Enhancing and Evaluating Software Test Case Diversity

Islam Elgendy

Supervisor: Professor Phil McMinn

Supervisor: Professor Robert Hierons

A thesis submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy

in the

Faculty of Engineering

School of Computer Science

October 2025

Dedicated to my wife, Fatma, my children, Farida, Perein, Hamza and Omar, my parents, and my entire family.

Declaration

Islam Elgendy
October 2025

Abstract

Diversity-based testing (DBT) has gained significant attention in software testing over the past two decades. DBT techniques use similarity metrics to exploit dissimilarity between software artefacts, such as requirements, inputs, or outputs, to address testing problems. This thesis advances the understanding and application of DBT to improve the diversity, effectiveness, and efficiency of software test suites. The research pursues three main objectives: (1) to analyse existing applications and artefacts used in DBT; (2) to develop automated approaches that apply DBT to test suite reduction and test case prioritisation; and (3) to explore new artefacts, such as bytecode, to enhance testing performance.

First, a systematic mapping study synthesised the landscape of DBT research, analysing 167 papers that employ 79 similarity metrics across 22 artefact types and 11 testing problems. The study identifies dominant trends, underexplored artefacts, and open challenges. Second, a string-based reduction technique was developed to minimise automatically generated test suites. This approach operates solely on test text, eliminating execution cost while maintaining fault detection capability. Third, the thesis investigates bytecode-level diversity as a novel artefact for test case prioritisation and mutant analysis. Results show that bytecode diversity improves fault detection by up to 7.8% and performs significantly faster than text-based approaches. Finally, the use of bytecode-based diversity in targeting stubborn mutants demonstrates superior efficiency, reducing test counts by up to 46% compared with coverage-based methods.

Overall, this thesis provides new insights into DBT theory and practice, introduces novel artefacts and techniques, and delivers scalable, effective solutions to key software testing challenges.

Publications

I. Elgendy, R. Hierons, and P. McMinn. Evaluating string distance metrics for reducing automatically generated test suites. In *Proceedings of the International Conference on Automation of Software Test (AST)*, pages 171–181, 2024. doi: 10.1145/3644032.3644455

I. Elgendy, R. Hierons, and P. McMinn. A systematic mapping study of the metrics, uses and subjects of diversity-based testing techniques. *Software Testing, Verification and Reliability*, 35(2):e1914, 2025. doi: 10.1002/stvr.1914

I. Elgendy, R. Hierons, and P. McMinn. Empirically evaluating the use of bytecode for diversity-based test case prioritisation. In *Proceedings of the International Conference on Evaluation and Assessment in Software Engineering (EASE)*, 2025. doi: 10.1145/3756681.3756969

I. Elgendy, R. Hierons, and P. McMinn. How effectively do test selection techniques kill mutants of varying stubbornness? Manuscript in preparation for submission to the International Conference on Software Testing Verification and Validation (ICST), 2026

Acknowledgements

All thanks and praise are due to Allah, God Almighty, the Most Gracious, the Most Merciful, and the All-Wise, for sustaining me and my family, and for aiding me throughout my studies.

I owe heartfelt thanks to my wife, Fatma, for her unwavering support during this journey; my children, Farida, Perein, Hamza, and Omar; my parents, Professor Taha and Dr. Somia; and my entire family, who stood by me every step of the way.

I would like to express my sincere gratitude to my supervisors, Professor Phil McMinn and Professor Robert Hierons, for their invaluable guidance, support, and advice throughout my PhD journey. I am especially grateful to Phil for being both my supervisor and mentor. I have learned so much from him, both academically and personally.

I am also thankful to the past and present researchers and academics in the Testing Lab at the University of Sheffield: Firhad, Ibrahim, Owain, Megan, Michael, Richard, Olek, Ben, Nasir, Abdulla, Ruizhen, Zalán, Neil, José, Donghwan, Tony, Kirill, Giulia, Yasmeen, Guannan, Arwa, Rimsha, and Sanjeetha. Thank you for sharing your knowledge and experiences with me. I also extend my thanks to my friends outside the lab who offered me much-needed moral support throughout.

Furthermore, I would like to thank the former Head of School, Professor Guy Brown, and the current Head of School, Professor Heidi Christensen, for their support during my time at the University.

Finally, I gratefully acknowledge the financial support I received from the University of Sheffield and the School of Computer Science.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

"Strength lies in **differences**,
not in similarities."
-Stephen R. Covey

This idea, although originally about human collaboration, applies equally well to software testing. One of the central challenges of testing is the intractability of testing each possible behaviour of a piece of software, since the number of behaviours a non-trivial software system is capable of exhibiting tends to be very large, if not infinite. For this reason, software testing researchers have devised many techniques to help a tester decide what to test. Approaches like boundary value analysis [318], risk [106], usage and complexity analysis [113], and history-based testing [94] can aid testers in designing test cases to target specific parts of the software, where it is more likely for failures to be exposed. However, when no such a priori information is available, and it is not possible to know where or how software failures will happen a priori, it is neither effective nor efficient to keep looking in the same or similar places [66] — i.e., to have tests that are overly-concerned with certain aspects of the software to the neglect of others.

Either explicitly or implicitly, one can argue that all software testing techniques embed the notion of *diversity*. Software testers commonly recognise that achieving thorough testing and ensuring system quality relies on the diversity of test cases [109]. Different code coverage metrics, for example, dictate that all statements or all branches of the software are executed, ensuring the diversity of code structures exercised by tests [23, 230]. Model-based techniques ensure diversity of tests by ensuring distinct abstract states of the system are tested along with the transitions between them [23, 304]. Even boundary-value analysis [23, 130], which advocates testing potentially similar inputs to encourage exploration around the thresholds of predicates in a program, does so in the knowledge that small mistakes in the conditions of predicates (for example, the use of " `>` " instead of " `>=` ", etc.) can trigger unexpected, divergent software behaviours.

However, the problem facing the tester is what to do when even the test requirements generated by these techniques result in too many tests. For these reasons, software testing researchers of late have become increasingly interested in a class of techniques that I refer to in this thesis as *diversity-based testing (DBT) techniques*.

1.1 Diversity-Based Testing (DBT)

At a high level, a DBT technique evaluates tests based on their *dissimilarity* to potential alternatives. A DBT technique measures dissimilarity using a *similarity metric* based on some aspect or set of *artefacts* related to the test; for example, the inputs it uses [45, 109, 282, 283, 284], the outputs it obtains given its inputs [18, 33, 212, 213, 215], the program structures it executes [31, 312, 336, 342]; or based on more abstract artefacts, such as the requirements the test exercises [24, 211], or parts of a model representation that it covers [54, 78, 143, 142, 139, 141]. Examples of similarity metrics used by researchers include, for example, the Euclidean distance between two numerical inputs [44, 45] or signals of a Simulink model [213, 215], the Edit distance between two regular expressions [51], or the Jaccard distance between two sets of selected product line features [1, 147]. More formally, I define a diversity-based testing technique as follows:

Definition 1. A **diversity-based testing technique** D is a function that takes as input a software system S, a set of artefacts A representing some aspects of the system (such as all of its possible inputs or outputs, its requirements, etc.), and a similarity metric *sim* that may be applied to two or more elements of A to measure their similarity. D uses *sim* in conjunction with A to minimise similarity between A and output a set of tests T for testing S, or to evaluate a test set for S.

DBT techniques have been used to tackle many different problems in software testing; for example, test data generation [18, 146, 213, 215], test case prioritisation [24, 184, 215, 226, 335], test suite reduction [71, 78, 80, 82], and test suite quality evaluation [52, 248, 286, 327]. They have also been applied to a wide range of subject domains, including deep learning models [4, 5, 229], compilers [63, 299, 192], web applications [18, 35, 208], as well as general code libraries [104, 109, 240, 226]. The advantages of DBT techniques have been reported in several studies on test case analysis, generation, and optimisation [110, 144, 240, 109, 78, 145, 226]. A common finding is that on the whole, the more diverse a test suite, the more likely it is to trigger failures and thereby detect faults [33, 35, 283, 109]. Several researchers have also shown that reordering test case execution order, based on diversity, or removing similar test cases is more likely to result in better test suites in terms of fault-finding than those re-ordered or reduced using other metrics, for example, coverage [24, 144, 54]. Finally, DBT techniques have been applied to different types of testing at different levels, including unit testing [110, 240], integration testing [246] and system testing [249, 297].

While DBT has been applied to many problems in software testing, prior research often studies these applications in isolation, focusing on different artefacts, metrics, or problem domains without connecting them together. The motivation for this thesis is therefore to develop a deeper, more integrated understanding of DBT, exploring not only its breadth but also how new artefacts and new problem settings can extend its power.

1.2 Goals

The primary goal of this thesis is to advance the understanding and application of diversity-based testing (DBT) as a principle for improving the effectiveness and efficiency of software testing. Rather than studying DBT in isolation, I progressively explore it across different problems and artefacts, with each chapter building on the findings of the previous one. To achieve this, the thesis is guided by the following three high-level objectives:

1. **Understanding:** To systematically investigate the landscape of DBT techniques, similarity metrics, artefacts, applications, and domains, identifying opportunities and gaps for further exploration.
2. **Utilising:** To develop and evaluate automated approaches that apply DBT to address key software testing problems such as test suite reduction and test case prioritisation.
3. **Exploring:** To identify and leverage underexplored diversity artefacts, such as bytecode, to enhance test suite effectiveness and efficiency.

1.3 Thesis Contributions

To achieve these objectives, this thesis makes four primary contributions, each chapter extending the overarching theme of how DBT can be harnessed to improve software testing. Figure 1.1 shows the relationships between the chapters, where the findings of one chapter motivate the next.

1.3.1 Literature Review (Chapter 2)

Chapter 2, which is based on my Software Testing, Verification and Reliability (STVR) paper [100], sets the foundation by surveying DBT research. This chapter is a systematic mapping study of DBT techniques, where I summarised the key aspects and trends of 167 papers that report the use of 79 different similarity metrics with 22 different types of software artefacts, which researchers have used to tackle 11 different types of software testing problems. I further presented an analysis of the recent trends in DBT techniques and reviewed the different application domains to which the techniques have been applied, giving an overview of the tools developed by researchers to do so. This study highlighted two key directions for subsequent chapters: the lack of studies

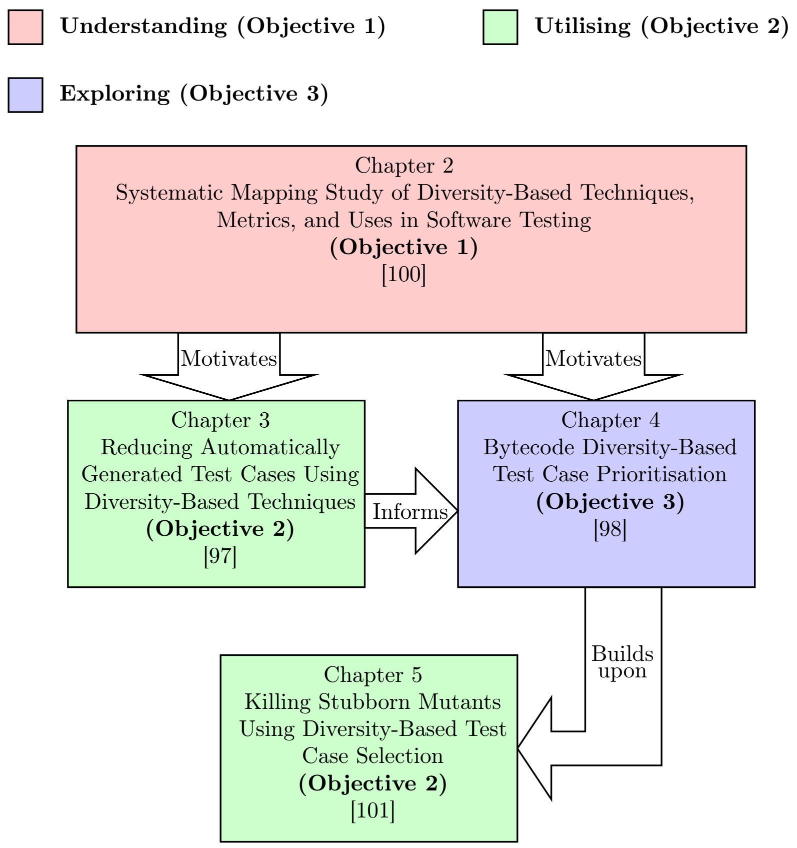

Understanding (Objective 1)
Utilising (Objective 2)
Exploring (Objective 3)
Chapter 2
Systematic Mapping Study of Diversity-Based Techniques, Metrics, and Uses in Software Testing
(Objective 1)
[100]
Motivates
Motivates
Chapter 3
Reducing Automatically Generated Test Cases Using Diversity-Based Techniques
(Objective 2)
[97]
Informs
Chapter 4
Bytecode Diversity-Based Test Case Prioritisation
(Objective 3)
[98]
Builds upon
Chapter 5
Killing Stubborn Mutants Using Diversity-Based Test Case Selection
(Objective 2)
[101]

Figure 1.1: *The relationships between the chapters of this thesis. Citations indicate that a chapter is based on my published work. Red blocks address "understanding", green blocks address "utilising", and blue blocks address "exploring".*

on automatically generated tests, and the limited exploration of alternative artefacts such as bytecode. The study offers the following contributions:

1. **Evaluation (Section 2.3)**: An evaluation of the growing research area and prominent papers and authors of DBT in software testing.
2. **Metrics (Section 2.4)**: A summary of the similarity metrics used in DBT.
3. **Artefacts (Section 2.5)**: An analysis of the software diversity artefacts used in DBT.
4. **Problems (Section 2.6)**: An analysis of the DBT techniques employed to address software testing problems.
5. **Subjects (Section 2.7)**: An overview of the subject domains where DBT were utilised in.
6. **Tools (Section 2.8)**: A summary of the DBT tools in the literature.

7. **Discussion (Section 2.9)**: A discussion of the research trends and future directions.

1.3.2 Diversity-Based Test Suite Reduction (Chapter 3)

Guided by the gap identified in Chapter 2, Chapter 3, which is based on a paper that I published in the proceedings of the International Conference on Automation of Software Test (AST) [97], investigates dissimilarity-based reduction of automatically generated tests. I evaluated string distance metrics for test reduction, showing that dissimilarity-based reduction outperforms random reduction in mutation detection, with normalised compression distance being the most effective metric. In this chapter, I used string distances on the text of the test cases as measures of similarity for reduction. I reduced test suites generated from Randoop [250] and EvoSuite [117]; two well-known test generation tools for Java programs. I implemented a pairwise string-based similarity reduction technique and compared it against random reduction. In the experiments, mutation scores using reduced test suites based on maximising string dissimilarity of test cases were higher than those for random reduction in over 70% of the test suites generated. Also, the results showed that, in one case, test suites generated by Randoop could be reduced by up to 99% using the string-based similarity reduction approach, while still maintaining the fault-finding capability of the original suite. Such a drastic reduction is feasible because Randoop often produces extremely large and redundant suites, with many tests offering little additional fault-detection value beyond others. Furthermore, on average, the normalised compression distance was found to be the best similarity metric choice in terms of fault-detection. Finally, I implemented a scalable similarity-based reduction approach from the FAST family, originally proposed by Miranda et al. [226]. While this method was significantly faster than the pairwise similarity reduction, its effectiveness was notably lower. The pairwise approach achieved higher mutation scores in 72–95% of the cases. The study offers the following contributions:

1. **Similarity-based reduction (Sections 3.5.1 and 3.5.2)**: I empirically evaluated, for the first time, the effectiveness of similarity-based reduction on automatically generated tests.

2. **Superior similarity metric (Section 3.5.3)**: I evaluated five different distance metrics in my experiments and found normalised compression distance to be the best out of them.

3. **Impact on tools (Section 3.5.4)**: I found that Randoop, which does not have a built-in minimisation technique, can benefit a lot from reducing test suites without affecting fault-finding capabilities.

4. **Scalability challenge (Section 3.5.5)**: A scalable approach can be more efficient, but that negatively affects its quality in terms of fault-finding abilities.

1.3.3 Bytecode Diversity in Test Case Prioritisation (Chapter 4)

In Chapter 2, I identified two key research gaps in the area of DBT: the limited exploration of automatically generated tests, and the lack of studies using alternative artefacts such as bytecode. The first of these gaps motivated the work in Chapter 3, where I investigated the potential of similarity-based reduction on automatically generated tests. The results of that study demonstrated the effectiveness of text-based similarity for reducing automatically generated tests, but also highlighted a practical challenge: the verbosity of test case text makes similarity calculations computationally expensive at scale. This scalability limitation motivated Chapter 4, where I also address the second gap by investigating bytecode as an alternative diversity artefact. This chapter, which is based on a paper that I published in the proceedings of the International Conference on Evaluation and Assessment in Software Engineering (EASE) [98], is the first to study bytecode as the basis of diversity in Test Case Prioritisation (TCP), leveraging its compactness for improved efficiency and accuracy.

I performed an empirical study on seven Java projects that shows that bytecode diversity improves fault detection by 2.3–7.8% over text-based TCP. It is also 2–3 orders of magnitude faster in one TCP approach and 2.5–6 times faster in FAST-based TCP. Furthermore, I found that filtering specific bytecode instructions improves efficiency up to fourfold while maintaining effectiveness, making bytecode diversity a superior static approach. However, dynamic coverage-based TCP techniques still achieve slightly higher fault-detection effectiveness (2.8–3.5% higher) due to richer runtime information, though bytecode diversity remains 6.5–8.3 times faster and avoids the instrumentation overhead required by coverage-based methods. The study offers the following contributions:

1. **A novel diversity artefact (Section 4.3.1)**: I introduced the use of bytecode of test cases as a new diversity artefact.

2. **A new filtering technique (Section 4.3.2)**: I proposed a new bytecode filtering method to enhance runtime.

3. **Empirical study (Section 4.5)**: An empirical comparison of static and dynamic TCP approaches with bytecode diversity-based TCP, using data from seven Java projects and 67 different classes, offering insights into the benefits of bytecode diversity.

1.3.4 Test Selection Strategies Under Varying Fault Difficulty (Chapter 5)

Building on the findings of Chapter 4, this chapter investigates why coverage-based selection appeared more effective in detecting faults. In Chapter 4, all faults were treated uniformly, without accounting for their relative difficulty. Here, I explore how different test selection strategies perform when faults vary in how hard they are to detect. To operationalise fault difficulty, I use *stubborn mutants*—mutants that are resistant to being killed and have been shown to correlate with real, hard-to-detect faults [165, 256]. Using these mutants, I compare dynamic coverage-based selection with lightweight, static diversity-based approaches that use both textual and bytecode representations of test cases. This analysis reveals that while coverage-based methods remain strong for easy or moderately stubborn faults, diversity-based techniques, particularly at the bytecode level, excel at identifying tests that kill highly stubborn mutants, requiring up to 46% fewer tests. To support this, I introduced the *Relative Stubbornness Thresholding Model (RSTM)*, which provides a principled, project-aware mechanism to identify stubborn mutants at varying levels of difficulty. Finally, I examined the characteristics and value of tests that exclusively kill stubborn mutants, finding that although they comprise only 2–25% of a suite, they achieve full mutation adequacy and reveal up to 88% of real faults. Overall, this chapter demonstrates that stubborn mutants serve as a valuable lens for understanding fault difficulty, and that lightweight, diversity-based selection offers an effective and scalable alternative for identifying high-value tests.

The chapter makes the following contributions:

1. **Empirical analysis of fault difficulty (Sections 5.5.1 and 5.5.2):** A systematic study of how different test selection strategies perform across varying levels of mutant stubbornness.

2. **Characterisation of stubborn mutants and their tests (Sections 5.5.3 and 5.5.4):** An investigation into the code characteristics of stubborn mutants and the value of tests that kill them.

3. **New identification model (Section 5.3.1):** The *Relative Stubbornness Thresholding Model (RSTM)* for identifying stubborn mutants in a project-sensitive and flexible way.

1.4 Summary

In this chapter, I introduced the motivation and goals of the thesis, highlighting the role of diversity as a fundamental principle in software testing. The contributions of the thesis are summarised in Figure 1.1, which illustrates how each chapter builds on the previous one. Starting with a systematic mapping study that clarifies the state of the art and research gaps, then addressing the gap of automatically generated tests through similarity-based reduction. Then, exploring bytecode as an alternative artefact for test case prioritisation, and finally connecting diversity to mutation testing through the identification and targeting of stubborn mutants. Together, these contributions advance the understanding of diversity-based testing both conceptually and practically, showing how it can be harnessed to improve test suite effectiveness and efficiency, while also expanding the scope of artefacts considered beyond traditional text and coverage.

Chapter 2

Literature Review

The contents of this chapter are based on “I. Elgendy, R. Hierons, and P. McMinn. A systematic mapping study of the metrics, uses and subjects of diversity-based testing techniques. Software Testing, Verification and Reliability, 35(2):e1914, 2025.”.

2.1 Introduction

A fundamental challenge in software testing lies in the sheer number of possible program behaviours. Exhaustively testing every potential behaviour of a non-trivial system is practically infeasible. As a result, numerous testing strategies have been developed to guide the selection of meaningful test cases, including techniques based on boundary values [318], risk [106], complexity [113], and historical usage [94]. These methods aim to increase the likelihood of uncovering faults by focusing testing effort where failures are most likely. However, in situations where such a priori knowledge is unavailable, repeatedly testing similar scenarios becomes inefficient [66]. Implicitly or explicitly, many test strategies rely on the principle of “diversity”, ensuring tests explore a broad and varied space of behaviours.

Diversity-based testing (DBT) is a family of techniques that makes this principle explicit. DBT approaches evaluate and select tests based on their dissimilarity from one another, using similarity metrics applied to test-related artefacts such as inputs [45, 109], outputs [18, 33], covered program structures [31, 312], requirements [24, 211], or model elements [54, 78]. By promoting variation among tests, DBT techniques aim to improve fault detection efficiency. They have been successfully applied to test case generation [146, 213], prioritisation [184, 215], reduction [71, 80], and evaluation across a wide range of domains, from compilers [63] and web applications [35] to deep learning systems [4]. The underlying assumption, supported by empirical evidence, is that more diverse test suites are more effective at revealing faults than those that are redundant or narrowly focused [33, 35, 283, 109].

DBT appears to have originated out of the work on Adaptive Random Testing (ART). ART was recently surveyed by Huang et al. [150], who summarise ART as aiming to *"enhance RT's [Random Testing's] failure-detection ability by more evenly spreading the test cases over the input domain".* Some ART techniques satisfy my definition of a DBT technique because they use a (dis)similarity metric that measures the "distance" between test cases to accomplish this spread. Others, however, do *not* fit my definition, because they partition the input domain instead — that is, *without* the use of a similarity metric [57, 67, 217], and potentially with the help of a human expert instead [274].

A systematic mapping study structures the types of research reports and results that have been published by categorising them [259]. It often gives a visual summary, the map, of its results. Systematic mapping studies can be utilised to provide a wide overview of a research area if the research area is very broad [173]. Given the wide range of types of DBT techniques, and the different problems and application domains to which they have been applied, I conducted a systematic mapping study of the field, collecting results from multiple search engines including IEEE Xplore, the ACM Digital Library, Scopus, and SpringerLink. I conducted my study following the guidelines by Petersen et al. [259] and Kitchenham [173]. The purpose of this study is to explore how DBT techniques in general have developed and broadened since its beginnings with ART. I do not, therefore, include ART itself as part of my review, referring the reader to the comprehensive survey of 140 ART papers by Huang et al. [150] instead. ART differs significantly in ethos from DBT. ART has traditionally focused on one problem — test input generation; and therefore one type of artefact — a program's input domain; while the definition of a similarity metric is not a necessary component of the technique, as already mentioned. In contrast, DBT focusses on tackling any problem in software testing where intractability is a key limiting characteristic, for which researchers have utilised many different software artefacts to assist in determining solutions, and have employed a wide variety of similarity metrics to measure their diversity in order to do so. In total, my study finds that researchers have tackled 11 different types of testing problems using DBT techniques, applying 79 different metrics to 22 different types of software artefacts.

This study, the first to define and survey the wider area of DBT techniques, summarises the work of 167 articles, which I list in a BibTeX file and make available for other researchers to use via a public GitHub repository and a replication package [95]. My study makes the following contributions:

1. The first definition of what constitutes a *diversity-based testing (DBT) technique* (Definition 1).
2. An analysis of the trends for DBT techniques and the prominent venues, papers, and authors in the research area (Section 2.3).
3. A compilation of the different metrics used to measure the similarity between testing artefacts (Section 2.4).
4. A summary of the artefacts used as a basis for DBT techniques (Section 2.5).

5. A summary of the different software testing problems where diversity has been applied (Section 2.6).
6. An overview of the different subject domains in which diversity-based testing has been utilised (Section 2.7), and the tools available for addressing them (Section 2.8).
7. A discussion of possible future research directions for DBT techniques (Section 2.9).

2.2 Methodology

The goal of this study is to get an overview of the DBT papers in software testing. I aim to answer the following research questions:

RQ1: ORIGINS, TRENDS, AND PROMINENT PAPERS (Section 2.3). What were the first papers in DBT? What are the publication trends, and most prominent venues and authors in the DBT field?

RQ2: SIMILARITY METRICS (Section 2.4). What similarity metrics have been used in the literature? Which ones have been used the most, and why?

RQ3: ARTEFACTS (Section 2.5). What artefacts have been used (e.g., test inputs, software requirements, abstract models) as the basis for applying similarity metrics to address the testing problem?

RQ4: PROBLEMS (Section 2.6). What are the software testing problems to which DBT techniques have been applied?

RQ5: SUBJECT DOMAINS (Section 2.7). To what subject domains have DBT techniques been applied?

RQ6: TOOLS (Section 2.8). What DBT tools have been developed by researchers?

2.2.1 Collection Approach

Detailed guidelines of how to perform a systematic mapping study that reports on relevant primary studies are given by Petersen et al. [259]. A primary study is an original first-hand research article reported by the author who generated the data used in the article. The systematic mapping process consists of five main steps. The first step is the definition of research questions which is the research scope. Second, find as many primary studies about the topic through searches using digital libraries, backward snowballing [321], and checking journal and conference proceedings. Third, select relevant primary studies by screening the list of primary studies obtained using inclusion and exclusion criteria. Fourth, make a classification scheme to extract relevant information from the primary studies to answer the research questions. Finally, collate and summarise the results of the included primary studies.

The formulation of my research questions was made during my initial set of previously known papers that used DBT in software testing [110, 18, 184, 79, 82, 283, 35]. I reformulated them again with the additional papers found from the search, and again during information extraction until I reached my final decision about the research questions reported earlier.

I collected papers for my study on the 24th of June, 2024, using searches on IEEE Xplore[1], the ACM Digital Library[2], Scopus[3], and SpringerLink[4]. I did not limit my searches to any particular year range. The search query I used for IEEE Xplore, Scopus, and ACM was {"`software test*`" AND ("`divers*`" OR "`similar*`" in title or keywords)}, where "`*`" is a wildcard matching any string. This means looking for the text "`software test*`" anywhere in the paper and a variation of the terms "`divers*`" (i.e., "diversity", "diversify", or "diversification", etc.) or "`similar*`" (i.e., "similar" and "similarity") in the title or the keywords. Initially, my search query contained only "`divers*`" in title or keywords, but I refined the search to include "`similar*`" before the screening process. I restricted the latter terms to the title or keywords due to their commonality, and hence the ability to match a very large number of papers that would (a) be intractable to screen manually, and (b) have, on average, a low probability of being related to DBT techniques. For example, a paper may claim to use a "diverse" set of subjects for an empirical study and would be returned in my search results without this constraint. SpringerLink did not allow me to search for certain terms in specific places like the title and keywords, so I used the closest search query possible, which was {"`software test*`" AND ("`similar*`" OR "`divers*`")} anywhere in the paper. I provide the exact search queries used for each search engine and the search operationalisation in my replication package [95]. I validated my search string against my initial set of primary studies [110, 18, 184, 79, 82, 283, 35]. Also, I validated the search against a suggested list from one of the reviewers of the study [249, 246, 248, 247, 110, 109, 184, 6, 55, 54, 168]. The combined lists contain 16 papers and 15 out of them were among my search results, while I identified the last one [184] during the snowballing phase.

Figure 2.1 gives an overview of the collection approach and the number of papers at each stage, reporting the number of papers retrieved from each search engine. The number of papers returned by SpringerLink was the most, due to the aforementioned difficulty in restricting the search terms. After I removed duplicate papers returned by more than one search engine, the set of papers collated numbered 3,569 in size, that I label the **Initial** set of papers in the figure.

I screened the papers by manually reviewing them to ensure they discussed or evaluated at least one approach fitting Definition 1, and thereby fell in the scope of this study. I decided whether to include or exclude papers based on titles, abstracts,

[1]`https://ieeexplore.ieee.org/Xplore`
[2]`https://dl.acm.org`
[3]`https://www.scopus.com/`
[4]`https://link.springer.com`

and detailed content if needed. Then, I reviewed and discussed the decisions with my supervisor to agree on each paper. Most decisions were obvious, and most "disagreements" were just minor mistakes due to the large number of papers to go through. I removed 3,302 papers from further consideration as part of this process. These included papers that were not focused on software testing or did not focus on software testing techniques; for example, being instead concerned with diversity in the sense of inclusion or representation of different types of people in a software engineering team. They also included papers discussing software testing techniques purporting to be "diverse", but did not satisfy my definition of a DBT given in Definition 1; for example, the technique did not use a similarity metric, or, I could not find whether the technique made use of a similarity metric in the text of the paper. Following these decisions, I then removed papers that did not appear in journals or conferences ranked "C" or above by the CORE Rankings Portal [196], with the aim of imposing a level of quality on the papers considered by my study. I chose not to do a quality assessment of my own since this could potentially introduce my own subjective biases. Using CORE rankings as a third party quality assessment is a route that has been used by others, including a recent systematic literature review published in TOSEM [317], a systematic literature review appearing at SIGCSE [296], and other reviews and mapping studies appearing in journals in related fields [263, 202, 201]. I included papers at any workshops with a history of co-locating with conferences ranked as "C" or above, because these workshops inherit the rigorous reviewing of the conferences hosting them, and I wanted to consider emerging novel ideas or results that are often presented at these venues. I excluded papers appearing in doctoral symposia. Only 15 papers in the study came from workshops, and they follow the same patterns and trends as those from conferences and journals. I removed a further 66 papers because they did not fulfil my CORE requirements. Also, I added four papers recommended by the reviewers. Furthermore, when I found duplicate extension papers that presented the same topic, I included only the most complete paper (i.e. the journal version). This left 205 papers, which I mark as the **Intermediate** set of papers in Figure 2.1.

I then removed a further 58 papers that related to Adaptive Random Testing (ART), which is not a focus of this study as discussed in Section 2.1. Finally, in case any DBT papers were missed by my original searches, we applied backward snowballing [321], whereupon I manually studied the references of each of the papers in the intermediate set of papers. If a reference involved a DBT technique, was in a CORE "C"-ranked venue or better, and was not an ART paper — i.e., it met all of my aforementioned criteria — I added it to the set of papers featured in this study. I found 20 additional papers via this method which I added to form the collection of papers marked as the **Final** set of papers in Figure 2.1. These papers did not include the search terms or were not in the database of the search engines. I read each paper in the final set and collated the statistics and information needed to answer my research questions as answered in the following sections. All of these papers in the final set are primary studies (i.e., it does not contain any secondary studies). The papers in my study are regular and short

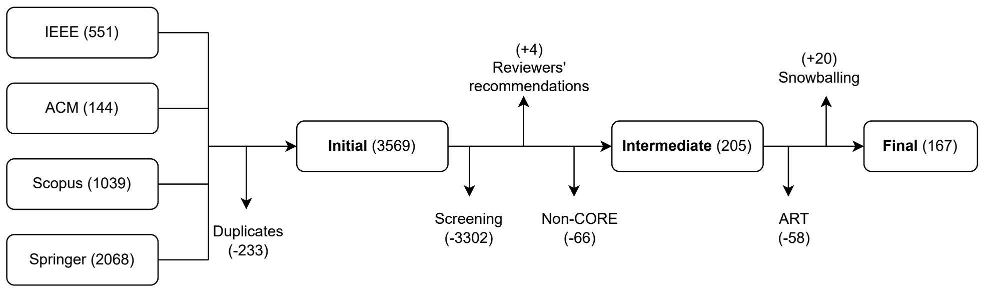


Figure 2.1: *Overview of my paper collection methodology, with the number of pages collected at each stage*

papers, with the majority of the papers being regular (93.4% of the papers are regular), and I did not include position papers. According to the guidelines by Kitchenham [173], I applied a test re-test approach, where I made a second extraction of the statistics and information on a random selection of papers to check the consistency of information extraction. The random set contained 17 papers (10% of the study) and they are specified in the replication package. The supervisor performed an independent check on the same random sample of the papers and cross-checked it with me to confirm the analysis results.

2.2.2 Threats to Validity

I hereby discuss validity threats and the steps I took to mitigate these threats.

1. **Some relevant papers might not have been included in the searches.** I mitigated this risk as much as possible by using a variety of search engines and I picked my search terms carefully to include different variations of the terms "diverse" or "similar". Also, I used backward snowballing to find papers that the search engines may not have indexed. However, I validated my search results against my initial set of primary studies and also against a list of papers suggested by one of the reviewers (both lists mentioned in Section 2.2.1). The combined lists consisted of 16 papers, where 15 out of them were among my search results, while I identified the last one during my backward snowballing phase. Some other search terms could have been used, but due to the vast possible results (3,569 papers in the initial set), I limited my search and relied on backward snowballing to include the papers that might have used a variation of diversity. To further mitigate this risk, I performed a search on the 2nd of September 2024, using IEEEXplore search engine with the addition of two proxy words ("`unique*`" and "`sparse*`"). After removing papers already considered as part of my original searches, I had a sample of 304 candidates. I did not find a single additional paper that fitted my scope and DBT definition. Therefore, it would not affect my study if such proxy words were included in the search. Finally, I did not perform forward snowballing as the guidelines [173, 259] did not prescribe it for mapping studies.

2. **The manual screening process may have eliminated some relevant papers.** I mitigated this risk by having the supervisor do the screening process separately and independently, then coming together to have a final decision on the papers where I had different opinions.

3. **Some papers may be of low quality and their results might be questionable**. I attempted to mitigate this risk by including papers that are only published in peer-reviewed journals and conferences ranked "C" and above by the CORE ranking portal.

4. **Information extraction of papers may contain mistakes**. I attempted to mitigate this risk by applying a test-retest approach, where I made a second data extraction of a random selection of primary studies to check data extraction consistency. Also, the supervisor performed an independent check on the same random sample of the papers and cross-checked it with me to confirm the analysis results.

5. **Reproducibility of the study**. I attempted to mitigate this risk by explaining my methodology as rigorously as possible and providing all search results and data in a replication package [95].

While I acknowledge that it is possible that I may have missed some DBT papers, I am confident that I have included the majority of relevant articles, and that my study provides an accurate account of the key trends and the state of the art of DBT.

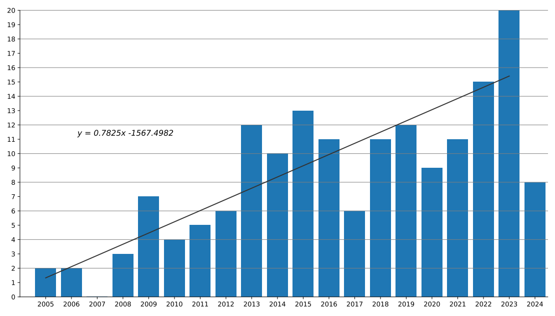
2.3 RQ1: ORIGINS, TRENDS, AND PROMINENT PAPERS

What were the first papers in DBT? What are the publication trends, and most prominent venues and authors in the DBT field?

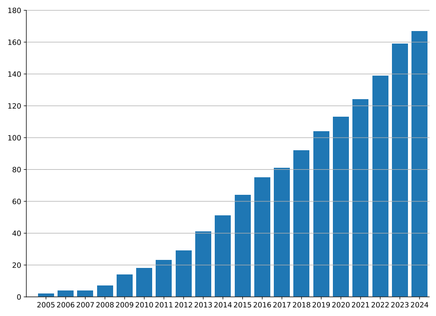
In this section I study the origins of DBT, studying the first papers. I also present and study trends in the number of DBT publications per year. Finally, I analyse the most cited DBT papers, the most prolific DBT authors, and the venues with the highest frequency of publishing DBT works.

2.3.1 First DBT Papers

The first DBT papers in my final collected set of papers appeared in 2005, are two papers by Xie et al. [328] and Hao et al. [132]. These first two papers are notable in that they apply the principles of DBT to assist two existing approaches, namely



(a) *Publications per year*



(b) *Cumulative sum of publications over the years*

Figure 2.2: *Papers published per year and the cumulative sum of papers found over the years for the final set of papers collected using my methodology. Since the data was collected partway through 2024, the bar for that year is incomplete and as such I plot the trendline through 2005–2023 only.*

search-based test data generation and fault localisation respectively, rather than being standalone DBT techniques in their own right. Xie et al.'s DBT work was part of a search-based strategy for maintaining population diversity so as to avoid premature convergence of the search on suboptimal test cases [328]. The approach monitors the similarity between the individuals in the population, and once it exceeds a certain threshold, their technique significantly increases the mutation probability to increase population diversity. I discuss this paper further in Section 2.5.1 (p.27). Hao et al.'s work [132] was a short paper proposing to detect similarity in test cases to remove bias from suspiciousness scores for fault localisation techniques. I discuss this paper in more detail in Section 2.6.6 (p.55).

However, peering into the intermediate set of papers collected by my methodology (Section 2.2) shows that authors started writing about ART techniques in terms of diversity and similarity in the year before these two papers appeared, in 2004 [66, 56, 67, 65]. ART itself, however, can be traced back earlier to 2001 by a paper by Chen et al. [68], as part of a review of proportional sampling strategies. Not all techniques discussed require a similarity metric. The first papers to evaluate ART using a similarity metric — Euclidean distance with numerical inputs — were various formulations by Chen et al. [66, 64, 65] in 2004, which were built on partition strategies.

2.3.2 Statistics of DBT Papers

Figure 2.2 displays the annual number of DBT publications and their cumulative total, showing an increasing trend in DBT research, with 2023 having the highest number of publications. The trend line only covers 2005–2023, since I performed the search in June of 2024.

Table 2.1 lists the top ten publishing venues, highlighting that DBT papers frequently

Table 2.1: *Most frequent publishing venues*

Rank / Venue		**# Papers**
1	Transactions on Software Engineering (TSE)	14
2	International Conference on Software Testing, Verification and Validation (ICST)	11
=3	International Conference on Automation of Software Test (AST)	9
=3	Information and Software Technology (IST)	9
=3	International Conference on Automated Software Engineering (ASE)	9
=3	International Conference on Software Engineering (ICSE)	9
=7	International Symposium on Search Based Software Engineering (SSBSE)	8
=7	Transactions on Software Engineering and Methodology (TOSEM)	8
9	International Symposium on Software Testing and Analysis (ISSTA)	7
10	International Symposium on Software Reliability Engineering (ISSRE)	5

appear in leading software engineering and testing journals and conferences. IEEE Transactions on Software Engineering (TSE) leads with 14 DBT papers, while the International Conference on Software Testing, Verification and Validation (ICST) follows with 11 papers. Conferences dominate the list with seven of the top ten spots, compared to three journals.

I collected citation counts from Google Scholar [124], with counts correct to 3rd July 2024. The paper with the second most citations (266) is that of Yoo et al. [335] on clustering test cases for test case prioritisation. However, this paper was also published in 2009, which is relatively old for a DBT paper, and therefore have more time than other works to gain citations. For this reason, I calculated the mean number of citations per year and list the top ten papers in terms of this metric in Table 2.2.

The papers with the highest mean citations per year are by Kim et al. [171], Henard et al. [146] and Hemmati et al. [141]. Kim et al. [171] introduce a new metric for testing Deep Learning systems that leverage on diversity. This paper is the most cited paper (431) in my study with an average of 71.8 citations per year. The proposed adequacy metric has been used a lot after the recent spike of interest to testing deep learning systems. Henard et al. [146] use a similarity-based approach for software product lines (SPL), using diversity between configurations to prioritise the set of configurations that should be tested. The paper is an important reference for other studies in testing SPL systems. I discuss it in more detail in Section 2.5.16 (p.37). Hemmati et al. [141] focussed on test case selection in the context of Model-Based Testing. I discuss this paper in Section 2.6.4 (p.54).

Test case prioritisation is the focus of four of the ten papers in Table 2.2, with the other six covering a variety of other topics including testing deep learning systems, test case selection, test generation, model-based testing, and new similarity metrics for measuring the diversity of test suites. I discuss the prominent problems to which DBT has been applied in Section 2.6.

Table 2.2: *Papers with the highest numbers of citations*

Rank / Authors		Title	Year	Section (Page)	# Cites	Mean cites per year
1	Kim et al. [171]	Guiding Deep Learning System Testing Using Surprise Adequacy	2019	2.5.1 (p.24)	431	71.8
2	Henard et al. [146]	Bypassing the combinatorial explosion: Using similarity to generate and prioritise t-wise test configurations for software product lines	2014	2.5.16 (p.37)	242	22.0
3	Hemmati et al. [141]	Achieving Scalable Model-Based Testing through Test Case Diversity	2013	2.6.4 (p.54)	259	21.6
4	Miranda et al. [226]	FAST Approaches to Scalable Similarity-Based Test Case Prioritisation	2018	2.6.2 (p.48), 2.5.2 (p.28)	121	17.3
5	Yoo et al. [335]	Clustering test cases to achieve effective and scalable prioritisation incorporating expert knowledge	2009	2.6.2 (p.50), 2.5.6 (p.31)	266	16.6
6	Arafeen et al. [24]	Test case prioritisation using requirements-based clustering	2013	2.6.2 (p.50), 2.5.4 (p.30)	194	16.2
7	Feldt et al. [109]	Test Set Diameter: Quantifying the Diversity of Sets of Test Cases	2016	2.5.1 (p.26)	143	15.9
8	Matinnejad et al. [215]	Test Generation and Test prioritisation for Simulink Models with Dynamic Behaviour	2019	2.6.1 (p.44), 2.5.3 (p.29)	87	14.5
9	Zohdinasab et al. [350]	Efficient and effective feature space exploration for testing deep learning systems	2023	2.5.13 (p.35)	29	14.5
10	Zohdinasab et al. [349]	Deephyperion: exploring the feature space of deep learning-based systems through illumination search	2021	2.5.13 (p.35)	57	14.3

2.3.3 Prominent Authors

The top five most prolific authors in the papers collected in my study are listed in Table 2.3. The author publishing the most papers is Lionel Briand [39], with 15 publications, focussed on applying DBT in model-based testing, and Simulink models in particular (see Section 2.6.1, p.44; and Section 2.6.4, p.53).

The second in the list is Robert Feldt [107] with 10 papers. Robert Feldt applied his work mostly in Search-Based Software Testing to diversity. This work is discussed in Section 2.6.1 (p.40). He also proposed a similarity metric named "Test Set Diameter" (TSDm) that can be applied to measure diversity of the whole test suite (discussed in Section 2.5.1, p.24), and proposed a new metric called "Surprise Adequacy for Deep Learning Systems" for testing deep learning system (discussed in Section 2.5.1, p.24).

Ranked third in the list with nine papers is Francisco Gomes de Oliveria Neto [245], who has mainly used DBT approaches to solve the problem of test case selection, discussed in Section 2.6.4 (p.53).

