp2 v2.7.0, commons-text v1.8, commons-imaging v1.0 and replace) shows that the result trends of most programs are very similar to the results of DBP without forgetting strategy. It matches the previous study by Chan et al. [73] that the forgetting strategy performs similarly to the basic algorithm, and it improves the complexity of FSCS-ART from $O(n^2)$ to $O(n)$.

TABLE X
DBP WITH FORGETTING WITH RP RESULTS

		DBP with Forgetting		
Row #	Programs	DBP/RP>1	DBP/RP=1	DBP/RP<1
1	commons-bcel v6.4.1	51.5%	3.0%	45.5%
2	commons-codec v1.13	91.2%	1.8%	7.0%
3	commons-collections4 v4.4	98.8%	0.6%	0.6%
4	commons-compress v1.19	39.6%	0.7%	59.7%
5	commons-csv v1.7	60.0%	13.3%	26.7%
6	commons-dbcp2 v2.7.0	63.0%	2.2%	34.8%
7	commons-text v1.8	77.5%	1.3%	21.3%
8	commons-imaging v1.0	98.2%	0.9%	0.9%
9	replace	29.4%	69.6%	1.0%
10	space	23.1%	3.3%	73.6%

From the Fig. 7, the biggest different result is from the space program. The maximum of DBP/RT reduced from 1.030 to 1.007, but the minimum increased from 0.994 to 0.995. The average for DBP with forgetting is 1.000, and it is 1.002 for without forgetting. There are 23.1% of the results (DBP/RP) that are larger than 1, and 73.6% are smaller than 1, the remaining 3.3% results are equal to 1. With 11852 test cases and DBP with forgetting, the coverage rates reached 100%, when the test cases executed reached 13409 for DBP without forgetting and 13300 for RP.

Table X lists the percentage of the results (DBP-with-Forgetting/RP) by three categories,



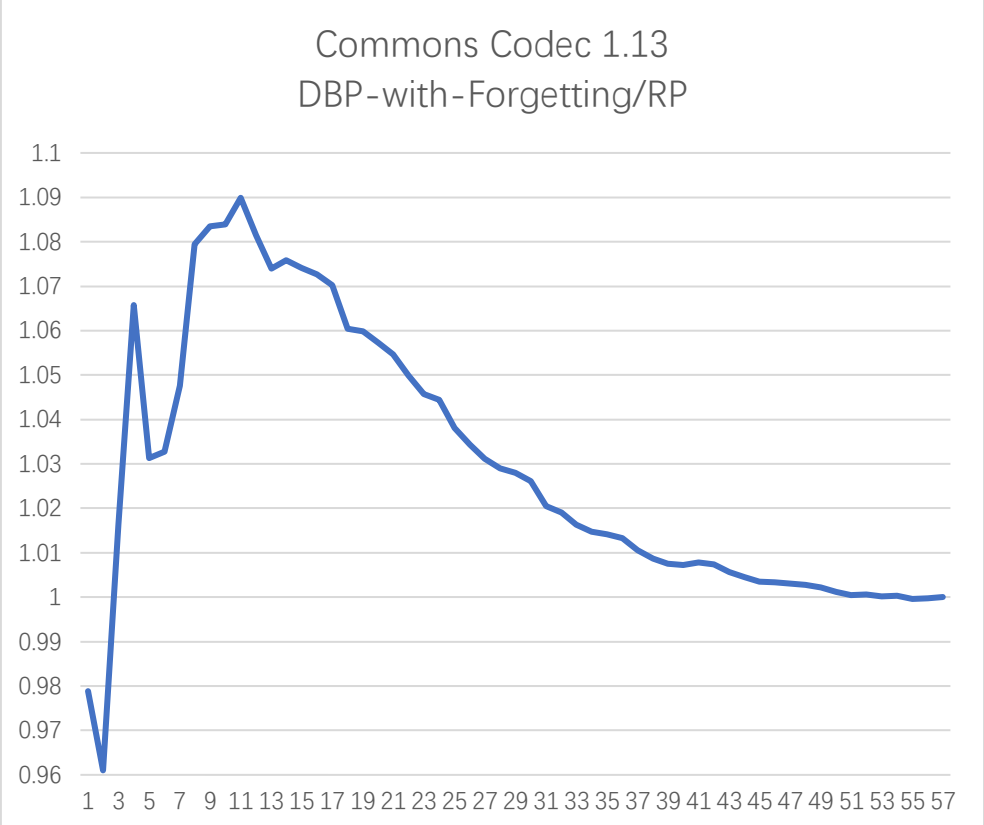
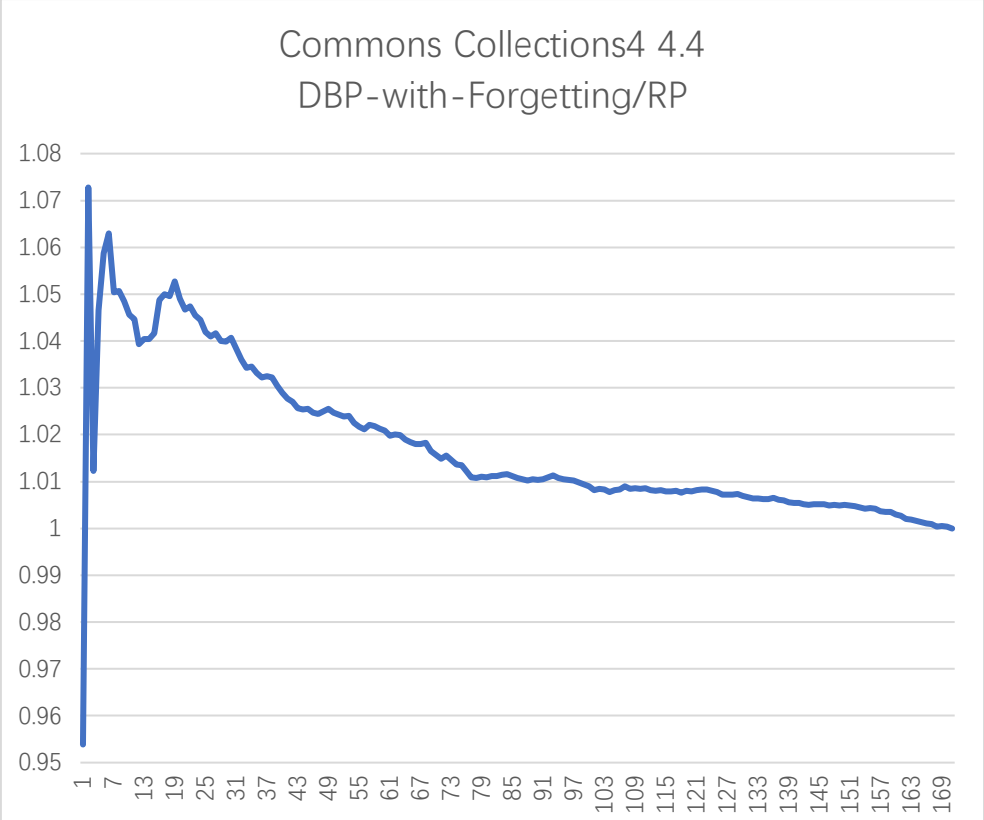


Fig. 7. DBP-with-Forgetting/RP Results



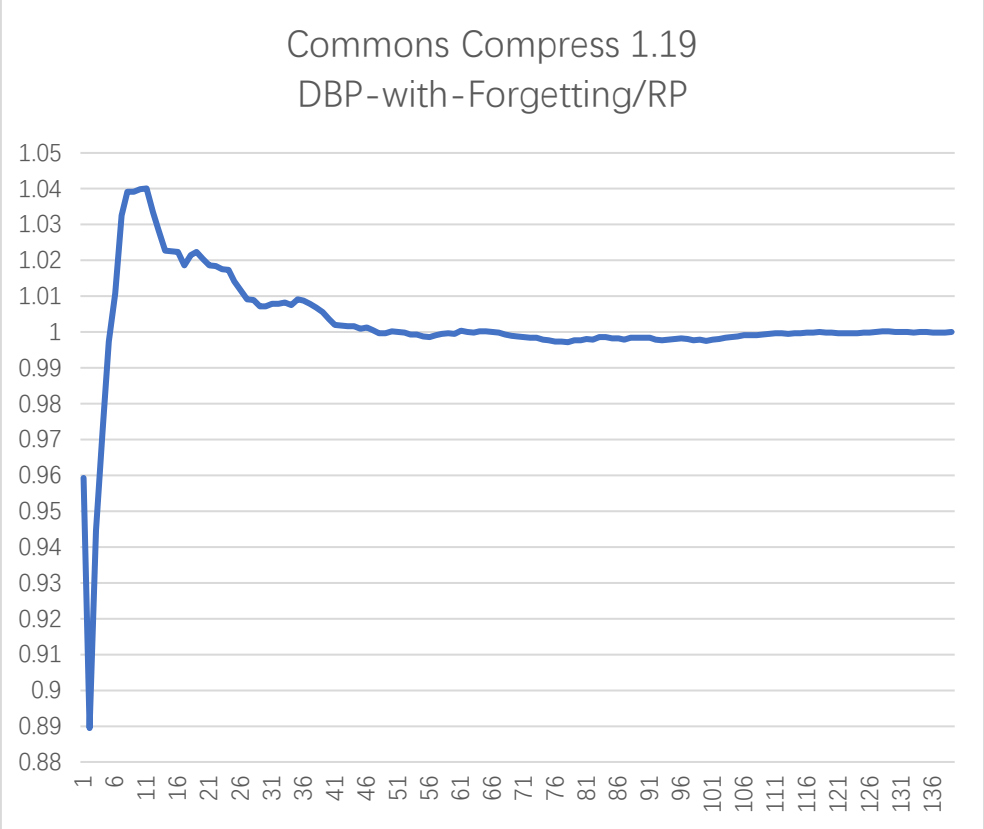
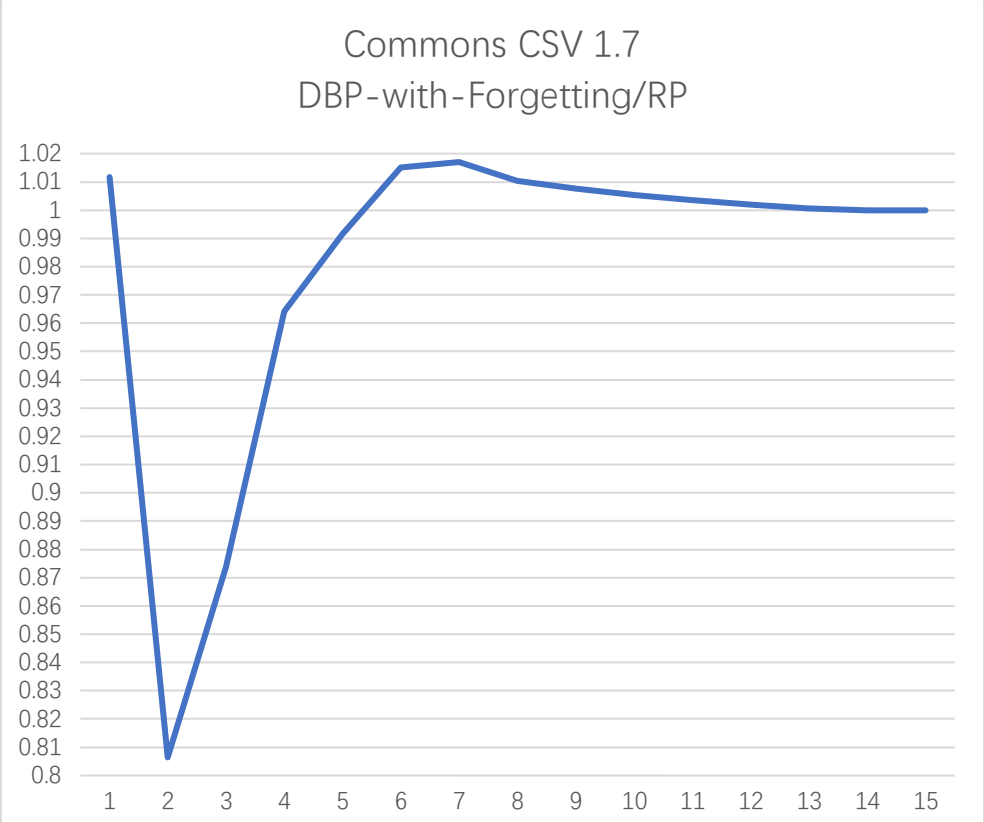


