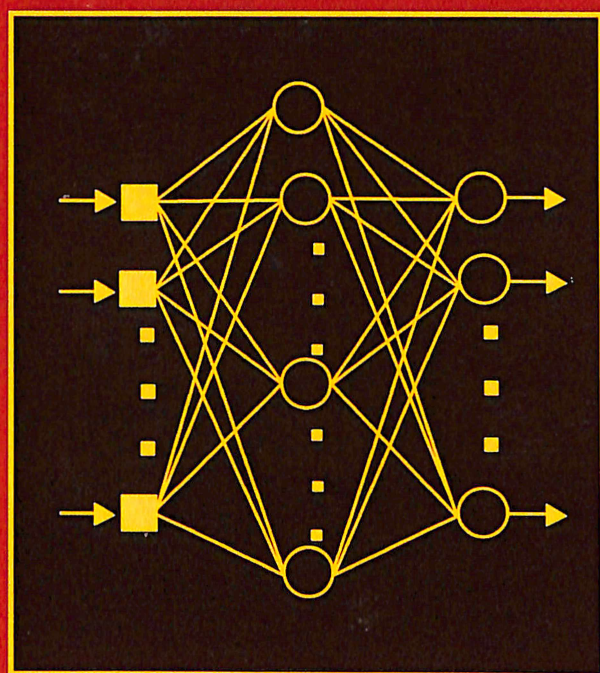
Advances in

COMPUTERS

Volume 113



Edited by

ATIF M. MEMON

Editors

Hurson and Atif M. Memon

CONTENTS

Preface *ix*

1. A Survey on Regression Test-Case Prioritization **1**

Yiling Lou, Junjie Chen, Lingming Zhang, and Dan Hao

1. Introduction 2
2. Framework 5
3. Criterion 9
4. Prioritization Algorithm 12
5. Measurement 17
6. Constraint 20
7. Application Scenario 23
8. Empirical Study 25
9. Some Discussions 28
10. Conclusion 33

Acknowledgments 33
References 33
About the Authors 45

2. Model-Based Test Cases Reuse and Optimization **47**

Mohamed Mussa and Ferhat Khendek

1. Introduction 48
2. Overall MBT Framework 49
3. Integration Test Generation 55
4. Acceptance Test Optimization 67
5. A Case Study: A Library Management System 70
6. Related Work 71
7. Conclusion 75

Acknowledgments 76
Appendix A: Properties of the Integration Test Generation Approach 77
References 84
About the Authors 87

3. Three Open Problems in the Context of E2E Web Testing and a Vision: NEONATE **89**

Filippo Ricca, Maurizio Leotta, and Andrea Stocco

1. Introduction 90
2. The Three Open Problems in the Context of E2E Web Testing 92

3. State of the Art on the Three Open Problems 97
4. Overcoming the Three Open Problems: The NEONATE Vision 104
5. Architecture of the NEONATE Integrated Testing Environment 107
6. NEONATE's Examples of Use 119
7. NEONATE's Long-Term Impact 124
8. Conclusions 126
References 128
About the Authors 132

4. Advances in Using Agile and Lean Processes for Software Development 135

Pilar Rodríguez, Mika Mäntylä, Markku Oivo, Lucy Ellen Lwakatare, Pertti Seppänen, and Pasi Kuvaja

1. Introduction 136
2. Trends on Agile, Lean, and Rapid Software Development 142
3. A Walk Through the Roots of Agile and Lean Thinking 148
4. Agile and Lean in Software Development 158
5. Beyond Agile and Lean: Toward Rapid Software Development, Continuous Delivery, and CD 182
6. DevOps 190
7. The Lean Startup Movement 197
8. Miscellany 202
9. Conclusions and Future Directions 209
References 212
About the Authors 221

5. Advances in Symbolic Execution 225

Guowei Yang, Antonio Filieri, Mateus Borges, Donato Clun, and Junye Wen

1. Introduction 226
2. Background 228
3. Constraint Solving 230
4. Path Explosion 236
5. Compositional Analysis 244
6. Memory Modeling 246
7. Concurrency 250
8. Test Generation 253
9. Security 259
10. Probabilistic Symbolic Execution 264
11. Tools Support 269

12. Conclusion 271
References 271
About the Authors 286

6. Symbolic Execution and Recent Applications to Worst-Case Execution, Load Testing, and Security Analysis 289

Corina S. Păsăreanu, Rody Kersten, Kasper Luckow, and Quoc-Sang Phan

1. Introduction 290
2. Symbolic Execution 290
3. Tools and Scalability Challenges 293
4. Applications 296
5. Conclusion 305

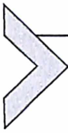
References 305
About the Authors 313

7. Experiences With Replicable Experiments and Replication Kits for Software Engineering Research 315

Steffen Herbold, Fabian Trautsch, Patrick Harms, Verena Herbold, and Jens Grabowski

1. Introduction 316
2. What Is Replication 318
3. Replication Kits 319
4. Experience Reports 320
5. Discussion 330
6. Conclusion 339

References 340
About the Authors 342



CHAPTER ONE

A Survey on Regression Test-Case Prioritization

Yiling Lou*,†, Junjie Chen*,†, Lingming Zhang‡, Dan Hao*,†

*Key Laboratory of High Confidence Software Technologies, Peking University, Ministry of Education, Beijing, China
†Institute of Software, EECS, Peking University, Beijing, China
‡Department of Computer Science, University of Texas at Dallas, Richardson, TX, United States

Contents

1. Introduction 2
2. Framework 5
3. Criterion 9
   3.1 Structural Criterion 9
   3.2 Model-Level Criterion 10
   3.3 Fault-Related Criterion 10
   3.4 Test Input-Based Criterion 11
   3.5 Change Impact-Based Criterion 11
   3.6 Other Criteria 12
4. Prioritization Algorithm 12
   4.1 Greedy Algorithm 13
   4.2 Search-Based Algorithm 14
   4.3 Integrate-Linear-Programming-Based Algorithm 15
   4.4 Information-Retrieval-Based Algorithm 15
   4.5 Machine-Learning-Based Algorithm 16
5. Measurement 17
   5.1 APFD 17
   5.2 $AFPD_C$ 17
   5.3 APXC 18
   5.4 WGFD 19
   5.5 HMFD 19
   5.6 NAPFD and RAPFD 20
6. Constraint 20
   6.1 Time Constraint 20
   6.2 Fault Severity 22
   6.3 Other Constraints 22
7. Application Scenario 23
   7.1 General Test-Case Prioritization 24
   7.2 Version-Specific Test-Case Prioritization 24

Advances in Computers, Volume 113
ISSN 0065-2458
https://doi.org/10.1016/bs.adcom.2018.10.001

© 2019 Elsevier Inc.
All rights reserved.

8. Empirical Study 25
8.1 Studies on Traditional Dynamic Prioritization 25
8.2 Comparison With Traditional Dynamic Techniques 26
9. Some Discussions 28
9.1 Existing Issues 29
9.2 Other Challenging Problems 31
10. Conclusion 33
Acknowledgments 33
References 33
About the Authors 45

Abstract

Regression testing is crucial for ensuring the quality of modern software systems, but can be extremely costly in practice. Test-case prioritization has been proposed to improve the effectiveness of regression testing by scheduling the execution order of test cases to detect regression bugs faster. Since its first proposal, test-case prioritization has been intensively studied in the literature. In this chapter, we perform an extensive survey and analysis on existing test-case prioritization techniques, as well as pointing out future directions for test-case prioritization. More specifically, we collect 191 papers on test-case prioritization from 1997 to 2016 and conduct a detailed survey to systematically investigate these work from six aspects, i.e., algorithms, criteria, measurements, constraints, empirical studies, and scenarios. For each of the six aspects, we discuss the existing work and the trend during the evolution of test-case prioritization. Furthermore, we discuss the current limitations/issues in test-case prioritization research, as well as potential future directions on test-case prioritization. Our analyses provide the evidence that test-case prioritization topic is attracting increasing interests, while the need for practical test-case prioritization tools remains.

1. INTRODUCTION

Modern software systems keep evolving to refine software functionality and maintainability, as well as fixing software flaws. Regression testing has been widely used during software evolution to ensure that software changes do not bring new regression faults. Although crucial, regression testing can be extremely costly [1–3]. In the research literature, it has been reported to consume 80% of the testing cost [4]. Furthermore, modern industry companies also suffer from regression testing cost due to the large number of accumulated test cases during software evolution. For example, Google engineers have witnessed a quadratic increase in their regression testing time, and the number of tests executed each day within Google already exceeds 100 million [5–7].