The fourth in the list are Hadi Hemmati [138] and Paolo Tonella [301] with eight papers each. Hadi Hemmati has a research interest in automated software testing and applying diversity in test case prioritisation and model-based testing (discussed in Section 2.6.2, p.48; and Section 2.6.4, p.53). Paolo Tonella worked on a handful of topics in DBT including deep learning testing (discussed in Section 2.5.13, p.35) and

Table 2.3: *Authors with the highest number of papers*

Rank / Author		# Papers	
1	Lionel Briand (`https://www.lbriand.info`)	15	[139, 143, 142, 140, 141, 212, 213, 215, 4, 269, 198, 200, 253, 28, 5]
2	Robert Feldt (`http://www.robertfeldt.net`)	10	[110, 109, 248, 247, 90, 209, 261, 231, 171, 108]
3	Francisco Gomes de Oliveira Neto (`https://www.gu.se/en/about/find-staff/franciscodeoliveiraneto`)	9	[54, 249, 246, 247, 248, 90, 229, 168, 6]
=4	Hadi Hemmati (`https://lassonde.yorku.ca/users/hhemmati`)	8	[143, 142, 139, 140, 141, 144, 240, 228]
=4	Paolo Tonella (`https://www.inf.usi.ch/faculty/tonella`)	8	[208, 170, 35, 221, 335, 265, 349, 350]

web testing (discussed in Section 2.6.1, p.45).

The most significant collaboration was between Lionel Briand and Hadi Hemmati, who co-authored five papers [143, 142, 139, 141, 140] on DBT in model-based testing and test case selection. Another key collaboration was between Robert Feldt and Francisco Gomes de Oliveira Neto, who co-authored three papers [248, 247, 90] on applying diversity for boundary value exploration [90], visualising test diversity [248], and measuring behavioural diversity with mutation testing [247].

Conclusions — RQ1: ORIGINS, TRENDS, AND PROMINENT PAPERS

The first paper on DBT techniques in my collected results was published in 2005, and the number of publications has had a strong upward trend year on year since then. The venue featuring the most papers to date (14) is the Transactions on Software Engineering (TSE) journal with the International Conference on Software Testing, Verification and Validation (ICST) appearing next, with 11 papers. The paper with the most citations (431) is that of Kim et al. [171] on proposing a new adequacy metric for testing a deep learning system that leverages diversity, published in 2019. Also, it has the highest mean number of citations per year followed by Henard et al. [146], on diversity for configurations of software product lines for testing, published in 2014; and Hemmati et al. [141] on test case selection for model-based tests, published in 2013. Finally, the most prolific author in the field is Lionel Briand, having published 15 papers on the topic of DBT.

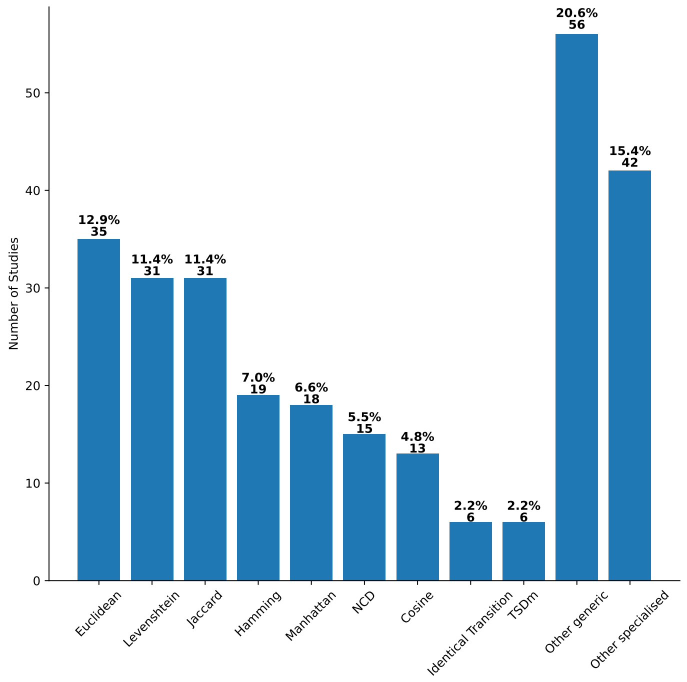


Figure 2.3: *Distribution of similarity metrics used in DBT techniques.*

2.4 RQ2: SIMILARITY METRICS

What similarity metrics have been used in the literature? Which ones have been used the most, and why?

All studies using diversity-based approaches rely on metrics to measure similarity or diversity, applicable to inputs, outputs, or other testing artifacts. I identified 79 similarity metrics in the literature, including well-known ones like Euclidean and Hamming distances, as well as domain-specific and novel metrics. I categorised these into two groups: generic metrics from other fields and specialised metrics for Software Engineering. Figure 2.3 shows the distribution of these similarity metrics in DBT papers. Sometimes a distance metric would appear under a different name, but would be synonymous with another. For example, “edit” and Levenshtein distance are commonly interchangeable, and in one paper, “overlapping” distance was found to be the same as Hamming distance [151]. Therefore, I merged the counts of these under the more common heading.

Table 2.4 lists the top five generic similarity metrics, while Table 2.5 covers the top five specialised metrics in software engineering, ordered by popularity and then alphabetically. Each entry includes a citation, a brief description, the number of papers using the metric, and references to all relevant papers. The citation for specialised metrics in Table 2.5 references the introducing paper. Extended tables with all 79 metrics are available in the replication package [95] and in Appendix A. Figure 2.3

Table 2.4: *A list of the generic similarity metrics citing the source, a brief description, the number of papers using them and citations*

ID	Metric & Source	Description	Suitable for	Total	Papers
G1	Euclidean distance [14]	The square root of the sum of the squared differences between the vectors X and Y. The formula is: $$Euc(X,Y) = \sqrt{\sum_{i=1}^{n}(x_i - y_i)^2}$$	Numbers	35	[7, 24, 26, 28, 45, 43, 53, 71, 79, 80, 82, 97, 105, 126, 127, 129, 144, 171, 182, 184, 206, 212, 213, 215, 231, 233, 241, 258, 265, 269, 283, 287, 312, 337, 345]
G2	Edit distance [189]	The minimum number of edits *(insertions, deletions or substitutions)* required to change one string into the other. It takes into consideration that parts of the strings can be similar even if not in corresponding places, and can work with strings of different sizes.	Strings	31	[2, 10, 27, 35, 37, 51, 79, 80, 78, 246, 97, 104, 115, 127, 143, 142, 139, 141, 168, 184, 205, 210, 228, 240, 231, 283, 288, 315, 314, 340]
G3	Jaccard distance [156]	The ratio of intersection over union between two sets A and B of values. The formula is: $Jac(A,B) = 1 - \frac{\vert A \cap B\vert}{\vert A \cup B\vert} = 1 - \frac{\vert A \cap B\vert}{\vert A\vert + \vert B\vert - \vert A \cap B\vert}$ Sometimes expressed [312] as: $Jac(A,B) = 1 - \frac{A.B}{A.B + \omega(\Vert A \Vert^2 + \Vert B \Vert^2 - 2(A.B))}$ with with $\omega = 1$.	Sets and strings	31	[22, 51, 78, 79, 80, 246, 248, 114, 141, 119, 127, 128, 142, 140, 147, 146, 168, 185, 198, 200, 210, 219, 225, 226, 299, 306, 312, 325, 336, 342, 344]
G4	Hamming distance [131]	The number of times when the corresponding characters in two strings are different. Huang et al. [151] referred to this as "Overlap" distance.	Vectors	19	[1, 52, 70, 79, 80, 97, 141, 112, 142, 151, 184, 185, 228, 240, 283, 309, 310, 330, 335]
G5	Manhattan distance [16]	The sum of the absolute differences between two vectors X and Y. The formula is: $$Man(X,Y) = \sum_{i=1}^{n}\vert x_i - y_i\vert$$	Numbers	18	[37, 52, 63, 60, 97, 127, 128, 144, 184, 192, 198, 200, 206, 241, 269, 283, 349, 350]

Table 2.5: *A list of the specialised Software Engineering similarity metrics.*

ID	Metric & Source	Description	Total	Papers
S1	Identical transition distance [54]	The number of identical transitions between two finite state machines divided by the average length of paths.	6	[54, 249, 143, 142, 139, 141]
S2	Test set diameter (TSDm) [109]	An extension of the pairwise normalised compression distance (G6) to multisets.	6	[109, 128, 129, 282, 284, 83]
S3	Identical state distance [143]	The number of identical states between two paths of finite state machines divided by their average number of states.	3	[143, 139, 141]
S4	Trigger-based distance [143]	An extension of identical transition similarity (S1) to account for triggers in the transitions.	3	[143, 139, 141]
S5	Average population diameter [307]	The average distance between all vectors in a population, where the distance between two vectors is the difference of their lengths.	2	[307, 308]

shows that 80.2% of DBT papers use generic metrics, while 19.8% use specialised ones. The largest bar is "other generic metrics" (20.6%), including 40 metrics coded G8 to G47, while “other specialised metrics” comprises 30 metrics coded S3 to S30.

The three most popular similarity metrics are Euclidean distance, Edit distance, and Jaccard distance used in 35, 31, and 31 papers, respectively. Numeric programs are used by many researchers to evaluate their techniques and Euclidean distance is a natural choice for such programs. Also, with string data, the Edit distance is very popular to use. Furthermore, Jaccard distance is widely used when the testing artefacts can be represented as sets, with an example being software product lines in which a product can be seen as being a set of features.

Conclusions — RQ2: SIMILARITY METRICS

Many similarity metrics were used in the literature, and I found 79 metrics in this study. The most used similarity metrics in the literature are generic metrics, where the most popular similarity metrics were Euclidean distance, Edit distance, and Jaccard distance found in 35, 31 and 31 papers, respectively. Since many of the testing subjects in the literature are either numeric or strings, it was natural for researchers to use Euclidean distance for numeric data and Edit distance for strings. Also, Jaccard distance is widely used when the testing artefacts can be represented as sets. The mapping study results are **not indicative** of which metric would be best to use, as different factors can affect this choice such as the nature of the data. As will be shown in Section 2.5 and Section 2.6, there is no clear indication of which metric performs the best, as different authors reported different results based on their empirical studies and the similarity metrics used in their studies.

2.5 RQ3: ARTEFACTS

What artefacts have been used (e.g., test inputs, software requirements, abstract models) as the basis for applying similarity metrics to address the testing problem?

Different software artefacts have been used as a basis for DBT techniques. Some of these artefacts include the input domain, output domain, etc. I discuss how these artefacts were used in the DBT techniques.

Table 2.6 presents a list of the diversity artefacts used, with a short description, the total number of papers using them, and citations of all these papers. The most used diversity artefacts are inputs and test scripts, while test report diversity, social diversity, function, and running time diversity were used in one study each. Figure 2.4 shows the distribution of testing artefacts used in DBT papers. Papers that use inputs or test scripts as diversity artefacts constitute almost 40% of the total DBT papers in my study. Also, DBT papers that use a hybrid of diversity artefacts form 7% of the collected papers.

Finally, all the papers in my study use only one of the following categories, except four papers [143, 142, 139, 141] that apply their experiments on transitions and triggers artefacts separately, and another paper [83] that use inputs and features in two separate approaches. Papers that use combinations of diversity artefacts in a single approach are categorised as hybrid (Section 2.5.20).

Table 2.6: *A list of the testing artefacts used as a basis for DBT techniques.*

Rank/Artefact	Description	Total	Papers

1	Input	Data provided to the software, which can be numbers, strings, objects, or images.	34	[21, 10, 20, 37, 45, 43, 69, 83] [86, 109, 171, 170, 209, 219, 221, 220, 229] [255, 258, 261, 264, 265, 277, 282, 283, 284] [288, 307, 308, 315, 313, 328, 333, 338]
2	Test Script	Information from the actual text of the test cases which may include for example setup, inputs, and assertions.	32	[6, 22, 29, 34, 47, 53, 60, 71, 79, 80] [82, 248, 97, 122, 126, 128, 129, 132, 140, 240, 144] [151, 168, 184, 226, 228, 253, 269, 286, 299, 133, 324]
3	Feature	The specific capabilities, or characteristics that test cases has in the software under test.	21	[1, 2, 4, 5, 28, 63, 70, 83, 105, 147, 161] [162, 182, 192, 233, 241, 311, 326, 337, 349, 350]
4	Hybrid	Approaches that use a combination of different artefacts as targets for applying diversity.	12	[8, 7, 26, 35, 110, 185] [198, 200, 205, 208, 236, 291]
=5	Execution	The actual runtime execution path of the software under test after running the test cases.	9	[52, 104, 127, 260, 323] [329, 335, 340, 344]
=5	Transitions	The edge connecting a source state to a target state. In model-based testing, two transitions are said to be similar if they both have the same source and target states, and trigger.	9	[27, 54, 78, 249, 143] [142, 139, 141, 314]
=5	Output	The results produced by the software under test from running the test cases.	9	[18, 33, 90, 108, 119] [175, 212, 213, 215]
=5	Coverage	Information about statement or branch coverage achieved from running the test cases.	9	[31, 246, 206, 235, 312] [336, 342, 345, 343]
9	GUI	The visual components and their properties' values that represent the interface of the software.	6	[111, 136, 187, 210, 231, 327]
10	Semantic	Statements that have different wordings but carry similar meanings are said to be *semantically* similar.	5	[50, 51, 247, 195, 244]
11	Trigger	A specific event or condition that causes the system to move from one state to another. In model-based testing, trigger similarity does not require the source and target states to be identical, unlike transition similarity.	4	[143, 142, 139, 141]

=12	Path	The possible flow of control and data within a program without executing it.	3	[49, 254, 330]
=12	Context	Data from software and environment to guide the responses and actions of the system.	3	[42, 309, 310]
=12	Test Steps	A test step corresponds to some manual action and consists of a stimulus and an expected reaction.	3	[115, 116, 306]
=12	Software product line configuration	A combination of specific features or components from a software product line to create a customized software product.	3	[114, 146, 325]
=16	Mutant	Small variations introduced to a software program using mutation operators that represent faults in the software.	2	[289, 290]
=16	Requirements	The system specifications and requirements, where test cases linked to requirements are specified through traceability matrix.	2	[24, 211]
=16	Graph	Graph models represent and analyse relationships between variables in a system, where the nodes of the graph are the variables and the edges are the relationships or dependencies.	2	[169, 281]
=19	Test Report	Reports written in natural language and contain screenshots that show the results of tests.	1	[112]
=19	Social	Coverage data collected from similar users who utilise a web service.	1	[225]
=19	Function	A process to perform a certain functionality given some inputs and possibly producing an output.	1	[293]
=19	Running Time	The runtime execution of test cases.	1	[287]

2.5.1 Input

Input diversity measures use the distances between the test data to calculate a diversity value for test sets. Inputs can be program inputs, or arguments passed to methods. The inputs can be numbers, strings, or complex objects like tree-structured data. Several authors used input diversity to guide the testing process [45, 283, 37, 333, 10, 35].

Input diversity is used in search-based software testing techniques to guide the search process. Bueno et al. [45] used Euclidean distance as a measure that characterises the diversity of a test set. The diversity of the test set is the sum of distances between each test data in the set and its nearest neighbour. Boussaa et al. [37] used the novelty score algorithm and suggested the use of diversity between methods' arguments of the individuals of a population. The Novelty metric measures the distance between test suites in the population. They used Manhattan distance for primitive types, and Levenshtein distance for string data types.

There has also been work that has used diversity metrics for strings. Shahbazi

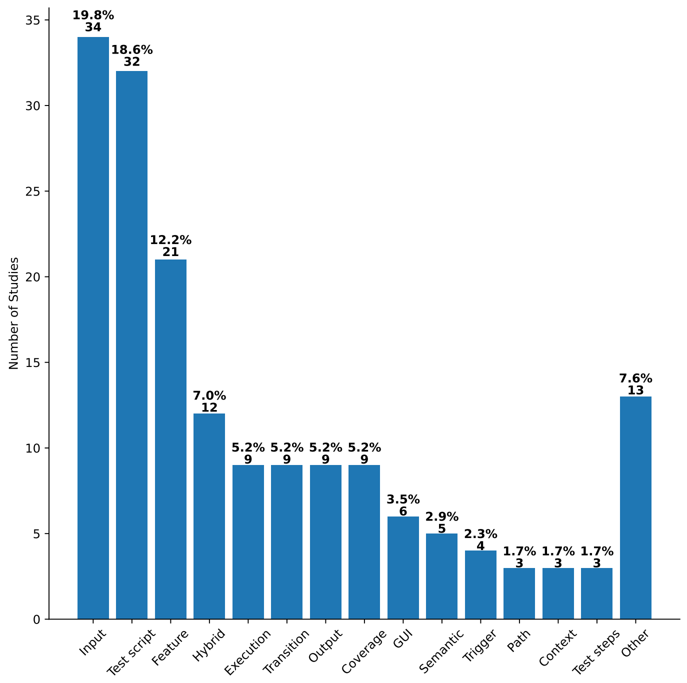


Figure 2.4: *Distribution of testing artefacts used in DBT techniques.*

and Miller [283] used string inputs as a basis for diversity and used six different string distance functions, which are Levenshtein distance, Hamming distance, Manhattan distance, Cartesian Distance, Cosine Distance, and Locality-Sensitive Hashing (LSH). The experiments showed that the Levenshtein distance was the best in terms of fault-detection, followed by Hamming distance and then Cosine distance. In another study, that deals with numeric and string inputs, Yatoh et al. [333] extended Randoop and used multiple pools simultaneously rather than a single pool to increase input diversity.

Deep learning (DL) systems gained a lot of interest recently. Kim et al. [171] proposed a new test adequacy criterion for deep learning systems called *"Surprise Adequacy for Deep Learning Systems"* (SADL), that leverages on diversity. One variant of the proposed SADL is a distance-based metric that finds the novelty of a new input in relation with the training data. It is based on the Euclidean distance of the activation trace (i.e. the neurons activated using the input) of the new input and the previous activation traces of the training data. Riccio et al. [265] proposed a way to generate new test inputs to augment the current test set and improve the mutation score. The proposed method uses an evolutionary algorithm, where two fitness functions are used, one of which is based on the distance of the new test input to previous inputs already found in the search process. It uses Euclidean distance or genotypic distance according to the domain used. For some DL systems, images of the dataset are considered input to the DNN. Mosin et al. [229] used input diversity (i.e. diversity between images) for a similarity-based test input prioritisation approach. The similarity between inputs (images) is calculated using the root mean square error. However, generating valid and diverse data using real-world media data is challenging. Yuan et al. [338] proposed a method for generating diverse and valid inputs for DNN testing. The media data can be mutated using exploration and exploitation increasing perceptual diversity while maintaining its validity. Distance between media data is used as part in calculating the diversity scores. Dai et al. [83] proposed a diversity-driven seed queue construction of fuzzing approaches for DL systems. The approach measures the similarity between test data using the normalised compression distance.

A new similarity metric was proposed by Feldt et al. [109], which used input diversity to evaluate the proposed approach. They proposed a metric to calculate the diversity of sets of test cases considered as a whole, rather than a metric just to calculate the diversity between a pair of test inputs, outputs, or traces. They called it "test set diameter" (TSDm), and it extends the pairwise normalised compression distance to multisets. The term "multiset" is used rather than set because the same string might occur more than once in the collection. However, this metric requires a higher computational cost to calculate compared to other similarity metrics.

Input diversity is used in fuzz testing to control the large number of tests generated [258], to generate inputs that exercise different traces [264], or to create a diverse set of tests [220]. Also, input diversity was employed for multiple dispatch languages. Poulding and Feldt [261] proposed use of diverse test inputs in terms of values and types.

Tree structured data

Tree-structured data are used in many systems like HTML and XML. Therefore, there is a need to have a similarity distance function for such data structures. Shahbazi and Miller [282] proposed a similarity metric for trees and named it *Extended Subtree* (EST). EST uses the edit base distance and generalises it using new rules for subtree mapping. In a later study, Shahbazi et al. [284] proposed a black-box test case generation technique of tree-structured programs (e.g. XML parsers or web browsers) based on the diversity of a tree test set, where each test case is a tree. The proposed approach used a Multi-objective genetic algorithm (MOGA) using a diversity-based fitness function and tree size as objectives. They used six tree distance functions in their experiments: Tree edit distance (TED), Isolated subtree (IST) distance, Multisets distance, Path distance, and Extended subtree (EST) distance. EST was reported to outperform the other distance functions.

Similarly, in another study, Shimmi and Rahimi [288] used the similarity of tree-structured data to generate test cases for Java applications. The source code is transformed into an XML structured format using a tool called srcML [224], then used another tool called Triang [75] to generate a meta-model of XML Schema Definition (XSD). A group of methods with the same exact XSD structure are similar to each other. The approach would recommend test cases of such similar methods to be adapted by the developers to cope with the changes in the software. They used the Levenshtein distance between the test case recommended and the developer's changes to estimate the effort for adapting the generated test cases.

One way of measuring structural similarity of code is through abstract syntax trees. Altiero et al. [21] proposed a similarity-based method for test case prioritisation capable of measuring structural similarity of changed code. The approach uses Tree Kernels to calculate the similarity between two abstract syntax trees to get a better measure of structural similarity rather than only a textual similarity that might not be relevant.

Population

Some studies implemented DBT techniques in an Evolutionary algorithm, and in their studies they managed to increase population diversity. In these studies, test inputs are represented as a population in the evolution algorithm.

One issue found in evolutionary algorithms used for test data generation is the *genetic drift* phenomena. The genetic drift problem occurs when individuals become very similar to one another, limiting the diversity of the population, and causing early convergence into suboptimal solutions. Xie et al. [328] proposed a dynamic self-adaptation strategy for evolutionary structural testing referred to as *DOMP* (**D**ynamic **O**ptimisation of **M**utation **P**ossibility). The approach analyses the evolution process and checks for premature convergence, then sharply increases the mutation probability to increase the diversity of the population. DOMP checks whether the proportion of gene-type exceeds a threshold value, and the mutation probability is adjusted to an enhanced mutation probability controlled by the tester until the diversity of the population is restored. Alshraideh et al. [20] proposed the use of multiple populations rather than a single population keeping the solutions diverse, and preventing the process of falling into a local optima. Kifetew et al. [170] proposed an orthogonal exploration of the input space approach to introduce diversity in the population to counter the genetic drift problem. The diversity of the population is increased by adding individuals in orthogonal directions via Singular Value Decomposition (SVD). Panichella et al. [255] proposed DIV-GA (**DIV**ersity based **G**enetic **A**lgorithm), that inject new orthogonal individuals to increase diversity during the search process for test case selection. DIV-GA uses an orthogonal design to generate an initial diversified population and then uses SVD to maintain diversity through the search process. Almulla and Gay [10] proposed an approach to increase diversity in test data generation referred to as adaptive fitness function selection (AFFS). AFFS changes the fitness functions used during the generation process to increase diversity. They used the Levenshtein distance as a fitness function and as a target for reinforcement learning.

An alternative way of maintaining population diversity is through using multiple subpopulations and ensuring that individuals with high diversity migrate between the subpopulations. Wang et al. [315] used this idea for web application testing and proposed a parallel evolutionary test case generation approach. With multiple subpopulations being evolved, the migration of individuals from one subpopulation to another keeps the diversity high. The diversity of an individual is calculated using the Levenshtein distance to all individuals in the target subpopulation, and the individuals with the farthest distances are selected for migration.

2.5.2 Test Script

Many studies applied DBT techniques using the test script as a target for measuring the diversity. This is achieved by considering the actual text of the test cases and measuring similarities between them. Mainly, many of these papers which use test script diversity perform test case prioritisation [184, 144, 29, 226, 126, 151, 53, 34, 60, 128, 129] or test suite reduction [79, 80, 82, 71, 97]. Since the test suite already exists, and the target is to reorder them or minimise the test suite, it is intuitive to use the actual text of the test cases. Ledru et al. [184] suggested the use of string distances on the text of test cases. They considered the use of four distance metrics, which are Euclidean distance, Manhattan distance, Levenshtein or Edit distance, and Hamming distance. They reported that Manhattan distance was the best choice for the similarity metric for test case prioritisation. Hemmati et al. [144] implemented the same algorithm used by Ledru et al. [184], using the Euclidean and Manhattan distances, to work on manual black-box system testing. Rogstad et al. [269] utilised test script diversity for test case selection in large-scale database applications. They used four similarity metrics in the study: Euclidean distance, Manhattan distance, Mahalanobis distance, and normalised compression distance. Burdek et al. [47] proposed a white-box test suite generation approach for software product lines based on the reuse of reachability information by means of similarity among test cases. Mondal et al. [228] used a few similarity measures in diversity-based test case selection, such as Hamming distance, Levenshtein distance, and Dice diversity formula. Khojah et al. [168] used unit, integration, and system level tests to prioritise tests based on text and semantic diversity. Jaccard, Levenshtein, and NCD distances are used to measure lexical similarity, while natural language processing techniques were used to measure semantic similarity.

The concept of program diversity was used by Tang et al. [299], where the distance between two test Programs is the graph edit distance between the control flow graphs of the two programs and the Jaccard distance between the statements of the test programs.

Miranda et al. [226] used the string representations of the test cases for similarity-based black-box prioritisation, and used Jaccard distance to measure the similarity between two test cases. While Cruciani et al. [82] applied test script diversity for test suite reduction. They modelled each test case as a point in some D-dimensional space, and used Euclidean distance to select evenly spaced test cases. Shi et al. [286] represented the test cases as nodes in a complete graph where the weights of the edges are the distance between each pair or nodes. Elgendy et al. [97] empirically evaluated multiple string distances as basis to reduce automatically generated regression tests. The string distance metrics were applied on the text of the test cases to measure similarity, and reduction was made to maximise diversity of test cases. Pan et al. [253] transformed the code of test cases into abstract syntax trees and used tree-based similarity measures as a fitness function to guide reduction of test cases.

2.5.3 Output

Output diversity or uniqueness is a black-box technique that minimises similarities between produced outputs. The key idea is that two test cases that produce different kinds of output have a higher probability of executing two different paths. Alshahwan and Harman [18] proposed the use of output diversity for web application testing aiming to minimise similarities between produced outputs. Output uniqueness can be measured by calculating the difference between a new page and the previously visited pages in terms of one or more of the web page components, which can be the appearance and structure of the web page or the content (i.e. the textual data that the user can view). In another study, Benito et al. [33] utilised output diversity for unit test data generation. However, they repurposed the XORSample' algorithm from sampling input diversity to sampling output diversity. The output domain is randomly partitioned into cells of equal size and randomly selects one cell. An input is selected that is able to produce an output in the selected cell, which in turn that input is the new test case.

Outputs of Deep Neural Networks were used by Gao et al. [119] who proposed an adaptive test selection method that leverages the differences between model outputs as a behaviour diversity metric. The approach is inspired from adaptive random testing, where they used a variation of Jaccard distance as a fitness metric to measure the differences between test cases. In another paper, Kong [175] used output diversity to generate more adversarial examples to test deep learning models. The approach applied Output Diversified Sampling (ODS) strategy, proposed by Tashiro et al. [300], to maximise the diversity of output space increasing neuron coverage while maintaining a high attack success rate by taking a step size in the diversification direction.

Also, output diversity is used in Model-based testing in general and in Simluink models in particular. Matinnejad et al. [212] proposed an algorithm for test case selection of state flow models based on output diversity. The output-diversity algorithm aims to build a new test suite from the original test suite by maximising the output signal diversity using the Euclidean distance to measure the diversity between two output signals. Later, Matinnejad et al. [213] proposed a new *feature-based* diversity measure. The feature of an output signal is a distinguishing shape in the signal, and a set of these features is stored in a feature vector. The Euclidean distance calculates the diversity between two feature vectors of two output signals. Test data generation using feature-based diversity detected more faults than vector-based diversity. Matinnejad et al. [215] used normalised Euclidean distance to calculate the diversity between output signals in terms of vector- and feature-based diversity.

Furthermore, output diversity is used in the automatic detection of boundaries. Dobslaw et al. [90] discussed Boundary Value Analysis (BVA) and proposed an algorithm for automatic detection of boundaries using distance functions for detection, especially when the specifications are incomplete or vague. The idea behind boundary detection is to use the output diversity.

2.5.4 Requirements

Test cases can be linked to requirements through a traceability matrix and some researchers investigated using software requirements as the target of diversity. Arafeen et al. [24] used requirements' diversity by using k-means clustering [135] algorithm, which uses Euclidean distance, to cluster the requirements, and using requirements-test cases traceability matrix, they linked the test cases with each cluster. They used the code complexity metric (cm) (explained in Table 2.5) to prioritise the test case in each cluster. Masuda et al. [211] proposed a syntax-tree similarity method based on natural language processing for test-case drivability in software requirements. They pre-process the requirements to select test-case derivable sentences using the syntax-tree similarity technique, then calculate the similarity between each sentence in the requirements and test- case-derivable sentences. Based on the selected requirements, conditions and actions are derived.

2.5.5 Program Internal Information

Internal information about the structure of the software can be used as a diversity artefact, such as code, branches, or paths. An interesting idea is to make use of the similarity of a tested program to test new ones. Pizzoleto et al. [260] introduced a framework to reduce the cost of testing a program using mutation testing. They used syntactic similarity to determine program similarities and to identify which program group is closest to a new program. The similarity metrics used were ACCM (Average of Cyclomatic Complexity per Method) and RFC (Response for Class). Guarnieri et al. [127] implemented the framework, experimented with 20 possible configurations and used Euclidean distance, Manhattan distance, Expectation Maximisation algorithm, Levenshtein distance, Jaccard Index, normalised compression distance, and a plagiarism detection tool called JPlag. Nguyen and Grunske referred to covering many branches evenly with diverse inputs as behavioural diversity [235]. They proposed a new metric to measure behavioural diversity, which is based on the Hill numbers ecology diversity metric.

Code coverage can be the target for diversity. Coviello et al. [80] proposed a clustering-based approach for test suite reduction based on their statement coverage. They used Euclidean distance, Hamming distance, cosine similarity, Jaccard distance, Levenshtein distance, and string Kernels-based dissimilarity to measure the similarity between test cases. Later, Coviello et al. [79] made an experimental study using many test suite reduction approaches where one of the approaches is based on the coverage similarity between test cases. They used the same six similarity metrics to cluster all similar test cases into one group, and then select the test case covering the largest amount of test requirements. Zhang and Xie [342] conducted an empirical study for testing deep learning systems, where one of the diversities measured is the diversity in Neuron Coverage (i.e. the number of activated neurons). Mahdieh et al. [206] used coverage information of test cases as a basis for grouping test cases together in clusters for test case priotrisation. The similarity between test cases is calculated using the Euclidean distance, Manhattan distance, and Cosine similarity of the coverage information of each test case. The empirical evaluation made on five projects from the Defects4J dataset reported that the Euclidean distance performed better than Manhattan and Cosine in terms of average first fail detected.

A program path is the possible flow of control and data within a program without executing it. The analysis is typically based on examining the program's source code, without considering actual runtime behaviour or specific input values. Xie et al. [330] proposed three algorithms for calculating the similarity between the test path and the target path for path-oriented testing. Panchapakesan et al. [254] used path diversity combining evolutionary strategy with evolutionary algorithm. The control flow graph (CFG) of the software under test is given, and the similarity between two paths are calculated from the CFG. Cai et al. [49] used path diversity to guide a search-based algorithm. The path similarity is determined by the approach level (AL) value. The smaller the AL value, the more similar two paths can be. The AL value between two paths is the number of mismatched branch predicates between them.

2.5.6 Execution

An execution path refers to a specific sequence of statements or instructions that are executed during the runtime of a program. It represents the flow of control within a program, outlining the order in which statements are executed, branches are taken, and conditions are evaluated. Yoo et al. [335] used test execution diversity for test case prioritisation by clustering test cases. The clustering is performed based on the similarity of the execution traces of test cases calculated using Hamming distance. Wu et al. [323, 104] aimed to use test execution diversity in similarity-based test case prioritisation techniques. An execution profile is defined as the ordered sequence of program elements sorted based on the execution count. They used Levenshtein distance as a similarity metric to measure the similarity between two ordered execution sequences. Zhang et al. [340] suggested a new similarity metric and referred to it as *test case similarity* metric to measure the execution similarity between test cases. It is based on the normalised edit distance between two execution paths.

Another way of utilising execution diversity is in metamorphic testing. Cao et al. [52] investigated the relation between the fault-finding capabilities of metamorphic relations (MRs) and the diversity of test case executions in the hope of identifying good MRs (i.e. MRs that can reveal faults). MRs are some expected properties of the target program. The dissimilarity of the test case executions is calculated by measuring the distance using coverage Manhattan distance, frequency Manhattan distance, and frequency Hamming distance. Their empirical study asserted the importance and significance of diversity of test case executions to effective MRs to reveal faults. Xie et al. [329] proposed a DBT prioritisation technique of Metamorphic test case Pairs (MPs) for Deep Learning (DL) software. The diversity between MPs is determined based on the execution of the internal states of DNNs using new diversity metrics suitable for DNNs.

The diversity metrics used are Kullback-Leibler (KL) divergence, Jensen-Shannon (JS) divergence, Wasserstein distance (WD), and Hellinger distance (HD).

2.5.7 Semantic

Software semantics refers to the meaning and interpretation of software artefacts, such as programs, libraries, frameworks, and other software components. Semantics can be static, which refers to the rules, constraints, and specifications that govern the structure and correctness of software artefacts. Also, semantics can be dynamic, that focus on the runtime behaviour and execution of software components. Lin et al. [195] presented a natural-language approach based on semantic similarity to test web applications using Cosine similarity.

Two tests that fail or pass together are said to have similar semantic [244]. Ojdanic et al. [244] investigated whether seeded faults (i.e. mutants) that are syntactically similar to real faults are also semantically similar. Syntactic similarity is measured through the Bilingual Evaluation Understudy, while semantic similarity is calculated using the Ochiai coefficient. They found that syntactic similarity does not reflect semantic similarity. In another paper, Gomes de Oliveira Neto et al. [247] introduced measures to calculate the behavioural (semantic) diversity of test cases, and referred to it as *b-div*. They used mutation testing to compare the executions and failure outcomes of the test cases to capture the behavioural diversity.

Error-revealing test cases can be stored in a library to be reused in the future. Retrieval methods can be based on keywords, attributes-values, and other syntactic measures. However, these methods lack important semantic information. Cai et al. [50] suggested the use of semantic diversity as a basis for test case reuse and retrieval.

Machine translation systems are systems that do automatic translation of text and speech from a source language to another target language, and like any other system, they must be tested properly. Usual approaches detect mistranslations by examining the syntactic (i.e., textual) similarities between the original sentence and the translated sentence, but ignoring the semantic similarity. Cao et al. [51] proposed an automatic testing approach for machine translation systems based on semantic similarity. The sentences are transformed into regular expressions and use Levenshtein distance to calculate the semantic similarity between two regular expressions, or transformed into deterministic finite automata where Jaccard distance is used to calculate the semantic similarity between their regular languages.

2.5.8 Context

Context-aware Pervasive Software (CPS) collects and analyses data from software and environment to guide the responses and actions of the system. Test cases for CPS consist of sequences of context values, such as $\langle location, activity \rangle$ in smartphones [310]. Context diversity measures the number of changes in contextual values of test cases. A context stream is a series of context instances taken at different time intervals. Each context instance is a series of variables, which consists of *value*, *type*, and *time*. Wang and Chan [309] introduced the use of context diversity paired with coverage-based criteria for constructing a test suite. The context diversity of a context stream is the sum of the Hamming distances between two successive context instances. In a later study, Wang et al. [310] proposed three techniques to use context diversity to improve data-flow testing criteria for CPS. Once more, context diversity is measured using Hamming distance.

In event-driven software (EDS), test cases are a sequence of events. Brooks and Memon [42] presented a new similarity metric, *CONTeSSi(n)* (**CON**text **Te**st **S**uite **Si**milarity), that uses the context of n preceding events in test cases. Thus, *CONTeSSi*(n) is a context-aware similarity metric, which is adapted from the cosine similarity metric, with values in the range $[0, 1]$, where the closer the value is to 1, the more similarity there is.

2.5.9 GUI

A GUI state is represented as a set of widgets that make up the GUI, a set of properties for the widgets, and a set of values for the properties. Feng et al. [111] used GUI state similarity as a basis for GUI testing. The GUI state similarity is the sum of all similarity values of its windows. A window similarity is the sum of all similarities of its widgets. A widget similarity is the similarity of its properties values.

Mariani et al. [210] made an empirical study on the reuse of GUI tests. They used the Jaccard Distance and Edit (Levenshtein) Distance to compute the similarities between source events and a set of target events, and reported that matching algorithm is the most impactful on the quality of the results. Xie and Memon [327] investigated GUI state diversity for evaluating the GUI test suite. In this study, the diversity of GUI states simply means using different GUI states.

Some DBT approaches address the GUI fragility problem (i.e. test execution failing for no apparent reason). Nass et al. [231] proposed a similarity-based approach for web element localisation to facilitate the test automation. The approach measures the similarity using Euclidean and Levenshtein distances between various attributes of each candidate web element and the target element to identify the candidate element that most closely matches the target.

2.5.10 Transition & Trigger Diversity

In the context of model-based testing, the test cases are considered to be paths traversing a Labelled Transition System (LTS) [166]. An LTS is a directed graph defined in terms of states and labelled transitions between states to describe system behaviour. Coutinho et al. [78] proposed using similarity-based approaches to reduce test suites in the context of model-based testing and focused on LTS. The reduction approach calculates a *similarity matrix*, which is the similarity degree between each pair of the test cases. They made an analysis of the effectiveness of six different string distances to compute the similarity between the test cases and applied them to three test subjects. The distance functions they used were the Similarity function, Levenshtein distance, Sellers algorithm, Jaccard index, Jaro distance, and Jaro-Winkler distance. They concluded that the choice of the similarity metric does not affect the size of the reduced test suite, but it affects the fault coverage. Cartaxo et al. [54] and Gomes de Oliveira Neto et al. [249] used transition diversity in similarity-based test case selection. The similarity is calculated using the *Similarity function* or identical transition similarity (S1) measure mentioned in Table 2.5. In a later study, Hemmati et al. [143] developed the similarity-based test case selection using trigger diversity rather than only transition diversity. They developed trigger-Based similarity (S4) measure explained in Table 2.5.

Another paper used transition diversity in search-based software testing. Asoudeh and Labiche [27] used search-based GA and represented the entire test suite of the model under test as an individual. The fitness function was defined based on transition diversity and calculated using the Levenshtein distance to measure the similarity of the individual solutions. Wang et al. [314] used Extended Finite State Machine (EFSM) to model Android apps and generate test cases using a genetic algorithm guided by test diversity. EFSM is able to represent the behaviour of the Android apps better, and account for more complex expressions, preconditions, and data constraints. Test cases are encoded as sequences of transitions, where they used as their fitness function the Levenshtein distance to measure the similarity between these sequences.

2.5.11 Graph

Graph models represent and analyse relationships between variables in a system. A graph model consists of a graph structure composed of nodes and edges, where nodes represent variables or entities, and edges represent relationships or dependencies between them. Some papers considered applying diversity on graph models, either diversity within a single model, or diversity between a set of models. In the case of a single model, internal diversity is measured through the number of shape nodes covered by the model. In the case of a set of models, the external diversity (i.e. diversity between pairs of models) is measured through graph distance functions. Semeráth et al. [281] proposed a number of shape-based graph diversity metrics to measure the model diversity. Three distance functions used in the study are Pseudo-distance, Symmetric distance, and Cosine distance.

2.5.12 Mutants

An interesting idea in mutation testing is to include the diversity of the mutants and tests in addition to the number of mutants killed. Shin et al. [289] proposed a new mutation adequacy criterion called *distinguishing mutation adequacy criterion*, where the idea is that fault-detection can be improved using stronger mutation adequacy criteria. Mutants can be distinguished by their killing tests, and the proposed mutation adequacy criteria aim to distinguish as many mutants as possible plus killing mutants.

Shin et al. [290] made a more comprehensive empirical study using 352 real faults from *Defects4J* on the distinguishing mutation adequacy criterion.

2.5.13 Feature

Features of software refer to the specific functionalities, capabilities, or characteristics that a software product or application offers to its users. Features differ depending on the type of the application or the software under test. Many researchers used the idea of feature diversity in many areas, such as testing Content-Based Image Retrieval (CBIR) [241], constructing meaningful search spaces [162], compiler testing [63, 192], and SPL [147, 1, 326].

CBIR is the process of finding content in a database most similar to an object image, which extracts features from images and calculates the similarities of these features with the images stored in the database using traditional distance functions. Nunes et al. [241] made an empirical study of 10 similarity functions in CBIR in the context of software testing systems with graphical outputs. Another study that also uses image datasets used feature diversity to test Deep Neural Networks. Aghababaeyan et al. [4, 5] used a feature extraction model to extract features from images and calculated the feature diversity using three metrics, which are the Geometric diversity, normalised compression distance (NCD), and Standard Deviation. Similarly, Jiang et al. [161] suggested an approach to generate test inputs by maximising feature diversity calculated by Geometric diversity and evaluated it on four image datasets.

In compiler testing, a feature vector of a test program refers to a representation that captures the characteristics or features of the program. It is a multidimensional vector that encodes specific attributes or properties of the program, which are relevant for testing and evaluation purposes. Chen et al. [63] used the idea of feature diversity for compilers and proposed HiCOND. HiCOND uses the Manhattan Distance to measure the diversity between the feature vectors that represent the test programs.

A meaningful search space (fitness landscape) is constructed by defining and organising possible solutions or configurations to represent the problem domain effectively and facilitate searching or exploring potential solutions. Joffe and Clark [162] proposed an approach to construct meaningful search space using neural networks (NNs). They used an auto-encoder for diversity based on the similarity of the data points salient features (i.e. features that are most important to differentiate one data point from another).

Feature diversity has been used in deep learning systems and testing autonomous vehicles. Zohdinasab et al. [349, 350] proposed a method to generate test inputs for deep learning systems. The proposed method uses an evolutionary algorithm, where the fitness function is based on the Manhattan distance to measure the diversity of feature space of the deep learning system. Neelofar and Aleti [233] proposed new test adequacy metrics for autonomous vehicles testing called *"Test suite Instance Space Adequacy"* (TISA) to measure the effectiveness of a test suite in terms of diversity and coverage. Wang et al. [311] proposed a diversity-based fuzzing method to speed up the fuzzing approach for testing autonomous vehicles. By measuring the similarity of a test sequence to the gold standard (i.e. non-crashing test sequence), the method can terminate the test early if it is similar to the non-crashing test, otherwise it continues. The similarity model sequence uses a "Diversity Inference" model to predict similarity where the absolute difference of the features extracted of the two test sequences is an input to the prediction model. Cheng et al. [70] developed a diversity-based technique for generating diverse scenarios to test autonomous vehicles. The approach extracts features from the given scenario and make groups of similar features to represent different behaviours. They used the Hamming distance to measure the similarity between two traces of test cases. Dai et al. [83] proposed another feautre-based diversity-driven seed queue construction. The approach extracts features from tests and then calculates the similarity using Cosine similarity. Adigun et al. [2] proposed a diversity and risk-driven test case generation method for safety-critical machine learning systems. The test cases here are considered to be a set of domain features and their values that describe the testing scenario. The feature diversity is measured using normalised Levenshtein distance.

In software product line engineering, a feature model (FM) is a structured representation of the common and variable features within a family of related software products. The model defines the presence or absence of features and allows for configuration and customisation of products by selecting or deselecting specific features. Henard et al. [147] used mutation testing on FMs of software product lines to confirm that diverse test cases are more capable of detecting the mutants. In this context, a product consists of n features of FM, and Jaccard Index can be used to measure the similarity degree between two products based on selected features. The test suite is ordered according to the similarity degree, such that the top test cases are the most dissimilar to all other test cases. Abd-Halim et al. [1] proposed an enhanced string distance function for similarity-based test case prioritisation in software product lines. The modified string distance is based on a hybrid between Jaro-Winkler and Hamming distance referred to as Enhanced Jaro-Winkler (EJW). They used their modified string distance on multiple of different prioritisation and evaluated them in terms of average percentage fault detection. In another study, Xiang et al. [326] developed a model for test suite generation of software product lines using Quality-Diversity optimisation. Each test case is represented a binary sequence of n-features, and the fitness function for test generation is guided by feature diversity that is calculated using Anti-dice distance.