Fig. 7. DBP-with-Forgetting/RP Results Continued



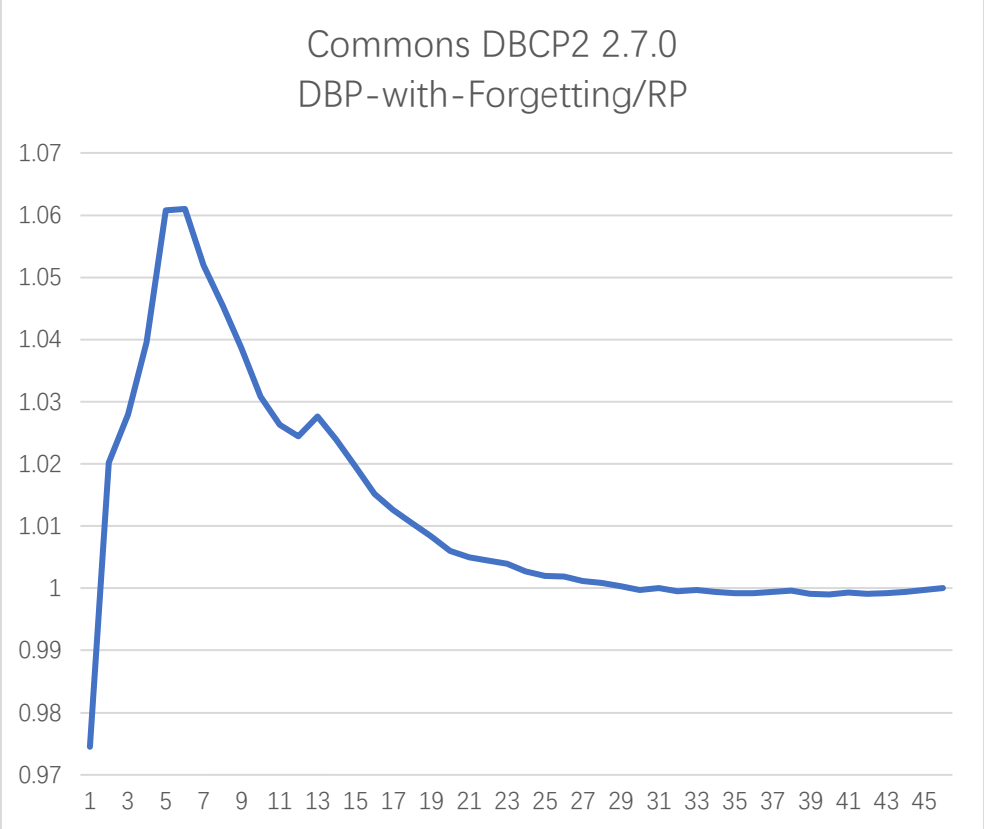


Fig. 7. DBP-with-Forgetting/RP Results Continued

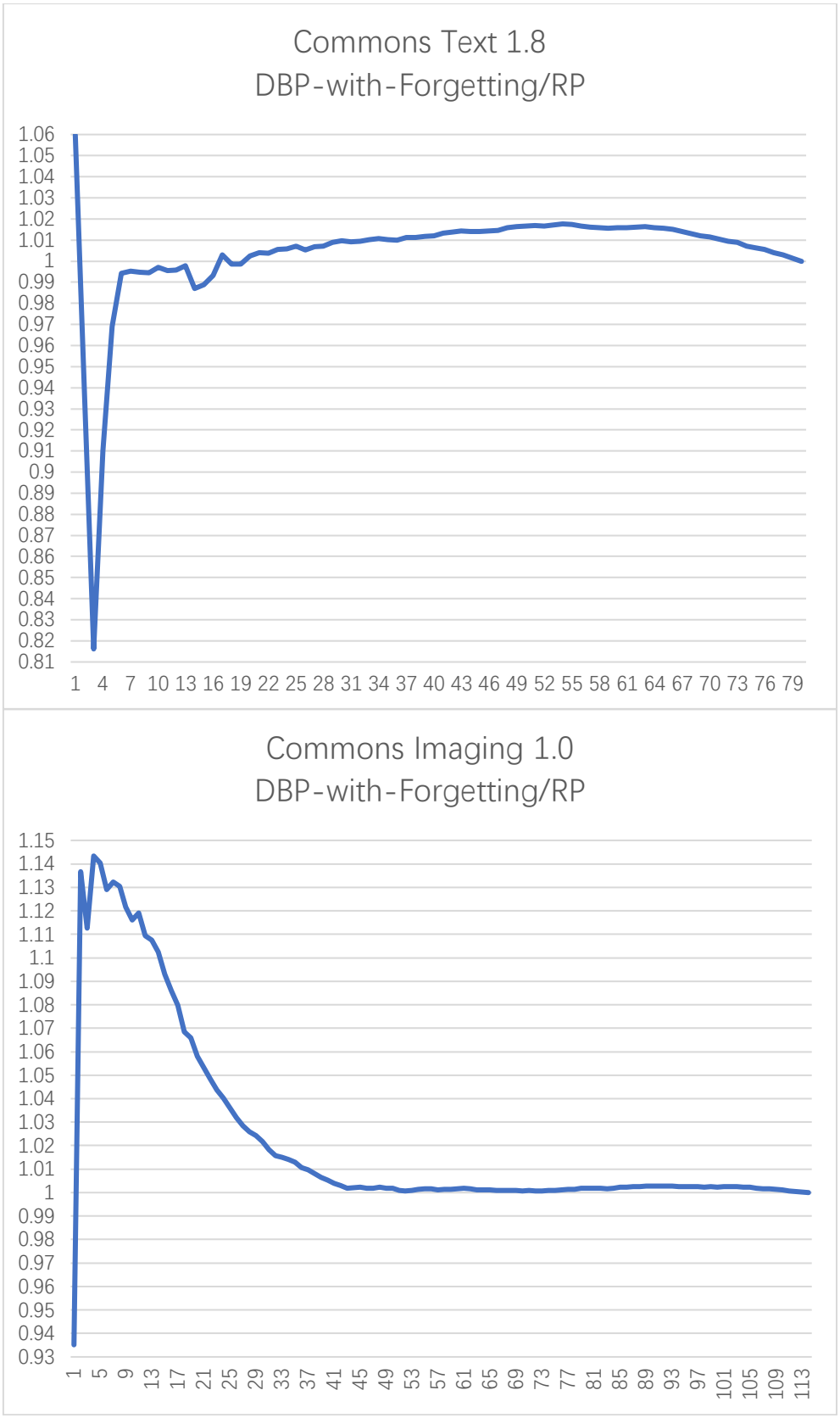


Fig. 7. DBP-with-Forgetting/RP Results Continued

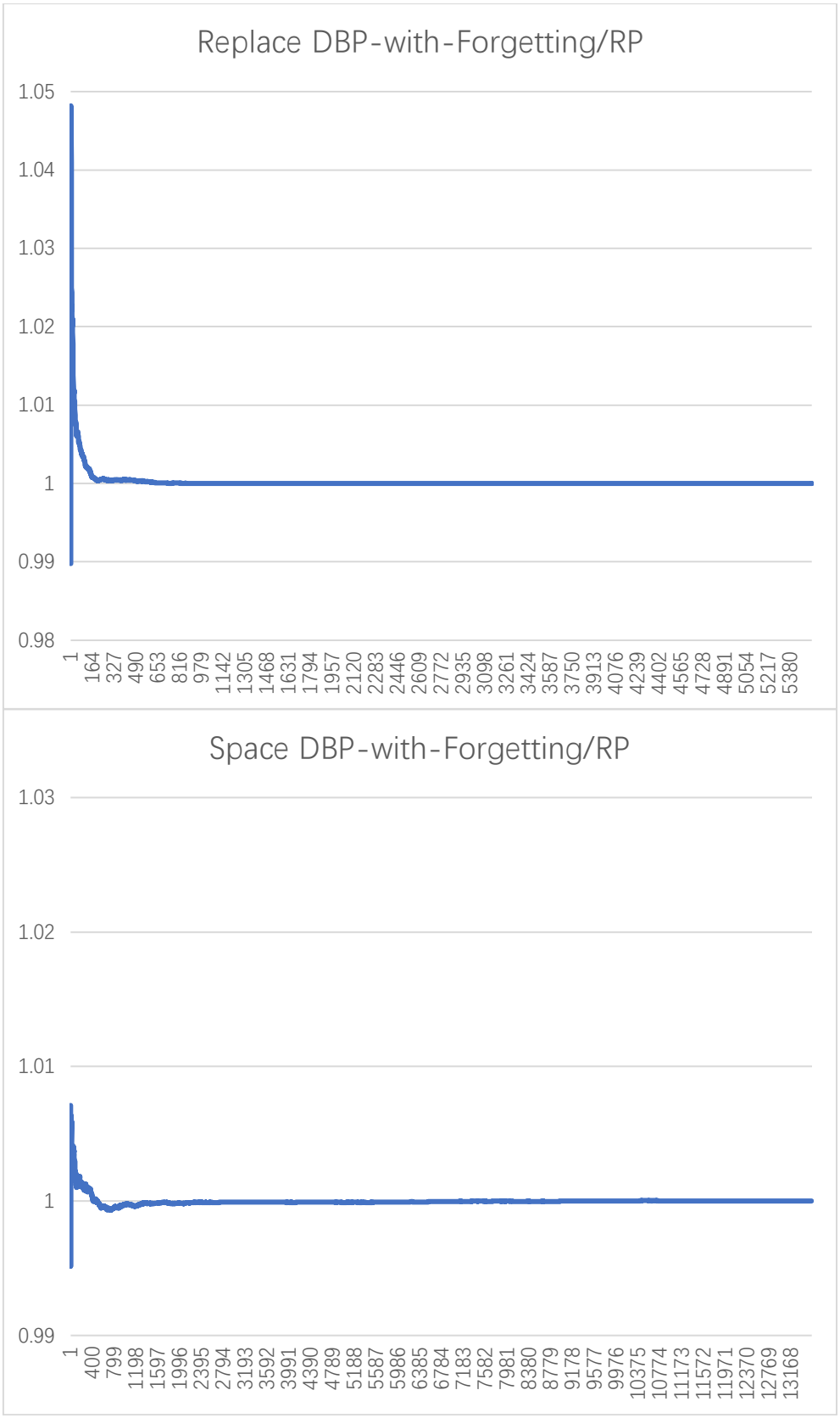


Fig. 7. DBP-with-Forgetting/RP Results Continued

DBP/RP > 1, DBP/RP = 1 and DBP/RP < 1. In addition to the space program, the commons-compress-1.19 has 59.7% of results smaller than 1, and 39.6% fall into the DBP/RP > 1, and the remaining 0.7% are equal to 1. The other 8 programs have higher percentages of results larger than 1, though there is no statistically significant difference between the DBP (with Forgetting Strategy) and RP in this research.

6.5 Summary

In summary, the empirical results show that DBP outperformed the RP in terms of the code coverage rate, on both coverage rate with same number of test cases, and the number of test cases to get 100% coverage rate. That is to say, with same number of test cases, DBP can achieve higher coverage rate than RP, and DBP requires less test cases than RP to reach 100% of the coverage rate.

Chapter 7

Practical Challenges and Unsuccessful Attempts

7.1 Practical Challenges

In this research, as empirical studies were conducted on the proposed approach using real-world projects, and one of the focus aspects is the applicability of the technique, this required many open source projects to study. The biggest challenge was to get test suites and test case information from sufficient projects. As open source projects are written in different languages, and most of them do not apply a generic standard on testing, it is challenging to study the projects and extract test information used for the study. To overcome this challenge, the following steps were used when selecting the open source projects:

1) Search on Internet for open source projects, with additional keywords, such as testing, test cases or similar, to increase the chance to find a new well-tested project.
2) Check the repository of the project, to see if there are multiple versions available. It is better to have multiple small version changes, e.g., 2.64, 2.65, …, 2.68, instead of big version changes 3.1, 3.2, 4.0, 4.1. This is because, for small changes, the structure of the project and the test suite will not have huge differences. However, when the main versions

are different, the structure could be totally different, and the test suite might not be able to execute on the previous version.

3) Download the source code and check if there is an obvious test suite, if so, check the size of the test suite and possible number of test cases.
4) Build the source code and check if there are documents for the test suite. May also try some common testing commands when no document is provided. If the test suite can run successfully, then check the output and determine the number of test cases.