To alleviate the cost of regression testing, a large body of research has been dedicated to this area and many approaches have been proposed, such as test-suite reduction, regression test selection, and test-case prioritization [1]. Test-suite reduction (also denoted as test-suite minimization) [2, 8–16] aims at reducing the number of test cases by excluding redundant test cases. Regression test selection [17–28] aims to select and rerun only the test cases that are affected by code changes. Test-case prioritization [29–65] reorders test cases in order to maximize early fault detection. Among the three areas, both test-suite reduction and regression test selection exclude some test executions and may suffer from *unsafe* test execution (i.e., missing regression faults). In contrast, test-case prioritization, the target area of this work, simply reorders test executions and does not discard any test case. Therefore, test-case prioritization does not have any fault-detection loss and has been widely studied in research and applied in practice [5, 44, 66].

Test-case prioritization was first proposed in regression testing to deal with the trade-off between what ideal regression testing should do and what is affordable by scheduling the execution order of test cases [67]. However, test-case prioritization is not the focus of that work. Later, Rothermel et al. [29] presented a widely known industrial case to show the necessity of test-case prioritization. Show in that work, the industry case has a product with about 20,000 lines of code consuming 7 weeks on running the entire test suite. Furthermore, that work also proposed various basic test-case prioritization techniques, including the total and additional techniques, which are usually taken as the control techniques in the evaluation of novel test-case prioritization techniques, and still represent state-of-the-art test-case prioritization according to three recent studies [68–70]. These two pieces of work witness the beginning of test-case prioritization, and a large amount of work has been proposed in the following two decades.

Briefly speaking, test-case prioritization aims to schedule the execution order of test cases so as to satisfy some testing requirements. Formally, test-case prioritization is defined as the following process: given any test suite T, test-case prioritization is to find a permutation T' of T satisfying $f(T') \geq f(PT)$, where PT represents any permutation of T and f is a function defined to map permutations of T to real numbers representing the prioritization goal [32]. Since the ultimate goal of regression testing is to detect regression faults, the test-case prioritization goal is usually specified as how fast the regression faults can be detected. That is, test-case prioritization is usually regarded as scheduling test cases to detect more faults earlier.

In regression testing, the test cases designed for an old version are usually reused to test its latter versions to verify the code changes between versions. That is, to reveal faults in the latter versions as early as possible, the reused test cases should be executed in some specified order, which is the aim of test-case prioritization. In other words, regression test-case prioritization (usually abbreviated as RTP) targets at scheduling the execution order of test cases designed for an old version so as to detect faults in its latter versions as early as possible. Besides regression testing, test-case prioritization is also applied to other testing scenarios where test cases are not designed for an old version but for the current version, which is called initial testing [71]. That is, test-case prioritization in initial testing (abbreviated as ITP in this chapter) targets at scheduling the execution order of test cases designed for the current version so as to detect faults in the current version as early as possible. Due to the characteristics of ITP (e.g., does not rely on old version information), its techniques are usually applicable to regression testing, whereas the techniques of the latter may not be applicable for the former.

As the ultimate goal of test-case prioritization, detecting more faults early is usually infeasible, because we can hardly know whether a test case detects faults without running the test case. Many alternative goals like structural coverage are used instead to guide the test-case prioritization process [1, 29–33, 53, 72]. However, due to the inherent difference between alternative goals and the ultimate goal, test-case prioritization becomes more difficult. Furthermore, even taking these alternative goals, test-case prioritization is also an NP-hard problem [73]. Therefore, test-case prioritization suffers from both the effectiveness and the efficiency issues.

To promote the long-term development of the test-case prioritization topic, it is necessary to review and summarize it systematically. However, the existing surveys either summarized this topic at a high level together with other topics (e.g., test-case selection and test-suite reduction) [1], or just reviewed test-case prioritization techniques before 2013 [74, 75]. During the recent years, researchers have still been making obvious achievements on this topic. For example, based on the papers collected for this survey (details shown in Section 2), recent 3 years witness another upsurge in test-case prioritization paper publications due to the popularity of continuous integration. Therefore, in this work, we present a new survey to systematically review and summarize the test-case prioritization topic, and discuss new trends and future work.

2. FRAMEWORK

In this section, we analyze the papers considered in this survey and present the analysis framework of this survey.

To conduct an extensive survey, it is necessary for us to collect a sufficient number of test-case prioritization papers, which represent the past and current status of test-case prioritization. To achieve this goal, we collected representative papers through two steps. First, we used keywords "`test`," "`prioritiz`," and "`prioritis`" to obtain an initial set of related papers. Second, we manually checked the initial set of papers to keep the most representative papers. Finally, we have a set of 191 papers on test-case prioritization in total. To the best of our knowledge, this is the most comprehensive study on test-case prioritization in the literature.

Fig. 1 shows the number of analyzed papers on test-case prioritization from 1997 to 2016. *X*-axis represents the year and *Y*-axis represents the number of papers. From Fig. 1, we observe that the number of test-case prioritization papers overall has a clear increasing trend since the first proposal of test-case prioritization. The reason is that software systems grow larger and larger during the last two decades (e.g., the Debian OS system [76] increased from 55 million LoC to 419 million LoC between 2000 and 2012), and more

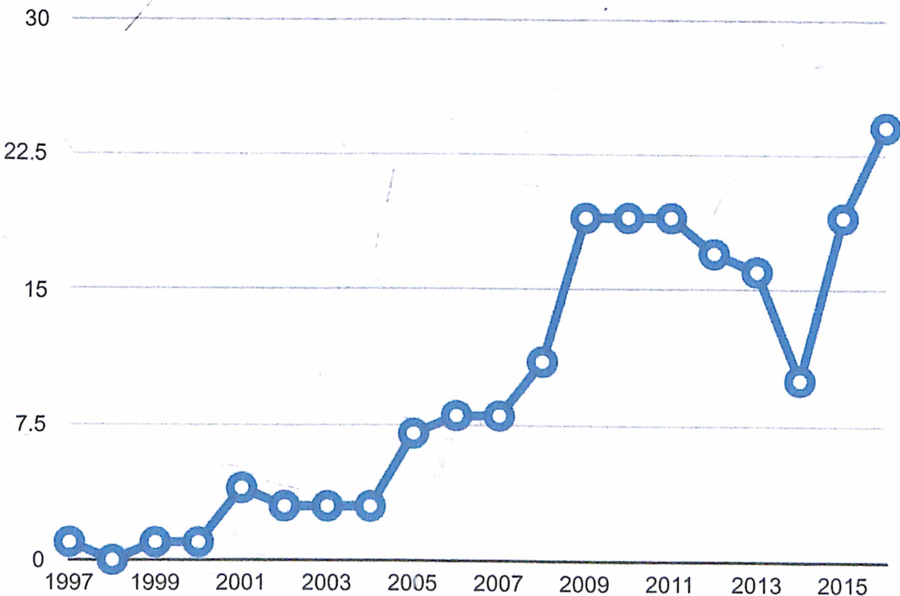


Fig. 1 Number of papers on test-case prioritization from 1997 to 2016.

and more regression test cases are also accumulated during the process, thus stimulating the development of efficient regression testing techniques including test-case prioritization. In addition, we also see several upsurges during the development of test-case prioritization in 2004–2005, 2008–2009, and 2014–2015. We looked into the phenomenon, and found the potential reasons for that. During 2004–2005, the modern distributed version control systems including Git [77] and Mercurial [78] were being proposed. With the advanced version control systems, more and more projects are hosted in code repositories, bringing regression testing techniques to the attention of the developers to test code revisions. During 2008–2009, the GitHub [79] open-source project hosting service (the largest source-code hosting website to date, with 20 million users and 57 million code repositories as of April 2017) was initially released, and the hosted projects usually use regression testing to validate code revisions. Another potential reason for the 2008–2009 resurge is that the financial crisis increased the graduate student population. Finally, we think that the resurge during 2014–2015 may be due to the recent development of mature Continuous Integration (CI) services, such as Travis [80] and Jenkins [81], which extensively use regression testing to provide fast quality feedback.

Following prior work on test-case prioritization [82], we also classify the existing test-case prioritization work according to the following aspects: algorithms, criteria, measurements, scenarios, constraints, and empirical studies. Fig. 2 shows the percentage of papers related to each aspect. Note that some

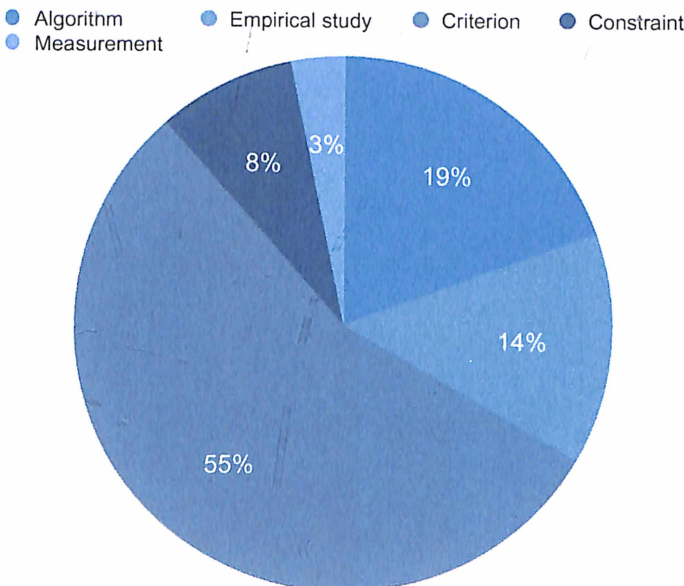


Fig. 2 Ratio of papers of each category.

papers cover multiple of those aspects, thus we categorize each paper based on the main contribution of the work. Also note that this figure does not show the percentage of papers on scenarios, because each test-case prioritization technique has to be evaluated on some specific scenario, such as version-specific test-case prioritization or general test-case prioritization (details shown in Section 7).