2.5.14 Test Report

Crowdsourced testing tasks are performed, and the results are given in reports referred to as "*test reports*", which are written in natural language and contain screenshots. The test reports are inspected manually to determine their fault-finding value. Feng et al. [112] proposed a diversity-risk-based prioritisation approach of the test reports for manual inspection and called it *DivRisk* strategy. The diversity aspect helps in removing duplicates and ensuring a wide range of reports are inspected, while the risk aspect ensures that the more likely fault-revealing reports are inspected first.

2.5.15 Test Steps

In system integration level testing, tests are represented as *"test steps"*, where each test step correspond to some manual action and a test step consists of a stimulus and an expected reaction [116]. The similarity between test steps can be calculated using any of the similarity distance functions. Flemstrom et al. [115] used the Levenshtein distance to measure the overlap between test steps. In a later study, Flemstrom et al. [116] used similarities between test steps to guide the ordering of test cases. Viggiato et al. [306] used test steps similarity for test suite reduction. They group similar test steps, and then they used the grouped test steps to find similar test cases using three different similarity metrics (simple overlap, Jaccard, and cosine metrics).

2.5.16 Software Product Lines Configuration

Software product lines (SPL) configuration refers to the process of selecting and combining specific features or components from a software product line to create a customised software product that meets specific requirements or user preferences. It involves choosing the desired features, their variations, and their configurations to create a tailored software solution. Diversity between SPL configurations is used in many papers [146, 114, 325].

2.5.17 Social

Miranda et al. [225] proposed a different coverage metric referred to as *Social Coverage*, that uses coverage data collected from similar users to customise coverage information. The goal is to identify a group of entities that are of interest to the user automatically without the need to provide such information a priori. Each user has a list of services performed, and the data coverage of these services is obtained through code instrumentation. In this work, the similarity between the target user and all other system users is calculated using the Jaccard Index measure. Given a similarity threshold, any service below that threshold is discarded, and the union of the remaining similar services serves as the targeted entities.

2.5.18 Function

If two libraries implementing the same functionality, and have two test suites, they can serve each other by applying one test suite on the other library or adapting it to reveal undetected bugs. Sondhi et al. [293] suggested exploiting the fact that many library projects overlap and have similar functionalities to use the associated test suites on similar projects with some adaption lowering the cost. They used cosine similarity distance to measure the similarity between library functionalities.

2.5.19 Running Time

The idea of monitoring the runtime execution of test cases, and applying diversity metrics on runtime information is only used by Shimari et al. [287]. They proposed a clustering-based test case selection using the execution traces of test cases for an Industrial Simulator. Runtime information of test cases was encoded into vectors of integers and clustered together into groups using k-means clustering algorithm. The Euclidean distance was used to measure the similarity between these vectors.

2.5.20 Hybrid

Approaches that use a combination of different artefacts as targets for applying diversity are referred to as hybrid diversity approaches. Feldt et al. [110] proposed a model for test variability referred to as VAT (VAriability of Tests), where they identified 11 variation points in which tests might differ from each other including "test setup", "test invocation", "test execution", "test outcome", and "test evaluation". They suggested using normalised compression distance to measure the diversity of the VAT trace. In another study, Albunian et al. [8, 7] defined three diversity measures based on fitness entropy, test executions, and Java statements. The first two referred to as *phenotypic diversity* are behavioural or semantic diversity, and the last one is referred to as *genotypic diversity* is a structural or syntactic diversity. They used Euclidean distance to calculate the semantic and syntactic similarities.

Liu et al. [198] used three test objectives in a search-based algorithm to increase test suite diversity and minimise size. The fitness functions used to guide the search process are referred to as coverage dissimilarity, coverage density, and number of dynamic basic blocks. The *coverage dissimilarity* increases diversity between test executions and uses Jaccard distance to measure it. The *coverage density* is an adaptation of the test coverage density which is the average size of test execution slices related to every output over the static backward slice of the output. High-density values indicate that large parts of the model are covered. In a later study, Liu et al. [200] added a new test objective based on output diversity. The *Output Diversity* aims to maximise the diversity between the output signals and uses a normalised Manhattan distance to measure the output diversity between two output signals. Also, Arrieta et al. [26] proposed using input and output diversity for Simulink models. They used the Euclidean distance as a basis for the two similarity metrics on input and output signals.

Ma et al. [205] used hybrid static and dynamic diversity metrics. The static diversity metric is based on Levenshtein distance, which makes sure that there is diversity in the program structures of the test cases. The other dynamic diversity metric enforces diversity in the test cases' ability to cover untested thread schedules.

Marchetto and Tonella [208] applied hybrid diversity for testing Ajax Web applications. Three fitness functions based on test diversity are *EDiv*, *PDiv*, and *TCov*. EDiv is based on the execution frequency of each event in the finite state machine (FSM). PDiv is based on the execution frequency of each pair of semantically interacting

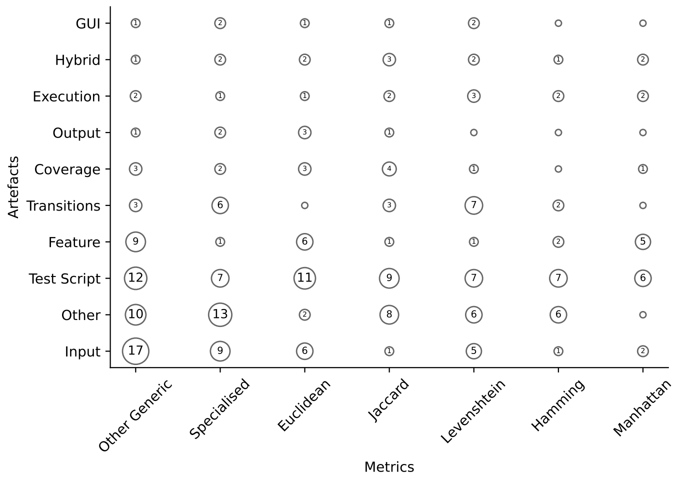


Figure 2.5: *Relationship between similarity metrics and diversity artefacts used in DBT techniques.*

events. TCov is based on the FSM coverage by the test cases. Biagiola et al. [35] used input diversity and navigation diversity for web testing. A test suite that covers the web application is generated using the distance metric between the test actions and input data, maximising the diversity of navigation sequence and input data. The distance metric used is composed of two parts. These are the difference between the action sequences in the two test cases, and the difference in input values utilised by the identical actions in those sequences.

2.5.21 Relationship between Similarity Metrics and Diversity Artefacts

Figure 2.5 shows the publications' relationship between similarity metrics and diversity artefacts. Test script artefact mainly uses generic similarity metrics, especially the most popular ones, which have been used almost in half the papers that used test script as a diversity artefact. Papers using transitions as diversity artefact mainly used Edit distance (G2) and specialised metrics (e.g. S1, S3, S4). Also, it is clear that generic metrics are used more often than specialised metrics.

Conclusions — RQ3: ARTEFACTS

DBT techniques used different software artefacts as a basis for the similarity calculations. These artefacts included input, output, test script, test execution, etc. The most used artefact reported in the literature was test inputs with 19.8% of the papers reported in this study. After that, 18.6% of the collected papers used test scripts as a basis for their approaches. Over one-third of the DBT techniques used either input or test script diversity, because both of them are simple to use as they are usually obtained without the need to instrument, run the tests, or build any models of the software under test, making them very appealing to use by testers.

2.6 RQ4: PROBLEMS

What are the software testing problems to which DBT techniques have been applied?

DBT techniques have been used in solving software testing problems, such as test data generation, test case prioritisation, etc. I explore the various studies made regarding these problems and discuss the impact of DBT techniques upon them.

Table 2.7 presents a list of the software testing problems addressed using DBT techniques, with a short description of the problem, the total number of papers solving them, and citation of all these papers. There are 66 studies focussing on test data generation, 33 concerning test case prioritisation, 22 studies about test case selection, 17 studies regarding test suite quality evaluation, 12 studies on test suite reduction and 8 studies with regard to fault localisation. Figure 2.6 shows the distribution of the papers tackling software testing problems. As it shown from the figure, almost 60% of the collected DBT papers deal with test data generation and test case prioritisation. Out of the 11 testing problems discussed, almost three-quarters of the DBT techniques are applied in test data generation, test case prioritisation, and test case selection. There are 10 other papers that introduced new similarity metrics [109, 225, 171, 233], investigated test reuse in software product lines [50, 47], utilised output diversity in boundary value testing [108, 90], used DBT in test adaptation [293], and argued for teaching diversity principles in software testing [69]. The paper by Matinnejad et al. [215] addressed both test data generation and test case prioritisation where it is cited in both problems in Table 2.7. All other papers in my study address only one type of problem in software testing.

2.6.1 Test Data Generation

Test data generation is the process of creating data to test a software program according to some adequacy criteria. It is the most researched area where DBT techniques have been proposed.

Table 2.7: *A list of the software testing problems addressed using DBT techniques.*

Rank/Problem	Description	Total	Papers
1 Test data generation	The process of creating data to test a software program according to some adequacy criteria.	66	[2, 8, 7, 10, 18, 20, 27, 33, 35, 37, 45, 43, 49, 51, 63, 70] [83, 110, 111, 114, 147, 146, 161, 162, 170, 175, 182, 185, 192, 195, 205] [208, 209, 210, 211, 213, 215, 221, 220, 235, 254, 261, 264, 265, 277, 282, 283, 284] [288, 299, 307, 308, 309, 310, 315, 313, 311, 314, 326, 328, 330, 333] [338, 340, 349, 350]
2 Test case prioritisation	The process of re-ordering test cases so that the tester gets maximum benefit in the case testing is prematurely stopped at some arbitrary point due to some budget constraints.	33	[1, 6, 21, 24, 29, 34, 53, 60, 247] [104, 112, 116, 126, 128, 129, 136, 144] [151, 168, 184, 206, 215, 219, 226, 229] [240, 281, 312, 323, 329, 335, 337, 345]
3 Test case selection	The process of selecting a subset of the current available tests that are relevant to some set of recent changes.	22	[4, 5, 26, 54, 86, 249, 246] [105, 119, 143, 142, 139, 140, 141, 212] [228, 255, 269, 287, 289, 290, 343]
4 Test suite quality evaluation	The process of judging on the quality of the current test suite and giving insight to testers about certain properties which can guide the tester to improve or fix some issues in the test suite.	17	[22, 28, 52, 248, 115, 127, 187, 236, 241] [231, 260, 244, 286, 291, 325, 327, 342]
5 Test suite reduction	The process of finding redundant test cases in the test suite and discarding them in order to reduce the size of the test suite.	12	[31, 42, 71, 78, 79, 80] [82, 97, 169, 253, 258, 306]
6 Fault localisation	The process of identifying the locations of faults in a program.	8	[122, 132, 133, 198] [200, 324, 336, 344]
7 New Metric	Introducing a new similarity metric that can be used to solve any of the software testing problems.	4	[109, 225, 171, 233]
=8 Test Reuse	Resuing part or the whole of a test suite on other systems.	2	[47, 50]
=8 Boundary value analysis	A software testing technique in which tests are designed to include representatives of boundary values in a range.	2	[90, 108]
=10 Test Adaptation	The process of identifying a test suite from another program that can be adapted to be used on the SUT.	1	[293]
=10 Teach DBT	Teaching software testing methods based on diversity principles.	1	[69]

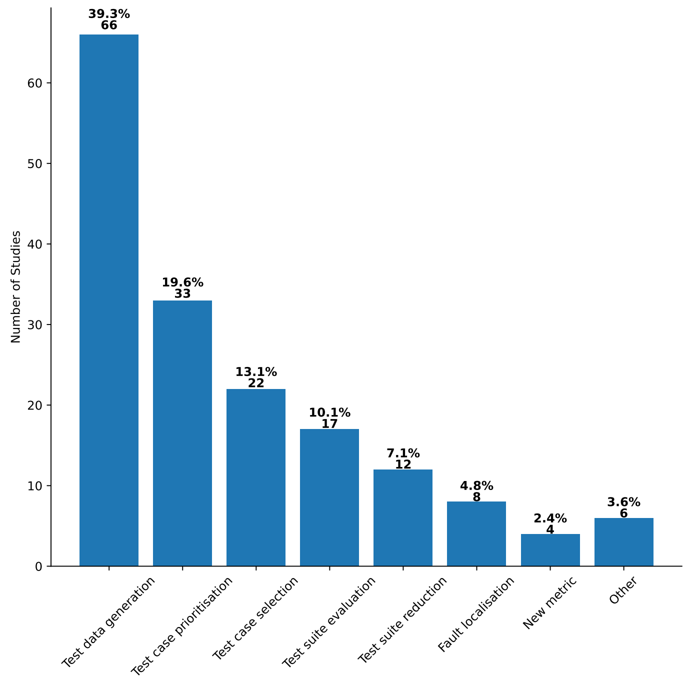


Figure 2.6: *Distribution of DBT papers addressing problems in software testing.*

Search-Based Approaches

Search-based approaches transform the testing objectives into search problems and solve these problems using an evolutionary process. The evolutionary process can use a single solution, such as hill climbing, or can use a population of individuals, such as genetic algorithms, that go through multiple iterations until the stopping criteria are met.

Researchers have applied diversity to construct individuals with higher fitness values in search-based approaches. Bueno et al. [45] employed a metaheuristic algorithm to evolve "randomly generated test sets" (RTS) towards "diversity oriented test sets" (DOTS), that aims to meticulously cover the input domain. They identified metaheuristics that can be utilised to generate DOTS and proposed an algorithm called "Simulated Repulsion" (SR). Bueno et al. [43] went on to implement that measure in a test data generation technique. They found that SR outperforms genetic algorithms, which in turn outperforms simulated annealing. The proposed SR found greater diversity values for the sets than genetic algorithms and simulated annealing in a lower number of iterations. However, the computational complexity of SR is higher than both simulated annealing and genetic algorithm. Boussaa et al. [37] used a Novelty Search (NS) algorithm for test data generation using statement coverage as an evaluation criterion. The test cases are selected based on their diversity, or based on a novelty score of how different the test cases are compared to all other solutions. This avoids the premature convergence that genetic algorithms suffer from.

Other approaches use fitness functions based on diversity metrics to guide the search-based algorithm in generating a diverse test suite. Shahbazi and Miller [283] investigated black-box testing methods where the input value is a string and used two fitness functions to guide the test case generation. The first fitness function was to control the diversity of the test cases and the other fitness function was to control the length distribution of the string test cases, and they employed a Multi-Objective Genetic Algorithm (MOGA) using both fitness functions to be minimised.

Some studies investigated the effects of population diversity in order to check if it helps in generating test suites achieving higher statement and branch coverage [8]. Albunian et al. [7] investigated the effect of holding the population diversity on the Many-Objective Sorting Algorithm (MOSA) for test data generation. Alshraideh et al. [20] proposed using multiple-populations rather than a single population to maintain the diversity of the solutions and to prevent the search process from being stuck in a local optima.

An alternative approach to assessing the diversity of a set of test inputs or the population diversity is to compare the resultant program execution traces. Feldt et al. [110] proposed a model for test variability referred to as VAT (VAriability of Tests), which I discussed in Section 2.5.20 (p. 38). The generated VAT trace of a test consists of all the values from the execution of a test with all the variation points in the VAT model.

A few more studies explored path diversity in evolutionary approaches. Xie et al. [330] used an evolutionary testing approach for test data generation for path-oriented testing. The fitness function is based on the similarity between the execution track and the target path. Panchapakesan et al. [254] proposed a white-box test data generation combining evolutionary strategy with an evolutionary algorithm based on path diversity. The control flow graph (CFG) of the software under test is given, and the similarity between the two paths is calculated from the CFG. Cai et al. [49] proposed a search-based test data generation algorithm using path coverage criterion called the "binary searching iterative algorithm". It is based on the idea that similar test cases usually cover similar paths. The proposed algorithm finds the closest (i.e. most similar) uncovered path to a discovered path to it, and then it searches for inputs left and right of the region of the already discovered path.

Fuzz testing

Fuzz testing, also known as fuzzing, is a software testing technique that involves feeding unexpected, random, and invalid inputs to a program to discover bugs, vulnerabilities, and other issues that might not be found through traditional testing methods. Fuzz testing is reported to improve the robustness and security of software [174, 303], and is widely used in industries such as cybersecurity, software development, and quality assurance. By generating a large number of test cases, fuzz testing helps uncover errors and weaknesses that might go unnoticed in traditional manual or automated testing, thereby helping to enhance the overall quality and reliability of software products. Zhang et al. [340] proposed a fuzzing approach that begins with similarity-based black-box fuzzing for test data generation, selects a subset of data based on similarity metrics, and generates new inputs using combination testing. Wang et al. [313] introduced an adaptive fuzzing method that reduces overall similarity by applying bit-flip mutations to highly similar bytes, using Word Rotator's Distance (WRD) for measurement. Lan et al. [182] used seed diversity, representing seeds as byte sequences, clustering similar sequences, and selecting diverse seeds to improve coverage. Lee et al. [185] also proposed a seed diversity-based fuzzing method, using syntactic and semantic similarity (measured by variants of Hamming and Jaccard distances, respectively) to cluster seeds.

Other researchers used fuzz testing for property-based testing, which is a software testing approach that focuses on checking the validity of general properties or invariants of a system. Instead of writing specific test cases, property-based testing uses properties, which are high-level assertions or conditions that should hold true for a wide range of inputs. Reddy et al. [264] proposed a black-box approach for quickly generating diverse, valid test inputs in property-based testing. Nguyen and Grunske [235] focused on increasing behavioural diversity and found their method superior to fuzzers like Zest [252] and RLCheck [264]. Menendez et al. [220] introduced *HashFuzz*, which combines coverage and diversity using hash functions to create diverse tests for C programs. Wang et al. [311] employed a diversity-based fuzzing method to accelerate fuzzing by terminating test sequences similar to non-crashing ones.

Fuzzing was used to test Deep Learning (DL) systems to generate diverse inputs. Dai et al. [83] proposed two diversity-driven *seed queue* construction approaches for fuzzing DL systems. Starting with an existing test suite, they randomly mutate some tests to generate more data. The seed queue is built by randomly selecting the first seed, then choosing k-random seeds to form a candidate set. The candidate with the highest distance from others is added to the queue. Both approaches outperformed random fuzzing and test set diameter (TSDm), with feature-based diversity showing slightly better results.

Model-based testing

Researchers have also applied diversity in model-based testing. Asoudeh and Labiche [27] proposed a genetic algorithm for test data generation of extended finite state machines (EFSMs) to maximise coverage and test case diversity and minimise overall cost. The fitness function guides the search, and it is defined based on the similarity of the individual solutions.

In Cyber Physical Systems (CPSs), models have time-continuous behaviour. Therefore, test inputs and outputs are defined as continuous signals representing values over time. Matinnejad et al. [213] proposed an approach for test data generation of Simulink models. The proposed approach aims to maximise diversity in output signals rather than maximising structural coverage, on the basis that test cases that yield diverse output signals are more likely to reveal different types of faults [213]. In a later study, Matinnejad et al. [215] modified the test data generation approach to be compatible with continuous behaviours, to fix unrealistic assumptions about test oracles, and more scalable using meta-heuristic search.

Deep Learning (DL) software has become widely used in many applications, and few works applied DBT techniques to generate test inputs for DL software. Jiang et al. [161] proposed an approach for generating valid and diverse test inputs for a testing task-specific DNN.

Focused testing

Sometimes it is important to generate a test suite that focuses on a specific part of the program under test. This part could be new code, or modified code. Also, I might want to focus on comparing new and previous functionality. This kind of testing is referred to as *focused testing.* The main issue with focused testing is that the inputs are not diversified enough. Menendez et al. [221] proposed an approach named *Diversified Focused Testing* that uses a search technique generating a diverse set of tests to cover a given program point (pp) repeatedly to reveal faults affecting the given pp.

Web testing

Web testing is the process of testing web-based applications to ensure that they function correctly and meet the intended requirements. Some researchers used DBT techniques in web testing. Marchetto and Tonella [208] used search-based testing to maximise test case diversity for exploring the large space of long interacting sequences in Ajax web applications, selecting only the most diverse cases. Alshahwan and Harman [18] proposed a black-box test criterion based on Output Uniqueness and evaluated it on six web applications. Later, Benito et al. [33] introduced "Output Diversity Driven" (ODD) unit test generation, showing that ODD achieves 50% higher output uniqueness and 63% better fault detection than "Input Diversity Driven" approaches.

A state flow graph can model a web application's navigation, with nodes representing dynamic DOM states and edges representing events triggering page transitions. Biagiola et al. [35] presented a DBT web test generation algorithm that selects the most diverse test cases, achieving better coverage, fault detection, and speed than crawling-based and search-based generators. Wang et al. [315] proposed a parallel evolutionary test case generation approach that enhances population diversity for web application testing. This parallel execution reduces test generation time and maintains diversity to avoid premature convergence.

Mobile applications testing

With the rise of mobile devices and apps, mobile application testing has become crucial to ensure apps function properly and meet requirements. Diversity concepts can enhance testing in this area. Vogel et al. [307] investigated diversity in test data generation for mobile apps using the SAPIENZ tool, which employs the NSGA-II heuristic. They found that SAPIENZ loses solution diversity over time, leading to premature convergence. To address this, Vogel et al. [308] proposed SAPIENZ^{DIV}, incorporating four mechanisms to maintain diversity. SAPIENZ^{DIV} achieved better or equal coverage compared to SAPIENZ, though it required more execution time.

Directed random testing

Pacheco et al. [251] proposed a technique called feedback-directed random test generation and implemented a tool called "Randoop" [250, 164] that generates unit tests for object-oriented programs. The main difference between this technique and random testing is the use of feedback from previously generated tests. However, one problem with Randoop, is that code coverage stops increasing after some point, due to the inability of the technique to find any new interesting tests that can contribute to the coverage [333]. Yatoh et al. [333] extended this work and proposed a method named *feedback-controlled random test generation*, which limits and controls the feedback to increase the diversity of generated tests. They used multiple pools simultaneously rather than a single pool, with different pools having different contents guiding the test generation into more diverse test cases.

Multiple dispatch language testing

In a multiple dispatch language, many methods can have the same name but differ in the number or type of parameters of the methods. The correct method call is determined by the data types of the actual arguments passed to the method. This concept is referred to as *function overloading* in many programming languages. Poulding and Feldt [261] proposed a probabilistic approach for generating diverse test inputs for multiple dispatch languages. They made an empirical study to evaluate the approach using 247 methods across nine built-in functions of the Julia programming language and found three real faults.

GUI testing

A Graphical User Interface (GUI) application is an event-driven software, where the input is a sequence of events that changes the state of the application, and then produces an output. The events can be clicks on buttons, messages in text fields, etc. A GUI test is made up of a sequence of events interacting with the GUI and some assertions on the GUI state. Feng et al. [111] proposed a 3-way technique for GUI test case generation based on event-wise partitioning. The event- wise partitioning is performed according to the level of similarity between the GUI states. They use k-means algorithm to cluster the states. Mariani et al. [210] made an empirical study on the reuse of GUI tests based on semantic similarities of GUI events. They proposed a semantic matching algorithm named *SemFinder*, and showed that it outperformed other algorithms. Events with the highest similarity score or all events above a certain threshold can be considered in the test reuse approach.

Compiler testing

Compilers like other software may contain bugs, and warning diagnostic messages could be faulty or missing. A specific type of testing called, compiler testing, is used to detect compiler bugs. An important task in compiler testing is to generate test programs that are able to reveal bugs. Chen et al. [63] proposed an approach called *HiCOND* (**Hi**story-Guided **CON**figuration **D**iversification) for test-program generation based on historical data that are bug-revealing and diverse in the sense that they are capable of revealing many types of bugs. In a latter study, Li et al. [192] developed an approach for testing Simulink models called *RECORD* (**R**einforcement l**E**arning-based **CO**nfigu**R**ation **D**iversification). The approach found eight more bugs than HiCOND. Another study by Tang et al. [299] proposed an approach to build diverse warning-sensitive programs to detect compiler warning defects. They named the approach DIPROM (**DI**versity-guided **PRO**gram **M**utation approach). They showed that DIPROM reveals more bugs by 76.74% than HiCOND, 34.30% than Epiphron, and 18.93% than Hermes all in less time.

Test case recommendation

Similar code structures tend to have similar test cases and co-evolve together, some researchers made use of this pattern to generate test cases for structurally similar methods. Shimmi and Rahimi [288] proposed such an approach that assumes an existing test suite that is either incomplete or outdated. In the former case, the approach recommends additional test cases to complete the test suite, or in the latter case, change the current test cases to adapt to the changes introduced in the code.

Highlights

Here are the highlights of DBT in test data generation:

- **Evolutionary processes**: DBT was used in search-based testing, such as hill climbing or genetic algorithms, to enhance test set diversity, aiming to improve coverage and avoid premature convergence [45, 43, 37, 7, 20, 330].
- **Fuzzing**: DBT was applied in fuzzing to improve coverage using behavioural diversity [235] or seed diversity and clustering based on similarity measures [311, 83].
- **Model-based testing**: DBT was used for testing extended finite state machines [27] and Simulink models [213, 213, 215] to enhance coverage and fault detection.
- **Web and mobile testing**: DBT was used to enhance web fault detection using output diversity [18, 33], and population diversity [315] to reduce test generation time. Also, diversity was utilised for testing mobile apps to improve coverage and fault detection et al. [307, 308].

- **Compiler testing**: Chen et al. [63] proposed HiCOND for test-program generation based on historical bug data. Li et al. [192] introduced RECORD, and Tang et al. [299] developed DIPROM, both finding more bugs than HiCOND.

2.6.2 Test Case Prioritisation

Test case prioritisation (TCP) re-orders test cases so that the tester gets maximum benefit in the case testing is prematurely stopped at some arbitrary point due to some budget constraints [334]. DBT techniques have been used in many test case prioritisation papers.

Prioritisation based on test cases

Some techniques use test cases themselves to guide prioritisation. Ledru et al. [184] proposed a prioritisation algorithm that enhances fault detection by maximising test diversity, using a greedy approach to select the most distant test cases based on a fitness function. Wu et al. [323] aimed to improve similarity-based prioritisation by considering the execution times of program elements in ordering test cases. Fang et al. [104] focused on ordered execution sequences for prioritisation. Wang et al. [312] proposed a branch coverage similarity-based approach using six similarity measures, finding Euclidean distance most effective. Khojah et al. [168] compared lexical and semantic diversity, finding semantic diversity superior for requirements coverage.

Prioritisation can be achieved by measuring how much a test case is covering a *changed* part of the code. Altiero et al. [21] proposed a similarity-based method for test case prioritisation capable of measuring structural similarity of changed code. Each test case is assigned a score based on the number of modified methods it covers, and the test suite is then sorted in descending order according to these scores.

Scalability and performance are critical for industrial applications. Miranda et al. [226] introduced the FAST family of test case prioritisation techniques, which are quick and scalable for large test suites. They use big data techniques like Shingling, Minhashing, and LSH for efficient similarity detection, applicable to both white-box and black-box testing. Greca et al. [126] later compared test case selection using Ekstazi [120] with test case prioritisation using FAST [226], finding that combining these approaches improved test suite efficiency by selecting and then prioritising tests.

Manual black-box system testing is a type of software testing in which the tester evaluates the functionality of a software application without having any knowledge of its internal workings or source code. Hemmati et al. [144] modified existing test case prioritisation techniques to work on manual black-box system testing. They implemented three different techniques for test case prioritisation based on topic coverage, diversity, and riskiness. They evaluated the using multiple Mozilla Firefox releases. Their results show that in rapid-release environments, risk-based prioritisation achieves up to 65% higher APFD, while no technique consistently outperforms the others in traditional development settings.

Some empirical studies have evaluated the effectiveness of DBT test case prioritisation (TCP) techniques. Huang et al. [151] studied 14 similarity measures and compared local and global TCP techniques, finding that global TCP outperformed local TCP in coverage and fault-finding capabilities. Haghighatkhah et al. [128] investigated whether similarity-based TCP (SBTP) is more effective than random ordering in locating faults and identified the best SBTP implementation. They tested Manhattan distance, Jaccard distance, Normalised Compression Distance (NCD), NCD-Multisets, and Locality Sensitive Hashing, using greedy algorithms based on Ledru et al.'s [184] work.

An alternative idea is based on the rationale that a previous faulty test case is more likely to reveal faults again. Thus, test cases that are similar to these previously faulty test cases have higher chance of detecting new faults as well. Noor and Hemmati [240] defined a class of metrics to measure the similarity of test cases to previously failing test cases, and proposed a test case prioritisation approach based on this metric. They used three distance functions to measure the similarity between test cases, which are Basic Counting, Hamming distance, and Levenshtein (Edit) distance.

Prioritisation based on test steps

"Test steps" refer to the individual actions or procedures that need to be followed in order to execute a test case. Test steps are typically written in a clear and concise manner and provide detailed instructions on how to carry out the test case. Flemstrom et al. [116] proposed ordering test cases based on their similarities in terms of test steps to reduce the cost of translating manual tests into executable code. The automation effort of translating the test steps into executable code can be reduced if the executable code of a test step can be reused for similar test steps.

Prioritisation based on program executions

Some researchers designed their prioritisation techniques based on program execution diversity. Gomes de Oliveira Neto et al. [247] proposed a test case prioritisation approach based on the behavioural diversity of test cases, and referred to it as *b-div*. They compared the *b*-div measures against usual diversity metrics between artefacts (*a-div*), such as test scripts, inputs, or outputs and found that *b*-div measures found more mutants than *a*-div measures in all subjects.

Prioritisation for deep learning software

Deep Learning (DL) is increasingly used in various applications, necessitating input prioritisation to quickly identify violations or incorrect predictions and reduce running costs[229, 329]. Some prioritisation approaches involve DBT techniques. Mosin et al. [229] compared white-box, data-box, and black-box approaches for DL systems, using surprise adequacy, autoencoder-based methods, and similarity-based methods, respectively.

Since Metamorphic Testing helps address the oracle problem in DL systems, another prioritisation method is based on the diversity between Metamorphic test case Pairs

(MPs). Xie et al. [329] proposed a DBT prioritisation technique for MPs, suggesting that diverse execution patterns between source and follow-up cases increase the likelihood of detecting violations.

Prioritisation for WS-BPEL programs

WS-BPEL (Web Services Business Process Execution Language) programs describe interactions between web services using XML. They include activities like invoking web services, data manipulation, and control structures. Mei et al. [219] proposed a similarity-based prioritisation technique for regression testing WS-BPEL programs, using tree edit distance and Jaccard distance to measure XML document similarity. Bertolino et al. [34] introduced a similarity-based approach for prioritising XACML access control test cases, employing two distance functions: a modified Hamming distance, referred to as simple similarity, and a metric considering both parameter values and XACML policies, referred to as XACML similarity.

Prioritisation in CI environments

Some studies investigated diversity-based test case prioritisation in Continuous Integration (CI) environments. Haghighatkhah et al. [129] proposed using diversity-based (DB-TCP) and history-based test case prioritisation (HB-TCP). They used three similarity measures in their approach, which are Manhattan distance, normalised compression distance (NCD), and NCD-Multiset. The NCD-Multiset was the best in terms of effectiveness, but it is slower to calculate than the other two measures. Also, Azizi [29] proposed a technique for regression test case prioritisation that selects test cases based on the test case's textual similarity to the changed parts of the code using Cosine Similarity measure. However, if a test case can reveal an error, but does not have matching terms with the changed code, that test case can be ignored in the ordering.

Clustering-based prioritisation approaches

Some studies have used clustering-based approaches for TCP. These methods group similar test cases, prioritise within each cluster, and construct a final list by selecting cases from each cluster, often in a round-robin fashion. Yoo et al. [335] proposed clustering based on dynamic runtime behaviour using Agglomerative Hierarchical Clustering. Test cases within clusters are prioritised by coverage and human experts, and the final suite is assembled by sequentially selecting the highest-ranked cases from each cluster. In another paper, Carlson et al. [53] used code coverage, code complexity, and fault history for clustering with Euclidean distance and Agglomerative Hierarchical Clustering. Test cases within clusters are ordered by these factors, and the final list is constructed by selecting cases from each cluster in a round-robin fashion. Zhao et al. [345] introduced a hybrid method combining code coverage similarities with Bayesian Networks. Test cases are clustered based on code coverage and prioritised using Bayesian Networks, which includes source code change information, test coverage data, and software quality metrics. The final test suite is created by selecting cases from each cluster in a round-robin method.

String Distance test case prioritisation (SD-TCP) ranks test cases based on similarity but can be skewed by test case length. Chen [60] proposed KS-TCP, which uses K-medoids clustering to group similar test cases and then applies a string distance-based prioritisation within clusters. Yu et al. [337] introduced a TCP method that leverages log analysis and test case diversity, using k-means clustering with Euclidean distance to group test cases based on features from log data.

An alternative clustering method groups test cases based on their requirements, prioritising them by client importance and code modifications. Arafeen et al. [24] implemented this approach, ranking clusters by requirement importance and selecting test cases based on cluster rank. Higher-ranked clusters contribute more test cases.

Some researchers cluster test cases based on coverage and fault-proneness. Mahdieh et al. [206] used hierarchical clustering and similarity measures (Euclidean, Manhattan, Cosine) on coverage data and predicted fault-proneness. Test cases within each cluster are prioritised by coverage, and the final list combines these priorities with fault-proneness.

Highlights

Here are the highlights of DBT in test case prioritisation:

- **Test script based**: Some approaches used the existing test suite to use a diversity-based TCP to improve average percentage of fault detection (APFD) [184, 104, 168, 323].

- **Clustering based**: DBT was applied in clustering-based TCP based on runtime behaviour [335], based on code complexity and fault history [53], based on coverage and fault-proneness [206], based on log analysis [337], or based on requirements and code modifications [24].

- **Empirical studies and comparisons**: Huang et al. [151] compared 14 similarity measures and found global TCP more effective than local TCP for coverage and fault detection. Haghighatkhah et al. [128] evaluated various similarity-based TCP methods to determine their effectiveness in fault location using greedy algorithms based on Ledru et al.'s [184] work. Mosin et al. [229] compared white-box, data-box, and black-box approaches for DL systems, using surprise adequacy, autoencoder-based methods, and similarity-based methods, respectively.

- **Scalability and performance**: Miranda et al. [226] introduced the FAST family of prioritisation techniques, using big data methods for efficient similarity detection in large test suites. Greca et al. [126] compared FAST [226] with Ekstazi [120] for test case prioritisation and selection, finding that combining these methods improved efficiency.

- **Continuous Integration (CI) Environments**: Some investigated diversity-based TCP in CI environments based on diversity and history [129], or based on textual similarity to the changed parts of the code [29].

2.6.3 Test Suite Reduction

Test suite reduction (TSR) techniques eliminate redundant test cases to reduce test suite size [334]. Coviello et al. [79] defined "adequate" TSR as maintaining test coverage and "inadequate" TSR as failing to do so. Their study found that inadequate approaches provide a better trade-off between size reduction and fault detection. Cruciani et al. [82] proposed scalable TSR methods using big data techniques, where testers specify the desired number of test cases and select them from a D-dimensional space based on even spacing.

Black-box techniques that relies only on the code of test cases were used for test suite reduction. Elgendy et al. [97] maximised textual diversity between test cases to reach a desired test suite size. Pan et al. [253] proposed a similarity-based, search-driven test suite reduction technique that used tree-based similarity measures as a fitness function to guide the genetic algorithm.

TSR techniques often use clustering to group similar test cases and select representatives from each cluster. Coviello et al. [80] proposed a method where similar test cases are clustered based on statement coverage, selecting the most comprehensive case from each cluster. Beena and Sarala [31] developed a multi-objective approach focusing on maximising coverage and minimising execution time using a similarity matrix for clustering. Chetouane et al. [71] combined k-means clustering with binary search to optimise cluster numbers while maintaining test suite effectiveness. Viggiato et al. [306] used clustering and text similarity to identify and reduce redundant manual testing steps based on natural language similarities.

Diversity has been used for test suite reduction in model-based testing. Coutinho et al. [78] employed a similarity-based approach with a matrix to identify and reduce redundant test cases by selecting pairs with the highest similarity.

Another form of reduction is to *"tame"* the inputs. Taming the fuzzer means controlling the large number of test cases generated, especially the tests that keep triggering the same bug. Pei et al. [258] introduced delta debugging to "tame" the fuzzer, focusing on retaining diverse test cases using the Furthest Point First (FPF) algorithm. This approach aims to reduce the number of test cases by maximising diversity, with distances measured using Euclidean metrics.

Some other works defined a set of test metrics to calculate the diversity between pairs of test cases in a regression test suite that can be based on interaction behaviour models [169], or based on block coverage criteria, control flow of a program, variable definition-usage, and data values [291].

2.6.4 Test Case Selection

Test case selection involves choosing a subset of available tests relevant to recent changes [334]. Rogstad et al. [269] proposed a black-box approach using similarity for large-scale database applications, dividing the input domain with classification trees and selecting test cases from each partition.

In some cases, emergency changes have to be made to the software quickly, but running the entire test suite can be time-consuming, and selecting a subset can be risky. Farzat [105] suggested a heuristic method to maximise coverage and diversity, reducing the risk of partial testing. Gomes de Oliveira Neto et al. [246] studied Continuous Integration (CI) and/or Continuous Delivery (CD) pipelines, using similarity functions like normalised Levenshtein, Jaccard distance, and normalised compression distance to optimise test selection. Ahmad et al. [6] later expanded on using DBT techniques for prioritising CI pipelines.

In a recent study, Shimari et al. [287] proposed a clustering-based test case selection using the execution traces of test cases for an industrial simulator. The goal was to select a subset of the regression test suite that is capable of achieving high code coverage and diverse runtime executions for the simulation (i.e. short and long simulations to reflect regular usage patterns). The case study showed that the proposed approach selects test cases with high coverage and high diversity of execution time.

Multi-objective algorithms

Multi-objective algorithms have been used for test case selection in a number of studies [86, 228, 255]. De Lucia et al. [86] discussed the concept of asymmetric distance preserving, useful to improve the diversity of non-dominated solutions produced by multi-objective Pareto efficient genetic algorithms. They proposed a multi-objective algorithm for test case selection by increasing population diversity in the obtained Pareto fronts. Mondal et al. [228] proposed using coverage-based and DBT techniques using NSGA-II multi-objective algorithm maximising both coverage and diversity of test cases in the context of test case selection. Panichella et al. [255] suggested an improvement in multi-objective genetic algorithms (MOGAs) by diversifying the solutions of the population for test case selection. They proposed DIV-GA (**DIV**ersity based **G**enetic **A**lgorithm), that inject new orthogonal individuals to increase diversity during the search process.

Model-based test selection

Model-Based Test Selection (MBTS) involves choosing a subset of test cases that adequately cover a system's functionality and requirements using a model of the system. DBT techniques are often utilised in this context. Cartaxo et al. [54] and Gomes de Oliveira Neto et al. [249] proposed similarity-based approaches for MBTS, detecting code modifications through test case similarities. Hemmati et al. [143, 139] introduced a similarity-based test case selection (STCS) for UML state machines using a

Genetic Algorithm, focusing on test case diversity and "Trigger-Based" similarity. Arrieta et al. [26] suggested a black-box approach for Simulink models with six effectiveness measures and similarity metrics based on input and output signals.

Empirical studies by Hemmati and Briand [142], and Hemmati et al. [140], explored the effectiveness of similarity-based techniques in industrial software. They found that these techniques are most beneficial when similar test cases detect common faults, and dissimilar cases detect distinct faults. Hemmati et al. [141] later introduced 320 STCS variants, optimised using different parameters, which reduced costs by $50 - 80\%$ and improved fault detection rates by up to 45% compared to coverage-based selection and 110% compared to random selection.

Deep neural network

Testing of Deep Neural Networks (DNNs) is a critical aspect of ensuring their reliability and accuracy in real-world applications. Zhao et al. [343] made an empirical study to find out the level of diversity of test input selection for deep neural network (TIS-DNN) methods. The level of diversity here means whether some classes in the original test set may have been missed by the selected subset. The test diversity is measured through the accuracy-based performance measure $AccEE_{avg}$, which focuses on the average value of accuracy over all classes. In another study, Gao et al. [119] proposed an adaptive test selection method for DNN models using output diversity. The selection process aims to evenly select test cases with maximum fitness values from a set of candidates. Aghababaeyan et al. [4] proposed a black-box strategy to test DNNs based on the diversity of inputs' features rather than the traditional neuron coverage criteria. The approach can be applied to address multiple areas of testing DNNs, including test case selection. In a later study, Aghababaeyan et al. [5] developed a test case selection approach called "DeepGD".

2.6.5 Test Suite Quality Evaluation

Test suite quality evaluation techniques aim to judge on the quality of the current test suite and give insight to testers about certain properties which can guide the tester to improve or fix some issues in the test suite. Diversity of tests is one of the properties that testers generally aim to have in the test suites. Nikolik [236] proposed a method for measuring the level a test suite executes on a program in diverse ways with respect to control and data, and implemented a tool named *The Diversity Analyzer*. Shi et al. [286] proposed a black-box metric for test suite quality evaluation referred to as *distance entropy* based on the diversification concept. They represented the test cases as nodes in a complete graph where the weights of the edges are the distance between each pair or node. The distance between the pairs of test cases is measured using the learned distance metric.

Many empirical studies were made to investigate the importance of diversity in test suites [327, 342, 115] and some of its challenges [248]. Xie and Memon [327] made an empirical study to investigate the characteristics of "*good*" GUI test suite evaluating the testing cost and fault detection effectiveness. They found that the diversity of states and the event coverage are the two major factors of fault-detection capabilities. Zhang and Xie [342] conducted an empirical study to investigate the essential diversities to be used as testing criteria for deep learning systems. They defined five metrics and leveraged metamorphic testing to reveal faulty behaviours. Flemstrom et al. [115] made an industrial case study on four real-world industrial projects in the railway domain to investigate if test overlaps exist and how they are distributed over different test levels, and reduce test effort. Test overlaps mean a high level of similarity between test cases, such that no additional value can be had by including the redundant test cases. Gomes de Oliveira Neto et al. [248] investigated some challenges of using diversity information for developers and testers since results are usually many-dimensional.

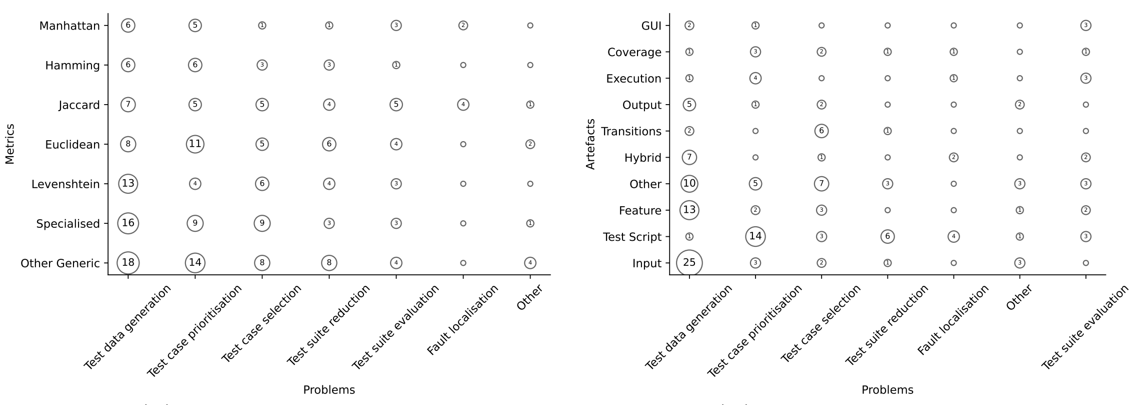
In another work, DBT technique was used to help the tester to evaluate a test plan. Aman et al. [22] proposed a similarity-based method for test case recommendation during a regression testing. Once the tester decides which tests to run, the method adds into the set more candidates that are similar to the selected tests using Jaccard distance.