5) When the size of the test suite is big enough, collect the test information; otherwise, exclude this project from the study. After that, check if there is a chance to execute test case separately, and study the method to collect coverage data. When both are available, coverage information can be collected.

The above steps only gave one project that could be used in the study. During this process, only a few of the projects could satisfy all the steps, and each step, especially the steps 4 and 5, are normally very time-consuming.

7.2 Unsuccessful Attempts

In early stage of the research, commercial testing tools, such as the products developed by Parasoft, were used to try to create test cases, in order to get a large number of test cases quickly. But later, it was noticed that the generated test cases may have a very high failure rate, such as 50% or above, which is not suitable for the experiment and not even close to a real-world situation. So instead of using auto-generated test cases, the focus was on using real-world projects to make the empirical study close to real-world situations.

The following list gives part of the downloaded source code packages that were studied but not used in the empirical studies. The other part of the packages have been lost or deleted directly after study. These projects may have minimum testing, lack of test information, or hard to collect coverage information.

account_1.1.tar.gz
adns-1.3.tar.gz
airline_1.1.tar.gz
alarm_clock_1.1.tar.gz
allocationVector_1.1.tar.gz
apache-ant-1.9.2-src.tar.gz
apache-jmeter-5.2_src.tgz
apache-lucy-0.3.3.tar.gz
archimedes-2.0.1.tar.gz
aspell-0.60.6.tar.gz
aspell-0.60.8.tar.gz
auctex-11.87.tar.gz
autoconf-2.69.tar.gz
autogen-5.18.16.tar.gz
autogen-5.18.tar.gz
avl-2.0.3.tar.gz
barcode-0.99.tar.gz
bash_1.0.tar.gz
bash-4.2.tar.gz
bayonne2-2.3.2.tar.gz
bcel-6.4.1-src.tar.gz
Binary-Heap_2.0.tar.gz
binutils-2.23.2.tar.gz
binutils-2.33.1.tar.gz
bison-2.7.1.tar.gz
bison-3.0.tar.gz
boost_1_54_0.tar.gz
bpel2owfn-2.0.4.tar.gz
bsf-src-2.4.0.tar.gz
ccaudio2-2.0.5.tar.gz
ccrtp-2.0.6.tar.gz
ccscript3-1.1.7.tar.gz
cgicc-3.2.10.tar.gz
classpath-0.99.tar.gz
cmake-3.7.0.tar.gz
cmake-3.7.0-Linux-x86_64.tar.gz
commoncpp2-1.8.1.tar.gz
commons-cli-1.4-src.tar.gz
commons-crypto-1.0.0-src.tar.gz
commons-daemon-1.2.2-src.tar.gz
commons-lang-2.4-src.tar.gz
commons-lang-2.5-src.tar.gz
commons-lang-2.6-src.tar.gz
commons-lang3-3.0-src.tar.gz
commons-lang3-3.1-src.tar.gz
commons-lang3-3.9-bin.tar.gz
commons-lang3-3.9-src.tar.gz
commons-math3-3.0-src.tar.gz
commons-math3-3.1.1-src.tar.gz
commons-math3-3.2-src.tar.gz
concordance_1.0.tar.gz
cpio-2.11.tar.gz
cups-1.6.2-source.tar.gz
curl-7.51.0.tar.gz
dap-3.8.tar.gz
ddd-3.3.12.tar.gz
dejagnu-1.5.1.tar.gz
denemo-1.0.4.tar.gz
denemo-2.3.0.tar.gz
diffutils-3.2.tar.gz
dopamine-1.tar.gz
ed-1.9.tar.gz
fajita-0.0.1b1.tar.gz
ferret-0.7.tar.gz
findutils-4.4.2.tar.gz
FreeCAD-0.18.4.tar.gz
freeipmi-1.2.9.tar.gz
freeipmi-1.6.4.tar.gz