According to Fig. 2, more than half of papers focus on investigating criteria for prioritization, followed by the papers proposing prioritization algorithms and the papers on empirical studies. Accessing the fault-detection capability of each test case is always a big challenge and the key for prioritization problem, which is hard to obtain in practice. Fault-detection capability interacts with many other capability such as coverage capability, mutant-killing capability. Thus researchers always keep figuring out many different ways to represent or simulate the fault-detection capability, and plenty of test criteria are newly proposed each year. Since prioritization problem is an NP-hard problem, the algorithm to find the optimal solution among the solution space also matters. Many advanced algorithms in other field can also be adopted to solve the test-case prioritization problem, thus there are also a large number of papers investigating prioritization algorithms. Naturally, due to the large number of test-case prioritization approaches, the comparison between these approaches is also crucial for providing practical guidelines in regression testing, leading to the large number of empirical studies.

To further analyze the trend of each category in test-case prioritization, Fig. 3 further shows the number of papers belonging to each category per year. Consistent with the ratio results of Fig. 2, most of the papers published each year work on investing effective test criteria for test-case prioritization, indicating the researchers' effort in finding optimal test criteria to simulate the fault-detection capabilities of tests a cross the last two decades. Besides, since 2009, prioritization algorithms and empirical studies also attracted increasing attentions, indicating the switch of research interests in this area. We suspect the reason to be as follows. In the initial stage of test-case prioritization (i.e., in 1997), there were not many works in this area, and thus the researchers mainly focused on the core problem of finding suitable surrogates (i.e., various test criteria) for real fault-detection capabilities. Later on, when the test-case prioritization area became more mature since 2009, researchers began to spend more efforts on designing new prioritization algorithms. Meanwhile, due to the large number of emerging papers on test-case prioritization, practitioners often found it hard to find the optimal technique. Therefore, a large body of research has also been dedicated to empirically evaluating and comparing various test-case prioritization techniques.

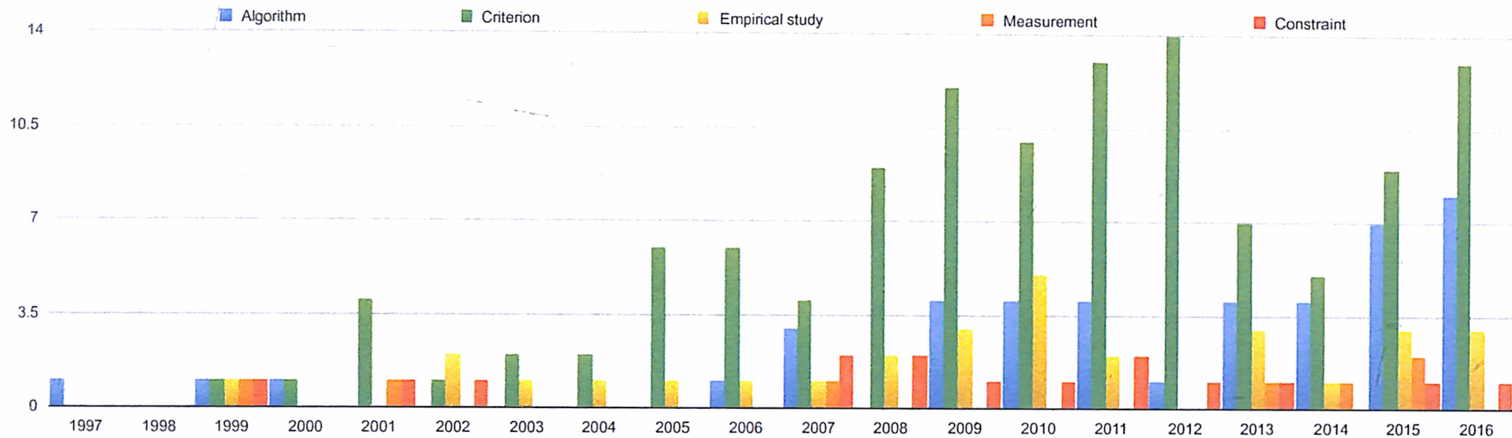
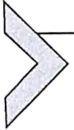


Fig. 3 Number of papers of each category per year.

In this survey, we discuss the development and future directions for each aspect of test-case prioritization in details. The remaining of this chapter is organized as follows. Sections 3–8 review test-case prioritization from the aforementioned six aspects, i.e., coverage criterion, prioritization algorithm, measurement, constraint, scenario, and empirical study. Section 9 discusses the challenges, issues, and future work in test-case prioritization, and finally Section 10 concludes this chapter.



3. CRITERION

Since it is hard to obtain the fault-detection capability of each test case directly in practice, various criteria are proposed to assess test-case fault-detection capability in prioritization. Besides test-case prioritization, criteria are also widely used in test generation, selection, and minimization [8, 17, 83].

Usually, large coverage criterion value means large probability of exposing faults in a program, and thus maximizing criterion values can be an intermediate goal of test-case prioritization. That is, criteria are actually used to guide the prioritization process. For example, branch-coverage-based prioritization [84] schedules the execution order of test cases based on the branch coverage of these test cases. Due to the importance of criteria, many of the existing work [34, 38, 60, 84–135] investigates their influence in the evaluation.

3.1 Structural Criterion

Among all criteria, structural coverage is the mostly used one. In particular, a structural coverage criterion is defined as the percentage of structural units covered by a test case [136–142]. For example, the widely used structural coverage criterion is statement coverage, which measures to what extent a test case covers statements during test-case execution. Higher statement coverage indicates larger fault-detection capability because without covering faulty statements a test case cannot reveal the corresponding faults.

Besides statement coverage [29], some other structural units like functions/methods [30], blocks [67], and modified condition/decision [32] have also been considered as a type of structural coverage criterion.

Interestingly, the experiment results in the work of Rothermel et al. [29, 30] showed that in most cases, branch coverage outperformed statement coverage using a set of C programs. But in more recent work of Lu et al. [68], statement coverage usually performs best among these coverage criteria on a

set of real-world Java programs. One potential reason could be that branches are less prevalent for the object-oriented Java programs than the procedural C programs, making branch coverage ineffective for Java.

3.2 Model-Level Criterion

Though structural criteria are widely used, sometimes, structural coverage can be unavailable for black-box or can be quite expensive to obtain for large systems. System models can capture the different behaviors of a system and there are some modeling languages proposed to model state-based software systems. Recently, model-based techniques have been adopted in software testing, such as test-case generation [143, 144], test-suite reduction [145], and test-case prioritization [117, 146–150]

Korel et al. [147] presented a novel test-case prioritization based on state-based models which execution information of the original and modified models is used for retesting the modified software system. Furthermore, Korel et al. [146] further proposed several model-based test-case prioritization heuristics and empirically investigated the improvements of these heuristics strategies. Xu and Ding [117] proposed an aspect-related test-case prioritization based on the incremental testing paradigm. Aspects are incremental modifications to the base classes, thus the tests targeting the aspects would be selected to execute first for they are more likely to detect the failures.

3.3 Fault-Related Criterion

As testing criteria are usually used to measure the fault-detection capability of a test case or a test suite, some researchers presented some fault-detection criteria directly because the preceding code-based coverage criteria cannot sufficiently assess the capability of a test case or a test suite [151–156].

In particular, Rothermel et al. [84] introduced mutation score to represent each test case's fault-exposing-potential, which regarded mutation-killing-capability as fault-detection-capability. Elbaum et al. [157] used the fault-index to estimate the fault proneness for each program unit, which had been proved effective in previous work [158, 159]. The calculation process of fault-index was as follows: (1), each function was associated with some measurable attributes; (2), all attribute values were standardized according to a group of baseline values; (3), the set would be reduced to a smaller one by principal components analysis [160]; (4), the left values were represented by a linear function which could generate one fault-index for each function in the program.