Exploratory testing (ET) is a software testing approach that complements automated testing by leveraging business expertise, and they are defined and executed on-the-fly by the testers [188], and can be considered as an evaluation technique. The main goals of ET are to find new bugs not detected using manual or automated tests, and improve the quality of the system under test. Leveau et al. [187, 188] proposed an approach to help testers explore any web application using Exploratory Testing. They used a prediction model based on n-gram language models, which tracks the online interactions that the testers do and based on the prediction model gives probabilistic future interactions. Testers who select lower probability interactions will increase the exploration diversity. Thus, achieving deeper investigation of the web application. In another paper, Nass et al. [231] proposed a similarity-based approach for web element localisation to facilitate the test automation. The approach aids the tester to determine the cause for why the test execution fail for no apparent reason.

Diversity-based approaches were used in assessing deep neural networks (DNNs). Attaoui et al. [28] made a clustering-based analyses that group together similar failure-inducing inputs in DNN testing, that uses a feature-based distance using Euclidean distance. The approach can identify root causes of DNN errors and aid tester to retrain and improve the DNN.

2.6.6 Fault Localisation

Fault localisation (FL) is the process of identifying the locations of faults in a program [322]. In this section, I cover testing-based fault localisation, which utilises the testing information to localise the faults [132], and in particular DBT techniques. Hao et al. [132, 133] proposed a technique named *similarity-aware fault localisation* (SAFL) by using the theory of fuzzy sets to evenly distribute the test cases. Zhao et al. [344] proposed a fault localisation framework to make use of execu-



(a) *Similarity metrics* **(b)** *Diversity artefacts*

Figure 2.7: *Relationship between DBT problems and similarity metrics (a) and diversity artefacts (b), respectively*

tion similarities of test cases to improve coverage-based fault localisation techniques. Gong et al. [122] presented a Diversity Maximisation Speedup (DMS) for test case prioritisation to address the problem of test cases without manual labelling. DMS requires testers to label much fewer test cases to achieve a competing fault localisation accuracy with other techniques thus reducing the cost of the test. Xia et al. [324] used the same DMS approach to locate faults in single-fault and multi-fault programs. They compared DMS with six test case prioritisation techniques on 12 *C* programs and found that DMS can reduce the cost while keeping the same accuracy of fault localisation. Later, You et al. [336] discussed a number of modified similarity coefficients in fault localisation to evaluate the significance of failing and passing test cases in similarity coefficients. The failing test cases provide more important information for fault localisation, and therefore it is given more weight than passing test cases.

Test cases have an important role in fault localisation, and adding more test cases to the test suite can help refine the suspiciousness scores of the statements. However, increasing the number of test cases adds to the cost of execution time and effort of the manual test oracle. Therefore, having a diverse and small number of test cases can improve fault localisation while reducing the cost. Liu et al. [198] proposed an approach of test case generation for fault localisation of Simulink models. They developed a prediction model based on supervised learning that prevents adding more test cases that are unlikely to improve fault localisation. Liu et al. [200] extended this work by adding a new test objective based on output diversity.

2.6.7 Relationship between DBT Problems and Diversity Artefacts and Similarity Metrics

Figure 2.7a shows the relationship between similarity metrics and software testing problems using DBT techniques. Fault localisation mainly used Jaccard (G3) and Manhattan (G5) distances. Over half the papers in test case prioritisation used the top generic metrics (G1-G5). Also, the top three generic metrics (G1-G3) are the most used similarity metrics in test data generation.

Figure 2.7b shows the relationship between diversity artefacts and software testing problems using DBT techniques. Input and feature diversity were primarily used for test data generation, test script diversity for test case prioritisation, and transition diversity for test case selection. Most papers using output or hybrid diversity artifacts applied DBT to test data generation.

Conclusions — RQ4: PROBLEMS

The collected papers in this study handled a range of software testing problems including: test data generation, test case prioritisation, test suite reduction, test case selection, test suite quality evaluation, and fault localisation. Test data generation is the most researched problem in software testing where DBT techniques have been used in 39.3% of the papers. Test case prioritisation comes next after test data generation in terms of the number of papers using DBT techniques with 19.6% of the papers. Followed by test case selection, test suite quality evaluation, test suite reduction, and fault localisation with 13.1%, 10.1%, 7.1%, and 4.8%, respectively. In test data generation, DBT techniques achieved an overall higher mutation scores than random testing [283] with an increase of up to 30% [43], assisted in preventing the search process from stagnating in search-based approaches [20, 7, 315], showed a significant increase in crash detection of between 28% to 97% [220], and decreased the average generation time of valid tests in deep learning systems by 15.7% to 47% [161]. Furthermore, DBT techniques increased the overall average percentage of fault detection compared to other test case prioritisation techniques [226, 144, 128], improved fault-detection rate by up to 45% over the coverage-based selection and 110% over the random selection [141], and achieved 82.2% test suite reduction maintaining both coverage and mutation score of the original test suites [71].

2.7 RQ5: SUBJECT DOMAINS

To what subject domains have DBT techniques been applied?

DBT techniques are applied across various domains, with each domain influencing the specific approaches used. This section highlights key domains where DBT has been explored. Table 2.8 summarises these domains, including descriptions, paper counts, and citations. Figure 2.8 shows the distribution of DBT applications across these domains. All the papers in my study fall under one of the following subject domains.

2.7.1 Console Programs & Libraries

Stand-alone programs and libraries have received significant attention in the context of DBT across various research areas. Notably, DBT techniques have been applied

Table 2.8: *A list of the software application domains addressed using DBT techniques.*

Rank/App	Description	Total	Papers
1 Stand-Alone	Console programs and libraries that are written to perform any specific task on a local machine.	113	[1, 6, 8, 7, 10, 20, 21, 22, 24, 29, 31, 33, 34, 37, 42, 45] [43, 47, 50, 49, 52, 51, 53, 69, 60, 71, 79, 80, 82, 86, 246, 248] [247, 90, 97, 104, 105, 110, 109, 108, 112, 114, 115, 116, 122, 126, 127, 128] [129, 132, 133, 140, 147, 146, 151, 169, 170, 168, 182, 184, 185, 198, 200, 205, 206] [209, 211, 219, 221, 220, 225, 226, 228, 235, 236, 240, 241, 244, 254, 255, 253] [258, 260, 261, 264, 281, 282, 283, 284, 286, 287, 289, 290, 288, 291, 293, 306, 309] [310, 312, 323, 324, 325, 326, 328, 330, 333, 335, 336, 337, 340, 344, 345]
2 NN Models	The heart of deep learning algorithms which are inspired by the human brain, mimicking the way that biological neurons signal to one another.	18	[4, 5, 2, 28, 83, 119, 161, 162, 171] [175, 229, 265, 329, 338, 343, 342, 349, 350]
3 Model-Based	Systems represented as models like FSM or Simulink models.	12	[26, 27, 54, 78, 249, 143] [142, 139, 141, 212, 213, 215]
4 Web	Applications that run on the web browser.	10	[18, 35, 144, 187, 188] [195, 208, 231, 315, 313]
5 GUI	A program with a graphical user interface that has labels, buttons, text fields, and other widgets which a user can interact with.	4	[111, 136, 210, 327]
=6 Autonomous Vehicles	Systems running autonomous vehicles, which perform various driving tasks, such as perception, localisation, planning, and control.	3	[70, 233, 311]
=6 Compilers	A program that translates a programming language's source code into machine code, bytecode or another programming language.	3	[63, 192, 299]
=6 Mobile	Applications designed to run on a mobile device.	3	[307, 308, 314]
9 Database	Applications that process and manipulate data stored and work as a database management system.	1	[269]

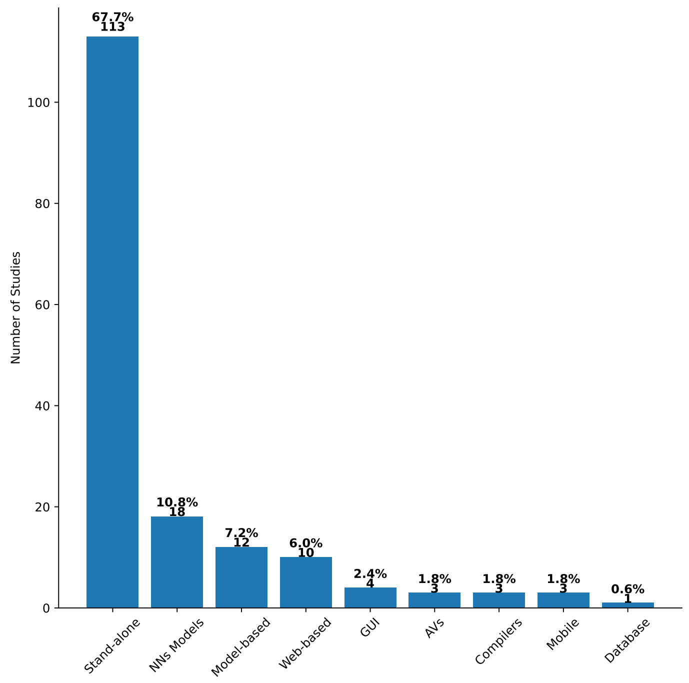


Figure 2.8: *Distribution of the subject domains of the DBT papers.*

in search-based software testing [45, 43, 110, 330, 283, 7, 37, 49, 254], mutation testing [289, 290, 281, 247, 260], continuous-integration environments [246, 129, 29], test case recommendation [288, 22], and machine translation systems [51].

A number of papers have explored the incorporation of diversity into fuzzers, particularly in terms of input generation. These include the utilisation of reinforcement learning to guide and control the input generator [264], used feedback-driven fuzzing [235], and exploration of deep program semantics [340]. Other approaches have aimed to enhance the diversity of tests in general [220], or reduce the number of generated tests [258].

Most papers used stand-alone programs or libraries in their experiments, but many proposed approaches are applicable to other domains. Notable datasets and frameworks used include Defects4J [164] (Java), Software-artifact Infrastructure Repository [89] (C), TravisTorrent [32] (continuous integration platforms), NL-RX-Synth [203] (regular expressions for natural language), and BugsInPy [319] (Python).

2.7.2 Model-Based Systems

DBT techniques have been employed in the field of model-based testing. Noteworthy contributions include the generation of test suites from finite state machines [27], test suite reduction for MBT [78], test case selection for MBT [54, 143, 142, 139, 141, 249], and the application of DBT techniques to Simulink models [212, 213, 215, 26, 198].

Most papers used case studies for evaluation but did not provide them. For example, Hemmati et al. [143, 142, 139, 141] used a C++ safety-critical control system that is not available. Public domain Simulink models, such as the Cruise Controller [155] and Clutch Lockup Controller from Mathworks [154], were used. Additionally, industrial Simulink models, Clutch Position Controller and Flap Position Controller, developed by Delphi Automotive Systems, were used [212, 213].

2.7.3 Web Applications

The realm of web testing has witnessed the utilisation of DBT techniques, as discussed in Section 2.6 and Section 2.5. Certain characteristics specific to web applications, such as the document object model (DOM) [35, 231], HTML tag structure [18], long interacting sequences [208], exploration of web applications [187, 188], and challenges related to string-matching-based rules in crawling-based web application testing [195], influence the application of DBT techniques. Other works used parallel evolutionary to generate test cases faster for web applications [315], and an adaptive fuzzing method for testing Modbus TCP/IP [313], which is an application layer messaging protocol.

Popular JavaScript frameworks from GitHub [163] and web applications used in web testing [19, 18] include FAQForge, Schoolmate, Webchess, PHPSysInfo, Timeclock, and PHPBB2.

2.7.4 Database Applications

DBT techniques have been applied to database applications in one paper, which focused on the selection of a diverse subset of test cases for large-scale databases [269]. The regression test suite was based on a legacy database application [268].

2.7.5 Graphical User Interface

Graphical User Interface (GUI) are applications that contain widgets (e.g. Buttons, Text fields, etc.) where a user interacts with. GUI can be applied in desktop, web, or mobile applications. However, some researchers focused on the characteristics of GUI components in testing and tried to investigate the characteristics of what makes a GUI test suite effective [327], generate GUI test cases [111, 136], or investigated the reuse of GUI tests based on semantic similarities [210].

2.7.6 Mobile Applications

Although limited, DBT techniques have also been applied to mobile application testing. Notably, Vogel et al. [307, 308] investigated diversity in the context of test data generation for mobile applications, leveraging a modified version of the heuristic NSGA-II algorithm to introduce diversity and achieve improved coverage, albeit at higher computational costs. Wang et al. [314] used Extended Finite State Machine (EFSM) to model Android apps and generate test cases using a genetic algorithm guided by transitions diversity. The approach achieved higher coverage and crash detection than other model-based approaches.

Some publicly available test subjects for Android Apps are ATM and CRAFT-DROID [210].

2.7.7 Neural Network Models

There has been growing interest in applying diversity-based testing (DBT) techniques within neural networks (NNs). For example, NNs have been used to create diverse search spaces by leveraging the diversity of salient features [162]. Additionally, NNs have been targeted for testing using neuron coverage diversity for deep learning (DL) models [342, 343] or through population diversity to generate diverse test cases [38]. Aghababaeyan et al. [4, 5] and Jiang et al. [161] demonstrated that using input feature diversity to test deep neural networks (DNNs) significantly improves fault detection and generation time compared to traditional coverage-guided, greedy, or random approaches. Kim et al. [171] proposed a new test adequacy criterion for DL systems based on input diversity.

For test input prioritisation in DL models, Xie et al. [329] found that a diversity-based approach outperformed a neuron coverage-based approach in terms of the average number of detected violations. Similarly, Mosin et al. [229] employed a similarity-based method for input prioritisation, finding it more efficient than white-box and data-box approaches, though with slightly lower effectiveness (APFD) than white-box but comparable to data-box.

Some widely used datasets in deep learning systems include ImageNet [177], MNIST [183], CIFAR10 [176], and SVHN [234].

2.7.8 Autonomous Driving Systems

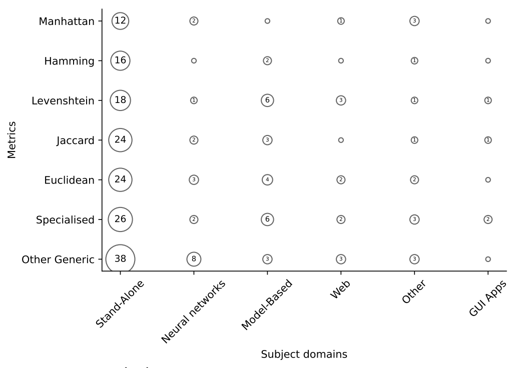
Autonomous Driving Systems (ADSs) replace human drivers, which perform various driving tasks, such as perception, localisation, planning, and control [70]. Applying DBT techniques for testing ADSs is playing a major role recently. Neelofar et al. [233] proposed new test adequacy metrics for autonomous vehicles based on coverage and diversity. Cheng et al. [70] developed a behaviour diversity-based technique for generating diverse scenarios to test autonomous vehicles. Wang et al. [311] proposed a diversity-based fuzzing method to speed up the fuzzing approach for testing autonomous vehicles.

Available datasets reported in the literature included Baidu Apollo [30] and LGSVL [270]. Some works in testing DL systems, mentioned in Section 2.7.7, used autonomous vehicles in their testing subjects [171, 349, 350, 265].

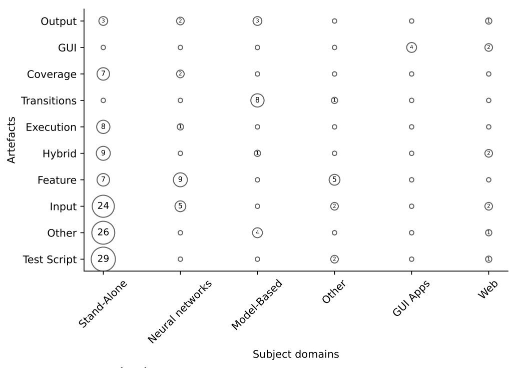
2.7.9 Compilers

Compilers are programs that translate source code written in some programming language into machine code, byte-code, or the source code of some other programming language. A few studies have explored DBT techniques in compiler testing, including history-guided bug revealing techniques [63], testing Simulink compilers [192], and the detection of compiler warning defects [299]. GCC, Clang and LLVM are popular C compilers used as subjects in the literature.

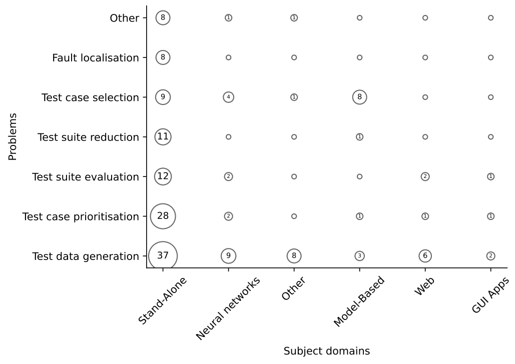
These varied subject domains highlight the wide range of contexts in which DBT techniques have been studied, reflecting ongoing efforts to improve the effectiveness and efficiency of software testing practices.



(a) *Similarity metrics*



(b) *Diversity artefacts*



(c) *DBT problems*

Figure 2.9: *Relationship between subject domains and similarity metrics (a), diversity artefacts (b), and DBT problems (c), respectively*

2.7.10 Relationship between Subject Domains and Similarity Metrics, Diversity Artefacts, and DBT Problems

Figure 2.9a shows the relationship between similarity metrics and software testing subject domains. Nearly half of the papers on stand-alone and console applications used the top generic metrics (G1-G5). Neural networks predominantly utilised other generic metrics (e.g., G6, G10, G11), while model-based applications primarily employed Edit distance (G2), Euclidean distance (G1), and specialised metrics (e.g., S1, S3, S4).

Figure 2.9b illustrates the relationship between diversity artifacts and software testing domains. Unsurprisingly, most diversity artifacts were used in stand-alone and console applications, as about 75% of the papers focus on this area. Transition diversity was mainly applied in model-based applications, while GUI diversity was used in GUI applications. Additionally, neural networks and deep learning systems heavily relied on feature and input diversity.

Figure 2.9c displays the relationship between DBT problems in software testing and subject domains. Stand-alone and console applications dominate DBT testing issues, except for test case selection, where model-based applications are nearly as prevalent. In neural networks, DBT mainly focuses on test data generation and test case selection.

Conclusions — RQ5: SUBJECT DOMAINS

The subjects reported in the collected papers were stand-alone programs and libraries, web applications, database applications, mobile applications, model-based applications, compilers, and neural network models. Almost two-thirds implemented their approaches on console programs and libraries written in different languages such as Java, C, Python, etc. In the last three years, there has been a significant increase in DBT techniques for testing neural networks and deep learning systems. Model-based systems come next in the popularity of reported studies. Moreover, a few studies reported the use of diversity in web applications, but one might have expected such systems to receive more attention since they are widely used in both industry and academia. Also, only three papers applied diversity in autonomous vehicles, three in compilers, and three in mobile applications. There is only one paper that used diversity in databases.

2.8 RQ6: TOOLS

What DBT tools have been developed by researchers?

In this section, I present the tools that were developed based on DBT techniques. Table 2.9 lists the tools encountered in the study. Figure 2.10 shows the number of developed tools since 2005. The number of tools started to increase in the last five years, which shows a level of maturity in the field of applying diversity in software testing. Also, the trend line shows an increase of DBT tools in the field, with the highest number of DBT tools developed in 2022. A possible reason for such an increase can be associated with the guidelines for conferences like ICSE and ICST that, since 2020, have required that papers include a replication package with the necessary software, data, and instructions.

Numerous research papers have contributed to the development of DBT tools aimed at generating test data for various applications. Many of these tools are categorised as fuzzers, such as "SimFuzz" [340], "RLCheck" [264], and "BeDivFuzz" [235]. Furthermore, "OutGen" [33] caters for a broad range of programs and software systems, focusing on the generation of diverse output data. In compiler testing, there is "HiCOND" [63] a test-program generation tool that leverages historical data for compiler testing, "DIPROM" [299], which constructs warning-sensitive programs to detect compiler

Table 2.9: *A summary of the tools encountered in the study.*

Tool	Problem	Application	Proposed	Used in	Open Source
The Diversity Analyzer	Test suite evaluation	General-purpose Applications	[236]	[237, 238, 239]	✗
DART	Test case selection	Database Application	[268]	[266, 267, 271]	✗
SimFuzz	Test data generation	General-purpose Applications	[340]	[275, 17]	✗
SIMTAC	Test case prioritisation	C & Access Control	[34]	-	✓
SimFL	Fault localisation	Simulink Models	[199]	[200]	✗
SimCoTest	Test data generation & Test case prioritisation	Simulink Models	[214]	[215, 331]	✓
FAST	Test case prioritisation	C & Java programs	[226]	[82, 126]	✓
HiCOND	Test data generation	Compilers	[63]	[262, 299, 192]	✓
DIG	Test data generation	Web Applications	[35]	[346]	✓
Sapienz^{DIV}	Test data generation	Mobile Applications	[307]	[308]	✓
SiMut	Test suite evaluation	Java programs	[260]	[127]	✗
RLCheck	Test data generation	Java programs	[264]	[235, 302]	✓
DeepMetis	Test data generation	Deep Learning Systems	[265]	[103, 347]	✓
DeepHyperion	Test data generation	Deep Learning Systems	[349]	[347, 350, 352, 351, 149, 36]	✓
OutGen	Test data generation	General-purpose Applications	[33]	-	✓
DIPROM	Test data generation	Compilers	[299]	-	✓
BeDivFuzz	Test data generation	Java programs	[235]	-	✓
TSE	Test data generation	Java programs	[288]	-	✓
Fastazi	Test case prioritisation	Java programs	[126]	-	✗
Metallicus	Test adaptation	Python programs	[293]	-	✓
DeepHyperion-CS	Test data generation	Deep Learning Systems	[350]	[353, 91]	✓
RECORD	Test data generation	Compilers	[192]	-	✓
DeepWalk	Test data generation	Deep Learning Systems	[338]	-	✓

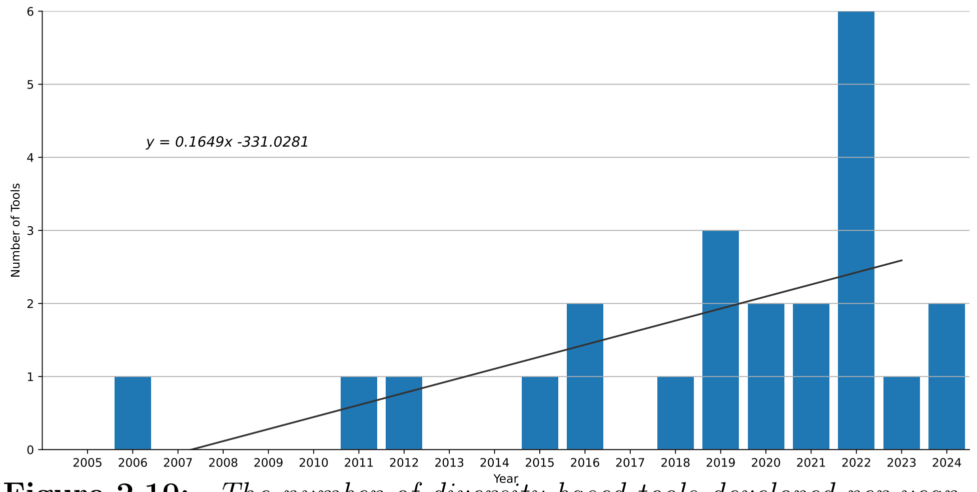


Figure 2.10: *The number of diversity-based tools developed per year.*

warning defects, "RECORD" [192] for testing Simulink compilers. Moreover, in test case recommendation, Test Suite Evolver (TSE) [288] augments an existing test suite to cope with changes made in the source code, or add more test cases to an incomplete test suite by exploiting structural similarity between methods.

Specifically targeting web applications, "DIG" [35] (**DI**versity-based **G**enerator) has been introduced for generating test cases. For mobile applications, SAPIENZ^{DIV} [307] used diversity to enhance the mobile application testing tool SAPIENZ resulting in improved coverage albeit at the cost of increased processing time. In another type of application, "SimCoTest" [214], is a tool for generating tests for Simulink models. Additionally, "SiMut" [260] is notable for its ability to generate a reduced set of mutants, offering efficiency gains in mutation testing. Furthermore, DEEPHYPERION [349] and DEEPHYPERION-CS [350] are tools that are capable of generating failure-inducing inputs for deep learning systems. DEEPMETIS [265] is another tool capable of augmenting the test set of a deep learning system to improve ability to kill mutants. DEEPWALK [338] is a DNN testing tool that generates diverse and valid inputs for high-dimensional media data.

Several additional papers have contributed to the realm of DBT tools in the context of test case prioritisation. One such tool, "SIMTAC" (Similarity Testing for Access-Control) [34], focuses on reordering test cases for Access Control systems based on their similarity. Similarly, "SimCoTest" [214] also addresses test case prioritisation, particularly within Simulink models, while also encompassing test data generation capabilities, as previously mentioned. Furthermore, two tools, namely "FAST" (Fast and Scalable Test Case prioritisation) [226] and "Fastazi" [126], utilise diversity to prioritise test cases within test suites for Java applications. FAST demonstrates improved efficiency without compromising effectiveness, while Fastazi combines file-based test case selection with similarity-based test case prioritisation.

Other areas in software testing had very few DBT tools. In 2006, Nikolik [236] introduced the first DBT tool named "The Diversity Analyzer" for quantifying the extent to which a test suite exercises a program in diverse manners with regard to control and data. In the domain of test suite reduction, "DART" (Database Regression Testing) [268] employs a similarity-based regression test reduction approach for large-scale database applications. Another tool, "SimFL" (Simulink Fault Localisation) [199], focuses on fault localisation within Simulink models. Lastly, METALLICUS [293] is a DBT test recommendation tool that generates a corresponding test suite given a query function.

These DBT tools showcased in the literature contribute to various aspects of software testing, ranging from test case prioritisation and reduction to fault localisation and test suite generation. Their adoption demonstrates the ongoing efforts to leverage diversity as a means to enhance the effectiveness, efficiency, and quality of software testing practices.

Conclusions — RQ6: TOOLS

The trend line shows an increase in DBT tools in software testing with a notable increase in the past 5 years. The highest number of developed DBT tools was in 2022. Most of the reported DBT tools (82.6%) were developed to deal with the problems of test data generation and test case prioritisation. Only one DBT tool was developed for test suite quality evaluation, one for test case selection, and one for test adaptation. The tool developed for test case selection is only for database applications.

2.9 Future Research Directions

I discuss a number of the challenges in DBT and future research that can be done to address these challenges.

2.9.1 Identifying Diversity and Handling Complex Data

Identifying what constitutes "diversity" in inputs, scenarios, and system states can be complex and domain-specific. Implementing effective diversity-based testing often requires deep domain knowledge to understand which types of diversity are relevant and how to simulate them accurately.

One of the challenges with DBT for test generation is managing complex data inputs (such as images [241] and abstract tree models [283]). Calculating diversity measures for these types of data can be resource-intensive, and researchers may struggle to create input data that is both valid and semantically meaningful (e.g., generating appropriate strings for form fields or diverse sets of arrays/XML files). Investigating other parameters in tree test generation are required that may affect failure detection performance [283]. Features of images can be used to measure similarity, but more types of features such as color and texture can utilised, and the assessment of the importance of these features [241]. Furthermore, test cases may include irrelevant information that can introduce noise when measuring similarity between them. DBT needs to pre-process test cases, especially when using test scripts as diversity artifacts.

2.9.2 Scalability

Scaling diversity-based testing for complex systems or large-scale applications may necessitate advanced tools and techniques. Some studies reported scalability issues [97, 168] and others proposed solutions [226]. Generating and running numerous diverse test cases can be resource-intensive, demanding significant computational power and time. Additionally, analysing the results from a wide range of test cases can be particularly time-consuming for complex systems. In Chapter 4, I use bytecode of test cases, which scales much better with bigger test suite sizes.

2.9.3 Areas of Applying DBT Techniques

Test data generation is the most researched area where DBT techniques were implemented, as I showed in Section 2.6. Although test suite reduction, test case prioritisation, and test case selection are closely related problems that deal with the cost of large regression test suites, many authors have applied DBT techniques in test case prioritisation and test case selection, but there are very few papers on test suite reduction. Since DBT techniques showed great promise in test case prioritisation [24, 226, 184, 151], it would make sense to use them to solve test suite reduction as well. More work needs to be performed in test suite reduction, and more extensive investigations need to be done to compare DBT techniques with other reduction techniques reported in the literature. Moreover, the literature shows very limited exploration of DBT techniques in the context of automatically generated test suites, which are often large and redundant. Addressing this gap, in Chapter 3 I apply DBT techniques to reduce the automatically generated test cases.

2.9.4 Diversity-Based Subjects

Most reported papers in this study used DBT techniques on stand-alone programs and in Model-Based testing. This was discussed in Section 2.7, and I showed that there was little work on the use of DBT techniques in web, database and mobile testing.

Although web applications are widely used, there are fewer papers applying DBT techniques on them. As discussed in Section 2.7.3, all of these address test data generation [208, 18, 195, 279, 35, 187, 188, 315], and none of the reported papers dealt with evaluating the quality of the current test suite or reducing the cost of test suites for web applications.

The limited memory and processing power of mobile devices make it difficult to use an extensive and more demanding techniques. This can be an obstacle for using DBT in test data generation for Mobile applications as was reported by Vogel et al. [308].

The modified tool SAPIENZ^{DIV} has more computational cost than SAPIENZ. However, applying test case prioritisation or test suite reduction for mobile test suites could be very appealing. A scalable prioritisation approach such as the one proposed by Miranda et al. [226], may be useful. Further investigations are needed to apply DBT techniques in mobile applications.

2.9.5 Tool Support

As discussed in Section 2.8, there has been a spike on the number of DBT tools developed since 2019, which indicates a possible maturity in the field. The trend line of Figure 2.10 also shows an increase of DBT tools developed. However, most of these tools were developed for test data generation. There is a need for more tools in other areas, especially for test case selection, test suite reduction, and test suite quality evaluation.

Furthermore, there are many Simulink DBT tools developed for a variety of areas, such as test data generation [214], test case prioritisation [214], and FL [199]. However, other applications, such as web and mobile, have very little tool support for DBT techniques. Only a single tool for Web applications was developed [35], and only a single tool for Mobile applications was developed [307]. These tools are only for test data generation, which leaves other areas such as test case prioritisation, test suite reduction, and test case selection open for DBT tools to be developed for both Web applications and Mobile applications. For Database applications, DART [268] is for test case selection. There is no available DBT techniques or tools to generate test cases for such applications.

2.9.6 Exploring Other Diversity Artefacts

Most current works focused on input and test script diversity as shown in Section 2.5. Other artefacts did not receive much attention, and some artefacts are used with other diversity artefacts, which makes it difficult to identify their importance and contribution to the effectiveness of the DBT technique. Non-functional properties are prime examples of such artefacts. Feldt et al. [110] used non-functional properties such as memory and processing time as one of 11 variation points in their model. However, since there were several other factors, it is difficult to determine how useful these properties can be. Shimari et al. [287] used the run time information obtained when executing the SUT with the test cases to develop a test case selection technique for an Industrial Simulator. Apart from their work, no other work investigated how non-functional properties would be beneficial. The same can be said about the program's state diversity, which is not investigated at all. In Chapter 4, I explore a new diversity artefact based on the bytecode of test cases, where I show that it improves effectiveness and hugely improves efficiency.

2.10 Conclusions

In this chapter, I covered 167 papers that described the use of DBT techniques to solve a software testing problem. My systematic mapping study discussed DBT publication trends showing the number of DBT publications per year, and I presented the prominent venues, papers and authors in the field. I presented 79 similarity metrics there are used in the literature to calculate the level of similarity between different testing artefacts, and found that generic similarity metrics are the most commonly used metrics. The most popular similarity metrics were Euclidean distance, Edit distance, and Jaccard distance used in 35, 31, and 31 papers, respectively. However, there is no clear evidence that a particular metric performs best, and it is clear that different factors, mainly the nature of the application, could affect the results.

The DBT papers attempted to address the problems of test data generation, test case prioritisation, test case selection, test suite reduction, test suite quality evaluation, and fault localisation. Test data generation and test case prioritisation were the most researched areas with 39.3% of the collected papers dealing with test data generation, then 19.6% dealing with test case prioritisation. Also, I presented the various testing artefacts used as a basis for DBT techniques, such as inputs, output, test scripts, test executions, features, etc. Of these, 19.8% used input diversity, then 18.6% used test scripts as artefacts for applying DBT techniques. Furthermore, I explored the various subject domains where DBT techniques were utilised. Stand-alone programs and libraries written in languages such as Java, C++, Python, and so on were the most used subject domain for the collected papers followed by model-based applications. Relatively few papers dealt with Web applications, Database applications, Mobile applications, and Compilers. There has been a clear increase in the use of DBT techniques for deep neural networks since 2019. Moreover, I presented the DBT tools reported in the collected papers and found that 82.6% of the tools presented in the study support test data generation and test case prioritisation. Finally, I discussed some challenges and future work in DBT that can be addressed by researchers in the future. For example, identifying what constitutes diversity can be complex and domain-specific, and managing complex data can be challenging. Also, for complex systems or large-scale applications, scaling DBT can be resource-intensive and may need advanced tools and techniques. Applying DBT in more domain-subjects is lacking, especially in web applications and mobile applications. Exploring other diversity artefacts, such as non-functional properties, is needed, and having better tool support.

Building on the findings and gaps identified in this literature review, the subsequent chapters of this thesis explore how DBT can be effectively applied to address practical testing challenges. Chapter 3 addresses the problem of excessive test cases generated by automated tools, which represented a gap in research on test suite reduction, by applying textual diversity to select a smaller, effective subset of tests. Chapter 4 builds on this by introducing a novel bytecode-based diversity measure for test case prioritisation, improving both effectiveness and runtime efficiency compared with traditional techniques. This naturally raises the question of whether coverage-based selection, while generally more effective at fault detection but less efficient than diversity-based strategies in terms of runtime (as shown in Chapter 4), achieves superior results because it detects many faults overall or because it targets the more challenging faults. Finally, Chapter 5 extends this investigation to the detection of difficult-to-find faults by evaluating DBT and coverage-based test selection strategies against stubborn mutants.

Chapter 3

Evaluating String Distance Metrics for Reducing Automatically Generated Test Suites

The contents of this chapter are based on "I. Elgendy, R. Hierons, and P. McMinn. Evaluating string distance metrics for reducing automatically generated test suites. In Proceedings of the International Conference on Automation of Software Test (AST), pages 171–181, 2024.".

3.1 Introduction

In the previous chapter, I reviewed the role of diversity-based testing (DBT) across different software engineering activities, including its application in test suite reduction (TSR). That review highlighted a gap: while similarity-based approaches have been explored for TSR, prior work has largely focused on developer-written or model-based test suites, with little attention to automatically generated ones. This gap is important because automatically generated test suites are typically much larger than manually written ones, leading to higher costs for regression testing and making them natural candidates for reduction.

Regression testing involves testing the software after changing the software or its execution environment. It ensures that modifications do not affect existing functionality. However, regression testing can consume a lot of resources and time [334]. This can even be more problematic if regression tests are generated automatically using test generation tools, because the sizes of the generated test suites are much larger, unless these tools have a minimisation approach built-in. To deal with this issue, TSR approaches originally aimed to reduce a test suite while maintaining the full set of test requirements it covered, by removing redundant test cases. However, subsequent studies [79, 80, 82], developed approaches that aim to gain more reduction at the expense of losing some of the power of the original test suite. A study on these approaches has been presented by Covieloo et al. [79], investigating the trade-off between the reduction in test suite size and the loss in fault detection capability. They found that approaches with the partial fulfilment of the test requirements have a better trade-off between reduction in test suite size and loss in fault detection capability than approaches maintaining the same test requirements as the original test suite.

There are different approaches to achieving the reduction of test suites. A well-known approach for reduction is based on removing "redundant" test cases, where a redundant test case is one that covers a test requirement already covered by another test case [334]. However, reduction using code coverage requires instrumentation of the program under test and requires the test suites to be executed first, which can be complicated and time-consuming. Similarly, model coverage requires one to have a model of the software under test. An alternative, that does not suffer from such problems, is to base reduction on textual diversity (i.e., test cases that "look" different from each other). The idea is that less similar test cases are more likely to exercise different parts of the system than more similar test cases. In this similarity-based approach, the degree of similarities between the test cases is estimated without the need to execute the test suites first. This way the source code of the test suite itself is the only requirement for the reduction to take place, which makes the approach faster to apply than other reduction techniques and also more widely applicable in contexts where execution or instrumentation is impractical.

Similarity-based approaches have been used in test case prioritisation in many works (e.g. [184, 226, 329]) showing the effectiveness of such approaches in terms of average percentage of fault detection. However, there has been relatively little work applying similarity-based techniques for TSR and no work for reducing automatically generated test suites. Coutinho et al. [77, 78] evaluated the use of distance functions for test suite reduction based on similarity, but within the scope of model-based testing. Cruciani et al. [82] proposed scalable approaches for test suite reduction based on similarity and some techniques developed for big data. These approaches used only developer-written test suites showing good results. Automatically generated tests also stand to gain from similarity-based reduction approaches, because these test suites are much larger than developer-written test suites, resulting in a higher cost of rerunning these tests. Therefore, it is very appealing to apply similarity-based reduction on such an automatically generated test suite. The extent to which similarity-based reduction can be utilised for automatically generated test suites is the focus of this chapter.

In my empirical study, I investigate the effectiveness of reducing automatically generated test suites based on textual diversity. I applied two state-of-the-art automatic test case generation tools for Java, Randoop [250] and EvoSuite [117] on subjects from Defects4J [164]. I reduced test suites based on the set of most diverse test cases and also on the set of least diverse test cases. Reduction based on least diverse test cases serves as a baseline. If the hypothesis is that maximising diversity would have a good fault-detection rate, then conversely one would expect that minimising diversity would result in a lower fault-detection rate. Also, I evaluate my string-based similarity reduction and compare it against the first-n test cases, where n is the desired size of the reduced test suite. Then, I evaluated the reduced test suites based on their fault-finding capability against random reduction and first-n test cases. I investigated the trade-off between the number of test cases lost in the reduction and the loss in mutation scores. In my experiments, I used Euclidean distance, Hamming distance, Levenshtein (edit) distance, Manhattan distance, and Normalised Compression Distance (NCD). I found that mutation scores using reduced test suites based on maximising string dissimilarity of test cases were higher than those for random reduction in over 70% of the test suites generated, and higher than first-n test cases in over 88% of the cases.

To investigate the scalability challenge, I conducted an experiment using the scalable FAST approach [82]. The goal was to evaluate how the traditional pairwise diversity-based maximisation compares with FAST. The results showed that the reduced test suites obtained by maximising pairwise diversity achieved higher mutation scores in nearly 95% of the Randoop-generated cases and 72% of the EvoSuite-generated cases.

The evaluation was conducted on test suites generated by Randoop and EvoSuite. A key distinction is that Randoop does not include any built-in minimisation, while EvoSuite integrates a minimisation mechanism. This study is the first to investigate string-based similarity reduction on automatically generated test suites applying NCD as a similarity metric. These findings not only address the gap identified in Chapter 2 regarding the underexplored application of DBT to automatically generated tests, but also motivate the exploration of alternative diversity artefacts. In particular, the results suggest that other compact representations of tests, such as bytecode, may offer further efficiency gains in similarity-based techniques, a topic that is the focus of the next chapter.

The chapter offers the following contributions:

1. The first work to evaluate the effectiveness of applying string-based similarity reduction on test suites generated by Randoop and EvoSuite against random reduction (Section 3.5.1).

2. An evaluation of string-based similarity reduction against first-n test cases and FAST reduction (Sections 3.5.2 and 3.5.5).

3. A comparison between the different similarity metrics in terms of time and fault-finding capability (Section 3.5.3).

4. An evaluation of string-based similarity reduction applied to test suites generated by Randoop and EvoSuite (Section 3.5.4).

3.2 Methodology

The goal of this chapter is to study the effectiveness of string-based similarity reduction on regression test suites automatically generated using tools. I developed a tool to perform the steps of my technique in an automated way. First, the tool generates regression test suites. Then, it calculates the similarities between the generated test cases using the similarity metrics, and applies string-based similarity reduction and random reduction using different reduction sizes. Finally, the tool evaluates the reduced test suites in terms of fault-finding capabilities.

3.2.1 Research Questions

I answer the following research questions:

- **RQ1**: How does reduction based on maximising and minimising diversity compare to random reduction?
- **RQ2**: How does string-based similarity reduction compare to a lower time budget for the test generation?
- **RQ3**: Which similarity metric performs the best in terms of time to compute and loss of fault-finding capability?
- **RQ4**: What is the effect of string-based similarity reduction on automatically generated test suites with no built-in minimisation (Randoop) compared to an already minimised test suites (EvoSuite)?
- **RQ5**: How does test suite reduction using FAST compare to reductions based on maximising diversity, minimising diversity, and random selection?

3.2.2 Test Suite Generation

I used two state-of-the-art tools, Randoop and EvoSuite, to automatically generate regression test suites using their default configurations on real-world Java applications from the Defects4J [164] framework. I set the time budget to 3000 seconds for the generation of test suites. Randoop and EvoSuite create unit tests in the JUnit format to cover the classes under test. Randoop generates unit tests using feedback-directed random test generation [250] for object-oriented programs. This technique iteratively extends sequences of method calls for the classes under test until the generated sequence raises an undeclared exception or violates a general code contract. Randoop executes the sequences it creates, using the results of the execution to create assertions that capture the behaviour of the program, then creates tests from the code sequences and assertions. EvoSuite uses search-based techniques applying a hybrid approach to evolve whole test suites towards satisfying a coverage criterion [117]. A fitness function guides the process based on a coverage criterion. When the search is completed, the highest code coverage test suite is minimised and regression test assertions are added [118].

```java
@Test
public void test0130() throws Throwable {
    if (debug)
        System.out.format("%n%s%n", "RegressionTest0.test0130");
    byte byte3 = org.apache.commons.lang3.math.NumberUtils.max( (byte) 0, (byte) 0, (byte) 1);
    org.junit.Assert.assertTrue("'" + byte3 + "' != '" + (byte) 1 + "'", byte3 == (byte) 1);
}
```

(a) *Test case 1*

```java
@Test
public void test0160() throws Throwable {
    if (debug)
        System.out.format("%n%s%n", "RegressionTest0.test0160");
    byte byte3 = org.apache.commons.lang3.math.NumberUtils.max( (byte) 0, (byte) 1, (byte) 100);
    org.junit.Assert.assertTrue("'" + byte3 + "' != '" + (byte) 100 + "'", byte3 == (byte) 100);
}
```

(b) *Test case 2*

```java
@Test
public void test0177() throws Throwable {
    if (debug)
        System.out.format("%n%s%n", "RegressionTest0.test0177");
    short short1 = org.apache.commons.lang3.math.NumberUtils.toShort("040404104-1");
    org.junit.Assert.assertTrue("'" + short1 + "' != '" + (short) 0 + "'", short1 == (short) 0);
}
```

(c) *Test case 3*

Figure 3.1: *An example of three test cases generated by Randoop from the* `Lang` *project. Test cases 1 and 2 call the same method, while test case 3 is different from them*

3.2.3 Similarity Metrics

A similarity metric is a function that quantifies the similarity between two objects in a numeric value. There are metrics based on string distance, while others are based on trace executions or coverage distance between the test cases.