From 2019 experiment folder:

gama-1.13.tar.gz
gawk-4.1.0.tar.gz
gc-7.0.tar.gz
gc-7.3alpha2.tar.gz
gcal-3.6.3.tar.gz
gcl-2.6.7.tar.gz
gengetopt-2.22.tar.gz
gettext-0.18.2.tar.gz
gettext-0.18.3.tar.gz
glibc-2.17.tar.gz
glibc-2.24.tar.gz
glibc-2.25.tar.gz
glibc-ports-2.16.0.tar.gz
glpk-4.52.tar.gz
gnash-0.8.10.tar.gz
gnats-4.1.0.tar.gz
gnubatch-1.8.tar.gz
gnuchess-6.0.3.tar.gz
gnu-c-manual-0.2.3.tar.gz
gnufdisk-2.0.0a1.tar.gz
gnuhealth-2.0.0.tar.gz
gnun-0.12.tar.gz
gnun-0.7.tar.gz
gnunet-0.11.6.tar.gz
gnunet-qt-0.8.1.tar.gz
gnuradio-3.2.tar.gz
gnurobots-1.2.0.tar.gz
gperf-3.0.4.tar.gz
gprolog-1.4.4.tar.gz
gprolog-1.4.5.tar.gz
grep_1.2.tar.gz
grep-2.9.tar.gz
groff-1.22.2.tar.gz
grpc-1.27.3.tar.gz
gsl-1.16.tar.gz
gsl-2.3.tar.gz
gtypist-2.9.tar.gz
guile-2.0.9.tar.gz
guile-gnome-platform-2.16.2.tar.gz
guile-gtk-2.1.tar.gz
gv-3.7.4.tar.gz
gzip-1.6.tar.gz
hello-2.8.tar.gz
help2man-1.43.3.tar.gz
ImageMagick.tar.gz
ImageMagick-7.0.3-6.tar.gz
indent-2.2.10.tar.gz
inetlib-1.1.tar.gz
inetutils-1.9.tar.gz
JaConTeBe_1.0.tar.gz
jboss_1.0.tar.gz
jfreechart-1.0.10.tar.gz
jfreechart-1.0.11.tar.gz
jfreechart-1.0.12.tar.gz
jfreechart-1.0.13.tar.gz
jfreechart-1.0.15.tar.gz
joda-time-1.6.1-src.tar.gz
joda-time-1.6.2-src.tar.gz
joda-time-2.1-dist.tar.gz
joda-time-2.2-dist.tar.gz
jpegsrc.v9a.tar.gz
jpegsrc.v9b.tar.gz
kawa-1.13.tar.gz
less-451.tar.gz
libarchive-libarchive-v3.2.2-10-g944b8aa.tar.gz
libbpg-0.9.7.tar.gz

libextractor-0.6.3.tar.gz
libextractor-1.9.tar.gz
libffi-3.0.13.tar.gz
libgsasl-1.8.0.tar.gz
libiconv-1.16.tar.gz
libidn-1.28.tar.gz
libmatheval-1.1.10.tar.gz
libmicrohttpd-0.9.28.tar.gz
libosip2-4.0.0.tar.gz
libosip2-5.1.0.tar.gz
librejs-7.20.1.tar.gz
libsigsegv-2.10.tar.gz
libtasn1-3.3.tar.gz
libtiff-cvsroot.tar.gz
libtool-2.4.tar.gz
libxml2-2.9.4.tar.gz
libzrtpcpp-2.3.4.tar.gz
lightning-2.1.3.tar.gz
lit-html-1.2.1.tar.gz
llvm-3.0.tar.gz
llvm-3.4.src.tar.gz
mailman-2.1.29.tgz
mailutils-2.2.tar.gz
make_1.4.tar.gz
make-3.82.tar.gz
mcron-1.1.tar.gz
mcsim-6.1.0.tar.gz
mpc-1.0.1.tar.gz
mpfr-3.1.2.tar.gz
mysql-5.6.12.tar.gz
mysql-cluster-gpl-7.3.2-linux-glibc2.5-i686.tar.gz
nano-2.3.2.tar.gz
nano-4.5.tar.gz
nasm-2.12.02.tar.gz
ncurses-5.9.tar.gz
octave-3.6.4.tar.gz
octave-5.1.0.tar.gz
openssl-1.1.0c.tar.gz
org.apache.sling.engine-2.6.22-source-release.zip
Panabit_1305_fb8x.tar.gz
PanabitFREE_SANGUOr1p1_20130515_FreeBSD8.0_dev.tar.gz
parted-3.0.tar.gz
perl-5.24.0.tar.gz
php-7.0.12.tar.gz
printtokens_2.0.tar.gz
printtokens2_2.0.tar.gz
proxyknife-1.7.tar.gz
pspp-0.8.0a.tar.gz
pth-2.0.7.tar.gz
putty-0.67.tar.gz
radius-1.6.tar.gz
readline-6.2.tar.gz
replace_2.1.tar.gz
roundcubemail-1.4.3.tar.gz
schedule_2.0.tar.gz
screen-4.0.3.tar.gz
screen-4.7.0.tar.gz
sed_2.0.tar.gz
sed-4.2.2.tar.gz
sharutils-4.13.tar.gz
shtool-2.0.8.tar.gz
sipwitch-1.6.1.tar.gz
SIR project:
smalltalk-3.2.tar.gz
solfege-3.22.0.tar.gz

source-highlight-3.1.7.tar.gz
source-highlight-3.1.tar.gz
src.git-refs_tags_80.0.3982.0.tar.gz
stow-2.3.1.tar.gz
swbis-1.11.tar.gz
tar-1.26.tar.gz
tcas_2.0.tar.gz
tcl8.6.0-src.tar.gz
texinfo-5.1.tar.gz
totinfo_2.0.tar.gz
tramp-2.2.7.tar.gz
tramp-2.4.2.tar.gz
ucommon-6.0.7.tar.gz
unrtf_0.20.4.tar.gz
userv-1.0.5.tar.gz
vim_1.0.tar.gz
wdiff-1.2.1.tar.gz
wget-1.14.tar.gz
wget-1.20.3.tar.gz
xboard-4.7.1.tar.gz
xhippo-3.5.tar.gz
xnee-3.16.tar.gz
xorriso-1.3.0.tar.gz
zile-2.4.9.tar.gz
zzuf-0.15.tar.gz

Chapter 8

Discussions and Conclusion

8.1 Discussions

In this section, we revisit the motivation of the research, summarize the findings, present the limitations of the study, and discuss several potential questions concerning the validity and completeness of the work.

8.1.1 Revisit of Motivation

Various TCP techniques (including input-based ones [21]) have been developed and reported in the literature, but their applicability, especially to real-world software projects, is limited. This is because, different techniques require different types of information and/or dissimilarity metrics on the input values, test cases, or the SUT prior to their application, but in practice, such details may not be available. For example, Yoo and Harman [4] pointed out that "component-based software development method tends to result in the use of many black-box components, often adopted from a third party. Any change in the third-party components may interfere with the rest of the software system, yet it is hard to perform regression testing because the internals of the third-party components are not known to their users."

Additionally, for real-life utilization, the complexity of sophisticated TCP techniques and their

computational overhead may be too high. For example, attempts were made to use $O(n^2)$ TCP algorithms in SQLite (row #1 of Table I) but the execution times were found to be prohibitive. Yoo and Harman [4] also pointed out that "the shorter life-cycle of software development, such as the one suggested by the agile programming discipline, also imposes restrictions and constraints on how regression testing can be performed within limited resources."

One may argue that in TCP, test cases are only ordered once, and then applied to all subsequent versions of the SUT, so that any excessive time spent in TCP is not an issue, and the time complexity of TCP algorithms is not too important. This concept is not valid for the following reasons: First, as reported earlier, when a test suite contains hundreds of thousands of test cases, the computational overhead of conventional nonlinear TCP algorithms may be prohibitive. Second, test suites are updated throughout the course of software evolution, and, hence, TCP is not a one-off activity; rather, it is repeated regularly. Finally, if test cases are always prioritized in exactly the same (deterministic) order, some lowly ranked test cases may never be run unless the entire test suite is executed. Therefore, some randomness in TCP techniques is desirable.

In fact, in the vast majority of real-life situations, RP is considered to be one of the most practical solutions, because it is simplest in concept, easiest, and cheapest to implement, and most importantly, requires no precondition for adoption. This explains why RP is used as the de facto benchmark in TCP studies.

8.1.2 Summary of Findings

In this research, a challenging goal has been raised, specifically, whether a TCP method can be developed to enhance RP. To achieve this goal, the technique must simultaneously satisfy all the following four requirements as stated in research goal RG1: It should be 1) more effective than RP, 2) quicker to detect failures than RP, 3) as efficient as RP, and 4) as readily applicable as RP. It is noted that none of the existing techniques in the literature can simultaneously satisfy all these four requirements.

To address the above research goal, the solution proposed uses the concept of natural distance for real-world test suites, and proposes a simple DBP algorithm. The method does not require any knowledge regarding the software requirements, the SUT, the test history, the test coverage, or even the values of the test cases. Thus, DBP is nonexecution-based and noninput-based and, hence, is as applicable as RP.

With 66 real-world programs collected from the public domain, a series of empirical studies have been conducted. The results show that the method significantly outperforms RP in effectiveness (in terms of both APFD and the F-measure) and the execution time to detect the first failure. Even in the worst case, the performance of DBP is still close to that of RP, and this observation is consistent with previous research results in the ART literature [91]. It is also shown in Table II that the F-measure appears to be a more suitable effectiveness metric than APFD when the test suite is large, as it more clearly shows the differences among different TCP techniques.

TABLE XI
SUMMARY OF RESULTS

	applicability	**effectiveness**	**execution time for failure detection**	**efficiency**
DBP vs. RP	Same.	DBP is better.	DBP is better.	These two algorithms have the same order of (linear) computational complexity, although the RP algorithm consumes less computation time than the DBP algorithm.
DBP vs. Additional	DBP is better.	Additional is better.	DBP is better.	DBP is better: DBP is in O(n), Additional is in O(n^2m).

In terms of efficiency, obviously, the RP algorithm involves fewer computation steps than DBP: When selecting a test case, the former only needs to generate a random number, whereas the latter needs to generate 10 random numbers and conduct 10 × 10 = 100 distance computations. Nevertheless, both of these algorithms have the same order of computational complexity, namely,

the linear complexity. Note that we are concerned with the testing of large and complex real-world systems. The SUT execution time, together with the result verification time, is generally far longer than the DBP driver time consumed for test case selection.[58] Hence, the difference in computation times between the RP and DBP algorithms has little impact on their relative overall execution times. This explains why the comparative results of effectiveness and execution times are similar.