Ma and Zhao [125] proposed a new prioritization index called testing-importance of module (TIM), which consisted of two factors: fault proneness and importance of module, which acted as a new metric to measure the severe fault proneness for module covered by test cases. Lou et al. [95] seeded mutation on the changed code between versions to imitate the real faults introduced during software evolution. Therefore, the capability of killing these mutants can represent the capability of detecting real faults to some extent.

3.4 Test Input-Based Criterion

Since the structural coverage, model information and mutation analysis can be costly to obtain, recently researchers started to measure the fault-detection capability of test cases based on the input data alone rather than the execution information of test cases. That is, this type of criteria measures the fault-detection capability by calculating the difference between test input data, which are usually regarded as strings or vectors.

In particular, Ledru [161] proposed a prioritization approach which compared the string distance between test cases with a greedy algorithm. Chen et al. [162] proposed a test-vector-based approach to prioritizing test programs for compilers by analyzing the extracted features of test programs to solve the efficiency problem of compiler testing [163]. Recently, Chen et al. [164] proposed to predict the bug-revealing probabilities per unit time of test programs for compilers via machine learning, and schedule the execution order of these test programs based on the descending order of these bug-revealing probabilities per unit time. Chen et al. [89] transformed test cases into a form of vectors for clustering. Jiang et al. [165] proposed a novel family of input-based prioritization techniques, which calculates the difference between test cases by three types of distance functions.

3.5 Change Impact-Based Criterion

Change information are also used very frequently in prioritization criteria [166, 167]. For example, the modified condition mentioned in structural unit level criteria also used the change information during program evolution. However, there is a category of criteria analyzing the change code in a more specific way, so this section introduce these criteria individually.

Haraty et al. [168] proposed a clustering prioritization approach based on code change relevance, which mainly prioritized clusters of test cases based on their relevance to code changes. Alves et al. [169] proposed a

refactoring-based approach to prioritizing tests for detecting refactoring bugs. The approach first collected the change edits between two versions of a program and then analyzed the change impact based on a number of refactoring fault models to determine the execution order of test cases.

Panda et al. [170] presented a static analysis approach to prioritizing test cases based on affected component coupling of object-oriented programs. It first constructed affected slice graph whose nodes had different fault-proneness and then scheduled execution order based on the nodes covered by each test case.

3.6 Other Criteria

Besides, some studies are hard to categorize into aforementioned categories.

3.6.1 *Risk*

Hettiarachchi et al. [171] proposed a risk-based test case prioritization approach, which applied a fuzzy expert system to estimate the risks systematically for requirements and prioritized test cases based on the risks they involved.

3.6.2 *Similarity*

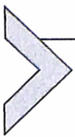
Fang et al. [100] proposed a similarity-based test case prioritization which transformed test case's execution profile into an ordered sequence of program entities and compared distance of the sequence of each test case.

3.6.3 *Service History*

Srikanth et al. [172] prioritized building acceptance test cases based on the service history data from several months, i.e., service interaction and historically failing services.

3.6.4 *Requirement*

Arafeen et al. [173] proposed a test-case prioritization approach which clustered test cases according to the requirement similarities in order to utilize requirements information.



4. PRIORITIZATION ALGORITHM

In this section, we introduce the algorithms used to guide test-case prioritization. Specifically, we classify the existing prioritization algorithms into several groups, i.e., greedy algorithm, search-based algorithm,

information-retrieval-based algorithm, integrate-linear-programming-based algorithm, machine-learning-based algorithm. Moreover, when introducing the prioritization algorithms, we take the statement coverage criterion as the representative, although many following algorithms can be applied to various criteria, e.g., method coverage, branch coverage, and even advanced data-flow coverage criteria [84, 85].

4.1 Greedy Algorithm

Greedy algorithms are widely used to address the test-case prioritization problem, which focus on always selecting the current "best" test case during test-case prioritization. The greedy algorithms can be classified into two groups. The first group aims to select tests covering more statements, whereas the second group aims to select tests that is farthest from the selected tests.

Regarding to the first group, the most popular greedy algorithms are the total and additional algorithms. In particular, the total algorithm prioritizes test cases based on the descendent order of statements covered by each test case, whereas the additional algorithm prioritizes test cases based on the descendent order of statements that are covered by each unselected test case but uncovered by the existing selected test cases. As the total and additional algorithms can have best performance in different cases, Zhang et al. [174, 175] proposed a unified prioritization model, which uses a probabilistic model to bridge the gap between the total and additional algorithms so that the total and additional algorithms can be regarded as its two extreme instances. Moreover, this model yields a spectrum of specific prioritization algorithms between the total and additional algorithms. Besides, Li et al. [72] proposed the 2-optimal strategy which was based on K-optimal algorithm [176] where $K = 2$. Different from approaches mentioned above, the 2-optimal approach tries to select "next two best" test cases according to coverage ability of each pair of test cases.

Regarding to the second group, the typical greedy algorithm is adaptive random test-case prioritization [177], which is proposed based on adaptive random testing [178, 179]. In particular, it first iteratively generates a candidate set of test cases and selects one test case based on a selecting algorithm. The selecting algorithm aims to select a test case that is the farthest from the already selected test cases based on a distance definition function $f1$ and a farthest selection function $f2$. In particular, this work proposed to use Jaccard distance to define $f1$ and defined three types of selection function $f2$.

The greedy algorithms focus on searching the local optimal solution to prioritization, and thus their prioritization results may not be the optimal solution.

4.2 Search-Based Algorithm

Since the prioritization problem is an NP-hard problem, greedy algorithms can not always obtain the optimal solution within the solution space. Therefore, some search-based algorithms are applied to solve the prioritization problem, aiming to achieve better prioritization results with acceptable computation cost.

In particular, Li et al. [72] applied meta-heuristic search-based algorithms to test-case prioritization. That is, they applied steepest ascent hill-climbing and genetic algorithms. In particular, steepest ascent hill climbing is a local search algorithm, where each test permutation is regarded as a state. This algorithm iteratively switches to best state among all neighbors of the current state. The genetic algorithm [72] is based on the processes of natural selection according to Darwinian theory of biological evolution. In this algorithm, each test sequence is encoded in an N-sized array representing an instance of chromosome. In the initial step, a group of test sequences is generated randomly as the initial individuals. Iteratively, a new generation is generated by combining selected individuals guided by the fitness function. The whole search process will be terminated until certain requirement is satisfied.

Besides the traditional single-objective test-case prioritization, there is another form of test-case prioritization problem, called multiobjective test-case prioritization. Given a test suite T, the set of T's permutations PT, and a vector of M objective functions, $f_i(i = 1, 2, \ldots, M)$, multiobjective test-case prioritization aims at finding $T' \subset PT$ such that T' is a Pareto-optimal permutation set with respect to the objective functions, $f_i(i = 1, 2, \ldots, M)$. The objective functions usually are some important prioritization criteria. Pareto-optimal means that strategy A improves strategy B without making things worse.

Epitropakis et al. [180] investigated multiobjective test-case prioritization through three objectives: average percentage of coverage, average percentage of coverage of changed code, and average percentage of past fault coverage and evaluated the fault-detection capabilities in the experiment.

Solving multiobjective problem in software engineering by multiobjective evolutionary algorithms usually faces with the challenge of scalability problem

due to the population size and iterations. Therefore, Li et al. [181] proposed a novel GPU-based parallel fitness evaluation algorithm for test-case prioritization, which implemented the fitness evaluation and crossover computation by graphic processing units on GPU.

Overall, the characteristics of search-based prioritization algorithms lie in searching for the optimal solution guided by the predefined fitness function within the searching space.

4.3 Integrate-Linear-Programming-Based Algorithm

Integrated linear programming (abbreviated as ILP) is a mathematical optimization or feasibility program where all the variables, objective functions, and constraints are linear, which is an NP-hard problem. Recently, researchers applied ILP to describe the problem of test-case prioritization and thus the solutions to the ILP formula are the prioritization results. That is, the problem of test-case prioritization is transformed into formula construction and solving process.

In particular, Zhang et al. [82] firstly applied ILP to solve time-aware test-case prioritization. In particular, this approach first selects a set of test cases by solving the ILP formula describing time-aware test-case prioritization, and then prioritizes selected test cases through some greedy strategies. Recently, to investigate the bound of coverage-based test-case prioritization, Hao et al. [182] used ILP to represent coverage-based test-case prioritization so as to learn the performance of optimal coverage-based test-case prioritization techniques.

4.4 Information-Retrieval-Based Algorithm

Information retrieval (abbreviated as IR) techniques [183] aim to obtain information needed from a collection of information resources, which have been fully studied in the last 40 years and applied to various domains, including software engineering. The main idea of information-retrieval-based algorithm is as follows: (1) it uses test case information such as execution information or source code of each test case to construct the corresponding document collection for each test case, namely, each document represents one test case; (2) it uses source code information (usually the changed part of source code) serving as the input query of IR, and IR will return a ranked list of the documents constructed in the first step, which in fact is a ranked list of test cases by the relevance to the input information.