A test case is a collection of string lines, which I can concatenate into a single string. If I do this for two test cases, I have two long strings that can be compared directly. Similarity is measured at the character level between the two strings. Figure 3.1 is an example to show how the notion of diversity can be used for reduction. The first two test cases are very similar invoking the `max` method, while the third one is more different invoking a different method (`toShort`). The faults found by the first two test cases might be similar compared to the third test case calling an entirely different method. If the goal is to remove one test case from these three, then from a static point of view, it would make more sense to remove one of the two similar test cases. This way, I have two test cases calling two different methods resulting in a higher probability of coverage and fault detection.

There are many similarity metrics that can be used in similarity-based approaches, as described in Section 2.4. I used five different similarity metrics: Euclidean, Hamming, Levenshtein, Manhattan, and Normalised Compression Distance (NCD) metrics. The selected metrics are used since the first four metrics are classical string distance metrics used in many software testing studies [74, 141, 184, 316], and NCD is a recent method proposed by Li et al. [190] and used by Feldt et al. [110] in measuring distance between test cases. NCD has been used in test case prioritisation [145, 129], in selecting between test suites for finite state machines [153], and to generate a minimised test suite [61, 62]. Furthermore, in Section 2.4 I found that the five similarity metrics I used were among the top-used similarity metrics in software testing. However, NCD has not been used in test suite reduction approaches before, and I use it in my study to compare it with the classical string similarity metrics.

Euclidean and Manhattan distances

The Euclidean distance is calculated as the square root of the sum of the squared differences between the vectors, while the Manhattan distance is computed as the sum of the absolute differences between two vectors. The two vectors must have the same length. It is possible to represent a string of characters as a vector of numbers, where each number is the ASCII code of the corresponding character. When two strings have different sizes, I can append to the smaller one $char(0)$ to make them the same size.

Hamming distance

The Hamming distance [131] between two strings is the number of times when the corresponding characters are different. Like Euclidean and Manhattan distances, the strings should be the same size. Again, I can solve this by appending $char(0)$ to the smaller one to make them the same size.

Levenshtein distance

The Levenshtein distance [189] or edit distance between two strings is the minimum number of edits *(insertions, deletions or substitutions)* required to change one string into the other. Levenshtein distance takes into consideration that parts of the strings can be similar even if not in corresponding places, and can work with strings of different sizes.

Normalised compression distance (NCD)

The Kolmogorov complexity of a string of symbols, x, is the length of the shortest program that outputs x [191]. The normalised compression distance (NCD) is based on the observation that the size of the output when compressing a string with real-world compression programs, such as gzip and bzip2, is a good approximation of its Kolmogorov complexity [109]. Let x and y be two strings, then NCD is calculated using:

$$NCD(x, y) = \frac{C(xy) - min\{C(x), C(y)\}}{max\{C(x), C(y)\}}$$

where $C(x)$ is the length of the compressed string x, $C(y)$ is the length of the compressed string y, and $C(xy)$ is the length of the concatenated strings x and y after compression. The NCD value is in the range $[0, 1]$.

3.2.4 Reduction

In this step, I explain the reduction using five methods: random reduction, first-n test cases, reduction based on maximising diversity, reduction based on minimising diversity, and reduction based on FAST [226]. To achieve random reduction, my tool randomly picks a test case to discard and continues until the desired size is reached. The first-n test cases are simply using the first generated test cases up to the desired size. For string-based similarity reduction, the tool calculates all similarity values between the test cases based on the desired similarity metric and returns the similarity values in a two-dimensional array that I call the similarity matrix. In this matrix, a cell with index $[2, 5]$ for example represents the similarity value between the third and sixth test cases. After calculating all the similarity scores, the tool reduces the test suite into the most diverse test suite and the least diverse test suite. I used a greedy algorithm for the reduction, which is based on the technique by Cartaxo et al. [55, 54], used for model-based testing and adapted to use on Java tests. My technique builds the most diverse test suite by discarding one of the pair of test cases found in the lowest value in the similarity matrix. This process continues until I reach the desired reduction size. Similarly, the least diverse test suite is built by discarding one of the pair of test cases found in the highest value in the similarity matrix until we reach the desired reduction size.

To explain this further, consider this example. Assume that the minimum value in the similarity matrix is in cell $[3, 7]$. The technique selects one of these indices randomly, say index three, and removes the entire row and column. Then it finds the next minimum value and continues until the desired size is reached. If there is more than one highest or lowest value in the matrix, the first one is selected as the target for reduction. Because of the random choice of removing between two possible test cases, the technique repeats the reduction 30 times for random, the most diverse, and the least diverse test suites. Later, I analysed this sample size of 30 using statistical testing to find out the significance level and effect size in order to justify the validity of the results.

Miranda et al. [226] introduced the FAST family of test case prioritisation approaches to address runtime concerns with large test suites by leveraging big data techniques such as Shingling, Minhashing, and Locality- Sensitive Hashing to compute test similarity for test case prioritisation. From the proposed family of FAST techniques, I used FAST-pw. I chose to use FAST-pw because they reported it to be the most precise in the ranking, as it guarantees the selection of the most dissimilar test case from the ones already prioritised. I will simply refer to it as FAST from now on. Algorithm 1

Algorithm 1: FAST-TCP algorithm [226]

```
    Input: Coded test suite $T$
    Output: Prioritised test suite $P$
 1  $P \leftarrow \emptyset$
 2  $I \leftarrow GetTextCaseIDs(T)$
 3  $M \leftarrow MHSignatures(T)$
 4  $B \leftarrow LSHBuckets(M)$ // $B$: The set of buckets
 5  $M(v) \leftarrow MHSignatures(\emptyset)$ // $M(v)$: Cumulative signatures of $P$
 6  while $|P| \neq |I|$ do
 7      $C_s \leftarrow LSHCandidates(B, M(v))$
 8      if $C_s = \emptyset$ then
 9          $M(v) \leftarrow MHSignatures(\emptyset)$
10          $C_s \leftarrow LSHCandidates(B, M(v))$
11      $C_d \leftarrow (I - P - C_s)$
12      $s \leftarrow \text{argmax}_{c \in C}\{JaccardDistance(M(v), M(c))\}$
13      $M(v) \leftarrow UpdateMHSignatures(M(v), M, s)$
14      $M \leftarrow Remove(M, s)$
15      $P \leftarrow Append(P, s)$
16  return $P$
```

outlines the steps involved in FAST [226]. Initially, the permuted test suite P is empty (Line 1), and the indices of the test cases are stored in I (Line 2). Using the encoded representation (i.e., text of test cases) of the test suite T, FAST generates the MinHash signatures M (Line 3). The algorithm then computes the Locality-Sensitive Hashing (LSH) buckets B (Line 4), while the cumulative signatures of the ordered test cases, denoted as $M(v)$, are initially set to empty (Line 5).

The candidate sets are generated within a while loop (Lines 6-15), where the prioritisation process occurs. $M(v)$ is divided into b bands, each of which is hashed. If a collision is detected with the corresponding bucket in B, the test cases within that bucket are added to the candidate set C_s (Line 7). The actual candidate set C_d, used by FAST, is calculated as the complement of C_s, excluding the test cases that have already been prioritised in P (Line 11). FAST then selects the candidate test case that is farthest from $M(v)$ based on the Jaccard distance (Line 12). $M(v)$ is updated to reflect the cumulative signatures of P (Line 13). Lastly, the selected test case is removed from M and added to P (Lines 14 and 15).

In order to use FAST in reduction, I used FAST to reorder the test cases, and then I selected the first-n test cases after ordering up to the desired size.

3.2.5 Analysis and Evaluation

Each project in Defects4J has faulty versions, where there are one or two real-faults in that faulty version. A test suite that has a very low coverage and mutation score can still detect the real fault, but that test suite would simply be unreliable since it did not even cover the rest of the program which might contain more faults. Therefore, I used mutants as representatives for faults because they are better distributed across the program and mutants are better suited to perform statistical testing.

To evaluate string-based similarity reduction, I determined the mutation score of each reduced test suite using the Defects4J framework. Defects4J uses the "*Major*" mutation framework to generate mutants and to run the mutation analysis. One issue that might occur after reduction is that some test suites fail due to test-dependency issues. I needed to fix these test suites first to remove the test cases causing the failure, and then run the mutation analysis. The tool provides bash scripts which I used to fix and analyse all the test suites. Finally, the tool parsed all the generated reports and log files producing a CSV file with all coverage and mutation scores for all 30 attempts. Also, during parsing, the tool calculated the statistical significance and effect size to verify the results.

3.3 Experimental Setup

Since EvoSuite generates minimised test suites, the number of test cases varies greatly between the generated test suites from EvoSuite and Randoop. I limited the number of generated tests to 1,500 tests to make it possible to run the experiments in a reasonable amount of time.

In order to compare Randoop and EvoSuite, I made the reduction using an absolute number of test cases. This decision was made to control for test suite size. If I used the same reduction percentages, the Randoop test suite would be much larger than the EvoSuite test suite. The reduced test suites of Randoop might perform better simply because they are larger.

For each test subject, I chose the smaller of the two generated test suites and then picked the size of the reduced test suite to be a percentage of the smaller test suite size. The selected percentages for the reduced test suites were 35%, 60%, and 85%. To illustrate this, consider the `"Lang"` project, for which given 3000 seconds, EvoSuite generated a test suite of size 123 test cases, while Randoop generated the upper limit of 1,500 test cases. Now based on the selected reduction percentages, the reduction sizes were 43 (35% of 123), 73 (60% of 123), and 104 (85% of 123), with these sizes being used with the test suites from both tools. However, it is important to note that in the case of reduced test suites for Randoop, a test suite of size 43 is 35% of 123 (the size of the smaller test suite), and 2.87% of 1,500 (the size of the test suite generated by Randoop).

The test generation algorithms used are stochastic, so there is a risk that any results might not hold more generally. In order to explore this potential threat and verify my

Table 3.1: *Information about the test subjects and the original test suites generated to cover them. MS is the mutation score*

Project (Bug ID)	Classes Under Test	Total No. of Mutants	Randoop		EvoSuite	
			Test Suite	MS	Test Suite	MS
Chart (2)	DatasetUtilities	972	504	27.47	106	36.4
Cli (1)	CommandLine	16	1500	31.25	14	62.5
Codec (2)	Base64	934	1500	57.81	37	32.1
Compress (1)	CpioArchiveOutputStream	395	1500	17.72	22	52.4
Csv (3)	Lexer	99	1091	25.25	22	54.5
Gson (2)	TypeAdapters	266	1209	24.06	66	50.0
JacksonCore (4)	TextBuffer	480	1500	50.83	53	56.5
JacksonDatabind (2)	TokenBuffer	617	341	24.79	128	53.5
Jsoup (5)	Parser	203	1496	51.23	19	17.7
JxPath (5)	NodePointer	274	1500	25.91	56	56.2
Lang (1)	NumberUtils	941	1500	29.30	123	49.7
Math (1)	BigFraction, Fraction	884	1500	59.84	112	63.7
Time (1)	Partial	415	1500	13.01	75	63.4

findings, I carried out an additional smaller study in which I generated five different test suites for 10 projects using the Defects4J framework by changing the random seed. Then, I applied the reduction approach to the generated test suites. Due to time constraints I excluded the "`Compress`", "`Csv`", and "`JxPath`" projects. I followed the same methodology described before, where I used the same absolute numbers for a reduction on both Randoop and EvoSuite. For example, in the "`Codec`" project, the first original test suite size was 14. Thus, I used test suites of sizes five, nine, and 12 on both test suites generated by Randoop and EvoSuite.

3.4 Results

In this section, I report the results of the experiments. I made my experiments on 13 test subjects, taken from the "Defects4J" framework. For each test subject, I generated test suites using EvoSuite and Randoop. Then, I used five similarity metrics and used three different sample sizes for reduction. I ran my experiments 30 times for random, least diverse, and most diverse test suites respectively. Finally, I made an evaluation based on mutation scores. In total, I ran 25,740 experiments and this took around 2,436 hours. Also, I ran another 1,100 experiments using different regression test suites in the smaller study to verify my findings. Table 3.1 presents all 13 test subjects with information about the classes under test, the total number of mutants in each subject, the original test suites' sizes and mutation scores generated from both Randoop and

Table 3.2: *The reduction analysis for the Cli project*

Similarity Metric	**Red. Test Suite Size**	**Least diverse**				**Most diverse**				**Random diverse**				**p-Value**				**Effect size**			
		Randoop		EvoSuite		Randoop		EvoSuite		Randoop		EvoSuite		Randoop		EvoSuite		Randoop		EvoSuite	
		Avg	SD	Avg	SD	Avg	SD	Avg	SD	Avg	SD	Avg	SD	LR	MR	LR	MR	LR	MR	LR	MR
Euclidean	5	25.0	0.00	30.2	15.90	26.9	2.86	51.5	6.79	23.1	7.24	32.9	14.43	0.65	0.24	0.42	α	0.14	0.39	0.41	0.84
	8	25.0	0.00	40.4	14.32	29.8	2.64	60.6	2.86	24.2	7.35	49.2	10.17	0.18	α	0.01	α	0.14	0.69	0.30	0.86
	12	25.0	0.00	59.6	5.29	31.0	1.12	62.5	0.00	27.3	5.93	58.3	5.43	α	α	0.30	α	0.14	0.69	0.30	0.86
Hamming	5	21.9	5.53	26.9	14.08	30.2	2.33	54.6	3.20	23.1	7.24	32.9	14.43	0.03	α	0.11	α	0.11	0.78	0.35	0.92
	8	22.1	5.53	37.5	8.54	31.3	0.00	59.0	3.10	24.2	7.35	49.2	10.17	α	α	α	α	0.11	0.81	0.18	0.79
	12	24.2	2.12	58.5	6.14	31.3	0.00	62.5	0.00	27.3	5.93	58.3	5.43	α	α	0.72	α	0.00	0.56	0.46	0.58
Levenshtein	5	18.1	5.19	22.7	10.76	28.1	3.12	53.5	6.39	23.1	7.24	32.9	14.43	α	0.02	α	α	0.06	0.52	0.26	0.88
	8	21.3	4.45	40.8	12.15	30.0	2.50	61.0	2.64	24.2	7.35	49.2	10.17	α	α	α	α	0.09	0.70	0.27	0.87
	12	23.5	2.64	60.0	5.00	31.0	1.12	62.5	0.00	27.3	5.93	58.3	5.43	α	α	0.19	α	0.00	0.55	0.50	0.58
Manhattan	5	25.0	0.00	28.1	14.77	28.1	3.12	50.0	6.04	23.1	7.24	32.9	14.43	0.65	0.02	0.18	α	0.14	0.52	0.37	0.80
	8	25.0	0.00	39.4	14.08	30.0	2.50	61.3	2.50	24.2	7.35	49.2	10.17	0.18	α	α	α	0.14	0.69	0.28	0.90
	12	25.0	0.00	60.6	4.88	30.6	1.88	62.5	0.00	27.3	5.93	58.3	5.43	α	0.02	0.09	α	0.00	0.52	0.53	0.58
NCD	5	24.4	1.88	25.8	10.42	25.8	5.53	46.7	8.50	23.1	7.24	32.9	14.43	0.34	0.63	0.07	α	0.13	0.45	0.33	0.76
	8	25.0	0.00	44.4	14.19	27.3	4.41	56.5	5.22	24.2	7.35	49.2	10.17	0.18	0.41	0.34	α	0.14	0.48	0.41	0.71
	12	25.0	0.00	50.8	7.86	31.0	1.12	62.5	0.00	27.3	5.93	58.3	5.43	α	α	α	α	0.00	0.55	0.20	0.58

EvoSuite given 3000 seconds and an upper size limit of 1,500. The first column shows the project ID as defined in the Defects4J framework. The second and third columns display the test suite size and test suite mutation score for the Randoop test suites, respectively. The fourth and fifth columns display the test suite size and test suite mutation score for the EvoSuite test suites, respectively.

Tables 3.2 and 3.3 show the analysis results of the "`Cli`" and "`Lang`" projects respectively. The remaining analysis tables for the remaining projects can be found in a public GitHub repository [1] and in Appendix B. In each table, the first column is the used similarity metric, and the second column is the size of the reduced test suite. The third, fourth, and fifth columns display the average (Avg) and standard deviation (SD) for the mutation scores using both Randoop and EvoSuite for least diverse, most diverse, and random reductions, respectively. The sixth column displays the p-values for both tools' statistical significance test between LR (least diverse and random), and MR (most diverse and random). I used the *The Mann-Whitney U test* as my statistical test because the two samples (MR or LR) are independent, and the observations are independent and not normally distributed satisfying the preconditions of the Mann-Whitney U test. An α represents a number lower than 0.001, which is in the 99% significance level. The last column displays the corresponding A_{12} *effect sizes* proposed by Vargha and Delaney [305]. Effect size informs you how meaningful the relationship between LR and MR is. The effect size is in the range $[0, 1]$, where higher values (> 0.8) mean large effect, values around 0.5 mean medium effect, and small values (< 0.2) mean small effect. My experiments show four patterns of performance.

[1]`https://github.com/islamelgendy/Diversity-test-suite-reduction`

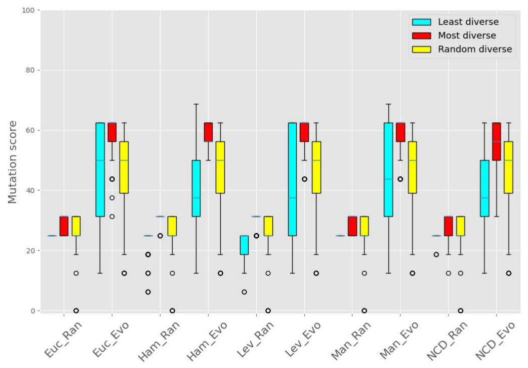
Table 3.3: *The reduction analysis for the Lang project*

Similarity Metric	**Red. Test Suite Size**	**Least diverse**				**Most diverse**				**Random diverse**				**p-Value**				**Effect size**			
		Randoop		EvoSuite		Randoop		EvoSuite		Randoop		EvoSuite		Randoop		EvoSuite		Randoop		EvoSuite	
		Avg	SD	Avg	SD	Avg	SD	Avg	SD	Avg	SD	Avg	SD	LR	MR	LR	MR	LR	MR	LR	MR
	43	2.1	1.13	18.8	0.60	8.7	0.73	19.6	1.03	7.9	1.74	24.5	2.24	α	0.12	α	α	0.00	0.62	0.00	0.00
Euclidean	73	3.6	1.17	32.4	1.75	9.9	0.67	29.6	1.29	11.4	1.90	34.7	2.73	α	α	α	α	0.00	0.21	0.26	0.03
	104	5.2	0.82	44.6	1.07	10.8	0.56	44.2	0.85	13.8	1.99	55.1	1.48	α	α	0.15	0.79	0.00	0.06	0.60	0.48
	43	4.0	1.47	21.1	2.42	8.9	0.73	19.4	1.09	7.9	1.74	24.5	2.24	α	0.05	α	α	0.05	0.65	0.15	0.01
Hamming	73	5.5	1.54	33.8	2.03	11.8	0.86	28.3	0.63	11.4	1.90	34.7	2.73	α	0.70	0.34	α	0.01	0.53	0.43	0.00
	104	6.4	1.77	45.8	0.59	12.6	0.92	42.9	1.74	13.8	1.99	55.1	1.48	α	α	α	0.01	0.00	0.28	0.88	0.31
	43	1.3	0.12	25.4	1.37	9.5	1.18	19.7	1.73	7.9	1.74	24.5	2.24	α	α	0.16	α	0.00	0.76	0.60	0.04
Levenshtein	73	2.0	0.23	35.4	0.76	11.1	0.86	27.6	1.58	11.4	1.90	34.7	2.73	α	0.16	0.13	α	0.00	0.39	0.62	0.00
	104	2.4	0.12	45.5	0.38	12.3	1.06	43.1	0.64	13.8	1.99	55.1	1.48	α	α	α	α	0.00	0.25	0.82	0.26
	43	1.9	0.68	18.9	0.91	8.5	0.76	18.7	1.23	7.9	1.74	24.5	2.24	α	0.33	α	α	0.00	0.57	0.00	0.00
Manhattan	73	3.0	1.13	32.8	1.88	10.3	0.77	29.0	1.16	11.4	1.90	34.7	2.73	α	α	α	α	0.00	0.26	0.28	0.01
	104	4.8	0.95	44.2	1.07	10.8	0.70	42.8	1.48	13.8	1.99	55.1	1.48	α	α	0.81	α	0.00	0.07	0.52	0.27
	43	2.7	0.22	24.9	1.95	10.9	0.82	20.7	2.09	7.9	1.74	24.5	2.24	α	α	0.60	α	0.00	0.94	0.54	0.12
NCD	73	3.6	0.43	36.1	0.70	13.5	1.45	28.9	1.22	11.4	1.90	34.7	2.73	α	α	α	α	0.00	0.81	0.70	0.00
	104	4.4	0.45	44.9	0.41	14.9	1.16	43.1	0.92	13.8	1.99	55.1	1.48	α	0.03	0.03	α	0.00	0.66	0.66	0.28

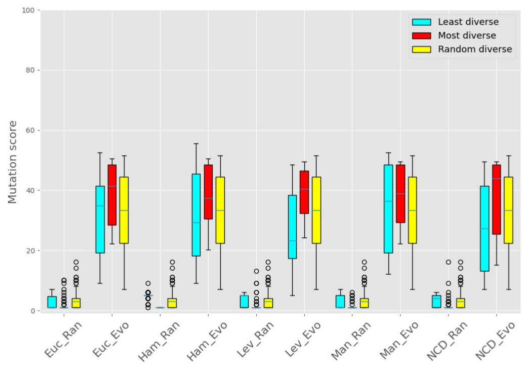
3.4.1 String-Based Similarity Reduction is Better on EvoSuite and Randoop (Pattern 1)

The first and most common pattern is where maximising diversity gave better mutation scores using tests from both tools. Table 3.2 shows the reduction analysis for "`Cli`" as an example of this pattern. Figure 3.2a shows the box plot for the same project showing the achieved mutation scores for least diverse, most diverse, and random reduced test suites. For "`Chart`", "`Cli`", "`Codec`", "`Gson`", "`JacksonCore`", "`Jsoup`", "`JxPath`", and "`Math`" projects, the achieved average mutation scores for the most diverse test suites are higher than randomly reduced test suites and the standard deviations are lower as well. Also, random reduction performs better than reduction based on the least diverse with few exceptions. However, the reduction in Randoop test suites caused a noticeable drop of mutation scores in "`JacksonCore`", "`Jsoup`", "`JxPath`", and "`Math`" projects.

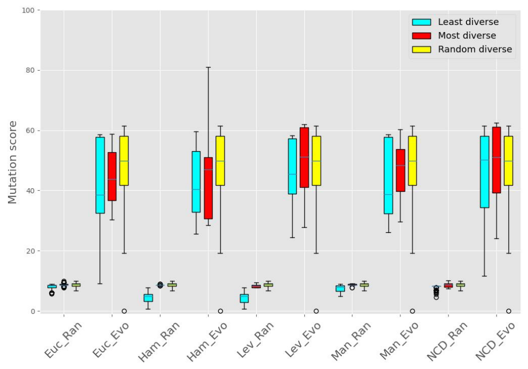
A slight advantage of string-based similarity reduction over random reduction occurred with "`Gson`" and "`JacksonDatabind`" projects. For "`Gson`" project, using Randoop, the achieved mutation score averages for the most diverse test suites are higher than randomly reduced test suites and the standard deviations are lower as well. However, using EvoSuite, for only the Levenshtein metric were the most diverse results better than those of random reduction. The results in other metrics are varying, where there is a slight advantage for the most diverse test suites over random reduction. For "`JacksonDatabind`" project, the achieved mutation score averages for the most diverse test suites are slightly higher than randomly reduced test suites, but the standard deviations are lower making the results more consistent. Also, random reduction is performing slightly better than reduction based on the least diverse across all metrics using both Randoop and EvoSuite.



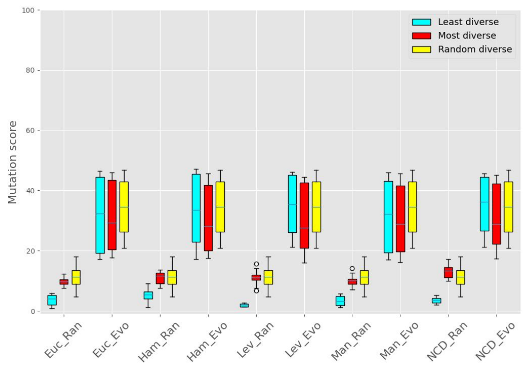
(a) *Example of pattern 1 "`Cli`" project*



(b) *Example of pattern 2 "`Csv`" project*



(c) *Example of pattern 3 "`Time`" project*



(d) *Example of pattern 4 "`Lang`"*

Figure 3.2: *The patterns observed in the experiments. Figure 3.2a shows the "`Cli`" project as an example of the pattern where diversity-based reduction performed better. Figure 3.2b shows the "`Csv`" project as an example of the pattern where diversity-based performed better in EvoSuite, but random reduction was better in Randoop. The "`Time`" project is shown in Figure 3.2c where diversity-based performed better in Randoop but not in EvoSuite. Finally, Figure 3.2d shows the "`Lang`" project where random reduction performed better than diversity-based reduction*

3.4.2 String-Based Similarity Reduction is Better on EvoSuite (Pattern 2)

The second pattern occurs where maximising diversity performed better in EvoSuite but not in Randoop. Figure 3.2b shows the box plot for "`Csv`" as an example of this pattern, showing the achieved mutation scores for least diverse, most diverse, and random reduced test suites. For "`Compress`" and "`Csv`" projects, using EvoSuite, the achieved average mutation scores for the most diverse test suites are higher than randomly reduced test suites in EvoSuite test suites. Also, the standard deviations are lower.

The reduction of the Randoop test suites caused a huge drop in mutation scores using any reduction technique. The mutation score drop from 17.72% to 4.4-6.8% in "`Compress`", while the mutation score dropped from 25.25% to 1.0-5.1% in "`Csv`". There was no clear-cut advantage to either random reduction or reduction based on maximising diversity in "`Compress`". However, for "`Csv`" project reductions based on random were better than most diverse in terms of mutation scores. These results suggest that the reduction was drastic for these two projects, and higher sizes would have achieved better mutation scores. This also might explain why random reduction was slightly better than string-based similarity.

3.4.3 String-Based Similarity Reduction is Better on Randoop (Pattern 3)

The third pattern occurs where maximising diversity performed better in Randoop but not in EvoSuite. Figure 3.2c shows the box plot showing the achieved mutation scores for least diverse, most diverse, and random reduced test suites of "`Time`" project. Using Randoop, the achieved mutation score averages for the most diverse test suites and randomly reduced test suites are very close to each other. However, the standard deviations for most diverse test suites are lower than randomly reduced test suites. On the other hand, using EvoSuite, reductions based on random were better than most diverse in terms of mutation scores. Also, random reduction performed better than the reduction based on the least diverse.

3.4.4 Random Reduction is Better on EvoSuite and Randoop (Pattern 4)

The last pattern is where random reduction performed better than string-based similarity reduction for both Randoop and Evosuite. This only occurred with "`Lang`" project, as the achieved mutation score averages for the randomly reduced test suites, are higher than both most and least diverse test suites with a few exceptions, but the standard deviations are lower for most and least diverse than those in randomly reduced. The exceptions occurred in Randoop using NCD and other metrics with sizes 43 (35% of

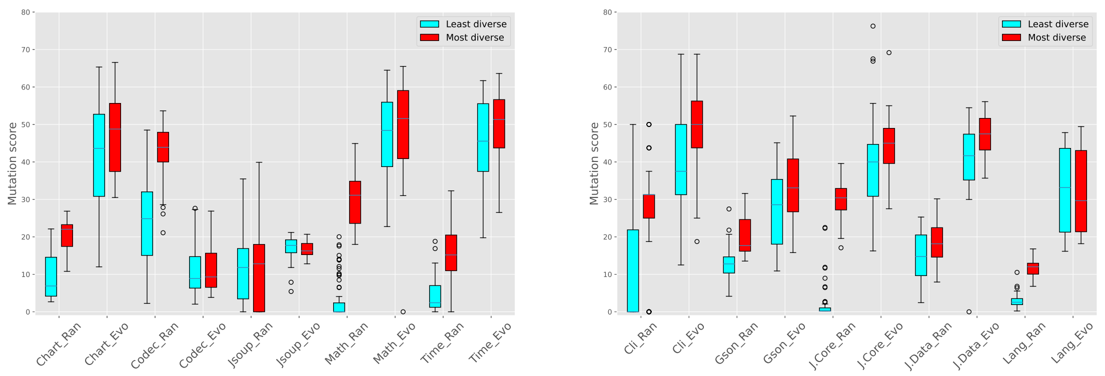


Figure 3.3: *The reduction analysis for all repeated test subjects*

the original size). Furthermore, the reduction of Randoop test suites caused a big drop in mutation scores from 29.3% to 7.9-14.9%. Table 3.3 shows the reduction analysis of the "`Lang`" project, and Figure 3.2d shows the box plot for the same project, showing the achieved mutation scores for least diverse, most diverse, and random reduced test suites.

Figure 3.3 shows the box plots of the reduction analysis of my smaller study. I included all the reduction analysis data using all similarity metrics and different reduction sizes in the plot. The mutation scores for the most diverse test suites are higher than the mutation scores for the least diverse except for the "`Lang`" project. These results are consistent with my findings reported earlier.

3.4.5 Generate Smaller Test Suite Sizes

To solve the problem of having a large regression test suite size, I can apply reduction as I did in this paper. However, another possibility is just to simply set a lower time budget for Randoop and EvoSuite to generate a smaller test suite. Thus, saving even more time than the reduction approach.

Therefore, I evaluated the string-based similarity reduction against using the first-n test cases. Figure 3.4 shows the box plots of all mutation scores for the first-n test case, the worst cases in string-based similarity reduction, and the average scores of string-based similarity reduction.

3.4.6 FAST Reduction

To evaluate the effectiveness of a more scalable reduction approach, I conducted experiments using the FAST method [82], following the procedure outlined in Section 3.2.4.

Table 3.4 presents the mutation scores achieved using Randoop and EvoSuite under four different reduction strategies: least diverse, most diverse (both based on NCD), random, and FAST. The results for the least diverse, most diverse, and random reductions are averaged over 30 runs due to their stochastic nature. In contrast, the

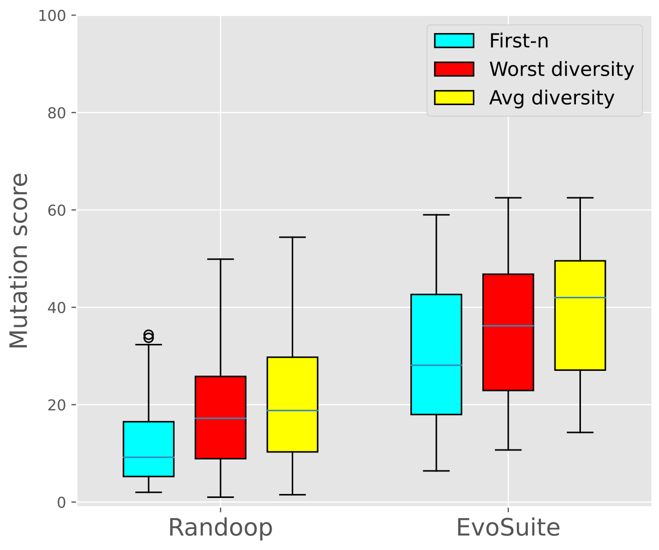


Figure 3.4: *Mutation analysis for all projects presenting the first-*n *test cases, worst cases of maximising diversity, and average scores of maximising diversity*

FAST reduction is deterministic, and its mutation score is reported from a single run. This setup allows for a fair comparison of FAST against both randomised and diversity-based reductions across all projects.

3.5 Discussion

I discuss the findings of my results and relate them to my research questions.

3.5.1 Answer to RQ1 - How does reduction based on maximising and minimising diversity compare to random reduction?

I had a total of 195 records of reduced test suites for each generation tool. Based on average mutation scores for test suites generated by Randoop, the reduction based on maximising diversity was better than random reduction in 71.28% of the cases, where random reduction performed better in 15.9%. In the remaining 12.82% random and most diverse gave almost identical mutation scores (±0.5 difference). For test suites generated by EvoSuite, the reduction based on maximising diversity was better in 70.26% of the cases than random reduction, where random reduction performed better in 17.44%. In the remaining 12.3% random and most diverse gave very close mutation scores.

On the other hand, for Randoop, reduction based on minimising diversity was better only in 11.28% of the cases than random reduction, where random reduction performed better in 78.46% of the cases. In the remaining 10.26% random and most diverse gave almost identical mutation scores (±0.5 difference). For test suites generated by

Table 3.4: *The reduction analysis for all the projects with FAST reduction.*

Project	**Red. Test**	**Least Div**		**Most Div**		**Random**		**FAST**	
	Suite Size	Rand	Evo	Rand	Evo	Rand	Evo	Rand	Evo
	37	7.5	10.2	**17.6**	**19.5**	12.5	16.1	3.8	11.2
Chart	63	10.9	20.8	**20.0**	**26.7**	16.5	24.1	8.0	18.4
	90	15.0	31.9	**31.9**	**34.5**	18.6	31.3	11.0	28.2
	5	24.4	25.8	**25.8**	**46.7**	23.1	32.9	12.5	12.5
Cli	8	25.0	44.4	**27.3**	**56.5**	24.2	49.2	12.5	18.9
	12	25.0	50.8	**31.0**	**62.5**	27.3	58.3	12.5	56.3
	12	37.0	17.7	**50.3**	**19.3**	43.3	18.5	34.6	14.6
Codec	22	45.3	25.1	**54.1**	**26.0**	46.3	24.8	34.7	23.6
	31	47.5	30.2	**55.6**	30.3	47.9	29.1	35.0	**30.4**
	7	1.4	13.1	**4.4**	**42.8**	**4.4**	31.9	2.3	41.3
Compress	13	2.1	25.2	5.4	**49.3**	**6.5**	41.4	2.5	47.8
	18	2.4	42.8	6.0	**51.7**	**6.7**	48.4	2.8	49.1
	7	2.0	11.3	2.0	**23.3**	**2.4**	18.2	1.0	15.2
Csv	13	3.1	27.4	1.3	**44.0**	**3.6**	33.4	3.0	37.4
	18	4.4	44.8	1.5	**49.1**	5.1	45.2	**7.0**	44.4
	23	5.3	15.9	**14.9**	24.1	11.4	**25.5**	5.3	24.8
Gson	39	6.6	25.9	**15.6**	**36.9**	12.3	35.2	5.3	35.7
	56	6.8	42.0	**15.8**	43.1	13.4	**44.0**	5.3	43.6
	18	0.2	31.3	**23.4**	**43.7**	11.4	35.2	7.5	24.8
JacksonCore	31	0.2	48.2	**26.6**	**51.1**	15.9	45.3	10.6	42.7
	45	0.2	**54.8**	**29.2**	54.6	18.4	53.6	11.3	49.0
	44	16.8	34.8	**17.9**	38.8	15.1	**39.7**	16.9	32.9
JacksonDatabind	76	19.3	45.2	**19.6**	**46.8**	18.7	**46.8**	18.6	39.1
	108	20.3	50.3	**20.8**	**51.3**	20.2	50.9	19.1	48.8
	6	12.1	14.7	17.0	14.3	**19.4**	14.2	3.4	**14.8**
Jsoup	11	15.3	15.8	**26.4**	15.8	20.4	**16.2**	3.4	15.8
	16	21.0	16.3	**29.9**	17.1	25.5	17.0	3.4	**17.7**
	19	7.5	21.2	**13.5**	**35.4**	9.1	26.6	5.5	16.4
JxPath	33	10.6	36.5	**16.2**	**46.2**	10.8	38.8	6.2	32.1
	47	12.9	45.7	**18.3**	**53.7**	12.9	50.8	6.9	49.6
	43	2.7	**24.9**	**10.9**	20.7	7.9	24.5	2.8	16.0
Lang	73	3.6	**36.1**	**13.5**	28.9	11.4	34.7	5.6	31.3
	104	4.4	44.9	**14.9**	43.1	13.8	**55.1**	8.2	44.5
	39	0.0	36.4	**28.8**	**40.0**	16.2	37.1	7.1	36.5
Math	67	0.0	51.6	**31.9**	50.1	21.0	49.8	11.2	**52.8**
	95	0.0	57.6	**34.3**	59.1	25.3	59.0	14.4	**61.0**
	26	7.4	26.2	8.2	36.0	**8.3**	**36.5**	2.4	31.6
Time	45	8.1	49.2	**8.6**	**53.0**	**8.6**	49.1	2.4	51.8
	63	8.2	59.1	**9.1**	61.4	**9.1**	58.9	2.4	**63.4**

EvoSuite, the reduction based on minimising diversity was better in 20% of the cases than random reduction, where random reduction performed better in 68.21%. In the remaining 11.79% of the cases random and most diverse gave very close mutation scores.

Conclusion for RQ1: The experiments show that reduction based on maximising diversity has higher mutation scores than random reduction in 71.28% of the cases for Randoop and 70.26% of the cases for EvoSuite. Also, reduction based on minimising diversity has lower mutation scores than random reduction in 78.46% of the cases for Randoop and 68.21% of the cases for EvoSuite. I conclude that reduction based on similarity plays a role in fault-finding capabilities.

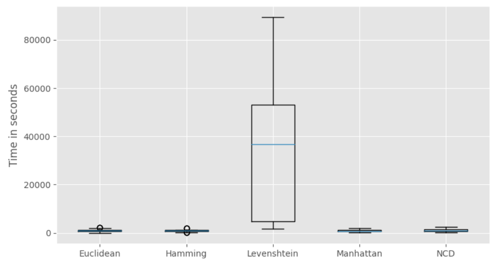
3.5.2 Answer to RQ2 - How does string-based similarity reduction compare to a lower time budget for the test generation?

As I mentioned before, setting a lower time budget will generate a smaller test suite. However, the generated test suites will have much smaller fault-detection capabilities. The average of string-based similarity reduction gave better mutation scores than the first-n test cases in 88.5% of the cases. Furthermore, even the worst cases of string-based similarity reduction were still better than the first-n test cases in 74.4% of the cases.

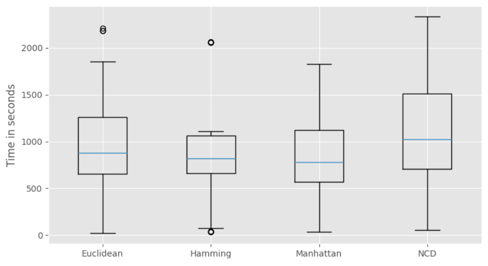
Conclusion for RQ2: String-based similarity reduction is better than using the first-n test cases in terms of fault-detection. Therefore, it is more effective to generate a large regression test suite, and then reduce it based on similarity.

3.5.3 Answer to RQ3 - Which similarity metric performs the best in terms of time to compute and loss of fault-finding capability?

For test suites generated by Randoop, NCD performed best in 38.5% of the cases giving the highest average mutation scores for “`Chart`”, “`Gson`”, “`JxPath`”, “`Lang`”, and “`Math`” projects. Levenshtein came in second performing best in 30.8% of the cases giving the highest average mutation scores for “`Codec`”, “`Compress`”, “`JacksonDatabind`”, and “`Jsoup`” projects. Manhattan, Hamming, and Euclidean gave the highest average mutation scores for “`JacksonCore`”, “`Cli`”, and “`Csv`” projects respectively. There was no clear advantage for any of the metrics in the “`Time`” project. Manhattan distance gave the highest for the 35% test suite size, while Euclidean distance gave the highest for the 60% test suite size, and NCD gave the highest for the 85% test suite size.



(a) *The reduction time for all metrics*



(b) *A closer inspection of the metrics without the Levenshtein metric*

Figure 3.5: *Test suite reduction time*

For test suites generated by EvoSuite, NCD performed best in 46.2% of the cases giving the highest average mutation scores for "`Codec`", "`Compress`", "`Csv`", "`JxPath`", "`Lang`", and "`Time`" projects. Manhattan distance came in second performing best in 30.8% of the cases giving the highest average mutation scores for "`Cli`", "`Chart`", "`Jsoup`", and "`Math`" projects. Hamming distance gave the highest average mutation score for "`JacksonCore`", and "`JacksonDatabind`" projects. Also, it is worth noting that Levenshtein and NCD both performed the best in the "`JxPath`", and "`Time`" projects. There was no clear advantage for any of the metrics in the "`Gson`" project.

Similarity metrics varied in effectiveness from one project to another and varied between Randoop and EvoSuite as well. However, NCD performed better than the other metrics at 42.3% across all experiments using both Randoop and EvoSuite. Also, NCD performed the best using Randoop and EvoSuite in the "`JxPath`", and "`Lang`" projects.

The box plots of Figure 3.5 show the time for the reduction of the compared similarity metrics. In Figure 3.5a, all similarity metrics are shown. However, it takes much more time to compute the Levenshtein metric than the others, so I can not observe the time spent on them properly. Thus, the box plots in Figure 3.5b show only the Euclidean, Hamming, Manhattan, and NCD metrics to get a better view of the time spent in reduction. As shown, Levenshtein takes the most time by far, which conforms with Ledru et al. [184]. NCD comes in next as it spends a little more time computing than Euclidean, Hamming, and Manhattan, which are very close to one another in time spent for reduction. NCD was nine to 53 times faster to compute than Levenshtein distance.

Conclusion for RQ3: The best similarity metric in terms of maintaining fault-finding capabilities is NCD. NCD performed best in 42.3% of cases in test suites generated by Randoop and EvoSuite. Also, the reduction time spent is close to other metrics. The Levenshtein comes in second performing best in 23% of cases, followed by Manhattan performing 19.2% of cases. However, Levenshtein distance spends a huge amount of time to compute compared to other metrics. It is nine to 53 times slower than NCD.

3.5.4 Answer to RQ4 - What is the effect of string-based similarity reduction on automatically generated test suites with no built-in minimisation (Randoop) compared to an already minimised test suites (EvoSuite)?

The test suite sizes generated by EvoSuite are already minimised, and further reduction of the test suites can negatively affect the fault-finding capabilities. However, if resources are limited, reduction based on the most dissimilar test cases is preferable. I managed to achieve a 15% reduction in size with only a 5% drop in mutation scores. On the other hand, the test suites generated by Randoop are considerably larger. In the "`Cli`" project, I managed to achieve the mutation score of the original test suite after reducing the size by 99.2%. In the "`Codec`" project, I maintained 97.2% of the mutation score of the original test suite after making a 97.9% reduction. Although other projects did not achieve strong results, in almost every project I made a huge reduction in size.

Considering all 13 test subjects, where I made reduction using five similarity metrics on three different size batches, I have a total of 195 records of most and least diverse test suites per tool (*Randoop and EvoSuite*). The most diverse test suites performed better at 175 for Randoop (*89.7%*), and 159 for EvoSuite (*81.5%*).

Conclusion for RQ4: Randoop can benefit much more from similarity-based reduction, often reducing the size of the test suites considerably and maintaining either the same mutation score of the original or a very close number. On the other hand, since EvoSuite has its minimisation techniques, any further reduction tends to drop the mutation score. However, string-based similarity reduction still achieved a 15% reduction in size with only a 5% drop in mutation scores.

3.5.5 Answer to RQ5 - How does test suite reduction using FAST compare to reductions based on maximising diversity, minimising diversity, and random selection?

To investigate the performance of a scalable reduction approach, I used FAST [82] and compared its mutation scores to those of the most diverse, least diverse, and random reductions. Unlike the other reduction techniques, FAST is deterministic and produces consistent results across runs, whereas the others are stochastic and averaged over 30 executions.

For test suites generated by Randoop, FAST achieved lower mutation scores than reduction based on maximising diversity in 95% of the cases. It also underperformed compared to random reduction in 95% of the cases, and even minimising diversity achieved better results in 61.5% of the cases.