For the traditional TCP algorithms, the additional algorithm is recognized as one of the best. Therefore, it is compared with DBP in research goal RG2. It is shown that DBP outperforms the additional algorithm in applicability, execution time for failure detection, and efficiency. In terms of effectiveness, it is not surprising to find that the lightweight approach cannot guarantee better effectiveness than the heavyweight additional algorithm. It should be noted that the execution time and effectiveness comparisons with the additional algorithm have been performed on only two of 15 projects listed in Table I. Further investigations with more subject programs are warranted in future research.

The comparative results are further summarized in Table XI. In short, with respect to research goal RG1, DBP is shown to simultaneously satisfy all four requirements, and can therefore be considered to be a promising enhancement of RP. In actual practice, testers or developers may often estimate the average time to execute a test case, and such information will be helpful in deciding whether DBP or RP should be used. The findings further suggest that DBP should be considered as a reference benchmark for the evaluation of new TCP techniques. With respect to research goal RG2, the case studies have also produced useful comparative results.

In terms of code coverage, the empirical study was extended with a new set of programs and the DBP compared with RP. From Fig. 6 and Fig. 7, in most cases, the RP has higher coverage rate in the first several test cases, and then the coverage rate of DBP starts to increase and

[58] This is because DBP is a linear algorithm. For nonlinear TCP algorithms, the situation may be very different.

outperform RP to reach the highest DBP/RP in the early stage of execution of the test suite. After that, the coverage rate of both DBP and RP is similar as they approach to 100% coverage. There is only one case, with the program Commons CSV, where the DBP outperforms the RP: at the first test case at the DBP/RP of 1.06, but then decreases to the lowest DBP/RP ratio of 0.84. This is possible because the size of the test suite is too small, it only has 15 test cases. But even in this case, there are still 60% of the results where the DBP has a higher coverage rate than RP. So basically, the DBP outperforms the RP in terms of the code coverage rate.

Observation I is the cornerstone of DBP. An illustrative example is given in Fig. 1 and Fig. 2 to support this observation. It should be noted that the test cases in Fig. 2 have been generated automatically by a tool. Test suites generated by human developers may be different. In some companies, for instance, failure-causing test cases can be added to a regression test suite after a reported bug has been fixed. In other situations, the developers may skip related scenarios. Some of the test suites in the empirical studies have been inspected in detail, and have indeed found obvious similarities among neighboring test cases. Furthermore, a further empirical study was conducted on Observation I, through a comprehensive examination of a new set of real-world projects, which provides the empirical evidence that support the Observation I.

8.1.3 Limitations

In Observation II, it is shown that in real-world test suites, it is not difficult to decide on the order among test cases. Nevertheless, a systematic methodology has not been presented. Indeed, some of the discussions in this thesis may be slightly oversimplified. For instance, the naming convention for test cases varies considerably among different programming languages, projects, frameworks, and organizations. The naming patterns or their implications have not been studied. In fact, test case priority determined by different heuristics may result in distinctive performances. In future research, it is planned to improve the DBP strategy by leveraging the structural information of test suites. For example, it is common in Java projects to have one test suite per class with multiple method invocations with the same naming as the source methods. So the tester

can easily incorporate such knowledge into their partitioning heuristic without affecting the complexity of the algorithm.

Next, the order of execution of a test suite in some frameworks is not necessarily sequential. For instance, the default test case execution order of JUnit is unpredictable when we completed our empirical study. As such, it may not be straightforward for users of such a framework to directly adopt DBP to prioritize their test cases; however, developers of the framework can implement DBP in the platform as an alternative to their original ordering algorithm.

In the case that Observation I is violated, the effectiveness of DBP and RP will become similar, that is, when there is no similarity (in terms of features tested, code coverage, or failure-detection capability, etc.) between neighboring test cases. This may happen if, for example, all the test cases have been randomly generated/sampled, or added to the test suite by different anonymous contributors working simultaneously on different parts of the project (in this situation, the ordering information may be unavailable or unreliable, and, hence, DBP should not be applied). Furthermore, DBP will not be applicable if test case IDs cannot be used—this may happen if the test driver cannot be edited, for example, if the test driver is a binary executable file or is provided as a black-box component by a third party (in this situation, all TCP techniques, except the sequential ordering, will be inapplicable). In addition, sometimes certain test cases may be given a higher priority; in this situation, DBP (as well as any other TCP technique) can only be applied to prioritize the test cases that have the same level of priority. An in-depth investigation into the above scenarios and development of further solutions is warranted in future research.

8.1.4 Why Did We Not Compare DBP With Other TCP Techniques?

As the main research goal is RG1, further comparisons between DBP and various other TCP algorithms are beyond the scope of this research. Such comparisons would actually be unfair because DBP does not require the extra information used by other TCP techniques, it does not even need to know the values of the test cases. Similarly, DBP is not compared with other TCP techniques reported to perform equally well or better than the additional algorithm [7, 18, 19, 24,

101, 102].

Actually, we have attempted to use some of the existing TCP algorithms, which have a quadratic time complexity. However, it was found that when the test suite is large, the test-case-generation time is not only much longer than the test-case-execution time but also prohibitive for any practical purpose. This confirms the finding of Miranda et al. [24] that "some TCP approaches [soon become] inefficient even for small-medium size benchmarks."

8.1.5 During TCP, Is It Appropriate to Consider All the Available Test Cases?

In TCP literature, researchers typically construct a set of test suites, by selecting a small number of test cases from a large test pool, which is created for the research project. Contrary to this research practice, in this thesis, it is advocated that where test suites are large, TCP experiments should be conducted using real-life packages; after all, if the test suites were small, there would be no need for TCP. Test suites with millions of test cases have been extensively reported in the industry (by Microsoft [103], IBM [104], and Google [105], to name a few), and have drawn researchers' attention in recent years [18]. In fact, Miranda et al. [24] noted that most TCP techniques "do not scale up to handle the many thousands or even some millions test suite sizes of modern industrial systems."

8.1.6 Is DBP Really Effective for the Detection of Meaningful Software Issues?

In this research, the empirical study involved testing but not debugging. In other words, the root causes for the failures detected have not been investigated. Nevertheless, it is known that the failures of all 17 faulty Space programs (in rows #27 to #43 of Table I) have been caused by genuine bugs collected during the real-life development of the software. Furthermore, all five GCC programs listed in rows #2 to #6 of Table I (namely, g++, gcc, gfortran, libmudflap, and libstdc++) have been tested using test suite v4.8.0, the same version as the SUTs. This means that

failures detected for these five GCC programs have revealed real and meaningful software issues. In addition, Firefox (row #11 of Table I), the largest SUT of this study, which contains 6 177 736 SLOC, has also been tested using a test suite that is of the same version as the SUT (namely, v31.0) and, therefore, its failures also indicate real and meaningful software issues.

Apart from the above programs, eight subject packages have been involved in the comparison of DBP and RP, in terms of effectiveness, execution time, efficiency and applicability, where a newer version of the test suite has been used to test an older version of the SUT. As listed in Table I, these packages are: SQLite in row #1, commons-lang in row #7, commons-math in row #8, jfreechart in row #9, joda-time in row #10, Autoconf in rows #12 to #16, Automake in rows #17 to #21, and MySQL in rows #22 to #26.

For the above eight packages, one may argue that the "failures" detected in this study might have been caused by compatibility issues (such as the old version SUT not supporting a new input parameter or a new component being absent in the old version), which may not necessarily indicate any defect in the SUT or any problem in the environment, so that such a testing practice is meaningless with respect to fault detection. It should be noted, however, that the purpose of the testing activities with the above eight packages is not direct detection of defects. As explained in Section 4.1.3.3, there could be various other testing objectives in running a newer test suite on an older SUT. These objectives include program comprehension, change impact analysis, behavioral comparison (which may or may not be caused by software faults) between two versions of the software, among others [82, 83, 106-108].

For the results shown in Table II, we can divide the table into two subtables: The first subtable consists of all 17 faulty Space programs (in rows #27 to #43), which includes genuine bugs, and all five GCC programs listed in rows #2 to #6 (namely, g++, gcc, gfortran, libmudflap, and libstdc++), for which the SUT and the test suite are of the same version, as well as row #11 (Firefox, the largest SUT of this study), which has also been tested using the same version of test suite.

The second subtable consists of all the remaining programs, that is, the eight subject packages where a newer test suite has been applied to an older SUT (namely, SQLite in row #1, commons-lang in row #7, commons-math in row #8, jfreechart in row #9, joda-time in row #10, Autoconf in rows #12 to #16, Automake in rows #17 to #21, and MySQL in rows #22 to #26).

For these two subtables, after examining the respective test results, it reveals that the first subtable results are much better than those of the second subtable. Consider column #5 (F.DBP ÷ F.RP), for example, the first subtable has mean, maximum, and minimum values of 0.84, 1.01, and 0.42, respectively, whereas the respective statistics of the second subtable are 0.97, 1.02, and 0.89, which means that the former is the dominating factor for the observed superior performance of DBP over RP.

When designing the experiments the old test suites were applied to the new versions of the SUTs. However, no failures could be detected. As explained in Section 4.1.3.3, this is because the new SUT versions must have already gone through regression testing and passed all of the old test cases before they were released.