In particular, Nguyen et al. [184] proposed an IR-based approach to prioritizing test cases for web services, which used the identifier documents extracted from the execution trace to represent each test case and used the web service change description as the input query of IR.

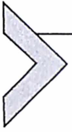
Kwon et al. [185] proposed an IR-based approach which adapted term frequency (TF) and inverse document frequency (IDF) to prioritize test cases. This approach considers not only code coverage information but also how many times a coverage element is executed by a test case (TF) and source code elements are tested by few test cases (IDF). Linear regression model is applied to weigh the value of the information.

Later on, Saha et al. [186] proposed an IR-based approach to prioritize JUnit test cases. Their approach used the test source code to construct the relative document for each test case and used the changed code of the program under test as the input query to get a ranked list of test cases by their relevance to program changes.

4.5 Machine-Learning-Based Algorithm

Machine learning is a data-analysis technique that builds a model from sample input to make prediction for new data. Typically, machine learning techniques consist of supervised learning and unsupervised learning (called clustering as well).

Tonella et al. [133] presented a machine-learning-based test-case prioritization approach which incorporated user knowledge by case-based ranking model. This approach used the indicator of priority, which was defined by user cases, and test case information such as coverage and fault proneness metrics as features to train a model to predict the priority of test cases. Chen et al. [162] proposed a test-vector based approach to prioritizing test programs for compilers, which did not need to collect coverage information but only analyze necessary features from each test program itself to prioritize test programs for compilers. More recently, Chen et al. [164] developed LET (short for learning to test), which learned from existing test programs to accelerate future test execution. LET first designed and extracted a lot of features from the source code of test programs (e.g., address features and pointer comparison features). Then, LET trained a capability model to predict the bug-revealing probability of each new test program, and a time model to predict the execution time of each new test program, based on these features. Finally, LET prioritized new test programs as the descending order of their bug-revealing probabilities in unit time.



5. MEASUREMENT

To access the performance of test-case prioritization techniques, it is necessary to propose a measurement for test-case prioritization, including efficiency and effectiveness.

With regard to the efficiency of test-case prioritization, researchers usually use the complexity analysis of a prioritization algorithm to measure its cost. For example, Elbaum et al. [157] analyzed that the complexity of the statement-coverage-based total prioritization technique is $O(mn + mlogm)$ and the complexity of the statement-coverage-based additional prioritization technique is $O(m^2n)$, where m represents the number of test cases and n represents the number of statements in a program.

With regard to the effectiveness of test-case prioritization, most of the existing work uses the average of percentage of faults detected (abbreviated as APFD). Besides, as this measurement suffers from the widely known problems, e.g., ignoring the impact of testing time and fault severities, many researchers further improved this measurement accordingly. In the following, we briefly introduce the measurements used in test-case prioritization.

5.1 APFD

Rothermel et al. [29] proposed the first measurement for assessing the effectiveness of test-case prioritization, which is called weighted average of the percentage of faults detected (APFD). APFD measures how rapidly a prioritized test suite detects faults. Higher APFD values mean faster fault-detection rates. Formula (1) shows how to calculate APFD values for a test-case prioritization technique. In this formula, TF_j refers to the first test case in prioritized test suite that detects the jth fault, n refers to the number of test cases, and m refers to the number of faults detected by the test suite. APFD has already become one of the most widely used measurements for assessing the performance of test-case prioritization in the literature [71].

$$APFD = 1 - \frac{\sum_{j=1}^{m} TF_j}{nm} + \frac{1}{2n} \tag{1}$$

5.2 AFPD$_C$

Actually, APFD does not reflect the practical performance of test-case prioritization, since it ignores the influence of test execution costs and fault

severity. Therefore, Elbaum et al. [33] further proposed another measurement to measure the practical performance of test-case prioritization by considering the influence of the two factors, which is called cost-cognizant weighted average percentage of faults detected ($APFD_C$). $APFD_C$ is actually adapted from APFD, which is defined as Formula (2). In this formula, f_i refers to the severity of the ith fault detected by the prioritized test suite, and t_j refers to the test cost of the jth test case in the prioritized test suite.

$$APFD_C = \frac{\sum_{j=1}^{m}\left(f_i * \left(\sum_{i=TF_j}^{n} t_i - \frac{1}{2} * t_{TF_j}\right)\right)}{\sum_{j=1}^{n} t_j * \sum_{j=1}^{m} f_j} \quad (2)$$

In practice, it tends to be quite difficult to know the severity of each fault in advance. Therefore, a simplified $APFD_C$ is usually used to measure the performance of test-case prioritization by treating all faults as sharing the same severity [180]. The simplified $APFD_C$ is shown as Formula (3).

$$APFD_C(\text{simplified}) = \frac{\sum_{j=1}^{m}\left(\sum_{i=TF_j}^{n} t_i - \frac{1}{2} * t_{TF_j}\right)}{\sum_{j=1}^{n} t_j * m} \quad (3)$$

5.3 APXC

In order to measure the performance of test-case prioritization before test-case execution, researchers [72, 182] also proposed to leverage the average percentage of some structural coverage (abbreviated as APXC) as a measurement. APXC has the similar formula with APFD. For APXC, TF_j in Formula (1) refers to the first test case in prioritized test suite that covers structural units (e.g., statement and block) j, and m refers to the total number of structural units covered by the test suite. In particular, higher APXC values mean faster coverage rates.

According to the general definition of APXC, we may have APBC to measure the rate at which a prioritized test suite covers the blocks, APSC to measure the rate at which a prioritized test suite covers the statements. Such measurements are defined based on structural units, which are not

the ultimate goal. Therefore, they are actually widely used as an intermediate goal (e.g., fitness function) during search-based test-case prioritization to guide test-case prioritization, rather than as a measurement for the performance of test-case prioritization.

5.4 WGFD

Higher APFD values mean faster fault detection. However, the problem is how to define "fastness." In different testing scenarios, "fastness" tends to have different definitions. Therefore, Lv et al. [187] proposed a new generalized measurement from a control theory viewpoint, which is called the weighted gain of faults detected (WGFD). The basic idea is to weight and sum fault-detection rates of different test cases so as to define "fastness" in different testing scenarios. That is, since the number of test cases detected at different time should have different impact on measuring the performance (fastness) of a prioritization technique, different weights should be assigned to the fault-detection rates of different test cases. In particular, WGFD is defined in Formula (4), where n refers to the number of test cases in the test suite, $r(i)$ refers to the fault-detection rate of test case i, and $w(i)$ refers to the assigned weight to the fault-detection rate of test case i.

$$\mathrm{WGFD} = \sum_{i=1}^{n} w(i) * r(i) \tag{4}$$

5.5 HMFD

According Formula (1), the APFD measure increases as the size of the test suite increases. In other words, APFD is affected by the size of a given test suite. To relieve this issue of APFD, Zhai et al. [99] proposed a new measurement to measure how quickly a prioritized test suite can detect faults, which is independent from the size of a given test suite. The new measurement is called the harmonic mean of the rate of fault detection (HMFD). In particular, HMFD is defined as Formula (5), where TF_j refers to the first test case in the prioritized test suite that detects the ith fault, and m is the number of faults detected by the test suite. Note that low HMFD values mean better performance of test-case prioritization.

$$\mathrm{HMFD} = \frac{m}{\sum_{j=1}^{m} \frac{1}{TF_j}} \tag{5}$$

5.6 NAPFD and RAPFD

In practice, there may be various constraints in test-case prioritization. Due to the existence of practical constraints in test-case prioritization, not all of the faults can be detected by a given test suite. Moreover, we may not execute the same number of test cases. Walcott et al. [188] proposed to assign a penalty to the missing faults so as to solve the first problem. In addition, Qu et al. [189] proposed normalized APFD (abbreviated as NAPFD) to measure the performance of test-case prioritization in order to solve the two problems. In particular, NAPFD is defined as Formula (6), where p refers to the value that is calculated by dividing the number of faults detected by the prioritized test suite by the number of faults detected by the full test suite. To further improve these measurements, Wang and Chen [190] proposed the relative average percent of faults detected (RAPFD) by considering the given testing resource constraint, which determines how many test cases could be run. Furthermore, Do and Rothermel [191, 192] further proposed many improved cost–benefit models for assessing regression testing methodologies (including test-case prioritization). In particular, these models incorporate context factors (e.g., the costs of some essential testing activities such as test setup and obsolete test identification) and lifecycle factors (e.g., the costs and benefits for techniques across system lifetimes).

$$\mathrm{NAPFD} = p - \frac{\sum_{j=1}^{m} TF_j}{nm} + \frac{p}{2n} \tag{6}$$

6. CONSTRAINT

As a practical problem, test-case prioritization tends to suffer from various practical constraints. Therefore, many studies investigated how to prioritize test cases when considering practical constraints [88, 121, 127, 188, 193].