For EvoSuite-generated test suites, which are already minimised, the difference in performance was slightly less dramatic but still significant. FAST underperformed compared to maximising diversity in 84.6% of the cases, to random reduction in 69.2% of the cases, and had comparable performance to minimising diversity, with the latter outperforming FAST in 48.7% of the cases.

This suggests that FAST, while scalable, does not preserve fault-detection effectiveness as well as the other methods, especially, in the context of Randoop-generated test suites.

Conclusion for RQ5: Although FAST is efficient and scalable, its reduction quality in terms of mutation scores is noticeably lower than that of maximising diversity using traditional pairwise and random reduction. In Randoop-generated test suites, FAST was outperformed in 95% of the cases by both maximising diversity and random reduction, and even minimising diversity performed better in 61.5% of the cases. In EvoSuite, FAST was also outperformed in 84.6% of the cases by maximising diversity and in 69.2% by random reduction, while performing similarly to minimising diversity. This suggests that FAST prioritises speed at the cost of fault-detection effectiveness.

3.5.6 Threats to Validity

I address validity threats that can affect my results.

1. ***Construct validity:*** The selected similarity metrics and the used test subjects can be a construct risk in my study. To mitigate this risk, I selected widely-used metrics in software testing research, and the dataset used [164] has been constructed by external researchers and is also used in many research studies such as [79, 82].

2. ***Internal validity:*** The accuracy of the results can be affected by random factors. To mitigate this risk, I repeated the reduction 30 times for each of the random, least and most diverse test suites. Also, I applied the same reduction approach using five different test suites for each test subject.

3. ***External validity:*** This risk is concerned with how much can I generalise the results to different Java programs and different test suites generated by different

tools, or even developer-written test suites. To mitigate this risk, I used 13 different projects. However, I still need to use more test subjects and conduct more experiments to try to draw general conclusions.

4. ***Reliability:*** This concerns how other researchers can replicate my study. To mitigate this risk, I explained the reduction approach and similarity metrics used. Also, I specified the test subjects which can be accessed from the web to replicate the study. Furthermore, all the data and test subjects are available on the web in a replication package [96].

3.6 Chapter Conclusions

Regression test suites usually have numerous test cases, and running the entire test suite will be costly. This chapter evaluates the effectiveness of applying similarity metrics to reduce the size of automatically generated regression test suites. The study used two widely used tools, Randoop and EvoSuite, that generate test suites. Five different similarity metrics were employed in the study, Euclidean distance, Hamming distance, Levenshtein distance, Manhattan distance, and NCD. I compared these similarity metrics, and compared the reduced test suites from random (Randoop) and search-based (EvoSuite) in terms of fault-finding capabilities. The reduction approach was evaluated on 13 test subjects from the Defects4J framework available on the web.

The results showed that reducing test suites based on maximising diversity is more efficient than random reduction in 71.28% of the cases in test suites generated by Randoop, and 70.26% of the cases in test suites generated by EvoSuite. When minimising diversity, random reduction gave higher mutation scores in 78.46% of the cases using Randoop and 68.21% of the cases using EvoSuite. Also, the comparison between the similarity metrics shows that NCD is the best similarity metric. The time to compute NCD is comparable to other similarity metrics and much faster than the Levenshtein metric, and NCD performed better than the other metrics in 42.3% of the experiments. Furthermore, string-based similarity reduction is more effective than using the first n test cases in terms of fault-detection. Also, I investigated the FAST approach, a scalable and deterministic TSR method. While FAST is computationally efficient, it underperformed in terms of mutation score compared to the other reduction methods. In test suites generated by Randoop, FAST was outperformed by maximising diversity and random reduction in 95% of the cases. For EvoSuite-generated test suites, FAST was outperformed by maximising diversity in 84.6% of the cases and by random reduction in 69.2% of the cases. These results suggest that although FAST offers scalability, this comes at the cost of reduced fault-detection effectiveness.

Overall, my findings indicate that it is possible to significantly reduce the size of automatically generated test suites, particularly those generated by Randoop, without sacrificing their fault-detection capability. The results also highlight the trade-off between efficiency and fault-detection, motivating further exploration of alternative representations and artefacts that can achieve scalability without losing effectiveness. In the next chapter (Chapter 4), I investigate bytecode as a compact and efficient diversity artefact to address these scalability challenges, building on the insights gained from textual similarity-based reduction.

Chapter 4

Empirically Evaluating the Use of Bytecode for Diversity-Based Test Case Prioritisation

The contents of this chapter are based on "I. Elgendy, R. Hierons, and P. McMinn. Empirically evaluating the use of bytecode for diversity-based test case prioritisation. In Proceedings of the International Conference on Evaluation and Assessment in Software Engineering (EASE), 2025.".

4.1 Introduction

In Chapter 3, I investigated how textual diversity can guide test suite reduction, showing that selecting maximally diverse tests improves fault detection while significantly reducing suite size. However, this study also highlighted a key challenge: textual representations are verbose and computationally expensive for large suites, and not all text contributes meaningfully to differentiating tests. These limitations motivated the exploration of more compact and semantically meaningful representations, leading to the study presented in this chapter.

Another widely used technique to optimise regression testing is Test Case Prioritisation (TCP). TCP reorders test cases to maximise early fault detection, particularly when testing has to be prematurely halted due to budgetary constraints [334]. In the domain of software testing research, TCP has garnered significant attention [204, 167, 145, 334], with diversity-based testing techniques playing a pivotal role [100, 184, 226], as discussed in Chapter 2.

TCP approaches that utilise the source code of the test cases can be classified mainly as static techniques (i.e., techniques that do not require test suite execution), and dynamic techniques (i.e., techniques that require test suite execution). An example of a static technique is using the source code of the test cases [184], while an example of a dynamic technique is using coverage information [273]. Static approaches may be less effective, but they do not require any complex program instrumentation and program execution, and can be used in black-box testing scenarios.

Many diversity-based TCP techniques leverage different testing artefacts, with static approaches that rely on test case text being commonly used and showing promising results [184, 226, 323, 312, 21]. These static techniques simplify the process by avoiding program instrumentation or model construction. However, relying on test case text has inherent limitations. The string-based representation of test cases is verbose and often large, increasing the time required for similarity calculations, particularly with complex distance metrics like Levenshtein distance [97]. Furthermore, I hypothesise that test cases contain extraneous information, such as variable names, comments, and assertion messages, which can distort similarity calculations and affect prioritisation.

Miranda et al. [226] introduced the FAST family of TCP approaches to address runtime concerns with large test suites by leveraging big data techniques such as Shingling, Minhashing, and Locality-Sensitive Hashing to compute test similarity for TCP. However, these techniques still measure similarity based on test case text, and as shown in Chapter 3, has a negative impact of fault detection. While preprocessing to remove comments and whitespace can mitigate some issues, it does not eliminate all irrelevant information. A more concise and efficient representation of test cases could address these limitations by reducing the size of the data used for similarity calculations and eliminating irrelevant information.

In this chapter, I propose to use the bytecode of the test cases as the basis for diversity calculations. Bytecode, the intermediate code executed by a virtual machine, offers a more abstract representation than machine code but is more concrete than high-level programming language code. The bytecode is smaller than the source code, as it is a condensed form of the original source code that strips away information such as comments, whitespace, and variable names. It retains only execution-critical details, making it more efficient to store and process than source code. While the use of bytecode may reduce diversity, my hypothesis it does not hinder effectiveness. Moreover, not all bytecode instructions differentiate tests. Some instructions, such as loading and storing variables in/from the stack, serve a mechanical role rather than providing insight into the logic of the test. Therefore, I further propose to filter out these types of bytecode instructions and extract only instructions that relate to the functionality being tested, such as setting up specific values or calling methods that are central to the test's purpose.

I implemented two state-of-the-art static TCP approaches, Ledru-TCP [184] and FAST-TCP [226], applying bytecode similarity for the first time instead of textual similarity. I evaluated the effectiveness and efficiency of bytecode diversity against traditional static TCP approaches using textual diversity. As I explained in Section 2.4, the Levenshtein distance suits string data representations and varying test sizes. I used the Levenshtein distance to measure textual similarity, and the same metric for bytecode similarity to ensure consistency. Results show that bytecode-based TCP outperforms text-based TCP, achieving 2.3%–7.8% higher APFD and up to 3.9 times better real-fault detection. The filtering bytecode approach improved APFD by 4.5% to 5.1% and doubled real fault detection compared to textual information.

Additionally, bytecode diversity-based methods are significantly faster than textual diversity-based ones. The use of all bytecode information was found to be 2–3 orders of magnitude faster than using text in Ledru-TCP and 2.5 to 6 times faster in FAST-TCP. Filtering bytecode information was 455–2798 times faster than text in Ledru-TCP and 4–17.9 times faster in FAST-TCP.

Finally, I evaluated how closely static bytecode diversity TCP matches dynamic TCP performance. I implemented two dynamic TCP techniques using greedy total and greedy additional algorithms [273]. As expected, dynamic approaches, with more information, outperformed bytecode diversity in fault detection, achieving 2.8%–3.5% higher APFDs. However, the best bytecode TCP approach was 6.5–8.3 times faster and avoids the complexity of program instrumentation required for coverage-based techniques and can be used in black-box testing scenarios. I provide a replication package [99] with detailed descriptions of the system.

The efficiency and effectiveness of bytecode diversity explored here lays the foundation for the next chapter (Chapter 5), where these static diversity techniques, together with dynamic coverage information, are applied to kill hard-to-kill mutants in test selection.

The chapter offers the following contributions:

1. The first known use of bytecode of test cases as a basis for measuring their diversity in a TCP technique (Section 4.3.1).

2. A novel bytecode filtering method for further enhancing TCP efficiency (Sections 4.3.2 and 4.5.2).

3. An evaluation of bytecode diversity against textual diversity-based approaches, providing new insights into their effectiveness in mutant and real-fault detection, as well as runtime performance (Section 4.5.1).

4. An empirical comparison of static and dynamic TCP approaches with bytecode diversity-based TCP, using data from seven projects and 97 versions from Defects4J, offering insights into the benefits of bytecode diversity (Section 4.5.3).

4.2 Background

In this section, I briefly explain TCP in general and diversity-based TCP approaches in particular.

4.2.1 Test Case Prioritisation

Test case prioritisation (TCP) re-orders test cases so that the tester gets the maximum benefit if testing is prematurely stopped at some arbitrary point due to some budget

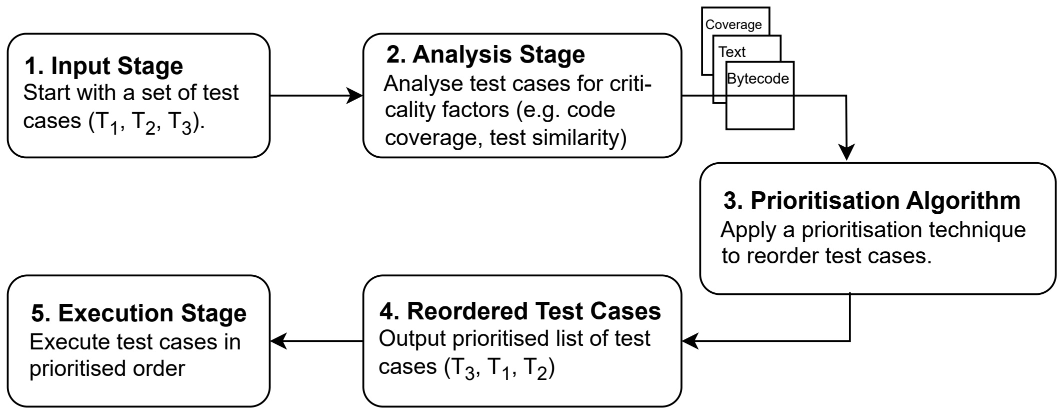

1. Input Stage
Start with a set of test cases (T1, T2, T3).
2. Analysis Stage
Analyse test cases for criticality factors (e.g. code coverage, test similarity)
Coverage
Text
Bytecode
3. Prioritisation Algorithm
Apply a prioritisation technique to reorder test cases.
4. Reordered Test Cases
Output prioritised list of test cases (T3, T1, T2)
5. Execution Stage
Execute test cases in prioritised order

Figure 4.1: *TCP process diagram*

constraints [334]. Figure 4.1 presents the general TCP process diagram showing the main stages of TCP. The primary goal of TCP is to identify and execute the most critical test cases earlier in the testing process. This leads to earlier fault detection and better resource utilisation. Rothermel et al. [273] formally defined TCP as follows: Given a test suite T, the set of permutations P of T, and an award function f from P to real numbers, find $(T' \in P)$ such that $\forall T''$, where $T'' \in P$ and $T'' \neq T'$, $f(T') \geq f(T'')$. The award function f can be based on several criteria, such as code coverage [272, 273], fault detection history [172, 129], high-risk features [93], software requirements [294, 24], or diversity of test cases [184, 226].

A well-known metric to evaluate TCP approaches is the Average Percentage of Faults Detected (APFD). The faults can be real faults or mutants. It is calculated using the following equation:

$$APFD = 1 - \frac{TF_1 + TF_2 + \cdots + TF_m}{nm} + \frac{1}{2n} \tag{4.1}$$

where n is the number of tests in the test suite, m is the number of killable faults, and TF_i is the position of the first test case that detects the fault i.

4.2.2 Diversity-Based TCP

In this study, I focus on diversity-based TCP, where the function f assesses test cases based on their diversity, either by rewarding those that are most dissimilar to others in an incrementally constructed subset P or by evaluating the overall diversity of the entire test suite. In Section 2.6.2, I covered diversity-based techniques in TCP. First, diversity-based TCP collects the relevant information of the test case. These can be the source code of the test cases, the inputs to the test, the outputs of the test cases, etc. Then, the pairwise dissimilarity values between test cases are measured according to some distance metric, such as Euclidean distance, Jaccard distance, Levenshtein distance, etc. A “dissimilarity value” is a measure that quantifies how different two test cases are, typically based on shared characteristics or features. In my study, a lower dissimilarity value indicates greater similarity, while a higher value reflects greater dissimilarity. Finally, test cases are ordered based on the dissimilarity values. The ordering occurs in iterations, where in each iteration a test case is removed from the test suite and added to the new permutation set. The first test case to be added to the permuted set can be selected randomly, or might be the most dissimilar test case from all others. Then, subsequent tests are selected one at a time, where the most dissimilar test from the ones already in the permuted set is selected, and the process continues until the test suite is completed.

Ledru et al. [184] proposed such a diversity-based TCP approach. They used five different distance metrics to calculate the pairwise dissimilarity values between test cases. The dissimilarity values are stored in a “similarity matrix”. Miranda et al. [226] developed the FAST family of TCP techniques, which are quick and scalable for large test suites. They applied some big data techniques, such as Shingling, Minhashing, and Locality-Sensitive Hashing to calculate the dissimilarity between test cases. In both works, the source code of the test cases is used as a basis to calculate dissimilarity. I will expand more on these works in Section 4.3.3.

4.2.3 Bytecode

Bytecode is an intermediate representation of source code, designed to be executed by a virtual machine rather than directly by the hardware. Bytecode is generated by a compiler from the source code and is typically more compact, as it omits non-essential elements like comments and variable names. This compactness enhances processing efficiency and reduces storage requirements, making bytecode an attractive artefact for TCP.

Java Bytecode is the Java Virtual Machine (JVM)-executable intermediate representation compiled from Java source code. Java Bytecode consists of a set of instructions and operands encoded in a binary format, providing a structured and efficient means of representing the program’s logic and structure. ASM [102], a widely used library, enables dynamic bytecode manipulation, supporting class transformation, instrumentation, and analysis.

4.3 Approach

In this section, I show a motivating example and describe the TCP approaches used in my study.

```java
public void testParseBig() {
    BigFraction f1 = improperFormat.parse("16721..." +
              " / " +  "532255...");
    Assert.assertEquals(FastMath.PI,
        f1.doubleValue(), 0.0);
    BigFraction f2 = properFormat.parse("3 " +
              "75363..." + " / " + "53225...");
    Assert.assertEquals(FastMath.PI,
        f2.doubleValue(), 0.0);
    Assert.assertEquals(f1, f2);
    BigDecimal pi = new BigDecimal("3.14159...");
    Assert.assertEquals(pi, f1.bigDecimalValue(99,
        BigDecimal.ROUND_HALF_EVEN));
}
```

(a) *Test case 1*

```java
public void testFormatNegative() {
    BigFraction c = new BigFraction(-1, 2);
    String expected = "-1 / 2";

    String actual = properFormat.format(c);
    Assert.assertEquals(expected, actual);

    actual = improperFormat.format(c);
    Assert.assertEquals(expected, actual);
}
```

(b) *Test case 2*

```java
public void testFormatZero() {
    BigFraction c = new BigFraction(0, 1);
    String expected = "0 / 1";

    String actual = properFormat.format(c);
    Assert.assertEquals(expected, actual);

    actual = improperFormat.format(c);
    Assert.assertEquals(expected, actual);
}
```

(c) *Test case 3 (original)*

```java
public void testFormatZero() {
    BigFraction bigFraction = new BigFraction(0, 1);
    // The expected output
    String output = "0 / 1";

    // The actual output
    String actual = properFormat.format(bigFraction);
    Assert.assertEquals(output, actual);

    actual = improperFormat.format(bigFraction);
    Assert.assertEquals(output, actual);
}
```

(d) *Test case 3 (variation)*

Figure 4.2: *An example of four test cases from the Math project, with test case 3 having two variations*

4.3.1 Motivating Example

Consider three test cases written by developers for the Apache Commons Math project shown in Figure 4.2 parts a)-c). I made a variation of test case 3, to showcase the problem, shown in Figure 4.2d. The test cases are designed to test the `BigFraction` class. Figure 4.3 shows the bytecodes of the test cases shown in Figure 4.2 in the human-readable and hexadecimal forms.

Let me take the example of the method `testFormatNegative` shown in Figure 4.2b and its corresponding bytecode shown in the second column of Figure 4.3a to illustrate how high-level Java instructions translate into low-level bytecode instructions. In the source code, a `BigFraction` object is created and initialised with the values -1 and 2. The bytecode reflects this sequence starting with the `new` instruction at index 0, which allocates memory for the new `BigFraction` object. The `dup` instruction at index 3 duplicates the top value of the stack to prepare for the constructor invocation, and the `iconst_m1` and `iconst_2` instructions at 4 and 5 push the values -1 and 2 onto the stack, respectively. The `invokespecial` instruction at 6 calls the constructor to initialise the object. Bytecode method invocation instructions are represented by `invokevirtual`, `invokestatic`, and `invokespecial`. Other bytecode instructions that involve loading and storing references to/from local variables are `aload` and `astore`, respectively. Field access instructions, like `properFormat` or `improperFormat.format`, are represented by `getfield`, `putfield`, and `getstatic`.

```
testParseBig:                     testFormatNegative:               testFormatZero:
0: aload_0                        0: new              #8            0: new              #8
1: getfield        #3             3: dup                            3: dup
4: ldc             #50            4: iconst_m1                      4: iconst_0
6: invokevirtual   #19            5: iconst_2                       5: iconst_1
9: astore_1                       6: invokespecial   #9             6: invokespecial   #9
10: ldc2_w         #52            9: astore_1                       9: astore_1
13: aload_1                       10: ldc            #13            10: ldc            #14
14: invokevirtual  #54            12: astore_2                      12: astore_2
17: dconst_0                      13: aload_0                       13: aload_0
18: invokestatic   #55            14: getfield       #2             14: getfield       #2
21: aload_0                       17: aload_1                       17: aload_1
22: getfield       #2             18: invokevirtual  #11            18: invokevirtual  #11
25: ldc            #56            21: astore_3                      21: astore_3
27: invokevirtual  #19            22: aload_2                       22: aload_2
30: astore_2                      23: aload_3                       23: aload_3
31: ldc2_w         #52            24: invokestatic   #12            24: invokestatic   #12
34: aload_2                       27: aload_0                       27: aload_0
35: invokevirtual  #54            28: getfield       #3             28: getfield       #3
38: dconst_0                      31: aload_1                       31: aload_1
39: invokestatic   #55            32: invokevirtual  #11            32: invokevirtual  #11
42: aload_1                       35: astore_3                      35: astore_3
43: aload_2                       36: aload_2                       36: aload_2
44: invokestatic   #12            37: aload_3                       37: aload_3
47: new            #57            38: invokestatic   #12            38: invokestatic   #12
50: dup                           41: return                        41: return
51: ldc            #58
53: invokespecial  #59
56: astore_3
57: aload_3
58: aload_1
59: bipush         99
61: bipush         6
63: invokevirtual  #60
66: invokestatic   #12
69: return
```

(a) *The human-readable form*

```
testParseBig:
19 00 B4 12 B6 3A 01 12 19 01 B6 0E B8 19 00 B4 12 B6 3A 02 12 19 02 B6 0E B8 19 01 19 02 B8 BB 59 12
     B7 3A 03 19 03 19 01 10 63 10 06 B6 B8 B1

testParseBig-filtered:
B4 B6 B6 0E B8 B4 B6 B6 0E B8 B8 59 B7 10 63 10 06 B6 B8 B1

testFormatNegative:
BB 59 02 05 B7 3A 01 12 3A 02 19 00 B4 19 01 B6 3A 03 19 02 19 03 B8 19 00 B4 19 01 B6 3A 03 19 02 19
     03 B8 B1

testFormatNegative-filtered:
59 02 05 B7 B4 B6 B8 B4 B6 B8 B1

testFormatZero:
BB 59 03 04 B7 3A 01 12 3A 02 19 00 B4 19 01 B6 3A 03 19 02 19 03 B8 19 00 B4 19 01 B6 3A 03 19 02 19
     03 B8 B1

testFormatZero-filtered:
59 03 04 B7 B4 B6 B8 B4 B6 B8 B1
```

(b) *The Hexadecimal form*

Figure 4.3: *The human-readable and the Hexadecimal formats of the bytecodes for the tests in Figure 4.2*

The textual test cases from Figure 4.2 are 705, 258, 252, and 320 bytes, respectively. Textual representations of test cases include additional elements such as variable names, comments, and formatting, which do not contribute to test behaviour but can inflate dissimilarity calculations. For instance, `testFormatZero` has two variations (Figures 4.2c and 4.2d), differing only in variable names and comments—purely informational elements to the tester that do not affect execution. However, textual diversity methods treat these variations as more dissimilar than they actually are. In contrast, bytecode diversity captures only the core executable logic, leading to a more compact and precise representation. The two variations of `testFormatZero` produce identical bytecode despite textual differences, as shown in the last column in Figure 4.3a.

The bytecode sizes are just 48, 37, and 37 bytes, respectively. The significant difference in size between the textual representation and the bytecode form highlights the potential inefficiencies of using textual diversity in TCP. Using Levenshtein distance, the textual dissimilarity between test cases 1 and 2 is 545, whereas their bytecode dissimilarity is only 31. For test case 2 (Figure 4.2b) and the original test case 3 (Figure 4.2c), the textual dissimilarity is 13, increasing to 69 in the second variation (Figure 4.2d) due to extraneous information (i.e., a difference of 56). In contrast, both variations of test case 3 produce identical bytecode, resulting in a consistent bytecode dissimilarity of just 2.

Clearly, bytecode is significantly smaller than its textual counterpart, reducing the computational overhead involved in calculating similarity values. This can lead to faster execution of TCP processes. Also, by focusing on the actual executable logic, bytecode diversity can avoid being potentially misled by superficial differences in the textual representation, leading to a more accurate assessment of the diversity between test cases.

4.3.2 Filtering Bytecode

When analysing the diversity of test cases based on their bytecode instructions, note that not all instructions contribute equally to the logical complexity or uniqueness of a test. Certain bytecode instructions, while essential for execution, are ubiquitous and serve a mechanical role rather than reflecting the test's unique logic. For instance, instructions like `aload` (loading a variable onto the stack) and `astore` (storing a variable from the stack) are necessary for code execution but are common across most tests and do not provide meaningful insight into the test's distinct behaviour.

In contrast, instructions such as `iconst` (pushing constant values onto the stack) or method invocations like `invokevirtual` and `invokestatic` better reflect the test's intent and logic. These instructions directly relate to the functionality being tested, such as initialising specific values or invoking methods that are central to the test's purpose. By focusing on these substantive instructions, diversity calculations can more accurately capture the variation in test logic and functionality by excluding routine operations that do not contribute to the core logic of the test case.

Filtering out routine bytecode instructions from diversity calculations will have a significant advantage: it reduces the overall size of the resulting bytecode. Many bytecode instructions, such as `aload` and `astore`, occur frequently and contribute to the bulk of the bytecode without adding meaningful diversity in terms of test logic. By excluding these routine instructions, the size of the bytecode representation for each test case becomes smaller and more concise. Figure 4.3b shows the hexadecimal forms of the bytecode and the filtered versions.

This filtering approach allows for a more meaningful analysis of the test suite's variety, highlighting the differences in test goals and behaviours rather than superficial variations in how variables are manipulated. Also, the reduction in bytecode size has a direct impact on the efficiency of diversity calculations. With a smaller set of instructions to analyse, the computational overhead for comparing test cases decreases significantly. This leads to faster processing times, making it feasible to evaluate large test suites more efficiently.

4.3.3 TCP Approaches

I implemented Ledru-TCP [184] and FAST-TCP [226], two state-of-the-art TCP approaches. Ledru-TCP and FAST-TCP are general TCP algorithms that can operate on either textual or bytecode information. When using textual information, they are referred to as Ledru-Text and FAST-Text, respectively, and when using bytecode, they are referred to as Ledru-Bytecode and FAST-Bytecode.

For Ledru-Bytecode, I used the ASM Java library to read the bytecode of the test cases and applied the Levenshtein distance [189] to calculate the pairwise dissimilarity values and save them in a similarity matrix. Algorithm 2 shows the steps of the Ledru-TCP [184]. The information I have is the pairwise similarity matrix Sim_T, and the list of tests T. The similarity matrix is a two-dimensional array, where each cell contains the textual/bytecode dissimilarity value between two test cases. I keep another list of the permuted tests P, which is initially empty (Line 1). First, using the similarity matrix, the minimum value of each row is saved into an array `minValues` (Line 2). For the first selection, I get the test that has the maximum value of the

Algorithm 2: Ledru-TCP algorithm, summarised from [184]

```
   Input: List of tests $T$; Similarity matrix $Sim_T$
   Output: Prioritised test suite $P$
 1 $P \leftarrow \emptyset$ // First selection
 2 $minValues \leftarrow getMinValues(Sim_T, T)$
 3 $maxKey = getMaxOfMins(minValues)$
 4 $P \leftarrow Append(P, T[maxKey])$
 5 $T \leftarrow Remove(T, T[maxKey])$ // Subsequent selections
 6 while $|T| \neq 0$ do
 7     $minValues \leftarrow getMinValues(Sim_T, P)$
 8     $maxKey = getMaxOfMins(minValues)$
 9     $P \leftarrow Append(P, T[maxKey])$
10     $T \leftarrow Remove(T, T[maxKey])$
11 return $P$
```

`minValues` (Line 3). This is the test that is the farthest away from all the rest. If multiple candidates have the same maximum value, I pick the first one. I add this test into P and remove it from T (Lines 4 and 5).

For the subsequent selections, I calculate the minimum values of P from the matrix and save them into `minValues` (Line 7). Then I get the test that has the maximum value of the new `minValues` array (Line 8). This is the test that is the farthest away from all the tests already selected in P. This test is added to P and removed from T (Lines 9 and 10), repeating until all tests are selected.

In FAST-Bytecode, I used the ASM Java library to read the bytecodes, which are then converted into a hexadecimal format to serve as the encoded representation of the test suite. In Chapter 3, I explained the steps involved in FAST-TCP in Algorithm 1. In this chapter, the encoded representation of the test suite T, can be either the text of the test cases or the bytecode representations.

4.4 Evaluation

I evaluated the proposed use of bytecode as a diversity artefact in TCP with the textual artefact in two static TCP approaches. I described the diversity-based TCP approaches in Section 4.3.3.

Additionally, I compared the static TCP approach using bytecode with dynamic TCP approaches. Thus, I implemented two coverage-based TCP techniques, Greedy Total and Greedy Additional. First, I selected the test case with the highest coverage. If two tests have the same coverage, the first one is selected. Then I selected the next test cases based on the total maximum coverage, in the case of greedy total, or based on the maximum additional coverage, in the case of greedy additional. If there is more than one possible test case with maximum coverage, the first one is selected. For greedy additional, when max coverage is reached, I used a stacking approach, where coverage information is reset and the selection process continues as before. This process continued until the last test case was selected.

4.4.1 Research Questions

This study introduces the use of bytecode diversity as a diversity artefact in diversity-based TCP techniques. I answer the following research questions:

RQ1: Bytecode vs. Textual Diversity How does bytecode diversity compare to textual diversity in the context of diversity-based TCP techniques, such as those proposed by Ledru and FAST? This RQ seeks to evaluate the effectiveness and applicability of bytecode diversity as an alternative to textual diversity.

RQ2: Filtering Bytecode What is the impact of filtering bytecode instructions on the effectiveness and efficiency of static TCP approaches? RQ2 explores whether reducing the set of bytecode instructions considered can improve performance (speed) while maintaining or enhancing prioritisation quality.

RQ3: Bytecode TCP vs. Dynamic TCP How does bytecode diversity, as a static TCP technique, perform compared to dynamic techniques that rely on coverage information? Finally, RQ3 aims to assess how close the performance of a static TCP approach based on bytecode is to that of dynamic TCP approaches.

4.4.2 Evaluation Metrics

To assess the effectiveness and efficiency of testing processes in my study, I used the following evaluation metrics. I used mutation testing evaluated by APFD defined in Equation 4.1 and described in Section 4.2.1. Also, I utilised the real faults recorded in Defects4J [164] and used the number of the first test to reveal the real fault as the evaluation metric. A test that can reveal a real fault is one that fails in the buggy version of the Defects4J but passes in the fixed version.

In order to evaluate the efficiency of the TCP approaches, I recorded the wall clock time of each TCP approach. For diversity-based techniques, there is a stage to calculate the similarity matrix for Ledru-TCP, or the signatures for FAST-TCP. This stage is referred to as *"preparation time"*. Then, there is the *"prioritisation time"*, which is the time taken to reorder the test suite based on the various TCP approaches. Obtaining coverage information requires either instrumenting the program under test and running the entire test suite or executing each test case individually to record coverage data. In this work, I adopted the latter approach. However, to prevent unfair treatment of coverage-based techniques, as instrumentation would be considerably faster, I excluded preparation times and concentrated solely on prioritisation times.

4.4.3 Subjects

I ran my experiments using Java projects from the Defects4J framework [164]. While I initially considered all available projects, I selected seven Maven-based projects where JaCoCo [157] successfully generated coverage reports and PIT mutation analysis [76] ran without issues. Other projects were excluded due to failures in meeting these requirements. I attempted to debug these issues for a week but ultimately did not pursue them further.

For my test suites, I used the developer-written tests augmented with automatically generated tests from Randoop [250] and EvoSuite [117]. I undertook this to create large, diverse test pools that possess high fault detection rates (i.e., mutation score) and can identify the real faults. Table 4.1 shows the projects and the size of the test suites for each project used in my study, where a total of 97 versions and 67 unique classes are used in my experiments. A complete list of the versions and the classes under test for each project is provided in Appendix C. Also, I reported the number of mutants generated using the PIT mutation tool, the number of Randoop tests, the number of EvoSuite tests, and the number of developer-written tests for each project. PIT is a widely adopted tool in both academic and industrial settings, and I employed

Table 4.1: *The test subjects used in my work, where #ver is the number of versions, and #Mut is the number of mutants*

ID	Project Name	#Ver	#Unique Classes	#Mut.	# Test Cases			
					Randoop	EvoSuite	Developer	Total
Cli	jfreechart	5	4	851	7425	279	242	7946
Compress	commons-compress	7	5	1051	10769	133	38	10940
Csv	commons-csv	10	5	767	15050	260	550	15860
Jsoup	jsoup	16	10	767	20575	278	632	21485
Lang	commons-lang	15	11	4136	25893	1103	688	27684
Math	commons-math	42	36	10986	27235	962	1612	29809
Time	joda-time	2	2	268	499	172	69	740
Total		97	67	18826	107446	3187	3831	114464

Randoop and EvoSuite, which are state-of-the-art tools for automated test generation in Java.

In Defects4J, the targeted classes of two different versions can be the same, but the real fault can be different. For example, the project Math version 26 (`Math26`) and version 27 (`Math27`) both target the "`Fraction`" class. The fault in `Math26` is in the constructor of the class, while the fault in `Math27` is in the method "`percentageValue`". As a result, the number of unique classes is lower than the number of versions.

I configured PIT to target the classes specified by Defects4J as target classes for which there are real faults. For example, in Math26, I configured PIT to target the "Fraction" class, since the real fault is in the "Fraction" class.

4.4.4 Methodology

In this section, I describe my methodology for obtaining the textual and bytecode information, mutation data, real fault detection, runtime, and coverage information, all of which are to be used in the static and dynamic TCP approaches. I developed a tool to automate all the steps for my experiments.

Data Collection

My tool parses the test files' source code and extracts each test case. The lines of each test case are concatenated into a single string, where the tool calculates the pairwise dissimilarity values between the strings of test cases using Levenshtein distance. I used the Levenshtein distance because I found it to be more appropriate with string data (as discussed in Section 2.4), which is the nature of my test cases. In Chapter 3, I found that Levenshtein distance performs well as a string distance metric.

Then, my tool generates a textual similarity CSV file that represents the textual similarity matrix. I compiled the source code using Apache Maven 3.6.3, which in turn used the `javac` compiler from OpenJDK 1.8.0_442. The tool also uses the ASM

Java library to read the bytecode files (.class) in their binary format and uses the same Levenshtein distance to calculate the pairwise bytecode similarity between test cases. A separate bytecode similarity CSV file is generated. The textual and bytecode similarity files are to be used in Ledru-TCP.

To collect data for FAST-TCP, the tool saves the concatenated strings representing test cases and the hexadecimal form of the bytecodes in text files. These text files are the input for FAST-TCP.

Additionally, the tool filters the bytecode instructions to include only instructions that push constant values onto the stack (like `iconst`), get field access instructions (like `getfield` or `putfield`), or invoke methods (like `invokevirtual` or `invokestatic`). I used the ASM Java library to target these specific instructions and discard the rest. As before, the tool produces separate filtered bytecode similarity files and text files that contain the hexadecimal form of the filtered bytecodes.

The tool obtains coverage, mutation, and error-revealing information about the test cases. The tool ran each test case independently and used the generated JaCoCo reports to gather coverage information (statement and branch) for that test case. The tool also runs PIT mutation analysis and parses the XML file generated by PIT to produce a mutation "kill map". A *mutation "kill map"* is a two-dimensional data structure that shows for each generated mutant which tests were able to kill that mutant. The tool also runs the test suite against the buggy versions of Defects4J to record all the error-revealing tests.

Prioritising Test Cases

Now, I'm ready to perform my TCP strategies, which I described in Section 4.3.3 and the coverage-based approaches described at the start of Section 4.4. I apply Ledru-TCP, described in Algorithm 2, using three different similarity matrices: textual similarity, all bytecode similarity, and filtered bytecode similarity. Also, I apply FAST-TCP, described in Algorithm 1, with three different input files, using the text of the test cases, the hexadecimal form of all bytecode, and the hexadecimal form of filtered bytecode.

TCP Evaluation

As described in Section 4.4.2, I evaluate the TCP strategies through mutation score, real-fault detection, and runtime. The tool uses the newly ordered test suite and the mutation information to calculate the APFD using Equation 4.1 for each TCP strategy.

With every test case added to the list of ordered tests, the tool checks that test case against the list of error-revealing tests. The tool records the index of the first test case to detect the real fault for each TCP strategy. The tool records the time taken to generate the similarity matrix, build the required signatures, and the prioritisation time.

To validate my results, I employed the non-parametric "Mann-Whitney U" statistical test, as the data are independent and I cannot assume a normal distribution. Additionally, I used the Vargha-Delaney (A_{12}) effect size test [305], a non-parametric measure that compares two groups by estimating the probability that a randomly selected value from one group is larger than a randomly selected value from the other group.

4.4.5 Threats to Validity

In this section, I address validity threats that can affect my results.

Construct validity: The selected string distance to measure test case similarity could be wrong. I mitigated this by using a widely known distance metric (Levenshtein), which is suitable given the nature of the data as reported by other works [97, 100, 184]. Another threat would be that the implementation of Ledru-TCP and FAST-TCP can be invalid. I mitigated this by running the implemented Ledru-TCP against the case studies provided by Ledru et al. [184], getting the same results, and by using the replication package for FAST-TCP provided by Miranda et al. [227]. Miranda et al. [226] used the text of the entire test classes as records for the input, while I used the individual test cases as records. I had an email correspondence with one of the authors to verify that I used the latest version of FAST, and that using individual test cases is valid.

Internal validity: The results can be affected by the nature of the used tests in my test suite, as the test suites I used are comprised of developer-written tests and automatically generated tests from Randoop and EvoSuite. To mitigate this risk, I ran my experiments on different test suites, where I used developer-written tests only, Randoop tests only, and EvoSuite tests only. I found negligible differences and my findings are not affected. These results are reported in my replication package [99].

External validity: The findings of my study may not generalise to different subjects. I mitigated this risk by using seven different Java projects from Defects4J, a publicly available and widely used framework, and ran my experiments on 97 versions of the projects and 67 different classes within the projects. The detailed results of each project are reported in my replication package. Another threat is whether my results give a valid summary of performance. To mitigate this risk, I go beyond single-dimensional summaries of performance (e.g., median) to include measures of variation (e.g., standard deviation), and confidence (e.g., using statistical testing and effect size).

Reliability: To address reproducibility concerns, I: (1) detailed all TCP approaches and subject characteristics, (2) thoroughly documented my methodology (including data collection and bytecode filtering criteria), and (3) provided a complete replication package on GitHub [99] with subjects, resources, implementation details, and an executable README.

Table 4.2: *APFDs for all static TCP approaches across all projects.*

Approach	**Median**	**SD**	**p-value**	A_{12}
Ledru-Text	87.6	12.8	-	-
Ledru-Bytecode	89.9	12.9	0.453	0.53*
Ledru-Bytecode-Filter	**92.1**	11.6	**0.040**	0.59*
FAST-Text	85.4	20.5	-	-
FAST-Bytecode	**93.2**	11.9	**0.000**	0.67**
FAST-Bytecode-Filter	90.5	14.8	**0.013**	0.60*

SD is the standard deviation and A_{12} is the effect size. The p-values and the effect sizes are between textual diversity and bytecode diversity for each TCP technique. A small effect size is marked with (*), while a medium effect size is marked with (**).

Table 4.3: *Real-fault detection for all static TCP approaches across all projects.*

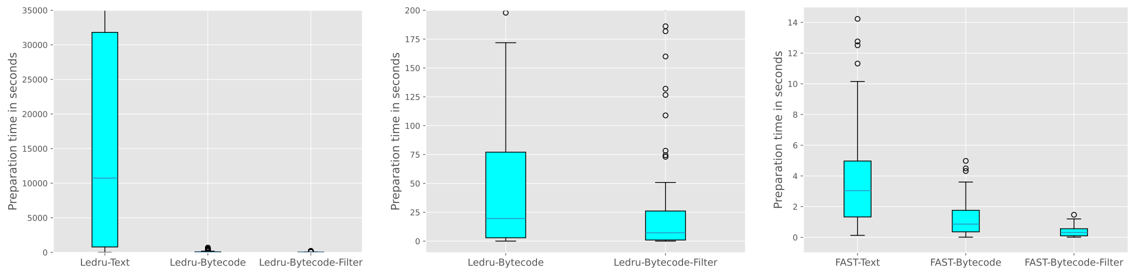
Approach	**Median**	**SD**	**p-value**	A_{12}
Ledru-Text	110.0	520.0	-	-
Ledru-Bytecode	124.0	565.0	0.744	0.49*
Ledru-Bytecode-Filter	**78.0**	559.5	0.211	0.45*
FAST-Text	125.0	890.0	-	-
FAST-Bytecode	**32.0**	523.9	**0.007**	0.61*
FAST-Bytecode-Filter	63.0	740.4	0.297	0.46*

4.5 Results

4.5.1 RQ1: Bytecode vs. Textual Diversity

Tables 4.2 and 4.3 refer to the APFDs and real-fault detection for the static diversity-based TCP approaches for all projects. The first column is the TCP approach and diversity artefact used. For instance, Ledru-Text applies Ledru-TCP using textual information, Ledru-Bytecode applies Ledru-TCP with all bytecode information, and FAST-Bytecode-Filter applies FAST-TCP employing the filtered bytecode information. The next two columns have the median and standard deviation (SD). The last two columns have the pairwise p-values and the Vargha-Delaney (A_{12}) effect sizes of the diversity-based TCP approaches. The highest APFD median value of the TCP approaches and the lowest median number of the first test case to detect the real faults are highlighted in the table. Also, I highlighted the p-values that are significantly different using $\alpha = 0.05$ and marked different effect sizes with stars, where 1, 2, and 3 stars correspond to a small, medium or large effect size, respectively. The p-values and effect sizes shown in the table are between the textual diversity and the corresponding bytecode diversity.

The median APFD of Ledru-Bytecode is 89.9, which is slightly higher than Ledru-Text. The median real fault detection is 124, which means that on median, after 124



(a) *Ledru-TCP* **(b)** *Ledru-TCP using bytecode* **(c)** *FAST-TCP*

Figure 4.4: *The preparation times in seconds for all static TCP approaches*

Table 4.4: *Runtime for all TCP approaches across all projects.*

Project	Ledru-Text	Ledru-Byte	Ledru-Byte-Filter	Avg P-Time	FAST-Text	FAST-Byte	FAST-Byte-Filter	Avg P-Time	Cov-Tot	Cov-Add
Cli	177780.8	405.2	99.8	1117.7	48.9	8.1	2.7	9.8	35.5	42.6
Compress	505855.0	1665.6	664.1	2792.6	36.7	14.7	5.4	18.7	29.3	37.6
Csv	317075.3	852.85	250.95	2145.5	49.5	16.6	5.9	19.6	36.1	46.7
Jsoup	1467823.5	2931.1	794.42	9587.5	80.4	25.3	6.4	43.7	446.1	568.7
Lang	930756.6	1142.6	332.6	5927.5	60.0	21.9	7.3	42.1	279.4	344.1
Math	347037.9	757.65	261.4	2818.7	88.9	33.2	9.2	29.8	89.2	118.5
Time	633.3	5.1	3.0	3.7	2.6	0.9	0.7	0.3	2.2	2.7

Avg P-Time is the average prioritisation time in seconds for the TCP approach. Cov-Tot is coverage-based greedy total, while Cov-Add is coverage-based greedy additional.

test cases run, the real fault can be detected. This is slightly worse than the 110 tests of Ledru-Text. However, the differences in the APFD and real fault detection values are not statistically significant and have small effect sizes over Ledru-Text.

FAST-Bytecode has a median APFD of 93.2 and a median real fault detection of 32 (i.e., only 32 test cases to run to find the real fault). The standard deviation is also the lowest, showing more consistency. Both values are statistically significant at the 99% significance level. APFD shows a medium effect size (0.67, which means that 67% of the cases favored FAST-Bytecode over FAST-Text), while the real fault detection demonstrates a small effect size.

In terms of runtime, Figure 4.4 shows the preparation times for all static TCP approaches. Table 4.4 shows the runtime for all TCP approaches in each project in my study. For Ledru-TCP and FAST-TCP, the diversity artefact has the biggest impact on the preparation time, while the prioritisation time is usually the same per TCP approach. Thus, I reported the preparation times of each TCP approach and the average prioritisation time for Ledru-TCP and FAST-TCP. Using bytecode rather than the text of test cases hugely improves the efficiency of the Ledru-TCP and FAST-TCP.