A supplemental experiment was conducted using the Replace program of the Siemens suite of programs [99], downloaded from SIR [85], to further confirm the finding that DBP is more effective than RP for the detection of software faults. According to SIR documentation, the Siemens suite of programs were initially assembled by Tom Ostrand and colleagues at Siemens Corporate Research for a study of the fault detection capabilities of control-flow and data-flow coverage criteria [99], and then modified by other researchers for further studies [100]. The Siemens suite of programs have long been used by the testing community for benchmarking testing strategies [109]. Among all seven Siemens programs, the Replace program, which performs regular expression matching and substitutions, is the most complex; despite having only 512 SLOC in C, it covers the most varieties of logic errors [110]. The Replace package includes

TABLE XII
RESULTS OF A SUPPLEMENTAL EXPERIMENT WITH THE REPLACE PROGRAM

project name & version	**F.RP**	**F.DBP**	**F.DBP ÷ F.RP**	**p-value (2-tailed) for F-measure, and effect size (d)**
replace v1	81.11	68.58	0.85	p=0.000, d=0.17 *
replace v2	145.93	135.96	0.93	p=0.000, d=0.07
replace v3	41.81	24.98	0.60	p=0.000, d=0.49 *
replace v4	38.55	22.82	0.59	p=0.000, d=0.51 *
replace v5	20.46	19.04	0.93	p=0.000, d=0.07
replace v6	56.80	54.55	0.96	p=0.004, d=0.04
replace v7	65.75	67.08	1.02	p=0.145, d=0.02
replace v9	178.70	181.60	1.02	p=0.242, d=0.02
replace v10	230.74	234.85	1.02	p=0.193, d=0.02
replace v13	34.30	21.01	0.61	p=0.000, d=0.48 *
replace v15	91.72	89.36	0.97	p=0.059, d=0.03
replace v17	225.32	230.15	1.02	p=0.115, d=0.02
replace v18	26.56	25.54	0.96	p=0.004, d=0.04
replace v19	1,373.69	1,422.78	1.04	p=0.001, d=0.05
replace v20	246.15	252.65	1.03	p=0.055, d=0.03
replace v21	1,372.45	1,139.15	0.83	p=0.000, d=0.24 *
replace v22	285.51	292.11	1.02	p=0.084, d=0.02
replace v24	32.72	32.54	0.99	p=0.687, d=0.01
replace v25	1,406.24	1,463.03	1.04	p=0.000, d=0.05
replace v27	20.69	20.90	1.01	p=0.472, d=0.01
replace v28	38.86	24.37	0.63	p=0.000, d=0.45 *
replace v29	85.69	77.78	0.91	p=0.000, d=0.10 *
		avg	0.91	
		max	1.04	
		min	0.59	

F.RP: F-measure of RP out of 10 000 trials; F.DBP: F-measure of DBP out of 10 000 trials. Where there is a statistically significant difference between the RP and DBP means (with a p-value below 0.05), the corresponding cells are highlighted (light gray: DBP outperformed RP; dark gray: RP outperformed DBP). “p = 0.000” means p < 0.0005. Where the effect size (Cohen’s d) is 0.10 or larger, the corresponding cells are starred (*). “d = 0.00” means d < 0.005.

a test driver that runs 5542 test cases. The order of test case execution in the test driver is, therefore, used as the order of the test cases in this experiment. The package contains a base version, which is used as the test oracle, and a total of 32 faulty versions; although the faults are created manually, the Siemens researchers have made them as realistic as possible [99]. Of the 32 faulty versions, 22 have recorded a failure rate below 5%. These 22 faulty versions, therefore, are used in the experiment. As with the previous experiments, for each faulty version, 10 000 trials of DBP and 10 000 trials of RP were conducted to estimate their respective F-measures. The experimental results are given in Table XII, which clearly shows that DBP has outperformed RP. These results are consistent with those collected from genuine real-life packages presented earlier in this thesis.

8.2 Conclusion and Future Work

The most important contribution of this research was the discovery of Observation I, a very simple but important property of real-world test suites that can support test case prioritization. More specifically, it was observed that neighboring test cases in real-world test suites often have similarities in certain ways. An empirical study was conducted on the validity of this observation with a new set of real-world projects, by inspecting 66 software projects and their test suites.

Based on Observation I, an extremely simple approach is proposed for prioritizing test cases. The algorithm itself is not novel, as it is a direct application of adaptive random testing. The novelty lies in the dispersity metric, which makes use of test case IDs (which are readily available) rather than concrete input values (for which the distance may not be easy to measure) or test case coverage data (which may not be available).

The results are consistent with recent studies. They suggest that diversity (and, therefore, dissimilarity) is a key concept underlying the foundations of successful software testing strategies [62, 111]. The proposed dispersity-based prioritization algorithm generates a sequence of test case IDs by using the “FSCS-ART with forgetting” method. Other linear ART algorithms [68] can also be adopted to replace “FSCS-ART with forgetting.” An empirical evaluation of these other

algorithms in the context of text case prioritization and test case selection using the natural distance is a future research topic.

This research has shown that DBP is more applicable than the additional algorithm. Also, even when conventional TCP techniques are applicable, DBP can still be a better choice, especially if the test suite is very large resulting in a high computational overhead. The proposed approach, therefore, provided an innovative direction and practical hints for testing engineers dealing with large test suites. It can be a very simple and yet useful solution to the TCP problem in real life.

The original order of test cases from real-world projects may be generated according to very different rules, and not all test cases are logically generated by tools. Furthermore, during software evolution, many new test cases are added to the test suite, making the test order different. However, as reported earlier in this thesis, even in such circumstances, applying DBP will do no harm to the test effectiveness and efficiency.

Bibliography

[1] G. Rothermel, R. H. Untch, C. Chengyun, and M. J. Harrold, "Prioritizing test cases for regression testing," *IEEE transactions on software engineering,* vol. 27, no. 10, pp. 929-948, 2001.

[2] S. Elbaum, A. G. Malishevsky, and G. Rothermel, "Test case prioritization: a family of empirical studies," *IEEE transactions on software engineering,* vol. 28, no. 2, pp. 159-182, 2002.

[3] G. Rothermel, M. Burnett, L. Li, C. Dupuis, and A. Sheretov, "A methodology for testing spreadsheets," *ACM transactions on software engineering and methodology,* vol. 10, no. 1, pp. 110-147, 2001.

[4] S. Yoo and M. Harman, "Regression testing minimization, selection and prioritization: A survey," *Software testing, verification & reliability,* vol. 22, no. 2, pp. 67-120, 2012.

[5] W. Shuai, S. Ali, Y. Tao, O. Bakkeli, and M. Liaaen, "Enhancing Test Case Prioritization in an Industrial Setting with Resource Awareness and Multi-objective Search," in *2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C)*, 2016: ACM, pp. 182-191.

[6] H. Mei, D. Hao, L. Zhang, L. Zhang, J. Zhou, and G. Rothermel, "A Static Approach to Prioritizing JUnit Test Cases," *IEEE transactions on software engineering,* vol. 38, no. 6, pp. 1258-1275, 2012.

[7] B. Jiang, Z. Zhang, W. K. Chan, and T. H. Tse, "Adaptive Random Test Case Prioritization," in *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, 2009: IEEE Computer Society, in ASE '09, pp. 233-244.

[8] Z. Li, M. Harman, and R. M. Hierons, "Search Algorithms for Regression Test Case Prioritization," *IEEE transactions on software engineering,* vol. 33, no. 4, pp. 225-237, 2007.

[9] M. B. Cohen, M. B. Dwyer, and S. Jiangfan, "Constructing Interaction Test Suites for Highly-Configurable Systems in the Presence of Constraints: A Greedy Approach," *IEEE transactions on software engineering*, vol. 34, no. 5, pp. 633-650, 2008.

[10] H. Srikanth, L. Williams, and J. Osborne, "System test case prioritization of new and regression test cases," in *2005 International Symposium on Empirical Software Engineering*, 2005: IEEE, p. 10 pp.

[11] B. Korel, G. Koutsogiannakis, and L. H. Tahat, "Application of system models in regression test suite prioritization," in *2008 IEEE International Conference on Software Maintenance*, 2008: IEEE, pp. 247-256.

[12] J.-M. Kim and A. Porter, "A history-based test prioritization technique for regression testing in resource constrained environments," in *Proceedings of the 24th International Conference on software engineering*, New York NY, 2002: ACM, in ICSE '02, pp. 119-129.

[13] Y.-C. Huang, K.-L. Peng, and C.-Y. Huang, "A history-based cost-cognizant test case prioritization technique in regression testing," *The Journal of systems and software*, vol. 85, no. 3, pp. 626-637, 2012.

[14] Z. Q. Zhou, A. Sinaga, W. Susilo, L. Zhao, and K.-Y. Cai, "A cost-effective software testing strategy employing online feedback information," *Information sciences*, vol. 422, pp. 318-335, 2018.

[15] S.-S. Hou, L. Zhang, T. Xie, H. Mei, and J.-S. Sun, "Applying Interface-Contract Mutation in Regression Testing of Component-Based Software," in *2007 IEEE International Conference on Software Maintenance*, 2007: IEEE, pp. 174-183.

[16] S. Elbaum, A. Malishevsky, and G. Rothermel, "Incorporating varying test costs and fault severities into test case prioritization," in *Proceedings of the 23rd International Conference on software engineering*, 2001: IEEE Computer Society, in ICSE '01, pp. 329-338.

[17] Z. Q. Zhou, "Using Coverage Information to Guide Test Case Selection in Adaptive Random Testing," in *2010 IEEE 34th Annual Computer Software and Applications Conference Workshops*, 2010: IEEE, pp. 208-213.

[18] Z. Q. Zhou, A. Sinaga, and W. Susilo, "On the Fault-Detection Capabilities of Adaptive Random Test Case Prioritization: Case Studies with Large Test Suites," in *2012 45th Hawaii International Conference on System Sciences*, 2012: IEEE, pp. 5584-5593.

[19] R. Saha, L. Zhang, S. Khurshid, and D. Perry, "An information retrieval approach for regression test prioritization based on program changes," in *Proceedings of the 37th International Conference on software engineering*, 2015, vol. 1: IEEE Press, in ICSE '15, pp. 268-279.

[20] B. Busjaeger and T. Xie, "Learning for test prioritization: an industrial case study," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on foundations of software engineering*, 2016, vol. 13-18-: ACM, in FSE 2016, pp. 975-980.

[21] B. Jiang and W. K. Chan, "Input-based adaptive randomized test case prioritization: A local beam search approach," *The Journal of systems and software*, vol. 105, pp. 91-106, 2015.