6.1 Time Constraint

The mostly studied constraint in test-case prioritization is the time constraint, also called time budget [188]. Ideally, all the test cases in the prioritized test suite are expected to be executed during the process of software testing, so as to avoid fault-detection capability loss of the test suite. However, under the practical environment of software testing, the allowed testing time may not be quite sufficient, which causes that the prioritized test suite may not

be totally executed. For example, in some companies, software testing is just allowed in night [188, 194], and thus if the time of executing the whole test suite is more than one night, some prioritized test cases will not be executed. Besides, new software development processes, e.g., extreme programming, also advocate a short testing cycle. Therefore, on this occasion, the time constraint is quite necessary to be considered when prioritizing test cases.

To make test-case prioritization more effective given the allowed testing time, various approaches have been proposed to select only a subset of test cases and schedule their execution order rather than all the test cases. Walcott et al. [188] proposed time-aware test-case prioritization. More specifically, they used a genetic algorithm to prioritize test cases in order to achieve two goals. The first goal is to ensure that the prioritized test cases can be executed within the given testing time. The second goal is to make the prioritized test cases achieve the largest fault-detection capability. To achieve the same goals, Alspaugh et al. [195] proposed to use 0/1 knapsack solvers to prioritize test cases, including greedy, dynamic programming, and the core algorithms. Zhang et al. [82] identified that time-aware test-case prioritization implied to select a subset of test cases from the test suite for prioritization. Therefore, they proposed to combine test-case selection and test-case prioritization to achieve the goals of time-aware test-case prioritization. More specifically, they first used integer linear programming [196] to select a subset of test cases that can achieve the maximum test coverage within the time budget, and then applied traditional test-case prioritization techniques to schedule the execution order of the selected test cases. Note that, in this way, the traditional total technique and the traditional additional technique are both adapted to be time-aware total technique and time-aware additional technique. Later on, Suri et al. [197] also applied ant colony optimization to prioritize test cases in the time constraint environment.

Based on the existing research [188, 191], considering the time constraint in test-case prioritization may influence the costs and benefits of test-case prioritization techniques. Do et al. [198] conducted a series of experiments to investigate such influence. Their experimental results demonstrated that the time constraint indeed has a significant influence on the cost-effectiveness of test-case prioritization techniques. Furthermore, You et al. [199] conducted an empirical study to investigate whether the time cost of each test case influences the effectiveness of time-aware test-case prioritization. Their experimental results showed that the effectiveness of the prioritization techniques considering the time cost of each test case has no significant difference with that of the prioritization techniques omitting the time cost of each test case.

That is, it tends to be not worth considering the time cost of each test case for time-aware test-case prioritization. In addition, Marijan [38] proposed a framework for optimal test-case prioritization in the time constraint environment by integrating three different perspectives, including business perspective, performance perspective, and test design perspective. More specifically, from a business perspective, failure impact is regarded as an important factor influencing test effectiveness; from a performance perspective, test execution time is regarded as an obvious factor of test effectiveness; from a technical perspective, both failure frequency and cross-functionality are regarded as important factors of test effectiveness. In particular, failure frequency refers to a measure of how often test cases detect failures, and cross-functionality refers to a measure of how much the functionality of the system under test is covered by a test case.

6.2 Fault Severity

Another widely studied constraint in test-case prioritization is fault severity. The fault severity reflects the costs or resources required if a fault persists in and influences the users/organization/developers. The existing test-case prioritization is based on the assumption that the severity of all the faults are considered equally. However, the assumption may not hold in practice, and thus the fault severity is also a practical constraint for test-case prioritization.

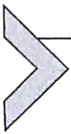
Elbaum et al. [33] firstly considered the fault severity constraint when measuring the effectiveness of test-case prioritization techniques. Park et al. [200] proposed to prioritize test cases by considering fault severity. In particular, they estimate the current fault severity using history information. Actually, their approach has an assumption, i.e., test costs and fault severities are not largely changed from one version to a later version. Malishevsky et al. [201] adapted traditional test-case prioritization (e.g., the total technique and the additional technique) to cost-cognizant test-case prioritization by considering the fault severity constraint and the time cost of each test case. Huang et al. [108] also proposed a history-based cost-cognizant test-case prioritization. More specifically, their approach collected the historical records from the latest regression testing and then used a genetic algorithm to schedule the most effective execution order of test cases.

6.3 Other Constraints

Besides, resource (e.g., hardware resource) is also a constraint in test-case prioritization. Kim and Porter [193] proposed a test-case prioritization based

on history information by considering the resource constraint and time constraint. That is, they assigned a selection probability for each test case based on history information, and selected a test case to run based on these probabilities until testing time is exhausted. More specifically, their utilized history information contains the execution history of each test case, the corresponding fault detection, and/or the covered program entities. Wang et al. [88] proposed a resource-aware multiobjective optimization solution to produce an optimal execution order of test cases by considering the resource constraint and the time constraint. In the multiobjective optimization solution, they defined a fitness function based on four cost-effectiveness measures, including (1) minimizing the time for executing prioritized test cases and allocating relevant test resources; (2) maximizing the number of test cases to be executed; (3) maximizing the usage of available test resources; and (4) maximizing fault-detection achieved by prioritized test cases.

Furthermore, there are some other constraints, e.g., testing requirement priorities and the request quotas of web service. To prioritize test cases by considering testing requirement priorities, Zhang et al. [127] proposed to utilize test history information to evaluate the priorities of test cases so as to prioritize test cases based on them. Here various types of code elements can be regarded as testing requirements, e.g., statements, basic blocks, methods; or features and attributions of system; or faults in system. About the constraint of the request quotas of web service (e.g., the upper limit of the number of requests that a user can send to a Web Service during a certain time range), Hou et al. [121] proposed quota-constrained test-case prioritization for service-centric systems by maximize testing requirement coverage. More specifically, they first divided the testing time into time slots, and then selected and prioritized test cases for each slot by using integer linear programming.



7. APPLICATION SCENARIO

Test-case prioritization aims to speed up fault detection for the new software version during software evolution. Balancing the overhead and effectiveness, two different application scenarios have been explored—(1) *general* test-case prioritization and (2) *version-specific* test-case prioritization.

General test-case prioritization techniques [68–70, 84, 202] usually compute the optimal test order once for one revision, and then reuse that test order for a number of subsequent revisions. On the contrary, version-specific test-case prioritization techniques [66, 95, 146, 186] compute the

optimal test order right before each revision in order to achieve effective test-case prioritization. While version-specific test-case prioritization may achieve more precise results, it may incur higher overhead due to the frequent test-case prioritization runs. In this section, we discuss the details for such two application scenarios.

7.1 General Test-Case Prioritization

Given a program P and its corresponding test suite T, general test-case prioritization [68–70, 84, 202] computes test execution order valid for a number of subsequent modified revisions of P. Therefore, they are usually based on general program/test information shared by various revisions, e.g., the set of program elements covered by each test.

For example, if test t_1 covers more program elements than t_2 on one program revision, the same may still hold for later program revisions. Therefore, traditional test-case prioritization techniques based on coverage information, e.g., the *total/additional* [84, 157], *adaptive-random-testing-based* [177], and *search-based* techniques [72], can all be directly utilized for general test-case prioritization.

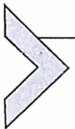
When prioritizing using coverage information obtained from historical revisions, software changes and test additions could make test-case prioritization techniques ineffective since coverage information can be obsolete (due to software changes) or absent (for newly added tests) during the software evolution. To study the impacts of software changes and test additions for general test-case prioritization, Lu et al. [68] recently performed a study on real-world evolving GitHub projects. The study results demonstrate that software changes do not impact general test-case prioritization much, whereas test additions, which incur tests without coverage information, may significantly impact the effectiveness of general test-case prioritization. The study provides practical guidelines for determining the intervals of applying general test-case prioritization—general test-case prioritization should be reapplied whenever there are nontrivial number of added tests.

7.2 Version-Specific Test-Case Prioritization

Given a program P and its corresponding test suite T, version-specific test-case prioritization [66, 95, 146, 186] computes optimal test execution orderings specifically for P', the next revision of P. Version-specific test-case prioritization is performed after changes have been made to P and prior to regression testing of P'. The prioritized test suite may be more effective for

testing P' than that computed by general test-case prioritization, but may be inferior on average on a succession of subsequent releases of P.