Using Ledru-TCP, the preparation times using all bytecode information were 124 to 814 times faster than using textual test cases, where the bigger the test suite size and the larger the test case, the more improvements can be observed. Figure 4.4a shows a substantial difference between using text and bytecode in Ledru-TCP. Similarly, Figure 4.4c highlights a significant reduction in preparation time when using bytecode instead of text in FAST-TCP. The smallest project in my study is the `Time` project, where the test suite size is 740 test cases. The preparation time in Ledru-TCP for building the similarity matrix dropped from 633.3 seconds (almost 10 minutes) using text to 5.1 seconds using all bytecode information. In FAST-TCP, the preparation time for building the signatures dropped from 2.6 seconds to 0.6 seconds. The `Math` project has the largest number of different versions and the largest number of test cases (29,610), but the test cases are small. In Ledru-TCP, the preparation times dropped from 347037.9 seconds (almost 96 hours) to 757.7 seconds (almost 12 minutes), while in FAST-TCP, the preparation times dropped from 88.9 seconds to 33.2 seconds. In FAST-TCP, the preparation times using all bytecode information were 2.5 to 6 times faster than using the text of test cases.

Conclusion for RQ1: The experiments show that bytecode diversity has higher APFDs than textual diversity, as it increased by 2.3% in Ledru-TCP and increased by 7.8% in FAST-TCP. The real fault detection in Ledru-Text was slightly better than Ledru-Bytecode, but FAST-Bytecode was 3.9 times better than FAST-Text. The most significant improvement was in efficiency, as using bytecode information is 124 to 814 times faster than using textual information in Ledru-TCP, while in FAST-TCP, bytecode is 2.5 to 6 times faster than text. This dramatic difference in runtime means that bytecode-based approaches can complete the entire prioritisation process while Ledru-Text is still calculating similarity values. Although textual diversity shows a marginal advantage in real fault detection for Ledru-TCP, the enormous runtime savings of bytecode diversity make it a more practical and scalable choice for large-scale regression testing.

4.5.2 RQ2: Filtering Bytecode

Filtering bytecode instructions has a positive effect in Ledru-TCP, as the median APFD increased from 87.6 in text to 92.1 using filtered bytecode information. This is also an increase of 2.2% over using all bytecode information. Furthermore, it is statistically significant at the 95% significance level and has a small effect size in favour of filtered bytecode over using text. Additionally, the median fault detection of Ledru-Bytecode-Filter is 78, which is 1.4 times better than Ledru-Text and 1.6 times better than Ledru-Bytecode.

In FAST-TCP, using filtered bytecode improved APFD by 5.1% compared with text, but it was 2.7% lower than all bytecode information. FAST-Bytecode-Filter demonstrates a statistically significant improvement over FAST-Text at the 95% significance level, with a small effect size. The median real fault detection is 63, which is almost twice as good as FAST-Text. However, the difference is not statistically significant and has a small effect size.

Once more, the biggest improvement is the efficiency of filtering bytecode information. Ledru-Bytecode-Filter is 455 to 2798 times faster than Ledru-Text and 2.5 to 4.1

times faster than Ledru-Bytecode. Figures 4.4b- 4.4c visualise these time reductions for both full and filtered bytecode approaches. FAST-Bytecode-Filter is 4 to 17.9 times faster than FAST-Text and 1.4 to 4 times faster than FAST-Bytecode. The preparation time in the `Math` project dropped to only 261.4 seconds (4 minutes) in Ledru-TCP, while it dropped to only 9.2 seconds in FAST-TCP.

Conclusion for RQ2: Filtering bytecode information increased the APFD in Ledru-TCP by 4.5%, and it was 1.4 times better in real fault detection than using the text of test cases. In FAST-TCP, the APFD increased by 5.1% and the real fault detection was twice as good as using text of test cases. Once more, the biggest improvement was in efficiency, where in Ledru it was 455 to 2798 times faster than Ledru-Text and 2.5 to 4.1 times faster than Ledru-Bytecode. In FAST-TCP, filtering bytecode was 4 to 17.9 times faster than text and 1.4 to 4 times faster than all bytecode information.

4.5.3 RQ3: Bytecode TCP vs. Dynamic TCP

I compared the best-performing static TCP approach (FAST-Bytecode) with the dynamic coverage-based TCP approaches. Tables 4.5 and 4.6 show the APFDs and real fault detection results. The median APFD of greedy additional is 96.7, which is slightly more than the greedy total (96), but the difference is not statistically significant. FAST-Bytecode has lower APFD by 2.8% to 3.5% than greedy total and greedy additional, respectively. Also, the standard deviation for the two coverage-based approaches is lower than bytecode diversity showing better consistency. Furthermore, the results are statistically significant at the 99% significance level, and there is a medium effect size for coverage-based TCP approaches.

In terms of real fault detection, the median numbers of the first test case to detect the real fault of greedy additional and greedy total are 9 and 10.5, respectively. The results of coverage-based TCP approaches are statistically significant at the 95% significance level over bytecode diversity, and there is a medium effect size in favour of bytecode diversity.

Finally, the overall time of FAST-Bytecode, which includes the preparation and prioritisation times, is better than coverage-based TCP. As explained earlier in Section 4.4.2, I included only the prioritisation times for greedy total and greedy additional, shown in Table 4.4. Nonetheless, even using prioritisation times only for coverage-based

Table 4.5: *APFDs for FAST-bytecode TCP and coverage-based TCP across all projects.*

Approach	Median	SD	p-value	A_{12}
FAST-Bytecode	93.2	11.9	-	-
Cov-Total	96.0	5.7	**0.003**	0.62**
Cov-Additional	**96.7**	6.3	**0.001**	0.64**

Table 4.6: *Real-fault detection for FAST-bytecode TCP and coverage-based TCP across all projects.*

Approach	Median	SD	p-value	A_{12}
FAST-Bytecode	32.0	523.9	-	-
Cov-Total	10.5	378.6	**0.011**	0.61*
Cov-Additional	**9.0**	589.2	**0.000**	0.65**

TCP, FAST-Bytecode is still up to 6.5 times faster than greedy total and up to 8.3 times faster than greedy additional. In the `Jsoup` project, the overall time using greedy total is 446.1 seconds (7.4 minutes) and using greedy additional was 568.7 seconds (9.5 minutes), while using FAST-Bytecode the overall time is 68.6 seconds (1.1 minutes).

Conclusion for RQ3: Coverage-based TCP outperform bytecode diversity TCP approaches, and there is little difference between greedy total and greedy additional. The results are statistically significant at the 95% significance level between bytecode and coverage-based approaches. However, the total runtime using FAST-Bytecode is up to 6.5 times faster than greedy total and up to 8.3 times faster than greedy additional. For larger test suites, the difference in runtime becomes larger.

4.6 Chapter Conclusions

This chapter introduced and evaluated, for the first time, the use of bytecode as a compact and semantically meaningful representation of test cases for Test Case Prioritisation (TCP). Building on Chapter 3, which demonstrated the benefits and limitations of textual diversity for reducing automatically generated test suites, this study addressed the efficiency challenges of similarity-based techniques.

The results show that bytecode diversity improves fault detection rates and significantly reduces processing times compared to traditional textual diversity-based methods. Also, the results show that filtering bytecode information greatly improves efficiency while achieving better fault detection than textual information and comparable effectiveness to using all bytecode information. Although coverage-based techniques yield better fault detection, they require extensive program instrumentation, making them less practical in certain contexts. Bytecode diversity thus provides a suitable alternative, offering a balance between effectiveness and operational simplicity. My findings suggest that bytecode-based TCP can be particularly advantageous in resource-constrained environments or where coverage data is unavailable.

These findings highlight that static diversity-based representations, like bytecode, can be both efficient and effective, making them a promising tool for targeted test selection. In the next chapter (Chapter 5), I build on this work to investigate how different test selection strategies, like static textual diversity, bytecode diversity, and dynamic coverage-based approaches, affect the types of faults detected. This study examines whether the better fault-detection performance of coverage-based techniques stems from simply finding more faults, or whether certain faults are inherently easier or harder to detect, providing a deeper understanding of how to prioritise tests effectively.

Chapter 5

How Effectively Do Test Selection Techniques Kill Mutants of Varying Stubbornness?

The contents of this chapter are based on "I. Elgendy, R. Hierons, and P. McMinn. How Effectively Do Test Selection Techniques Kill Mutants of Varying Stubbornness? Manuscript in preparation for submission to ICST 2026.".

5.1 Introduction

Building on the discussions in Chapters 3 and 4, which examined strategies for lowering the cost of regression testing through test suite reduction and test case prioritisation, this chapter turns to the question of *which faults* different selection strategies are most effective at detecting. Prior work found that *coverage-based* techniques have consistently been shown to achieve strong fault detection performance [273, 334]. These approaches are inherently *dynamic*, requiring pre-execution of the entire test suite to collect coverage information, which can be costly for large systems. In contrast, DBT methods — discussed in previous chapters — have been shown to compete with coverage-based techniques in terms of fault detection [21, 184, 226, 312, 323], while being *static* and *lightweight* (i.e., not requiring test execution), which makes them substantially less costly, as I discussed in Chapter 4. While coverage-based selection tends to perform well empirically, it remains unclear *what kinds of faults* these approaches are most effective at detecting.

Mutation testing provides a powerful way to explore this question. It evaluates test suite effectiveness by introducing small artificial faults (*mutants*) into a program and observing whether tests can detect them [160]. However, not all mutants contribute equally to this evaluation. Some are trivial (i.e., easily detected by many tests), others are equivalent (i.e., undetectable by any test), a particularly interesting category are *stubborn mutants* (i.e., those that are hard to kill), and ordinary (i.e., neither hard nor

```
 1    public static Fraction getReducedFraction(int numerator, int
          denominator) {
 2  ⊖     if(denominator == 0) { // original line
 3  ⊕     if(denominator != 0) { // mutated line (M1)
 4            throw new MathArithmeticException(...);
 5        }
 6  ⊖     if(numerator == 0) { // original line
 7  ⊕     if(numerator != 0) { // mutated line (M2)
 8            return ZERO; // normalize zero.
 9        }
10        // allow 2^k/-2^31 as a valid fraction (k>0)
11        if(denominator == Integer.MIN_VALUE && (numerator & 1) == 0) {
12  ⊖         numerator /= 2; denominator /= 2; // original line
13  ⊕         numerator *= 2; denominator *= 2; // mutated line (M3)
14        }
15        .
16        .    // simplify fraction.
17        .
18        return new Fraction(numerator, denominator);
19    }
```

Figure 5.1: *An example of the `getReduceFraction` method in the `Fraction` class*

easy to kill) [197].

Stubborn mutants have been shown to offer higher utility in assessing test quality than other mutant types [165, 197, 256]. They resist detection [332] and often require targeted effort to be killed, making them meaningful indicators of testing effectiveness. They subsume other, easier-to-kill mutants [180, 197] and correlate more strongly with real fault detection [165, 256]. Yet, identification of stubbornness vary. Some works identify them as mutants that can be killed but survive thorough (i.e., full branch coverage) test suites [59, 194, 332], are hard to reach [84], or, most commonly, are killed by very few tests [92, 123, 152, 197]. In this thesis, I adopt the latter view, focusing on mutants that are killed by only a small proportion of tests.

5.1.1 An example

Figure 5.1 shows a real-world example from the widely used Apache Commons Math library's `Fraction` class. This class is part of my experimental subjects described later in Section 5.4, but I will use the class and the associated test suite here as my example. The test suite covering this class consists of 1,997 test cases, comprising both developer-written tests and automatically generated ones from Randoop [250] and EvoSuite [117]. The method `getReducedFraction` creates a Fraction object in the simplest form using two integers. I used the PIT mutation tool [76] to generate mutants for `Fraction` class, and I consider three of these mutants in my example. The first mutant (M_1), located at Line 2, is produced by a Relational Operator Replacement

(ROR, e.g., `>` → `<=`) operator. The original line is highlighted in red with a '–' icon, and the mutated line in green with a '+' icon. A total of 432 tests kill this mutant. The second mutant (M_2) is at Line 6, and it is also based on the ROR, but 67 tests kill this mutant. The final mutant (M_3) is at Line 12, and it is based on the Arithmetic Operator Replacement (AOR, e.g., `+` → `-`)), and only one test kills it.

M_3 is a stubborn mutant as it is only killed by one test out of the 1,997 tests, and the obvious reason for its stubbornness is the difficulty of reaching this line. This is because reaching and triggering its faulty behaviour requires a very specific combination of inputs (an even numerator and a denominator equal to `Integer.MIN_VALUE`). On the other hand, M_1 is a trivial mutant (i.e., easy to kill), since any test that passes a denominator of a value 0 and is expecting an exception will not find it, or any test that has a non-zero denominator value and should not trigger an exception will find an exception occuring. Almost any test that executes the method will kill this mutant. M_2 lies somewhere between the two extremes, and can be considered stubborn or not based on the desired stubbornness threshold. It introduces a fault where the method returns the canonical `ZERO` fraction even when the numerator is non-zero. Thus, any test that supplies a non-zero numerator and asserts a correct (non-zero) fraction would detect this error. The mutant is therefore not deeply hidden, but it still requires that the test assert the return value, rather than merely executing the method. This explains why the mutant is relatively easy to kill but not trivial.

5.1.2 Motivation

The relationship between *test selection strategies* and *mutant difficulty* has not been systematically studied. Prior work has focused primarily on overall mutation scores or fault detection rates, implicitly treating all mutants as equally important [184, 226, 98]. As a result, we know little about whether coverage-based techniques are more effective because they detect *many* faults, or because they detect the *hard ones*. Understanding this relationship is crucial for assessing the true value of different test selection strategies, particularly in cost-sensitive contexts such as regression testing and continuous integration. In this chapter, I therefore investigate whether diversity-based approaches—being static and inexpensive—can match or surpass coverage-based methods in detecting the hardest (stubborn) faults.

In the literature review (Chapter 2), I highlighted that diversity-based testing (DBT) techniques often rival or even outperform coverage-based approaches in terms of fault detection [21, 184, 226, 312, 323]. Earlier chapters of this thesis extended DBT to new domains: Chapter 3 applied textual diversity to reduce automatically generated test suites, and Chapter 4 introduced bytecode diversity for more efficient test case prioritisation. Building on these foundations, this chapter investigates whether diversity can also help address the challenge of stubborn mutants: can dissimilarity-based test selection kill them earlier and more effectively than traditional coverage-based methods?

To answer this, I study the performance of different test selection strategies across mutants of varying levels of stubbornness. My goal is to understand not only which strategies achieve high mutation scores overall, but also which ones are better at exposing stubborn, high-utility faults. I evaluate stubborn mutants across seven real-world Java projects from Defects4J [164] from three perspectives: (1) how coverage-based and diversity-based approaches compare in killing stubborn mutants; (2) the structural and semantic characteristics of stubborn mutants; and (3) the practical utility of tests that kill them. Results show that DBT using bytecode information achieves comparable or superior performance to coverage-based selection, requiring 46% fewer tests on median to kill the most stubborn mutants. Moreover, stubborn mutants are disproportionately located in private ($12\times$) and void ($10.6\times$) methods, deeply nested code (up to $5.8\times$), and are often produced by *method-level mutation operators* (i.e., mutate the whole method call, like `VoidMethodCall`) (up to $7.4\times$). These findings not only provide empirical insight into where and how stubborn mutants occur, but can also assist practitioners in designing more effective mutation operators that focus on challenging, high-utility faults, improving the effectiveness of mutation testing. Finally, I show that tests killing only stubborn mutants make up just 2–25% of a suite, yet they achieve full mutation adequacy and up to 88% real fault reveal rate.

These findings provide the first empirical evidence that diversity-based test selection can effectively detect hard-to-find faults, while also deepening the understanding of their properties and practical significance. The chapter offers the following contributions:

1. The first empirical evaluation of diversity-based and coverage-based test selection strategies for killing stubborn mutants across seven real-world projects (Sections 5.5.1 and 5.5.2).

2. An empirical study of stubborn mutants' characteristics (e.g., code location, mutation operators, method properties) and their utility and prevalence (Sections 5.5.3 and 5.5.4).

5.2 Background

5.2.1 Mutation Testing

Mutation testing is a fault-based software testing technique that evaluates the effectiveness of a test suite by introducing small, syntactical changes, known as *mutants*, to the source code of a program [160]. The mutants are designed to simulate real faults made by developers. Mutation testing operates on two hypotheses: the Competent Programmer Hypothesis (CPH) [87], and the Coupling Effect [243]. The CPH assumes that programmers write code that is close to correct, and that faults are often small deviations from the intended logic. The coupling effect suggests that tests that cover simple errors are effective in detecting more complex errors.

A mutant is said to be *killed* when at least one test case in the test suite produces a failing result when executed against the mutated code. A mutant is said to be live (survive) if all test cases pass when executed against the mutated code. A live mutant indicates a potential weakness in the test suite. However, there is a type of mutant that can never be killed, referred to as *equivalent mutant.* An equivalent mutant is one that is syntactically different, but semantically equivalent to the original program.

5.2.2 Stubborn Mutants

In principle, the idea of a stubborn mutant is the same across different works, but the definition of these stubborn or hard-to-kill mutants varies. A stubborn mutant is one that is very hard to kill, while an equivalent mutant is impossible to kill. All mutants lie on a spectrum of "killability" from easy to kill to impossible to kill [332]. Their position on the "killability" spectrum makes stubborn mutants a critical focal point. They subsume easier-to-kill mutants [180, 197], making them a stronger indicator of test suite effectiveness, and their difficulty correlates more strongly with real fault detection [165, 256].

Papadakis et al. [256], Hernandez et al. [123], and Du et al. [92] consider a mutant to be stubborn if very few tests kill that mutant. Hernandez et al. [123] introduced the rank-stubbornness model (RSM), where they evaluate the stubbornness rank of each mutant. The RSM assigns each mutant a rank based on the number of test cases that kill it. Mutants killed by fewer test cases are assigned higher ranks, with those killed by only one test case receiving the highest rank. I will refer to the RSM later in Section 5.3.1. Also, Lindström et al. [197] reported that a stubborn mutant is one that is killed only by one test or very few tests. My work uses a similar definition of stubborn mutants.

Other works defined stubborn mutants as ones that can be killed but are still alive after a *thorough* test. Usually, a thorough test suite is one that covers all feasible branches. This way of identifying stubborn mutants is used in several works [332, 194, 59] in the context of devising new techniques to generate tests capable of killing these stubborn mutants.

Another way of deciding whether a mutant is stubborn is by calculating how difficult it is to reach and execute the mutated line. Dang et al. [84] inspect the line where the mutant occurs and evaluate all the conditions and variables in every possible path. Their approach computes the *execution probability* of a program path by multiplying the probabilities of all conditional branches along that path. They factor in *variable complexity*, noting that paths involving more input variables are harder to target with tests due to the larger input space required to reach the mutant. They determine reachability by these two factors, and the difficulty of killing a mutant is the lowest reachability value of all possible paths to the mutant's statement.

5.2.3 Test Selection Techniques

Test selection techniques aim to identify a subset of test cases from a larger suite that maximises fault detection while reducing execution cost [273, 334]. These techniques

Table 5.1: *Example illustrating mutant stubbornness and killability scores*

Tests	m_1	m_2	m_3	m_4	m_5	m_6	m_7
T_1	✓	✓	✗	✓	✗	✗	✓
T_2	✓	✗	✗	✓	✗	✗	✗
T_3	✓	✗	✗	✓	✓	✗	✓
T_4	✓	✓	✓	✗	✗	✓	✗
T_5	✓	✓	✗	✗	✗	✗	✗
T_6	✓	✗	✗	✓	✓	✓	✓
T_7	✗	✓	✗	✓	✗	✗	✓
Killability score	6	4	1	5	2	2	4

are widely used in regression testing and continuous integration, where executing the full suite can be expensive.

Coverage-based techniques select tests based on the program elements (e.g., statements, branches, or methods) they exercise [273]. These approaches are *dynamic*, requiring pre-execution of the test suite to collect coverage information, which can be costly for large systems. Despite this, coverage-based selection has consistently been shown to achieve strong fault detection performance.

Diversity-based testing (DBT) techniques, in contrast, are *static* and *lightweight*, as they do not require test execution, making them substantially cheaper than coverage-based strategies, as shown in Chapter 4.

5.3 Approach

This section outlines my approach to identifying and evaluating stubborn mutants and analyses how different test selection strategies detect them early. I present my model for identifying stubborn mutants and describe the test selection strategies employed in my study. Then, I assess the significance of the tests killing only stubborn mutants.

5.3.1 Stubborn Mutant Identification

I follow the same notions of stubbornness introduced by Hernandez et al. [123] (described in Section 5.2.2), where I implemented the Rank-Stubbornness Model (RSM).

Table 5.1 shows an example of seven tests and seven mutants. Each cell entry indicates whether a test T_i kills a mutant m_j ('✓') or not ('✗'). The term rank may be misleading, as it can suggest that each mutant is assigned a contiguous ordinal position. To avoid this confusion, I use the term **Killability Score**, which reflects the number of distinct test cases that kill a mutant, allowing multiple mutants to share the same score. The last row in Table 5.1 (*Killability Score*) reports the number of distinct tests that kill each mutant. Mutant m_1 is killed by six tests, indicating that it is easy to

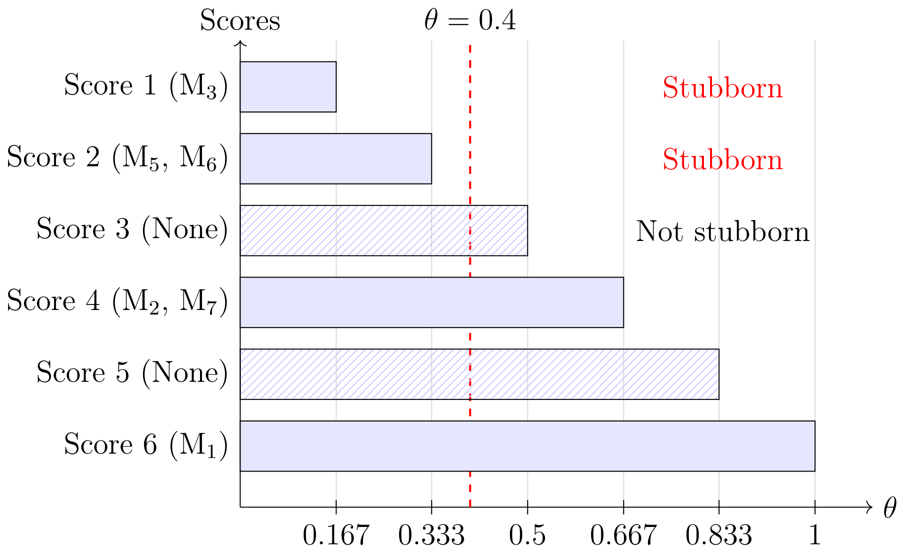


Figure 5.2: *Illustration of RSTM: grouping mutants by threshold levels. Mutants with θ below the threshold (here $\theta = 0.4$) are classified as stubborn. Filled bands represent mutants present in the example; hatched bands (Score 3 and 5) are added for illustration but contain no mutants in this scenario.*

detect. In contrast, m_3 is killed by only one test, and both m_5 and m_6 are killed by two tests, suggesting that they are more difficult to detect.

RSM provides a scoring of mutants based on their resistance to being killed. However, it lacks a clear decision boundary for identifying which specific mutants are stubborn. As a result, the methodology lacks a principled mechanism for defining a stubbornness boundary, relying instead on arbitrary score selection.

Papadakis et al. [256] addressed this by introducing fixed thresholds. For example, treating mutants killed by no more than 5% or 2.5% of tests as stubborn. However, such thresholds are rigid and can misclassify mutants when the overall test effectiveness is low. In cases where even the easiest mutants are only killed by a small portion of tests, this approach may falsely label many as stubborn.

Stubbornness is project-dependent, as a score that is considered highly resistant in one project might represent an average or even easy-to-kill mutant in another. Thus, using absolute scores across diverse codebases leads to inconsistent and potentially misleading classifications. To address these limitations, I propose the **Relative Stubbornness Thresholding Model (RSTM)**.

RSTM introduces a user-defined threshold that selects stubborn mutants based on their difficulty relative to the easiest mutant within the same project. This enables consistent, flexible, and project-aware identification of stubborn mutants. By adjusting the threshold, testers can tailor the level of stubbornness sensitivity to their specific needs.

For a set of n mutants and a test suite T, the Relative Stubbornness Thresholding

Model (RSTM) assigns each mutant m_i a stubbornness score $s_i \in \mathbb{N}$, which is the number of test cases in T that kill m_i. Let:

- $S = \{s_1, s_2, \ldots, s_n \mid 1 \leq s_i \leq |T|\}$ be the multiset of stubbornness scores assigned to each of the n mutants.
- $s_{\min} = \min(S)$ be the minimum score value, corresponding to the easiest-to-kill mutant (i.e., the one killed by the most tests).
- $\theta \in (0, 1]$ be a user-defined threshold that determines the level of relative stubbornness.

A mutant m_i is classified as *stubborn* under the RSTM if:

$$s_i \leq \theta \cdot s_{\min}$$

Using the example from Table 5.1, $s_{\min} = 6$ and if $\theta = 0.4$, then the threshold is $6 \times 0.4 = 2.4$, and any mutant with $s_i \leq 2.4$ is considered stubborn. So, m_3, m_5, and m_6 are stubborn mutants. If the stubbornness threshold is 0.25, then only m_3 is stubborn. RSTM enables flexible identification of relatively stubborn (i.e., harder-to-kill) mutants.

Figure 5.2 visualises this example. The y-axis shows the mutant scores (number of killing tests), while the x-axis represents the corresponding threshold levels (θ). Mutants are grouped into horizontal bands by score, and the θ line defines which groups fall below the threshold. For instance, with $\theta = 0.4$, mutants in groups with $\theta \leq 0.4$ (e.g., M_3, M_5, M_6) are classified as stubborn, while those in higher bands are not.

5.3.2 Test Case Selection Approaches

In my work, I implemented six test case selection approaches, two of which are coverage-based and four are diversity-based.

The first coverage-based selection approach is based on maximising statement coverage, as the hypothesis could be that stubborn mutants are killed by comprehensive, broadly-scoped tests that cover large portions of the code. I implemented a greedy algorithm where the test case with the maximum coverage is selected first. If multiple test cases have the same maximum coverage, the first one is selected. Then, the test case with the next highest additional coverage is selected in the same way, and this continues until the stubborn mutant under consideration is killed. Since all stubborn mutants in my study are by definition killable (as identified by RSTM), this process always terminates successfully. The second coverage-based selection approach is based on minimum statement coverage. The selection is exactly as before, except that the minimum coverage is used to guide the selection process. I did this to investigate whether tests with a narrow, focused coverage have any effect on killing stubborn mutants over tests that have a wider focus. By implementing these two opposing strategies, I can analyse which coverage-based property (breadth or focus) is more strongly associated with the ability to reveal stubborn mutants.

I implemented two state-of-the-art diversity-based selection approaches based on FAST [226] (explained in Chapter 3) and Ledru [184] (explained in Chapter 4). The motivation is that stubborn mutants are hard to kill and may require diverse test inputs or behaviours to be triggered (as described in Section 5.1.1). For each approach, I used two types of diversity information. Textual diversity (Ledru-Text and FAST-Text) measures differences in the text of test cases, capturing variations in inputs and method sequences. Bytecode diversity (Ledru-Bytecode and FAST-Bytecode) measures differences in compiled test instructions, capturing variations in actual program behaviours while being more compact and efficient. Comparing these four variants allows me to evaluate how both the algorithm and the choice of diversity representation affect the ability to kill stubborn mutants. I explained these approaches in detail in Chapters 3 and 4.

5.3.3 Stubborn Mutants Impact

I want to identify the impact of stubborn mutants on test suite effectiveness and real fault detection. An interesting observation from Table 5.1, is that some mutants *subsume* others. Just et al. [165] found that subsuming mutants have higher utility, as they are less likely to be equivalent or trivial. Other works found that stubborn mutants are more likely to subsume other mutants [197]. Killing m_5 (e.g., by T_3 or T_6) also results in the killing of m_1, m_4, and m_7. Similarly, killing m_3 (via T_4) also kills m_1, m_2, and m_6. This subsumption effect suggests that some stubborn mutants may be more valuable than their apparent rarity implies, because they represent faults whose detection indirectly kills multiple other mutants.

Returning to the example, selecting T_4 (which kills m_3) and either T_3 or T_6 (which kill m_5) is sufficient to kill all seven mutants. This demonstrates the potential utility of stubborn mutants in test suite optimisation. Focusing on tests that can kill the stubborn mutants could not only reduce the test suite size but also improve overall test effectiveness by collapsing redundant mutant coverage.

In my approach, I identify the set of tests that kill the set of stubborn mutants, denoted as (T_{Stb}). I analyse the impact of killing the stubborn mutants by analysing the mutation score achieved by T_{Stb}. I can identify whether these tests can reveal real faults in the program by calculating the *Fault Reveal Rate* (FRR), which is the proportion of error-revealing tests that appear in T_{Stb}. The set of error-revealing tests is denoted as (T_E), which I identify using the Defects4J benchmark [164]. The FRR value is in the range $[0, 1]$ and calculated using this equation:

$$FRR = \frac{|T_{Stb} \cap T_E|}{|T_E|} \tag{5.1}$$

An FRR of 1 indicates that all error-revealing tests are included in the stubborn-killing set, whereas a value of 0 means none are.

Table 5.2: *The test subjects used in my work, where #ver is the number of versions, and #Mut is the number of mutants*

ID	Project Name	#Ver	#Unique Classes	#Mut.	# Test Cases			
					Randoop	EvoSuite	Developer	Total
Cli	jfreechart	5	4	851	7425	279	242	7946
Compress	commons-compress	7	5	1051	10769	133	38	10940
Csv	commons-csv	10	5	767	15050	260	550	15860
Jsoup	jsoup	16	10	767	20575	278	632	21485
Lang	commons-lang	15	11	4136	25893	1103	688	27684
Math	commons-math	42	36	10986	27235	962	1612	29809
Time	joda-time	2	2	268	499	172	69	740
Total		97	67	18826	107446	3187	3831	114464

5.4 Evaluation

To validate the potential of diversity-based test selection as a lightweight means of killing stubborn mutants and to deepen my understanding of their properties, this study is guided by the following research questions:

- **RQ1:** How do different test selection strategies (coverage-based, textual diversity-based, and bytecode diversity-based) compare in their ability to kill stubborn mutants earlier in the test execution process?
- **RQ2:** Does varying the stubbornness threshold impact the ability of each selection strategy in identifying and killing stubborn mutants early?
- **RQ3:** What is the impact of the tests that kill stubborn mutants on test suite effectiveness and real fault detection?
- **RQ4:** Where do stubborn mutants usually occur, and what are their characteristics?

RQ1 investigates if lightweight diversity-based techniques (textual and bytecode) are an effective proxy for identifying tests that kill stubborn mutants, potentially outperforming more expensive coverage-based techniques. RQ2 investigates how the identification of "stubbornness" influences the performance of each selection strategy. RQ3 quantifies the practical value of the tests that kill stubborn mutants, assessing their contribution to overall test suite quality and their correlation with real fault detection. Finally, RQ4 provides a foundational analysis of stubborn mutants themselves, seeking to identify patterns in their location and nature that can inform future testing and debugging efforts.

5.4.1 Subjects

I ran my experiments using seven real-world Java projects from the Defects4J framework [164]. I used the same dataset from Chapter 4, utilising the reported coverage, similarity, and mutation information. These projects are listed in Table 5.2, where a total of 97 versions were considered in my experiments.

To ensure my study included a sufficient number of stubborn mutants, I required large and diverse test suites capable of revealing subtle or subsuming behaviours. For this reason, I augmented developer-written tests with automatically generated tests from Randoop [250] and EvoSuite [117]. Larger and more comprehensive test suites increase the likelihood of killing a broad range of mutants, including those that are harder to detect. As noted by Just et al. [164], more complete test suites provide a better approximation when identifying dominator sets (mutants that subsume others), a concept closely related to stubborn mutants.

5.4.2 Methodology

I developed a tool to automate the identification of the stubborn mutants and the application of the different test case selection approaches. I explain how the data was collected, the analysis of stubborn mutants and their characteristics, and the implemented test case selection approaches.

Data Collection

I used the data I collected in Chapter 4, which includes error-revealing tests, coverage, similarity, and mutation information for the seven projects reported in Table 5.2. The coverage information contains statement and branch coverage, with coverage per test case provided. More details about the format and the structure of the data can be found in the replication package [99].

Stubborn Mutant Impact

As I explained in Section 5.3.1, I used the RSTM model to identify the stubborn mutants. In my experiments, I investigated different thresholds, which are 0.75, 0.5, 0.25, 0.1, 0.05, 0.01, 0.005, 0.003, 0.002, and 0.001. For all stubbornness thresholds, the tool determined T_{Stb} (i.e., the set of tests that kill the set of stubborn mutants). Then, it recorded the total mutation score, the mutation score achieved only by T_{Stb}, the FRR values, the size of the full test suite, and the size of T_{Stb}. To establish a baseline, I also generated random test subsets ($\mathcal{R}$) of the same size as T_{Stb}. Each random selection was repeated 31 times, and the results were averaged to reduce variance. This comparison with $\mathcal{R}$ allows me to distinguish the specific impact of stubborn-mutant-killing tests from effects that could arise merely due to test suite size.

Since each project consists of multiple program versions, the tool computed average values across all versions. My tool counted how many times a *perfect FRR* (i.e., a value of 1 indicating that all error-revealing tests are selected) is achieved.

Stubborn Mutant Characteristics

I analysed some properties of the most stubborn mutants in each project. The most stubborn mutants are Score1 (i.e., killed only by one test) through Score5 (i.e., killed by five tests).

For each stubborn mutant, the tool identified the source file and line number where the mutant occurred. The tool parsed the abstract syntax tree to identify the type of the line (e.g., an `if` statement, a `return` statement, a method invocation, etc). The tool also recorded the nesting level of that line. In addition, the tool identified the method containing the mutated line and recorded the access modifier (i.e., `private`, `protected`, `public`), whether it is a `void` or non-`void` method, and whether it is a `static` or instance method. Finally, the tool recorded the mutation operator that generated the mutant. All this information was used to give me an idea of what might cause a mutant to be stubborn.

Test Case Selection Approaches

My tool implemented the six coverage and diversity-based test selection approaches explained in Section 5.3.2. For each stubborn mutant, the tool records the number of the first test case to kill that mutant.

5.4.3 Threats to Validity

This section addresses validity threats.

Construct validity: My implementations of Ledru and FAST may contain bugs or deviate from the intended designs. To mitigate this risk, I validated my implementations in Chapter 4.

Internal validity: The results could be influenced by how stubborn mutants are identified using RSTM. To assess this threat, I also applied the fixed threshold-based identification used by Papadakis et al. [256] and compared outcomes. I observed negligible differences, indicating my findings are not sensitive to the choice of identification model. Detailed comparisons are available in my replication package [99].

External validity: My findings may not generalise beyond the selected subjects. I mitigated this risk by evaluating my approach on seven diverse Java projects from the Defects4J benchmark, covering 97 versions and 67 classes. Project-level results are included in my replication package. Additionally, to ensure robustness, I reported not only median results but also variability (e.g., standard deviation) and statistical significance (e.g., p-values and effect sizes).

Reliability: The ability of other researchers to reproduce my results is critical. I mitigate this risk by providing a comprehensive replication package on GitHub [99], including source code, experiment scripts, and datasets. The repository includes a README with setup and execution instructions to aid reproducibility. I also documented my methodology and evaluation metrics in detail in this paper.

5.5 Results

5.5.1 RQ1: Test case selection strategies to kill stubborn mutants

Table 5.3 shows my analysis of the test selection strategies in killing the stubborn mutants for all projects in my study. The first column shows the stubbornness threshold levels, and the second column shows the number of stubborn mutants that fall in that threshold. The rest of the table shows the median (Med) and standard deviation (SD) of the first test to kill stubborn mutants for every test selection strategy used in my study. The table presents how quickly different test selection strategies are able to kill stubborn mutants, measured by the median number of selected tests needed. Lower medians are better, indicating faster detection. The bolded values highlight the strategy with the lowest median at each threshold level, i.e., the most efficient in killing stubborn mutants early.

Maximum coverage (Max-Cov) is the best-performing strategy at low stubbornness levels, and starting from threshold level 0.005, it falls behind FAST-Bytecode. Minimum coverage (Min-Cov) is consistently the worst selection strategy, indicating that tests with a narrow focus have no utility in killing stubborn mutants. FAST-Text performed the worst out of all diversity-based strategies. Ledru-Text and Ledru-Bytecode are very close to each other, and their performance is relatively close to that of maximum

Table 5.3: *Median and standard deviation of the first test to kill stubborn mutants per selection strategy for all projects. T-L is the Threshold Level, Led. is Ledru and MxC/F-B is the Max-Cov/FAST-Bytecode.*

T-L	**#Stb Mut.**	**Led.-Txt**		**Led.-Byte**		**FAST-Txt**		**FAST-Byte**		**Max-Cov**		**Min-Cov**		**MxC/F-B**	
		Med	SD	Med	SD	Med	SD	Med	SD	Med	SD	Med	SD	p-value	A_{12}
0.75	13196	16.0	332.3	14.0	253.1	50.0	599.8	20.0	330.5	**5.0**	308.0	444.0	686.7	**0.0000**	0.64
0.50	12690	17.0	337.5	16.0	257.2	55.0	605.7	21.0	336.3	**6.0**	313.2	446.0	691.5	**0.0000**	0.64
0.25	11111	25.0	355.8	21.0	271.9	67.0	629.7	24.0	354.6	**9.0**	331.5	511.0	706.0	**0.0000**	0.63
0.10	8633	38.0	393.3	30.0	300.9	111.0	676.6	33.0	391.7	**13.0**	368.5	626.0	729.4	**0.0000**	0.61
0.05	6772	55.0	427.2	40.0	328.8	167.0	714.0	39.0	423.9	**17.5**	395.4	716.0	737.4	**0.0000**	0.60
0.01	2649	175.0	539.7	105.0	446.2	539.0	829.2	65.0	558.8	**57.0**	497.7	857.0	704.2	**0.0000**	0.55
0.005	1564	244.0	621.8	157.0	482.2	714.0	836.1	**44.0**	661.0	63.5	560.7	934.5	717.0	**0.0004**	0.54
0.003	1079	256.0	640.0	179.0	499.9	730.0	852.2	**42.0**	720.9	87.0	615.1	905.0	695.6	0.6695	0.51
0.002	739	361.0	695.7	235.0	549.8	758.0	914.3	**61.5**	828.1	128.0	688.7	1006.0	741.7	0.5282	0.51
0.001	236	408.0	853.5	288.5	724.9	1664.0	1107.1	**217.5**	1095.8	404.0	862.6	1601.0	814.4	0.7467	0.49

coverage at low stubbornness thresholds, and slightly better than FAST-Bytecode. However, after threshold 0.05, they fall behind FAST-Bytecode, and the gap gets bigger with higher threshold levels.

In Chapter 4, I reported that FAST-Bytecode was by far the best performing in terms of runtime. That makes FAST-Bytecode the best candidate of the diversity-based strategies overall.

To assess whether these observed differences are statistically significant, I conducted Mann-Whitney U tests and the Vargha-Delany effect size on all strategies. Given that six selection strategies yield 15 pairwise comparisons, it is not feasible to discuss all results in detail. Instead, I focus on the two best-performing strategies, FAST-Bytecode and Max-Cov, which also represent the two families of approaches considered in this study: static lightweight diversity-based and dynamic coverage-based.

Comparison between FAST-Bytecode and Max-Cov

Although Max-Cov achieves the lowest median number of tests needed to kill stubborn mutants at higher threshold levels (e.g., threshold = 0.75, median = 5), the median values achieved by FAST-Bytecode are often close in magnitude. For example, at threshold levels of 0.75, 0.50, and 0.25, FAST-Bytecode yields medians of 20, 21, and 24, respectively. This observation is significant because FAST-Bytecode is a lightweight strategy, requiring no prior execution of the test suite, whereas Max-Cov depends on dynamic coverage information. The closeness of their performance suggests that lightweight diversity-based techniques like FAST-Bytecode can be competitive with heavier, execution-dependent approaches in the early stages of test selection.

The last two columns in Table 5.3 show the statistical comparison between FAST-Bytecode and Max-Cov using Mann-Whitney U test and the Vargha-Delaney A_{12} effect size. The results show that the performance of Max-Cov is statistically significantly better than FAST-Bytecode for thresholds 0.75 through 0.01, with p-values ≤ 0.001, and the effect size shows a medium to small effect. For stricter stubbornness (e.g., threshold ≤ 0.005), the statistical difference diminishes, with p-values increasing (e.g., 0.6695 at threshold 0.003) and A_{12} effect sizes nearing 0.5, implying no practical performance advantage of Max-Cov over FAST-Byte. In fact, from threshold 0.005–0.001, FAST-Bytecode outperforms Max-Cov in terms of median, though the difference is not statistically significant.

Conclusion for RQ1: My findings indicate that while Max-Cov demonstrates strong performance at broader stubbornness thresholds by quickly killing stubborn mutants with fewer tests, its advantage decreases as the threshold becomes stricter. FAST-Bytecode is lightweight and does not require pre-execution of the test suite and consistently approaches Max-Cov in early-stage effectiveness and eventually surpasses it in the stricter thresholds (e.g., ≤ 0.005). FAST-Bytecode has up to 46% fewer tests required at the strictest threshold than Max-Cov.

5.5.2 RQ2: Varying stubbornness threshold

At higher thresholds (e.g., 0.75–0.10), maximum coverage (Max-Cov) consistently has the lowest median (e.g., 5 at 0.75, 13 at 0.10), showing strong performance when many mutants are considered stubborn. This suggests that coverage-based strategies are more effective when the bar for stubbornness is low (i.e., more mutants are labelled as stubborn).

At mid-range thresholds (0.05–0.005), FAST-Bytecode starts performing better. FAST-Bytecode has the lowest median at 0.005 (median = 44.0), indicating that it becomes more efficient as stubbornness gets higher.

At very strict thresholds (0.003–0.001), FAST-Bytecode is the best performing with medians of 42, 61.5, and 217.5, respectively. These values are lower than Max-Cov and significantly lower than other strategies at the same levels, suggesting that FAST-Bytecode scales better with harder-to-kill mutants.

Conclusion for RQ2: The effectiveness of test selection strategies varies with the strictness of the stubbornness threshold. While Max-Cov demonstrates better performance when many mutants are considered stubborn, FAST-Bytecode emerges as the most effective and scalable approach under stricter thresholds, where mutants are harder to kill. Thus, FAST-Bytecode presents a very good, lightweight alternative for early stubborn mutant detection, particularly in resource-constrained or large-scale testing scenarios.

5.5.3 RQ3: Impact of stubborn mutants

Table 5.4 summarises the effectiveness of selecting tests that target only stubborn mutants (T_{Stb}) at various stubbornness threshold levels across all projects in my study. While my setup considered thresholds from 0.75 down to 0.001, in practice I only report results for thresholds $\geq$0.05. At lower thresholds (0.01–0.001), the number of identified stubborn mutants was too small to yield meaningful or generalisable results, and in some cases, T_{Stb} degenerated to an empty or near-empty set. Therefore, for clarity and space reasons, I focus on the thresholds that consistently produced non-trivial results across projects. The complete results, including the lower thresholds, are available in my replication package [99]. The first column lists the project name along with the average test suite size and mutation score (MS) per version. The second column indicates the threshold level (TL) used to identify stubborn mutants based on RSTM. The third and fourth columns report the average MS achieved by T_{Stb} (tests that kill only stubborn mutants) and by $\mathcal{R}$, a baseline consisting of randomly selected tests of the same size as T_{Stb}. The next two columns show the proportion of versions in which T_{Stb} and $\mathcal{R}$ achieve the same MS as the full test suite ("Perfect" MS). The next columns present the average FRR of fault-revealing tests (as defined in Section 5.3.3) and the proportion of versions with perfect FRR (i.e., FRR of 1.0). The final column shows the average size of T_{Stb} (same size as $\mathcal{R}$).