[22] Y. Ledru, A. Petrenko, and S. Boroday, "Using String Distances for Test Case Prioritisation," in *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, 2009: IEEE Computer Society, in ASE '09, pp. 510-514.

[23] S. W. Thomas, H. Hemmati, A. E. Hassan, and D. Blostein, "Static test case prioritization using topic models," *Empirical software engineering : an international journal*, vol. 19, no. 1, pp. 182-212, 2014.

[24] B. Miranda, E. Cruciani, R. Verdecchia, and A. Bertolino, "FAST approaches to scalable similarity-based test case prioritization," in *Proceedings of the 40th International Conference on software engineering*, 2018, vol. 2018-: ACM, in ICSE '18, pp. 222-232.

[25] W. E. Wong, J. R. Horgan, S. London, and H. Agrawal, "A study of effective regression testing in practice," in *Proceedings The Eighth International Symposium on Software Reliability Engineering*, 1997: IEEE, pp. 264-274.

[26] G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold, "Test case prioritization: an empirical study," in *Proceedings IEEE International Conference on Software Maintenance - 1999 (ICSM'99). 'Software Maintenance for Business Change' (Cat. No.99CB36360)*, 1999: IEEE, pp. 179-188.

[27] J. Chi *et al.*, "Relation-based test case prioritization for regression testing," *The Journal of systems and software*, vol. 163, p. 110539, 2020.

[28] M. Arnold and B. Ryder, "A framework for reducing the cost of instrumented code," in *Proceedings of the ACM SIGPLAN 2001 conference on programming language design and implementation*, 2001: ACM, in PLDI '01, pp. 168-179.

[29] A. Srivastava and J. Thiagarajan, "Effectively prioritizing tests in development environment," in *Proceedings of the 2002 ACM SIGSOFT international symposium on software testing and analysis*, 2002: ACM, in ISSTA '02, pp. 97-106.

[30] J. A. Jones and M. J. Harrold, "Test-suite reduction and prioritization for modified condition/decision coverage," *IEEE transactions on software engineering*, vol. 29, no. 3, pp. 195-209, 2003.

[31] S. Elbaum, D. Gable, and G. Rothermel, "Understanding and measuring the sources of variation in the prioritization of regression test suites," in *Proceedings Seventh International Software Metrics Symposium*, 2001: IEEE, pp. 169-179.

[32] B. Korel, L. H. Tahat, and M. Harman, "Test prioritization using system models," in *21st IEEE International Conference on Software Maintenance (ICSM'05)*, 2005: IEEE, pp. 559-568.

[33] D. Xu and J. Ding, "Prioritizing State-Based Aspect Tests," in *2010 Third International Conference on Software Testing, Verification and Validation*, 2010: IEEE, pp. 265-274.

[34] Z. Ma and J. Zhao, "Test Case Prioritization Based on Analysis of Program Structure," in *2008 15th Asia-Pacific Software Engineering Conference*, 2008: IEEE, pp. 471-478.

[35] R. Huang, Q. Zhang, D. Towey, W. Sun, and J. Chen, "Regression test case prioritization by code combinations coverage," *The Journal of systems and software*, vol. 169, p. 110712, 2020.

[36] C. Lu, J. Zhong, Y. Xue, L. Feng, and J. Zhang, "Ant Colony System With Sorting-Based Local Search for Coverage-Based Test Case Prioritization," *IEEE transactions on reliability*, vol. 69, no. 3, pp. 1004-1020, 2020.

[37] R. Abou Assi, W. Masri, and C. Trad, "Substate Profiling for Effective Test Suite Reduction," in *2018 IEEE 29th International Symposium on Software Reliability Engineering (ISSRE)*, 2018, vol. 2018-: IEEE, pp. 123-134.

[38] R. Abou Assi, W. Masri, and C. Trad, "Substate Profiling for Enhanced Fault Detection and Localization: An Empirical Study," in *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*, 2020: IEEE, pp. 16-27.

[39] B. Korel, G. Koutsogiannakis, and L. Tahat, "Model-based test prioritization heuristic methods and their evaluation," in *Proceedings of the 3rd international workshop on advances in model-based testing*, 2007: ACM, in A-MOST '07, pp. 34-43.

[40] N. Gökçe, F. Belli, M. Eminli, and B. T. Dinçer, "Model-based test case prioritization using cluster analysis: A soft-computing approach," *Turkish Journal of Electrical Engineering and Computer Sciences*, vol. 23, no. 3, pp. 623-640, 2015.

[41] K.-W. Shin and D.-J. Lim, "Model-Based Test Case Prioritization Using an Alternating Variable Method for Regression Testing of a UML-Based Model," *Applied sciences*, vol. 10, no. 21, p. 7537, 2020.

[42] R. Krishnamoorthi and S. A. Sahaaya Arul Mary, "Factor oriented requirement coverage based system test case prioritization of new and regression test cases," *Information and software technology*, vol. 51, no. 4, pp. 799-808, 2009.

[43] W. Zhang, Y. Qi, X. Zhang, B. Wei, M. Zhang, and Z. Dou, "On Test Case Prioritization Using Ant Colony Optimization Algorithm," in *2019 IEEE 21st International Conference on High Performance Computing and Communications; IEEE 17th International Conference on Smart City; IEEE 5th International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*, 2019: IEEE, pp. 2767-2773.

[44] M. Abbas, I. Inayat, M. Saadatmand, and N. Jan, "Requirements Dependencies-Based Test Case Prioritization for Extra-Functional Properties," in *2019 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, 2019: IEEE, pp. 159-163.

[45] C. Hettiarachchi and H. Do, "A Systematic Requirements and Risks-Based Test Case Prioritization Using a Fuzzy Expert System," in *2019 IEEE 19th International Conference on Software Quality, Reliability and Security (QRS)*, 2019: IEEE, pp. 374-385.

[46] A. Arrieta, S. Wang, G. Sagardui, and L. Etxeberria, "Search-Based test case prioritization for simulation-Based testing of cyber-Physical system product lines," *The Journal of systems and software*, vol. 149, pp. 1-34, 2019.

[47] X. Wang and H. Zeng, "History-Based Dynamic Test Case Prioritization for Requirement Properties in Regression Testing," in *2016 IEEE/ACM International Workshop on Continuous Software Evolution and Delivery (CSED)*, 2016: ACM, pp. 41-47.

[48] T. Noguchi, H. Washizaki, Y. Fukazawa, A. Sato, and K. Ota, "History-Based Test Case Prioritization for Black Box Testing Using Ant Colony Optimization," in *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*, 2015: IEEE, pp. 1-2.

[49] Y. Yang, Z. Li, L. He, and R. Zhao, "A systematic study of reward for reinforcement learning based continuous integration testing," *The Journal of systems and software*, vol. 170, p. 110787, 2020.

[50] J. A. d. Prado Lima and S. R. Vergilio, "A Multi-Armed Bandit Approach for Test Case Prioritization in Continuous Integration Environments," *IEEE transactions on software engineering*, pp. 1-1, 2020.

[51] D. Pradhan, S. Wang, S. Ali, T. Yue, and M. Liaaen, "Employing rule mining and multi-objective search for dynamic test case prioritization," *The Journal of systems and software*, vol. 153, pp. 86-104, 2019.

[52] S. Dirim and H. Sozer, "Prioritization of Test Cases with Varying Test Costs and Fault Severities for Certification Testing," in *2020 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, 2020: IEEE, pp. 386-391.

[53] D. Hao, L. Zhang, and H. Mei, "Test-case prioritization: achievements and challenges," *Frontiers of Computer Science*, vol. 10, no. 5, pp. 769-777, 2016.

[54] R. W. Hamming, "Error detecting and error correcting codes," *The Bell System technical journal*, vol. 29, no. 2, pp. 147-160, 1950.

[55] V. I. Levenshtein, "Binary codes capable of correcting deletions, insertions, and reversals," *Doklady Akademii Nauk SSSR*, vol. 163, no. 4, pp. 845-848, 1965.

[56] A. Kuhn, S. Ducasse, and T. Gîrba, "Semantic clustering: Identifying topics in source code," *Information and software technology*, vol. 49, no. 3, pp. 230-243, 2007.

[57] T. Y. Chen, H. Leung, and I. K. Mak, "Adaptive random testing," in *9th Asian Computing Science Conference*, Berlin, 2004, vol. 3321: Springer, pp. 320-329.

[58] J. Chen *et al.*, "Test Case Prioritization for Compilers: A Text-Vector Based Approach," in *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, 2016: IEEE, pp. 266-277.

[59] C. Fang, Z. Chen, K. Wu, and Z. Zhao, "Similarity-based test case prioritization using ordered sequences of program entities," *Software quality journal*, vol. 22, no. 2, pp. 335-361, 2014.

[60] D. Leon and A. Podgurski, "A comparison of coverage-based and distribution-based techniques for filtering and prioritizing test cases," in *14th International Symposium on Software Reliability Engineering, 2003. ISSRE 2003*, 2003: IEEE, pp. 442-453.

[61] T. Bin Noor and H. Hemmati, "A similarity-based approach for test case prioritization using historical failure data," in *2015 IEEE 26th International Symposium on Software Reliability Engineering (ISSRE)*, 2015: IEEE, pp. 58-68.

[62] T. Y. Chen, F.-C. Kuo, R. G. Merkel, and T. H. Tse, "Adaptive Random Testing: The ART of test case diversity," *The Journal of systems and software*, vol. 83, no. 1, pp. 60-66, 2010.

[63] T. Y. Chen, F.-C. Kuo, H. Liu, and W. E. Wong, "Code Coverage of Adaptive Random Testing," *IEEE transactions on reliability,* vol. 62, no. 1, pp. 226-237, 2013.

[64] T. Y. Chen, D. H. Huang, and F. C. Kuo, "Adaptive random testing by balancing," in *Proceedings of the 2nd international workshop on random testing,* 2007: ACM, in RT '07, pp. 2-9.