In the literature, researchers have also applied traditional coverage-based test-case prioritization techniques to the version-specific scenario. Furthermore, since regression faults are mainly due to software changes, researchers have also proposed various version-specific test-case prioritization techniques [66, 95, 146, 186] based on the detailed change information during software revision for more effective test-case prioritization. For example, Srivastava and Thiagarajan [66] analyzed the binary-level basic block changes to execute tests covering more changes earlier for faster regression fault detection. Korel et al. [146] analyzed the system models and computed model-level modifications for precise version-specific test-case prioritization. Lou et al. [95] presented a mutation-based version-specific test-case prioritization technique, which simulates faults occurred in software evolution by mutants on the change and prioritizes test cases based on their killing information on these simulation faults. Recently, Saha et al. [186] transformed the version-specific test-case prioritization problem into an information retrieval problem by treating source-code level changes as queries and test-case source code as documents. Then, the tests with more textual similarities with software changes are executed earlier to detect regression bugs faster.



8. EMPIRICAL STUDY

Due to the large number of existing test-case prioritization techniques, it can be hard to make the right/optimal choices in practice. Therefore, researchers have also performed various studies on test-case prioritization techniques to provide practical guidelines for test-case prioritization.

8.1 Studies on Traditional Dynamic Prioritization

Due to the dominant position of traditional dynamic test-case prioritization techniques, the vast majority of studies explore various factors around these techniques.

Rothermel et al. [84] empirically compared various dynamic test-case prioritization techniques (including coverage-based and mutation-based techniques) against unordered or randomized test suites on a suite of C programs. Later on, Elbaum et al. [33] further studied the impacts of fault severities and test execution time on test-case prioritization. Elbaum et al. [157] also investigated the impacts of program versions, program types, and different coverage granularities on test-case prioritization on C programs. Do et al. [203]

performed the first study of test-case prioritization on JUnit tests for Java programs. The study demonstrated the effectiveness of dynamic test-case prioritization on Java programs besides C programs, and also revealed divergent behaviors of test-case prioritization on Java and C programs. Do et al. [198] also investigated the effect of time constraints on the cost-effectiveness of test-case prioritization, as well as demonstrating the validity of using mutation faults for test-case prioritization experiments [156, 204]. Recently, Lu et al. [68] investigated the impacts of real-world software evolution on test-case prioritization and found that code changes do not impact the effectiveness of test-case prioritization much while test additions can significantly lower the effectiveness of traditional dynamic test-case prioritization.

In terms of effectiveness, various studies have confirmed that the traditional additional [84] and search-based [72] test-case prioritization techniques represent the state of the art [68, 72, 174, 177].

8.2 Comparison With Traditional Dynamic Techniques

Besides the traditional dynamic test-case prioritization techniques, researchers have also proposed various other test-case prioritization techniques. In the next, we present two recent but important studies comparing traditional dynamic test-case prioritization with other static or black-box techniques.

8.2.1 Dynamic vs Static

Traditional dynamic test-case prioritization techniques [72, 84, 177] mainly reply on dynamic execution information (e.g., statement or method coverage) to prioritize tests. Although effective, they may not be suitable for all the cases. For some software systems, it may not be possible to collect dynamic execution information via code instrumentation, e.g., code instrumentation may interrupt normal test run for real-time systems. For some software systems, dynamic execution information may not be always precise, e.g., code with concurrency and randomness. Even it is possible to collect precise dynamic execution for some software systems, dynamic instrumentation may incur high overhead, e.g., even the coarse file/class-level dynamic information may incur 8× slowdown for `commons-math` [205]. Finally, the dynamic execution information may not always be available on the old version [137, 206]. Therefore, Zhang et al. [137] firstly proposed to use static analysis to simulate the dynamic execution information. More specifically, they used the static call graph information of each test to simulate the method-level coverage of the test, since the static call graph is always a superset of the actual method cover Later on, Mei et al. [206] further extended the call-graph-based test

prioritization techniques via considering the method body information. Ledru et al. [161] directly treated each test (e.g., test source code or test input) as a string and prioritized tests to maximum string distances of the executed tests. The main insight is that executing more diverse tests may have higher probability to detect unknown regression bugs. Thomas et al. [148] found that simply treating each test as a string may include useless terms while missing important latent terms of the test. Therefore, they proposed to further use topic model to infer the latent semantic representation of each test. Then, they computed the string distances between test semantic representations, and prioritized tests to execute more diverse tests.

Although various static test-case prioritization techniques have been proposed, there lack extensive studies comparing different static techniques as well as comparing static techniques against dynamic techniques. For example, the call-graph-based techniques [137, 206] were not compared against other static techniques since there were no other static techniques before, while the more recent topic-model-based technique was only evaluated using only two subject systems. Therefore, recently, Luo et al. [70] performed an extensive study on state-of-the-art static and dynamic test-case prioritization techniques using 30 modern real-world GitHub projects. The study results show that the call-graph-based techniques outperform all the studied dynamic and static techniques at the test-class level, while the topic-model-based technique performs better than other static techniques but worse than two dynamic techniques at the test-method level. The call-graph-based techniques have also been shown to incur the lowest prioritization overhead among all the static techniques. Overall, while almost all techniques perform better at the test-method level, the static techniques perform comparatively worse to dynamic techniques at the test method level as opposed to the test class level. Finally, the study results show that there is minimal overlap between the detected faults by the static and dynamic techniques, e.g., top 10% prioritized tests only share less than 30% of detected faults, indicating a promising future for applying static and dynamic test-case prioritization in tandem.

8.2.2 *Block-Box vs White-Box*

Since the first proposal of test-case prioritization two decades ago [29, 67], white-box test-case prioritization techniques have been intensively studied. Such white-box techniques rely on the source code or dynamic execution information (obtained via code instrumentation) of the program under test to perform effective test-case prioritization. However, such techniques may not be applicable when the program source code and dynamic execution

information are not accessible or available. Furthermore, white-box techniques can be expensive due to the collection of dynamic execution information [137, 206]. Therefore, researchers have also proposed black-box test-case prioritization techniques which do not require accessing source code or performing code instrumentation. Bryce and Colbourn [130, 134] proposed the first black-box test-case prioritization technique inspired by combinatorial interaction testing (CIT). Based on the test input information, they adopted a "one-test-at-a-time" greedy approach to prioritize test cases to achieve high pair-wise interactions of the test inputs faster. Bryce et al. [115, 207] later used *t*-wise interaction from CIT to prioritize test cases for GUI applications. Qu et al. [208, 209] also used the notion of CIT to prioritize tests for the highly configurable software systems (e.g., software product lines). Henard et al. [210] recently proposed a search-based technique to prioritize the configurations for testing highly configurable software systems based on CIT.

Due to the presence of various black-box and white-box test-case prioritization techniques, it can be hard for the developers or testers to choose the right technique. Therefore, recently, Henard et al. [69] systematically studied and compared the existing white-box and black-box test-case prioritization techniques. They studied 20 state-of-the-art test-case prioritization techniques, including 10 white-box techniques and 10 black-box techniques. The study was performed on six real-world C programs, widely used in prior work on test-case prioritization. The study results reveal a number of practical guidelines. First, the CIT and diversity-based techniques perform the best among all studied black-box test-case prioritization techniques. Second, although white-box techniques outperform black-box techniques for the majority of the cases, surprisingly, the performance (in terms of APFD) difference between white-box and black-box techniques is negligible, e.g., at most 4% APFD difference. Third, the overlap between the faults detected by the black-box and white-box techniques tend to be high: the first 10% prioritized tests agree on over 60% of the detected faults. Overall, the study provides practical guidelines that the developers or testers who may not have source code information available can use black-box test-case prioritization as a reliable substitute of white-box test-case prioritization.

9. SOME DISCUSSIONS

In this section, we first discuss existing issues in test-case prioritization following the previous classification and then point out some other challenge problems and potential future work in test-case prioritization.

9.1 Existing Issues

In this section, we discuss the existing issues in test-case prioritization through three aspects—criteria, measurements, and empirical studies.

9.1.1 Criteria

Testing criteria are used to guide the selection of test cases in test-case prioritization. Most of the widely used testing criteria can be classified into two categories, structural coverage-based criteria and mutation-based criteria. However, these two types of testing criteria are either less precise or costly. In particular, the structural coverage criteria (e.g., statement coverage or branch coverage) actually measure the percentage of code elements (e.g., statements or branches) covered by a test case or a test suite. That is, these coverage criteria measure the effectiveness of only test input, ignoring test oracle [211, 212]. Therefore, such a type of criteria is less precise. On the contradictory, mutation-based criteria tend to measure the effectiveness of a test case or a test suite based on the output of the program. Therefore, mutation-based criteria consider both test input and test oracle, which seem to have higher precision than coverage-based criteria. However, mutation testing suffers from the widely known cost issue. To sum up, neither structural coverage-based criteria nor mutation-based criteria are good enough serving as testing criteria, and thus another precise but less costly testing criterion is needed. Recently, Zhang et al. [213] proposed predictive mutation testing (PMT). The approach built predictive models based on a series of lightweight features related to mutants and tests, and predicts mutant execution results without executing the mutants. It greatly reduces the cost of mutation testing while incurring only minor loss of accuracy, which may provide effective but efficient supports for future test-case prioritization.

9.1.2 Measurement

First, the effectiveness measurement taken by the existing work has obvious flaws. In the past, most of the existing work evaluated test-case prioritization techniques based on APFD [84]. However, APFD assumes that all the tests have the same execution time and treats them equivalently, which is usually not true in practice. For example, for project MapDB [214], the test with the longest running time spends $8.8*10^5$X more time than that with the shortest running time. To address this measurement issue, Elbaum et al. [33] proposed a cost-cognizant version of APFD, $APFD_C$, which considers different test costs and fault severities. Since fault severities can be hard to determine in practice, Epitropakis et al. [180] simplified this measurement by assuming all

faults have the same severity. We encourage researchers to evaluate future test-case prioritization work using $APFD_C$ or simplified $APFD_C$ to explicitly consider test execution time. Meanwhile, $APFD_C$ may also not be suitable for all cases, since its values are influenced by various factors like the number of tests, the number of faults. Therefore, it is hard to use the values of such measurements to explain the effectiveness of a prioritization technique in different cases. To illustrate, we can hardly tell whether a prioritization technique whose $APFD_C$ value is 0.7800 is good or not for a particular test suite. Furthermore, such measurements do not explicitly consider the actual switching costs between test executions (e.g., time to load and schedule the next test). In the future, we suggest researchers to also consider measuring test-case prioritization techniques based on the actual time spent on fault detection, e.g., TTFF (time to detect the first fault) and TTLF (time to detect the last fault), since such measurements precisely measure the actual time cost during regression testing.

Second, the efficiency measurement is mostly ignored in test-case prioritization, although its results influence the usage of test-case prioritization techniques. In the past, the efficiency of test-case prioritization is mostly evaluated through complexity analysis rather than the actual prioritization time. However, the complexity of some prioritization algorithms (e.g., genetic algorithm [72]) can be hard to estimate. Furthermore, although the time complexity of some algorithms (e.g., integer linear programming-based algorithm [82]) is large, their actual prioritization time may be acceptable since the test-case prioritization process is usually performed *offline* beforehand, i.e., before the new version is ready. On the other hand, the efficiency of test-case prioritization can also be crucial for some cases (e.g., version-specific test-case prioritization). In such cases, test-case prioritization is usually performed *online* (e.g., after the new version is ready), making it unbearable when the prioritization time is close to the time spent on test-case execution. Therefore, it is necessary to study the end-to-end testing time (i.e., including the prioritization time and the test execution time) for the online test-case prioritization techniques.

Finally, besides the prioritization cost, it is also important to measure the cost on collecting the necessary data required by test-case prioritization techniques. Most prioritization techniques require extra information besides test cases (e.g., structural coverage) for test-case prioritization. Apparently, obtaining such information may occur extra cost. However, many studies simply take the information as given and do not report the collection cost. In particular, some prioritization techniques require structural coverage

[84], static coverage [206], or mutation execution information on some early version [95]. Although such information is usually collected offline, i.e., before test-case prioritization, it still consumes computing resources and should be measured to provide practical guidelines.

9.1.3 Empirical Studies

In the literature, existing empirical studies investigated the various factors (e.g., programming languages [203], coverage granularity and type [203, 206], fault type [156, 204], test granularity [95, 206], and constraints [192]) that may influence the effectiveness and efficiency of test-case prioritization. Besides these factors, it is also important to investigate the following (but not limited to) factors.

Some experimental factors have been recognized as threats in the past, e.g., subjects, faults, and test cases, but they are seldom studied. For example, subjects are a widely recognized external factor, but the early work of test-case prioritization (especially the papers published around 2000) mostly used the seven small programs (whose number of lines of code is smaller than 600) in Siemens as the subjects. Fortunately, this threat is reduced to some extent after 2000, because researchers started to use larger projects, e.g., *grep* and *gzip* whose number of lines of code is about 10,000. Furthermore, most prior work uses mutation faults or seeded faults, which may be a nonnegligible threat, since there might be some gap between mutants and real faults during software evolution. In other words, we suggest considering using real regression faults in test-case prioritization.

Besides these well-recognized threats, researchers started to notice the difference between practice and existing experimental setup of test-case prioritization. For example, recently Lu et al. [68] identified another one important flawed setting in the existing evaluation, evolution of source code and test cases. That is, previous work on test-case prioritization is usually evaluated based on the source code and test cases with artificial changes simulated via mutation testing, which do not represent real software evolution. Lu et al. [68] investigated the influence of this factor on the effectiveness of many existing general prioritization techniques, and found that changes on source code do not have much influence on the effectiveness of test-case prioritization, but changes on test code (e.g., test additions) do have.

9.2 Other Challenging Problems

Besides these issues in the current work, test-case prioritization, test-case prioritization also suffers from other challenging problems.

9.2.1 Intermediate/Ultimate Goal

Test-case prioritization has been studied for long, and a large number of prioritization techniques have been proposed and investigated in the literature. However, most of the prioritization techniques are less effective than the simple greedy algorithm, such as the additional algorithm, resulting from the difference between the ultimate goal and the intermediate goal of test-case prioritization. In particular, as the ultimate goal of test-case prioritization can hardly serve to guide prioritization, existing prioritization techniques actually use an intermediate goal instead, and thus these "well-designed" prioritization techniques do not optimize the execution order of test cases in terms of the ultimate goal. In recent years, researchers in test-case prioritization started to notice this fact [72] and investigated this fact [182]. Unfortunately, no work in the literature actually solves this problem, and it becomes a fundamental challenge for test-case prioritization. In the future, researchers should investigate other intermediate goals (e.g., detection of mutation faults or detection of similar real faults), which have closer relationship with the ultimate goal rather than the existing intermediate goals (e.g., structural coverage).

9.2.2 Practical Values

Test-case prioritization is a practical problem raised from industry, and thus it is important to study test-case prioritization in practice.

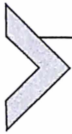
Test-case prioritization aims to facilitate fault detection in software testing, and thus it brings more benefits when the time spent on test-case execution is not ignorable (e.g., several days or months). In other words, when the total execution time of all test cases is small (e.g., several minutes), it does not matter so much whether a fault is detected by the first test case or the last test case. However, to our knowledge, most of the existing research work is actually evaluated on the subjects whose total execution time of test cases is not large at all. That is, the existing techniques are not evaluated in its most possible application scenario. In other word, to facilitate practical usage, it is necessary to investigate test case prioritization in a proper practical scenario.

Besides, test-case prioritization may have variants besides its default setting. Traditionally, test-case prioritization aims to address the test effectiveness problem when the total execution time of test cases are long. However, in practice, it may be costly to run an individual test case. In particular, a test suite may consist of only several test cases, each of which consumes long execution time. Therefore, it is also interesting to study how to optimize the execution of an individual test case, e.g., transferring a test case with long execution time to several test cases with short execution time by modifying its components

(e.g., test input data). Apparently, this problem is different from the existing prioritization problem, and thus a totally new method for this problem is needed.

Furthermore, surprisingly, to the best of our knowledge, although test-case prioritization techniques have been studied for decades, there still lack practical test-case prioritization tools that are effective and easy to use. For example, despite the large number of papers on JUnit test-case prioritization, there is no practical test prioritization technique fully integrated with JUnit. To demonstrate the practical value of test-case prioritization, we encourage the researchers to provide practical tool supports on test-case prioritization in the near future.

In summary, although test-case prioritization has been studied for decades, it is yet not fully explored and evaluated, leaving many future work in this promising area. In addition, to gain practical impacts, we encourage researchers to investigate this problem in real practical scenarios and provide practical tool supports.



10. CONCLUSION

To alleviate the cost of regression testing, test-case prioritization is proposed, which aims to achieve some testing requirements by scheduling the execution order of test cases. This domain has been studied for decades and dedicated efforts have been made accordingly. In this work, we conduct a survey to systematically investigate the existing work on test-case prioritization.

More specifically, in this survey, we review the existing work by classifying them into six categories: algorithms, criteria, measurements, constraints, scenarios, and empirical studies. Based on these analyses, we further discuss challenges, issues, and future opportunities in test-case prioritization.

ACKNOWLEDGMENTS

This work is supported in part by NSF Grant No. CCF-1566589, UT Dallas faculty start-up fund, Google Faculty Research Award, Samsung GRO Award, and generous supports from Huawei, the National Key Research and Development Program 2016YFB1000801, and the National Natural Science Foundation of China under Grant No. 61522201.

REFERENCES

[1] S. Yoo, M. Harman, Regression testing minimization, selection and prioritization: a survey, Softw. Test. Verification Reliab. 22 (2) (2012) 67–120.

[2] G. Rothermel, M.J. Harrold, J. Von Ronne, C. Hong, Empirical studies of test-suite reduction, Softw. Test. Verification Reliab. 12 (4) (2002) 219–249.