[65] T. Y. Chen, F. C. Kuo, R. G. Merkel, and S. P. Ng, "Mirror adaptive random testing," *Information and software technology,* vol. 46, no. 15, pp. 1001-1010, 2004.

[66] T. Y. Chen, F.-C. Kuo, and H. Liu, "Adaptive random testing based on distribution metrics," *The Journal of systems and software,* vol. 82, no. 9, pp. 1419-1433, 2009.

[67] A. Shahbazi, A. F. Tappenden, and J. Miller, "Centroidal Voronoi Tessellations- A New Approach to Random Testing," *IEEE transactions on software engineering,* vol. 39, no. 2, pp. 163-183, 2013.

[68] S. Anand *et al.,* "An orchestrated survey of methodologies for automated software test case generation," *The Journal of systems and software,* vol. 86, no. 8, pp. 1978-2001, 2013.

[69] K. P. Chan, T. Y. Chen, and D. Towey, "RESTRICTED RANDOM TESTING: ADAPTIVE RANDOM TESTING BY EXCLUSION," *International journal of software engineering and knowledge engineering,* vol. 16, no. 4, pp. 553-584, 2006.

[70] T. Y. Chen, R. Merkel, P. K. Wong, and G. Eddy, "Adaptive random testing through dynamic partitioning," in *Fourth International Conference onQuality Software, 2004. QSIC 2004. Proceedings,* 2004: IEEE, pp. 79-86.

[71] T. Y. Chen, D. H. Huang, and Z. Q. Zhou, "On Adaptive Random Testing through Iterative Partitioning," *Journal of information science and engineering,* vol. 27, no. 4, pp. 1449-1472, 2011.

[72] H. Liu, X. Xie, J. Yang, Y. Lu, and T. Y. Chen, "Adaptive random testing through test profiles," *Software: Practice & Experience,* vol. 41, no. 10, pp. 1131-1154, 2011.

[73] K. P. Chan, T. Y. Chen, and D. Towey, "Forgetting Test Cases," in *30th Annual International Computer Software and Applications Conference (COMPSAC'06),* 2006, vol. 1: IEEE, pp. 485-494.

[74] S. Khurshid, C. S. Păsăreanu, and W. Visser, "Generalized symbolic execution for model checking and testing," *Lecture notes in computer science,* vol. 2619, pp. 553-568, 2003.

[75] K. Sen, D. Marinov, and G. Agha, "CUTE: a concolic unit testing engine for C," in *Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on foundations of software engineering,* New

York NY, 2005: ACM, in ESEC/FSE-13, pp. 263-272.

[76] M. Pezzè and M. Young, *Software testing and analysis : process, principles, and techniques*. Hoboken, N.J: Wiley, 2008.

[77] A. C. Barus, T. Y. Chen, F.-C. Kuo, H. Liu, R. Merkel, and G. Rothermel, "A Cost-Effective Random Testing Method for Programs with Non-Numeric Inputs," *IEEE transactions on computers*, vol. 65, no. 12, pp. 3509-3523, 2016.

[78] E. Selay, Z. Q. Zhou, T. Y. Chen, and F.-C. Kuo, "Adaptive Random Testing in Detecting Layout Faults of Web Applications," *International journal of software engineering and knowledge engineering*, vol. 28, no. 10, pp. 1399-1428, 2018.

[79] T. Y. Chen and R. Merkel, "An upper bound on software testing effectiveness," *ACM transactions on software engineering and methodology*, vol. 17, no. 3, pp. 1-27, 2008.

[80] B. A. Kitchenham *et al.*, "Preliminary guidelines for empirical research in software engineering," *IEEE transactions on software engineering*, vol. 28, no. 8, pp. 721-734, 2002.

[81] Y. Li, C. Zhu, M. Gligoric, J. Rubin, and M. Chechik, "Precise semantic history slicing through dynamic delta refinement," *Automated software engineering*, vol. 26, no. 4, pp. 757-793, 2019.

[82] W. Jin, O. Alessandro, and T. Xie, "Automated Behavioral Regression Testing," in *2010 Third International Conference on Software Testing, Verification and Validation*, 2010: IEEE, pp. 137-146.

[83] G. Soares, R. Gheyi, and T. Massoni, "Automated Behavioral Testing of Refactoring Engines," *IEEE transactions on software engineering*, vol. 39, no. 2, pp. 147-162, 2013.

[84] SQLite website, "How SQLite is tested," https://www.sqlite.org/testing.html, Accessed on: March 1, 2019.

[85] H. Do, S. Elbaum, and G. Rothermel, "Supporting Controlled Experimentation with Testing Techniques: An Infrastructure and its Potential Impact," *Empirical software engineering : an international journal*, vol. 10, no. 4, pp. 405-435, 2005.

[86] Software-artifact Infrastructure Repository, https://sir.csc.ncsu.edu/portal/index.php, Accessed on: March 1, 2019.

[87] F. I. Vokolos and P. G. Frankl, "Empirical evaluation of the textual differencing regression testing technique," in *Conference on Software Maintenance*, 1998, pp. 44-53.

[88] A. Field, *Discovering Statistics Using IBM SPSS Statistics: North American Edition*. SAGE Publications, 2017.

[89] J. Cohen, *Statistical power analysis for the behavioral sciences*, 2nd ed. ed. Hillsdale, N.J: L. Erlbaum Associates, 1988.

[90] J. A. Durlak, "How to Select, Calculate, and Interpret Effect Sizes," *Journal of pediatric psychology*, vol. 34, no. 9, pp. 917-928, 2009.

[91] T. Y. Chen, F.-C. Kuo, and Z. Q. Zhou, "ON FAVOURABLE CONDITIONS FOR ADAPTIVE RANDOM TESTING," *International journal of software engineering and knowledge engineering*, vol. 17, no. 6, pp. 805-825, 2007.

[92] K.-H. Yuan and S. Maxwell, "On the Post Hoc Power in Testing Mean Differences," *Journal of educational and behavioral statistics*, vol. 30, no. 2, pp. 141-167, 2005.

[93] A. Ghasemi and S. Zahediasl, "Normality tests for statistical analysis: A guide for non-statisticians," *International journal of endocrinology and metabolism*, vol. 10, no. 2, pp. 486-489, 2012.

[94] D. G. Altman and J. M. Bland, "Statistics notes: The normal distribution," *BMJ*, vol. 310, no. 6975, pp. 298-298, 1995.

[95] C. A. Boneau, "The effects of violations of assumptions underlying the t test," *Psychological bulletin*, vol. 57, no. 1, pp. 49-64, 1960.

[96] J. H. Zar, *Biostatistical analysis*, 2nd ed. ed. Englewood Cliffs, N.J: Prentice-Hall, 1984.

[97] D. M. Levine, D. F. Stephan, T. C. Krehbiel, and M. L. Berenson, *Statistics for managers using Microsoft Excel*, 4th ed. Upper Saddle River, NJ, USA: Pearson Prentice Hall, 2005.

[98] J. F. Pallant, *SPSS survival manual : a step by step guide to data analysis using SPSS for Windows (Version 15)*, 3rd ed. Crows Nest, N.S.W: Allen & Unwin, 2007.

[99] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand, "Experiments of the effectiveness of dataflow- and controlflow-based test adequacy criteria," in *Proceedings of the 16th international conference on software engineering*, 1994: IEEE Computer Society Press, in ICSE '94, pp. 191-200.

[100] Software-artifact Infrastructure Repository, "C Object Biographies: Siemens," Siemens.

[101] D. Hao, L. Zhang, L. Zhang, G. Rothermel, and H. Mei, "A Unified Test Case Prioritization Approach," *ACM transactions on software engineering and methodology*, vol. 24, no. 2, pp. 1-31, 2014.

[102] D. Mondal, H. Hemmati, and S. Durocher, "Exploring Test Suite Diversification and Code Coverage in Multi-Objective Test Case Selection," in *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*, 2015: IEEE, pp. 1-10.

[103] V. Vangala, J. Czerwonka, and P. Talluri, "Test case comparison and clustering using program profiles and static execution," in *Proceedings of the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on the foundations of software engineering*, 2009: ACM, in ESEC/FSE '09, pp. 293-294.

[104] IBM, "Address system-on-chip development challenges with enterprise verification management," IBM, Enterprise Verification Management Solutions White Paper, ftp://public.dhe.ibm.com/software/cn/rational/pdf/Enterprise_verification_management.pdf, Accessed on: March 1, 2019.

[105] 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*, 2014, vol. 16-21-: ACM, in FSE 2014, pp. 235-245.

[106] P. Braione, G. Denaro, O. Riganelli, M. Baluda, and A. Muhammad, "Static/Dynamic Test Case Generation For Software Upgrades via ARC-B and Deltatest," in *Validation of Evolving Software*. Cham: Springer International Publishing, 2015, pp. 147-184.

[107] Y. Noller, "Differential program analysis with fuzzing and symbolic execution," in *Proceedings of the 33rd ACM/IEEE International Conference on automated software engineering*, 2018: ACM, in ASE 2018, pp. 944-947.

[108] B. Danglot, "Automatic Unit Test Amplification For DevOps," 2019.

[109] B. Kurtz, P. Ammann, J. Offutt, M. E. Delamaro, M. Kurtz, and N. Gökçe, "Analyzing the validity of selective mutation with dominator mutants," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on foundations of software engineering*, 2016, vol. 13-18-: ACM, in FSE 2016, pp. 571-582.

[110] T. Y. Chen, T. H. Tse, and Z. Q. Zhou, "Semi-Proving: An Integrated Method for Program Proving, Testing, and Debugging," *IEEE transactions on software engineering*, vol. 37, no. 1, pp. 109-125, 2011.

[111] T. Y. Chen, F.-C. Kuo, D. Towey, and Z. Q. Zhou, "A revisit of three studies related to random testing," *Science China. Information sciences*, vol. 58, no. 5, pp. 45-53, 2015.