Across all projects, selecting tests that target only stubborn mutants (T_{Stb}) leads to a substantial reduction in test suite size. For instance, the average number of tests drops from over 1600 in `Cli` and `Compress` to fewer than 150, and even to under 50 in projects like `Csv` and `Jsoup`.

Importantly, at a threshold level of 0.75, T_{Stb} consistently achieves high mutation scores across all projects. The average mutation score (MS) at this level is either equal to or very close to that of the full test suite, with "Perfect MS" values at or near 1.0 in most cases. In contrast, the random baseline $\mathcal{R}$ (tests of the same size as T_{Stb}) achieves noticeably lower average MS and much lower average real fault FRR. Moreover, $\mathcal{R}$ never matches the full suite: both its "Perfect MS" and "Perfect FRR" values remain zero across all thresholds. This confirms that the effectiveness of T_{Stb} stems from targeting stubborn mutants rather than merely reducing suite size. Also, it indicates that targeting only stubborn mutants at higher thresholds maintains test adequacy while significantly reducing suite size.

In terms of real fault detection, FRR varies by project. Four of the seven projects (`Csv`, `Jsoup`, `Lang`, and `Math`) exhibit moderate to high FRR at threshold 0.75, suggesting that many real fault-revealing tests are captured within T_{Stb}. In contrast, `Cli` and `Compress` show consistently low FRR, and the `Time` project reveals a limitation: none of the real fault-revealing tests were included in T_{Stb} across its two versions (FRR = 0). However, these three projects contain relatively few real faults in my evaluation (five, seven, and two for `Cli`, `Compress`, and `Time`, respectively), which may limit the representativeness of their FRR values.

While a drop in FRR is observed in some projects, this trade-off appears more pronounced in cases with limited real fault data. In contrast, projects with more extensive real fault sets tended to show consistently high FRR.

These results demonstrate that stubborn mutants are valuable indicators of test effectiveness, and prioritising their detection, particularly early in the testing process, can be highly rewarding in terms of both adequacy and cost-efficiency. My empirical results indicate that a threshold of 0.75 consistently yields a strong trade-off between FRR and test suite size across the studied projects. Therefore, I recommend 0.75 as a practical default threshold, while encouraging practitioners to adjust it based on their specific context and testing goals.

Conclusion for RQ3: Stubborn mutants play an important role in guiding effective test selection. At moderate thresholds (e.g., 0.75), targeting only the stubborn mutants often leads to high mutation adequacy and, in many cases, substantial FRR, while also significantly reducing test suite size. Across my study subjects, the subset of tests that killed only stubborn mutants constituted just 2–25% of the total test suite, yet they achieved full mutation scores in all cases but one. Moreover, in six out of seven projects, this subset achieved an average real

Table 5.4: *Impact analysis of stubborn mutants for all projects. T_{Stb} is the set of testing killing stubborn mutants and $\mathcal{R}$ is a random set of tests*

Project (Avg Size / Avg MS)	**Threshold Level**	**Avg MS**		**Perfect MS**		**Avg FRR**		**Perfect FRR**		**Avg Size**
		T_{Stb}	$\mathcal{R}$	T_{Stb}	$\mathcal{R}$	T_{Stb}	$\mathcal{R}$	T_{Stb}	$\mathcal{R}$	
Cli (1738 / 0.9)	0.75	0.9	0.85	1.0	0.0	0.68	0.40	0.50	0.0	141
	0.50	0.9	0.84	1.0	0.0	0.68	0.38	0.5	0.0	136
	0.25	0.9	0.82	1.0	0.0	0.64	0.37	0.25	0.0	125
	0.10	0.9	0.76	1.0	0.0	0.63	0.24	0.0	0.0	107
	0.05	0.9	0.73	1.0	0.0	0.5	0.16	0.0	0.0	95
Compress (1617 / 0.72)	0.75	0.72	0.54	1.0	0.0	0.63	0.33	0.2	0.0	112
	0.50	0.72	0.56	1.0	0.0	0.63	0.33	0.2	0.0	108
	0.25	0.70	0.46	0.6	0.0	0.63	0.24	0.2	0.0	104
	0.10	0.69	0.34	0.4	0.0	0.43	0.06	0.2	0.0	98
	0.05	0.69	0.31	0.4	0.0	0.43	0.03	0.2	0.0	97
Csv (1902 / 0.69)	0.75	0.69	0.64	0.8	0.0	0.73	0.38	0.6	0.0	32
	0.50	0.69	0.63	0.8	0.0	0.73	0.38	0.6	0.0	32
	0.25	0.69	0.62	0.6	0.0	0.53	0.27	0.4	0.0	27
	0.10	0.68	0.50	0.6	0.0	0.53	0.06	0.4	0.0	20
	0.05	0.63	0.44	0.4	0.0	0.53	0.02	0.4	0.0	19
Jsoup (1798 / 0.86)	0.75	0.83	0.67	0.9	0.1	0.82	0.31	0.8	0.0	51
	0.50	0.83	0.67	0.9	0.0	0.82	0.27	0.8	0.0	49
	0.25	0.83	0.56	0.9	0.0	0.72	0.15	0.7	0.0	47
	0.10	0.83	0.54	0.9	0.0	0.72	0.11	0.7	0.0	43
	0.05	0.79	0.49	0.8	0.0	0.6	0.06	0.5	0.0	40
Lang (1811 / 0.84)	0.75	0.84	0.77	1.00	0.27	0.88	0.46	0.82	0.00	268
	0.50	0.77	0.68	0.82	0.18	0.78	0.39	0.73	0.00	259
	0.25	0.76	0.67	0.73	0.18	0.72	0.37	0.64	0.00	246
	0.10	0.67	0.57	0.64	0.09	0.67	0.25	0.64	0.00	202
	0.05	0.58	0.50	0.45	0.00	0.58	0.16	0.55	0.00	155
Math (625 / 0.82)	0.75	0.82	0.73	0.94	0.10	0.84	0.43	0.74	0.00	157
	0.50	0.79	0.66	0.90	0.06	0.81	0.38	0.71	0.00	151
	0.25	0.78	0.64	0.84	0.06	0.77	0.32	0.68	0.00	125
	0.10	0.73	0.57	0.61	0.00	0.69	0.20	0.61	0.00	90
	0.05	0.64	0.44	0.35	0.00	0.54	0.11	0.48	0.00	69
Time (370 / 0.8)	0.75	0.80	0.72	1.0	0.0	0.0	0.0	0.0	0.0	102
	0.50	0.80	0.70	1.0	0.0	0.0	0.0	0.0	0.0	98
	0.25	0.80	0.65	1.0	0.0	0.0	0.0	0.0	0.0	83
	0.10	0.80	0.62	1.0	0.0	0.0	0.0	0.0	0.0	64
	0.05	0.78	0.50	0.5	0.0	0.0	0.0	0.0	0.0	34

fault FRR ranging from 63% to 88%, indicating that many fault-revealing tests are captured when focusing only on stubborn mutants. By contrast, randomly selected subsets of the same size consistently underperformed, with lower mutation scores, lower FRR, and never achieving perfect MS or FRR.

5.5.4 RQ4: Characteristics of stubbornness

To investigate whether certain code characteristics are associated with stubborn mutants, I measured the proportion of the stubborn mutants with those characteristics compared to all mutants and performed a series of Chi-Square tests of independence [257]. Since my data has different categories (access modifier, method types, node types, mutation operator, and nesting levels) and count data (how many mutants are Score1 vs. not Score1), a Chi-Square Test of Independence is very appropriate. It tests if there is a significant association between a category and the likelihood of being a Score1 or Score2 mutant, etc. For each characteristic, I analysed the distribution of mutants within the top 5 stubbornness scores (i.e., mutants killed by at most 1–5 tests) compared to all other mutants. I report both the statistical significance (p-value) and the effect size using Cramér's V [81], which measures the strength of association. Possible values of Cramér's V are 0.1 (i.e., small effect), 0.3 (i.e., medium effect), or 0.5 (i.e., large effect).

Although a category value may have the highest observed proportion within a subset, this does not necessarily imply it has the highest statistical significance. To account for this, I use the standardised residuals (z-scores) from the Chi-Square test of independence. These residuals reflect how much the observed count deviates from the expected count under the assumption of independence. A higher z-score indicates a more significant deviation from expectation, regardless of the raw proportion. This helps identify which values are truly over- or underrepresented beyond what would be expected by chance. While proportions highlight raw representation, standardised residuals (z-scores) also account for sample size. A value may have a lower proportion but a higher z-score if it significantly deviates from expected counts in a large sample. Conversely, high proportions in small samples may yield lower z-scores due to greater uncertainty. A standardised residual with an absolute value greater than 2 indicates a statistically significant deviation from the expected count.

Table 5.5 shows the proportions and statistical analysis for each characteristic. Figure 5.3 shows the bar plots of the distribution of stubborn mutants.

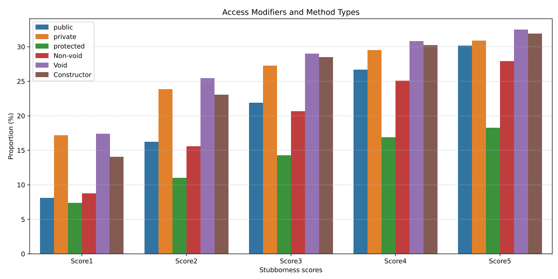
Zhu et al. [348] investigated the relation between achieved mutation score and code testability and observability. In another study, Yao et al. [332] investigated the correlation of five mutant operator classes (ABS, AOR, LCR, ROR, UOI) with equivalent and stubborn mutants. ABS is not part of the default mutation operators in PIT, which I used, and some PIT mutation operators are method-level operators that are not in any of the five general classes. It would be interesting to find a correlation

Table 5.5: *Proportions and statistical analysis of each category across the top five scored subsets. Bold values indicate the highest proportions and standardised residuals (z-scores) within each category for the corresponding subset. p denotes the p-value, C is Cramér's V (effect size), z represents standardised residuals, and % indicates the proportion of the subset. An asterisk (*) denotes* $p < 0.001$.

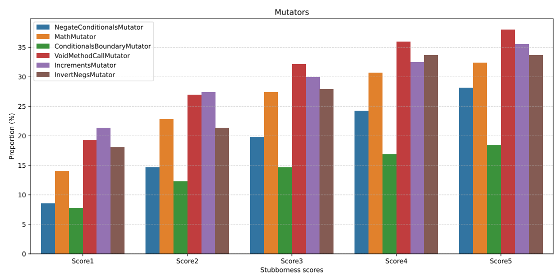
Category	Value	Score1				Score2				Score3				Score4				Score5			
		%	z	p	C	%	z	p	C	%	z	p	C	%	z	p	C	%	z	p	C
Access Modifier	public	8.1	-7.0			16.2	-3.6			21.9	-1.5			26.7	0.3			30.2	1.7		
	private	**17.2**	**12.0**	*	0.13	**23.8**	**8.5**	*	0.11	**27.2**	**5.8**	*	0.09	**29.5**	**3.6**	*	0.08	**30.8**	**2.1**	*	0.08
	protected	7.4	-4.0			11.0	-6.3			14.3	-6.7			16.9	-7.1			18.3	-7.6		
	package-private	11.0	0.1			19.4	0.7			24.6	0.8			28.5	0.8			30.6	0.6		
Method Type	Non-void	8.7	-6.3			15.6	-5.7			20.6	-4.4			25.1	-2.8			27.9	-2.1		
	Void	**17.4**	**10.6**	*	0.12	**25.4**	**9.3**	*	0.11	**29.0**	**6.9**	*	0.09	**30.8**	**4.5**	*	0.06	**32.4**	**3.3**	*	0.04
	Constr	14.1	2.1			23.1	2.6			28.5	2.6			30.2	1.6			31.9	1.1		
Mutator	Negate-Cond	8.6	-4.5			14.6	-5.3			19.7	-4.2			24.3	-2.9			28.1	-1.1		
	Math	14.0	5.8			22.8	**6.6**			27.3	**5.7**			30.7	4.7			32.4	3.7		
	Cond-Boundary	7.8	-3.6			12.3	-5.3			14.6	-6.6			16.9	-7.3			18.4	-7.7		
	NullRtnVal	5.8	-4.8			12.6	-4.0			16.8	-3.9			19.1	-4.5			20.4	-5.0		
	Void-MthdCall	19.2	**7.4**			26.9	**6.1**			32.1	**5.7**			**36.0**	**5.4**			37.9	-3.1		
	Primitive-RtnCall	10.0	-0.6	*	0.12	16.3	-0.9	*	0.13	23.6	0.4	*	0.13	27.8	0.6	*	0.13	29.9	0.4	*	0.13
	EmptyObj-RtnVals	8.3	-1.5			26.4	3.9			**33.2**	4.3			**36.5**	3.8			**41.5**	**4.5**		
	BoolTrue-RtnVals	9.9	-0.5			16.5	-0.7			22.4	-0.1			29.0	0.9			30.1	0.4		
	Increments	**21.3**	4.5			**27.4**	3.1			30.0	2.1			32.5	1.6			35.5	1.7		
	BoolFalse-RtnVals	9.5	-0.5			20.1	0.6			27.8	1.4			**36.7**	2.6			40.2	2.7		
	InvertNeg	18.0	2.4			21.3	0.9			27.9	1.2			33.6	1.5			33.6	0.9		
Nesting Level	0	8.3	-6.2			13.3	-8.8			17.6	-8.6			21.0	-8.4			23.6	-7.9		
	1	10.8	0.1			20.1	3.0			25.8	3.8			29.6	3.7			31.9	3.2		
	2	14.1	4.0			24.3	**5.9**			29.0	**5.2**			32.4	**4.6**			35.2	**4.6**		
	3	17.6	**5.8**	*	0.11	23.8	3.9	*	0.13	29.6	4.1	*	0.13	35.3	**4.9**	*	0.13	37.4	**4.4**	*	0.12
	4	**19.7**	**5.4**			**30.1**	**5.7**			**33.1**	**4.4**			36.6	4.0			39.1	3.8		
	5	**20.4**	3.7			29.3	3.4			**33.1**	2.7			37.6	2.7			41.4	2.9		
	6	17.9	1.8			**29.9**	2.3			**32.8**	1.7			**38.8**	2.0			**44.8**	2.4		
Node Type	BinaryOpr	13.6	5.6			22.1	6.2			26.5	5.1			29.6	4.0			32.2	3.8		
	IfStmt	8.0	-5.6			13.4	-7.0			17.9	-6.6			22.4	-5.2			25.3	-4.5		
	RtnStmt	7.9	-4.4			16.6	-1.7			22.5	-0.3			26.5	0.0			28.7	-0.4		
	ForStmt	8.1	-2.5			13.8	-2.9			17.3	-3.4			19.4	-4.1			21.4	-4.2		
	MthdInvo.	20.7	**8.8**	*	0.13	28.3	**7.1**	*	0.12	33.5	**6.6**	*	0.12	37.3	**6.2**	*	0.11	39.3	**5.6**	*	0.10
	TernaryE.	17.5	2.2			21.9	1.0			29.8	1.6			36.0	2.0			**42.1**	2.6		
	StmtExp.	3.5	-1.7			6.9	-2.0			10.3	-2.0			17.2	-1.4			17.2	-1.7		
	WhileStmt	26.3	2.1			**36.8**	1.9			**42.1**	1.8			**42.1**	1.3			**42.1**	1.1		
	VarDeclare	**28.6**	2.0			28.6	0.9			35.7	1.0			35.7	0.7			35.7	0.5		

between code observability and the PIT mutation operators with stubborn mutants using different test subjects.

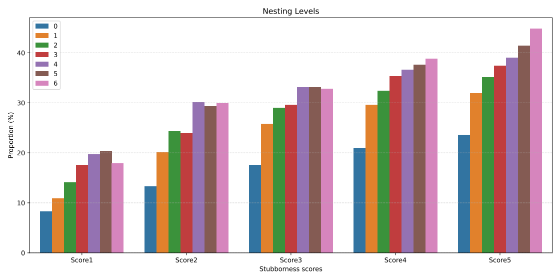
Across all five stubbornness scores, chi-square tests confirmed statistically significant



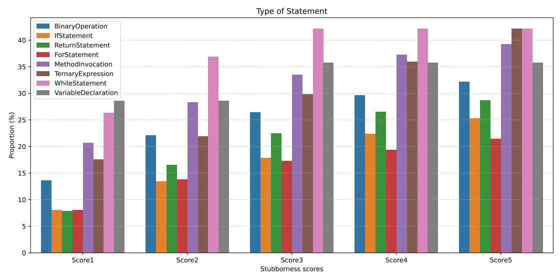
(a) *Access Modifiers and Method Types*



(b) *Mutators*



(c) *Nesting Levels*



(d) *Type of Statement*

Figure 5.3: *Bar plots showing the distribution of stubborn mutants proportions across different categories and ranks.*

associations ($p < 0.001$, denoted by an asterisk in Table 5.5) for each factor, though effect sizes were generally small (Cramér's V: 0.08–0.13), indicating subtle but meaningful trends.

Access Modifier

Stubborn mutants, especially at higher scores, are more likely to appear in *private* methods and less likely in *public* or *protected* methods. This is clearly shown in the first three bars of Figure 5.3a of each score. Private methods consistently showed strong overrepresentation in the top three scores (e.g., Score1 $z = 12.0$), while protected methods were notably underrepresented across all scores. This suggests that reduced visibility may correspond to less observable behaviour, making mutants harder to detect. It is also consistent with Zhu et al. [348] recommendation of refactoring private methods to public methods to increase chances of killing mutants in these methods.

Method Type

Void methods are significantly more prone to hosting stubborn mutants across all scores, particularly Score1 ($z = 10.6$), while *non-void* methods are underrepresented, likely due to the presence of return values that aid assertion-based testing. Constructors show moderate overrepresentation, but the trend is strongest and most consistent for void methods. The last three bars of Figure 5.3a of each score show that void and constructor methods are surpassing non-void methods. Once more, this conforms with the recommendation by Zhu et al. [348] to refactor void methods to non-void methods, since void methods could be more difficult to test.

Mutation Operator

Certain operators are disproportionately responsible for stubborn mutants. `VoidMethodCallMutator`, `MathMutator`, and `EmptyObjectReturnValsMutator` are strongly overrepresented at all scores, suggesting these introduce subtle semantic changes that evade detection. In contrast, mutants from operators like `NegateConditionalsMutator` and `NullReturnValsMutator` are more easily caught, showing consistent underrepresentation in the top scores. The proportions of the distributions are shown in Figure 5.3b, where the `VoidMethodCallMutator` and `IncrementsMutator` have higher proportions than the other mutators.

Yao et al. [332] reported that LCR was the best class to generate stubborn mutants. However, I found that `BooleanTrueReturnValsMutator` and `BooleanFalseReturnValsMutator` (both LCR) show near-expected proportions ($z = -0.5$), indicating their stubbornness aligns with baseline expectations.

Nesting Level

Mutants in more deeply nested code (nesting levels 3–5) are significantly more likely to be stubborn, with Score1 overrepresentation peaking at nesting level 3 ($z = 5.8$). Conversely, flat (level 0) code is associated with low stubbornness across all scores. This is shown in Figure 5.3c. This supports the hypothesis that structural complexity contributes to mutation resilience.

Node Type

Stubborn mutants often arise in specific syntactic constructs. `MethodInvocation` and `BinaryOperation` nodes are consistently overrepresented, particularly at higher scores (e.g., `MethodInvocation` in Score1: $z = 8.8$). In contrast, control-flow constructs like `IfStatement`, `ReturnStatement`, and `ForStatement` are underrepresented, suggesting that test suites more effectively detect mutations in branching logic. Figure 5.3d shows the proportions of the stubborn mutants according to the node types where they occurred.

While my analysis identifies strong associations between certain code characteristics and stubborn mutants, explaining why stubborn mutants preferentially occur in some constructs and not others is beyond the scope of this study and is left for future work.

Conclusion for RQ4: My results demonstrate that certain code characteristics are associated with mutant stubbornness. First, **access modifiers** show that *private* methods are more likely to contain stubborn mutants, likely due to their limited visibility, which restricts test observability. This is evidenced by an over-representation score of 12 at the strictest stubbornness level (Score1), gradually decreasing to 3.3 by Score5. Second, **method return** types reveal that *void* methods are more prone to stubborn mutants, with an over-representation of 10.6 at Score1, similarly decreasing to 3.3 at Score5, as the absence of return values makes it harder for tests to assert correct behaviour. Third, at the level of **mutation operators**, I find that method-level mutation operators generate a higher number of stubborn mutants, with an over-representation of 7.4 at Score1 and maintaining a relatively high 5.4 even at Score4, suggesting that such mutations are harder to detect. Finally, **code nesting depth** is positively correlated with stubbornness, where mutants in nesting level 3 are overrepresented with scores ranging from 5.8 to 3.9 across Score1 to Score5, while level 4 exhibits similar patterns (5.7 to 3.8).

5.6 Chapter Conclusions

In this chapter, I presented the first empirical study examining the relationship between test selection strategies and fault difficulty, using stubborn mutants as a proxy for hard-to-find faults. I compared dynamic coverage-based and lightweight, static diversity-based selection techniques across seven real-world Java projects. My results show that coverage-based approaches (Max-Cov) perform well at broader thresholds, but under stricter definitions of stubbornness, FAST using bytecode, a lightweight diversity-based method, outperforms them, requiring up to 46% fewer tests. Also, I observed strong correlations between mutant stubbornness and structural code properties, including access modifiers, return types, mutation operator categories, and nesting depth. The test cases that exclusively kill stubborn mutants constitute only 2–25% of the total suite, yet achieve full mutation adequacy in nearly all cases and up to 88% fault-revealing rate.

These findings have important implications for regression testing and continuous integration. They suggest that lightweight diversity-based selection can be a scalable alternative to coverage-based approaches, reducing execution and instrumentation costs while still targeting high-utility faults.

Chapter 6

Conclusion and Future Work

In this thesis, I enhanced the understanding and evaluation of diversity-based techniques in software testing, especially in regression testing. I explored the role of diversity-based techniques in software testing, focusing on test suite reduction, prioritisation, and mutant analysis across a series of empirical investigations. Table 6.1 presents a summary of the primary research chapters of this thesis.

In Chapter 2, I conducted a systematic mapping study covering 167 papers that applied diversity-based testing (DBT) to various testing problems. This study revealed that DBT has been most frequently applied in the areas of test data generation and test case prioritisation, with similarity metrics like Euclidean, Edit, and Jaccard distances being the most commonly used. Despite their popularity, no single metric emerged as clearly superior, suggesting that the effectiveness of a given metric may be context-dependent. The mapping also showed that input diversity and test scripts were the most widely used artefacts for diversity calculation, with growing interest in DBT for deep learning systems. However, gaps remain, especially in applying DBT to web, mobile, and large-scale industrial software, highlighting several avenues for future research.

In Chapter 3, I investigated the use of diversity-based reduction on regression test suites generated by Randoop and EvoSuite. I applied five similarity metrics and found that maximising diversity often outperformed random and FAST-based reduction in terms of mutation score, especially for Randoop-generated suites. Among the metrics, Normalised Compression Distance (NCD) emerged as the most effective. While the FAST technique was computationally efficient, it generally underperformed in fault detection. These findings demonstrate that automatically generated test suites can be significantly reduced using diversity-based techniques without compromising fault-detection capability, provided that the reduction strategy is carefully selected.

In Chapter 4, I introduced a novel use of bytecode-level diversity for test case prioritisation. The results showed that bytecode diversity not only improved fault detection rates but also significantly reduced computational overhead compared to traditional textual approaches. Furthermore, filtering bytecode yielded even better efficiency and effectiveness while avoiding the instrumentation costs associated with

Table 6.1: *Summary of the primary research chapters of this thesis.*

Chapter	Summary
3	**Problem:** Test Suite Reduction (TSR) **Subjects:** 13 projects from Defects4J, 13 classes, 6,496 mutants, 16,641 Randoop tests, and 833 EvoSuite tests **Findings:** 1. Reduced test suites based on maximising string dissimilarity outperformed random reduction in over 70% of Randoop and EvoSuite-generated test suites. 2. Outperformed first-n test selection in over 88% of the cases. 3. Normalised Compression Distance (NCD) was the most effective similarity metric on average in terms of fault detection. 4. The reduced test suites based on traditional maximising diversity achieved higher mutation scores in nearly 95% of Randoop-generated cases and 72% of EvoSuite-generated cases.
4	**Problem:** Test Case Prioritisation (TCP) **Subjects:** 7 projects from Defects4J, 67 classes, 18,826 mutants, 114,464 test cases **Findings:** 1. First study to use bytecode-level diversity in TCP, leveraging compactness for enhanced efficiency. 2. Bytecode diversity improves fault detection by 2.3–7.8% over text-based diversity. 3. Bytecode diversity is 2–3 orders of magnitude faster in Ledru-TCP and 2.5–6x faster in FAST-based TCP. 4. Filtering bytecode instructions improves efficiency up to 4x while preserving effectiveness, making bytecode diversity a strong static technique.
5	**Problem:** Killing Stubborn Mutants and Test Case Selection (TCS) **Subjects:** 7 projects from Defects4J, 67 classes, 18,826 mutants, 114,464 test cases **Findings:** 1. Lightweight bytecode-based selection surpasses coverage-based selection under stricter stubbornness thresholds, reducing tests required by up to 46%. 2. Tests that kill only stubborn mutants comprise just 2–25% of the test suite, yet achieve full mutation adequacy and up to 88% real fault reveal rate. 3. Stubborn mutants are valuable testing targets, and bytecode-based diversity techniques offer a scalable path for practical mutation testing.

coverage-based techniques. This positions bytecode diversity as a scalable and practical alternative for environments with limited resources or unavailable coverage data.

In Chapter 5, I presented the first empirical study examining how different test selection strategies affect the types of faults detected, using stubborn mutants as a proxy for hard-to-find faults. I compared dynamic coverage-based approaches with lightweight, static diversity-based methods across seven real-world Java projects. While coverage-based selection performs well at broader stubbornness thresholds, under stricter definitions of stubbornness, FAST using bytecode outperformed coverage-based approaches, requiring up to 46% fewer tests. I also observed strong correlations between mutant stubbornness and structural code properties, including access modifiers, return types, mutation operator categories, and nesting depth. Importantly, the subset of tests that exclusively kill stubborn mutants constitutes only 2–25% of the total suite, yet achieves full mutation adequacy in nearly all cases and up to 88% real fault reveal rate. These findings demonstrate that targeting hard-to-kill faults can effectively guide efficient test selection. They suggest that lightweight diversity-based approaches provide a scalable alternative to coverage-based selection, reducing execution and instrumentation costs while still prioritising tests that reveal high-utility faults.

In summary, this thesis demonstrates the versatility and effectiveness of diversity-based testing across several dimensions of the software testing lifecycle. It provides empirical evidence that diversity, particularly at the bytecode level, can be leveraged to reduce test suite size, improve prioritisation efficiency, and identify test cases capable of killing hard-to-detect faults. These findings contribute both practical insights and methodological innovations to the field of automated software testing.

6.1 Restrictions and Limitations

6.1.1 Empirical Uncertainty

The selection of subject systems in my thesis is from the Defects4J benchmark, while widely adopted in the software engineering research community, but may not fully capture the diversity of real-world software, particularly large-scale or proprietary industrial systems. This limitation affects the generalisability of the results to domains with significantly different development practices, architectures, or scales. Furthermore, not all classes or components within Defects4J projects were analysable due to compatibility or tooling limitations, which may have inadvertently introduced selection bias into the experimental subjects and results.

6.1.2 Open-Source Projects

All empirical evaluations presented in this thesis are based on open-source Java projects. As a result, the findings may not be directly applicable to software written in other programming languages, such as JavaScript, C++, or Python. Moreover, open-source projects may not reflect the same testing constraints, development processes, or codebase complexity found in industrial settings, which could impact the applicability and scalability of the techniques studied in this thesis.

6.2 Future Work

6.2.1 Extended Replication

An important direction for future work is to replicate the empirical evaluations presented in this thesis across a broader range of subject programs, including proprietary software and systems developed in languages other than Java. Such replication would allow for a more comprehensive assessment of the generalisability of the findings. If similar trends were observed, it would strengthen the evidence that the proposed techniques are applicable across diverse software ecosystems. Conversely, significant deviations would prompt deeper investigation into the language- or domain-specific characteristics that influence the techniques presented in this thesis.

Conducting this extended replication poses several challenges. These include the need to establish collaborations with industrial partners willing to provide access to internal codebases and the effort required to reimplement the core analysis techniques for use in other programming environments. Despite these hurdles, this line of inquiry could yield valuable insights and enhance the broader applicability of the models and methodologies developed in this thesis.

6.2.2 Incorporating Diversity-Based Reduction in Randoop

In Chapter 3, I demonstrated that test suites generated by automatic tools like Randoop can be significantly reduced, particularly since Randoop lacks built-in reduction mechanisms. As part of my future work, I aim to extend this reduction approach by incorporating lightweight diversity-based techniques to minimise redundancy in generated tests while preserving fault detection and code coverage. Initial findings suggest that such diversity-based reduction can substantially shrink Randoop-generated suites with minimal or no loss in effectiveness. Further investigation could focus on integrating this method directly into the generation process, enabling Randoop to produce more compact and diverse test suites from the outset.

6.2.3 Investigation of Other Diversity Artefacts

In Chapter 4, I demonstrated that leveraging bytecode representations for diversity calculations significantly enhanced both the efficiency of test case prioritisation and its effectiveness in early fault detection. However, additional artefacts of diversity merit investigation. One promising direction involves the use of non-functional properties. For example, Feldt et al. [110] incorporated metrics such as memory usage and processing time as part of a broader model with 11 variation points. Due to the complexity of their model, it remains unclear how much non-functional properties alone contributed to the results, leaving room for more focused exploration. Shimari et al. [287] also utilised runtime information, gathered during the execution of the system under test (SUT), to inform test case selection in an industrial simulation context. This suggests that execution-time behaviour can be a valuable source of diversity.

Another underexplored area is program state diversity, which has yet to be systematically studied in the context of test selection and prioritisation. Investigating how differences in program state during execution contribute to diversity could offer new insights and further improve selection techniques.

6.2.4 Bytecode Diversity Enhancements

Future work could investigate different ways to filter bytecode information and gain insights into which instructions can be the most beneficial in test case prioritisation. An important enhancement to the approach presented in Chapter 4 is the use of *normalised bytecode*, which can mitigate the effects of compiler-specific optimisations and structural differences that do not impact program behaviour. Raw bytecode may introduce noise due to such non-semantic variations, potentially degrading the quality of diversity metrics. To address this, techniques such as bytecode canonicalisation, variable renaming, and control-flow normalisation, employed by tools like SootDiff [85] and the approach by Schott et al. [278], can be applied to produce more behaviourally consistent representations. These techniques may enhance the effectiveness of diversity-based selection by focusing on semantically meaningful differences. Additionally, future experiments could incorporate datasets containing multi-fault programs and use cost-cognizant versions of evaluation metrics, such as the weighted APFD, to assess the impact of fault severity on prioritisation performance.

6.2.5 Bytecode-Aware Idiom Mining for Test Generation

"Idiom mining" is the process of identifying frequently recurring patterns, known as *idioms* in code, typically to support tasks like code synthesis, recommendation, or comprehension [9]. These idioms often capture common developer practices or usage conventions in specific contexts. An interesting direction for future work is the integration of bytecode-level information in idiom mining to uncover recurring patterns in unit tests. While idiom mining has traditionally focused on source code, incorporating bytecode can reveal structural and semantic test behaviours that may be obscured at the syntactic level, such as specific control flows, exception handling patterns, or API usage semantics. By mining bytecode-level test idioms across diverse projects, it may be possible to identify language-independent testing conventions that can guide automated test generation. These mined idioms could serve as blueprints for synthesising effective and idiomatic test cases in new or under-tested projects, improving both test relevance and coverage.

6.2.6 Identifying Reachability-Stubborn Mutants

As part of future work, I aim to investigate the use of semantic models, such as the Reachability-Infection-Propagation (RIP) model, for detecting stubborn mutants. Unlike frequency-based approaches, RIP-based analysis may reveal deeper execution-level characteristics that contribute to a mutant's resistance to being killed. This could enable the identification of a new subclass of mutants, which I refer to as *reachability-stubborn* mutants, those that are executed (i.e., reached) by multiple tests, yet are only killed by very few of them. This perspective shifts the focus from overall suite size to the relative difficulty of killing a mutant among tests that exercise it. I plan to explore whether diversity-based techniques can effectively identify and eliminate these reachability-stubborn mutants, potentially improving fault detection precision and test suite optimisation.

6.2.7 Higher-Order Stubborn Mutants

Harman et al. [134] demonstrated that Higher Order Mutants (HOMs), mutants composed of multiple first-order mutations, can reduce the number of mutants and required tests, while preserving test effectiveness. Building on this idea, I plan to investigate whether the stubborn mutant characteristics identified in Chapter 5 can be leveraged to construct *higher-order stubborn mutants* that subsume multiple individual stubborn mutants. Such mutants could provide a more compact yet representative set for mutation analysis, potentially reducing computational cost while maintaining or improving the guidance provided to test selection and generation techniques.

Bibliography

[1] S. Abd Halim, D. N. A. Jawawi, and M. Sahak. Similarity distance measure and prioritization algorithm for test case prioritization in software product line testing. *Journal of Information and Communication Technology*, 18(1):57–75, 2019. doi: 10.32890/jict2019.18.1.4.

[2] J. Gbolahan Adigun, Tom P. H., M. Camilli, and M. Felderer. Risk-driven online testing and test case diversity analysis for ml-enabled critical systems. In *Proceedings of the International Symposium on Software Reliability Engineering (ISSRE)*, pages 344–354, 2023. doi: 10.1109/ISSRE59848.2023.00017.

[3] C. C Aggarwal, Al. Hinneburg, and D. A Keim. On the surprising behavior of distance metrics in high dimensional space. In *Database Theory—ICDT*, pages 420–434, 2001. doi: 10.1007/3-540-44503-X_27.

[4] Z. Aghababaeyan, M. Abdellatif, L. Briand, S. Ramesh, and M. Bagherzadeh. Black-box testing of deep neural networks through test case diversity. *IEEE Transactions on Software Engineering*, 49(5):3182–3204, 2023. doi: 10.1109/TSE.2023.3243522.

[5] Z. Aghababaeyan, M. Abdellatif, M. Dadkhah, and L. Briand. Deepgd: A multi-objective black-box test selection approach for deep neural networks. *ACM Transactions on Software Engineering and Methodology*, 33(6), 2024. doi: 10.1145/3644388.

[6] A. Ahmad, d F G Oliveira Neto, E. P. Enoiu, K. Sandahl, and O. Leifler. An industrial study on the challenges and effects of diversity-based testing in continuous integration. In *Proceedings of the International Conference on Software Quality, Reliability, and Security (QRS)*, pages 337–347, 2023. doi: 10.1109/QRS60937.2023.00041.

[7] N. Albunian, G. Fraser, and D. Sudholt. Measuring and maintaining population diversity in search-based unit test generation. In *Proceedings of the International Symposium on Search Based Software Engineering (SSBSE)*, pages 153–168, 2020. doi: 10.1007/978-3-030-59762-7_11.

[8] N. M. Albunian. Diversity in search-based unit test suite generation. In *Proceedings of the International Symposium on Search Based Software Engineering (SSBSE)*, pages 183–189, 2017. doi: 10.1007/978-3-319-66299-2_17.

[9] M. Allamanis, E. T Barr, C. Bird, and C. Sutton. Learning natural coding conventions. In *Proceedings of the SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, pages 281–293, 2014. doi: 10.1145/2635868.2635883.

[10] H. Almulla and G. Gay. Generating diverse test suites for gson through adaptive fitness function selection. In *Proceedings of the International Symposium on Search Based Software Engineering (SSBSE)*, pages 246–252, 2020. doi: 10.1007/978-3-030-59762-7_18.

[11] Wolfram Alpha. Canberradistance - wolfram: Alpha. `https://reference.wolfram.com/language/ref/CanberraDistance.html`, Retrived Sep. 19, 2024.

[12] Wolfram Alpha. Chebyshevdistance - wolfram: Alpha. `https://reference.wolfram.com/language/ref/ChebyshevDistance.html`, Retrived Sep. 19, 2024.

[13] Wolfram Alpha. Cosinedistance - wolfram: Alpha. `https://www.wolframalpha.com/input?i=CosineDistance`, Retrived Sep. 19, 2024.

[14] Wolfram Alpha. Euclideandistance - wolfram: Alpha. `https://mathworld.wolfram.com/EuclideanMetric.html`, Retrived Sep. 19, 2024.

[15] Wolfram Alpha. Kronecker delta - wolfram: Alpha. `https://mathworld.wolfram.com/KroneckerDelta.html`, Retrived Sep. 19, 2024.

[16] Wolfram Alpha. Manhattandistance - wolfram: Alpha. `https://www.wolframalpha.com/input?i=ManhattanDistance`, Retrived Sep. 19, 2024.

[17] S. Alqahtani. A study on the use of vulnerabilities databases in software engineering domain. *Computers & Security*, 116:102661, 2022. doi: 10.1016/j.cose.2022.102661.

[18] N. Alshahwan and M. Harman. Coverage and fault detection of the output-uniqueness test selection criteria. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, pages 181–192, 2014. doi: 10.1145/2610384.2610413.

[19] Nadia Alshahwan and Mark Harman. Automated web application testing using search based software engineering. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*, pages 3–12, 2011. doi: 10.1109/ASE.2011.6100082.

[20] M. Alshraideh, B. A Mahafzah, and S. Al-Sharaeh. A multiple-population genetic algorithm for branch coverage test data generation. *Software Quality Journal*, 19:489–513, 2011. doi: 10.1007/s11219-010-9117-4.

[21] F. Altiero, A. Corazza, S. Di Martino, A. Peron, and L. Libero Lucio Starace. Regression test prioritization leveraging source code similarity with tree kernels. *Journal of Software: Evolution and Process*, page e2653, 2024. doi: 10.1002/smr.2653.

[22] H. Aman, T. Nakano, H. Ogasawara, and M. Kawahara. A test case recommendation method based on morphological analysis, clustering and the mahalanobis-taguchi method. In *Proceedings of the Workshop on Testing: Academia-Industry Collaboration, Practice and Research Techniques (TAICPART)*, pages 29–35, 2017. doi: 10.1109/ICSTW.2017.9.

[23] P. Ammann and J. Offutt. *Introduction to software testing.* Cambridge University Press, 2016.

[24] Md J. Arafeen and H. Do. Test case prioritization using requirements-based clustering. In *Proceedings of the International Conference on Software Testing, Validation and Verification (ICST)*, pages 312–321, 2013. doi: 10.1109/ICST.2013.12.

[25] M. Arjovsky, S. Chintala, and L. Bottou. Wasserstein generative adversarial networks. In *Proceedings of the International Conference on Machine Learning (ICML)*, pages 214–223, 2017.

[26] A. Arrieta, S. Wang, U. Markiegi, A. Arruabarrena, L. Etxeberria, and G. Sagardui. Pareto efficient multi-objective black-box test case selection for simulation-based testing. *Information and Software Technology*, 114:137–154, 2019. doi: 10.1016/j.infsof.2019.06.009.

[27] N. Asoudeh and Y. Labiche. A multi-objective genetic algorithm for generating test suites from extended finite state machines. In *Proceedings of the International Symposium on Search Based Software Engineering (SSBSE)*, pages 288–293, 2013. doi: 10.1007/978-3-642-39742-4_26.

[28] M. Attaoui, H. Fahmy, F. Pastore, and L. Briand. Black-box safety analysis and retraining of dnns based on feature extraction and clustering. *ACM Transactions on Software Engineering and Methodology*, 32(3):1–40, 2023. doi: 10.1145/3550271.

[29] M. Azizi. A tag-based recommender system for regression test case prioritization. In *Proceedings of the International Workshop on Software Test Architecture (InSTA)*, pages 146–157, 2021. doi: 10.1109/ICSTW52544.2021.00035.

[30] Baidu. Apollo: Open source autonomous driving. `https://github.com/ApolloAuto/apollo`, Retrived July. 29, 2024.

[31] R. Beena and S. Sarala. Multi objective test case minimization collaborated with clustering and minimal hitting set. *Journal of Theoretical and Applied Information Technology*, 69(1):200–210, 2014.

[32] M. Beller, G. Gousios, and A. Zaidman. TravisTorrent: Synthesizing travis CI and GitHub for full-stack research on continuous integration. In *Proceedings of the International Conference on Mining Software Repositories (MSR)*, pages 447–450, 2017. doi: 10.1109/MSR.2017.24.

[33] H. M. Benito, M. Boreale, D. Gorla, and D. Clark. Output sampling for output diversity in automatic unit test generation. *IEEE Transactions on Software Engineering*, 48(1):295–308, 2020. doi: 10.1109/TSE.2020.2987377.

[34] A. Bertolino, S. Daoudagh, D. El Kateb, C. Henard, Y. Le Traon, F. Lonetti, E. Marchetti, T. Mouelhi, and M. Papadakis. Similarity testing for access control. *Information and Software Technology*, 58:355–372, 2015. doi: 10.1016/j.infsof.2014.07.003.

[35] M. Biagiola, A. Stocco, F. Ricca, and P. Tonella. Diversity-based web test generation. In *Proceedings of the Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 142–153, 2019. doi: 10.1145/3338906.3338970.

[36] M. Biagiola, A. Stocco, V. Riccio, and P. Tonella. Two is better than one: Digital siblings to improve autonomous driving testing. *Empirical Software Engineering*, 29(4):1–33, 2024. doi: 10.1007/s10664-024-10458-4.

[37] M. Boussaa, O. Barais, G. Sunyé, and B. Baudry. A novelty search approach for automatic test data generation. In *Proceedings of the International Workshop on Search-Based Software Testing (SBST)*, pages 40–43, 2015. doi: 10.1109/SBST.2015.17.

[38] H. B. Braiek and F. Khomh. Deepevolution: A search-based testing approach for deep neural networks. In *Proceedings of the International Conference on Software Maintenance and Evolution (ICSME)*, pages 454–458, 2019. doi: 10.1109/ICSME.2019.00078.

[39] L. Briand. Personal web page. `https://www.lbriand.info/`, Retrived Sep. 19, 2024.

[40] F. Britannica. Mean squared error - britannica. `https://www.britannica.com/science/mean-squared-error`, Retrived Sep. 19, 2024.

[41] A. Z Broder, S. C Glassman, M. S Manasse, and G. Zweig. Syntactic clustering of the web. *Computer networks and ISDN systems*, 29(8-13):1157–1166, 1997. doi: 10.1016/S0169-7552(97)00031-7.

[42] P. A Brooks and A. M Memon. Introducing a test suite similarity metric for event sequence-based test cases. In *Proceedings of the International Conference on Software Maintenance (ICSM)*, pages 243–252, 2009. doi: 10.1109/ICSM.2009.5306305.

[43] P. MS Bueno, M. Jino, and W E. Wong. Diversity oriented test data generation using metaheuristic search techniques. *Information sciences*, 259:490–509, 2014. doi: 10.1016/j.ins.2011.01.025.

[44] P. MS Bueno, W E. Wong, and M. Jino. Improving random test sets using the diversity oriented test data generation. In *Proceedings of the International Workshop on Random Testing (WRST)*, pages 10–17, 2007. doi: 10.1145/1292414.1292419.

[45] P. MS Bueno, W E. Wong, and M. Jino. Automatic test data generation using particle systems. In *Proceedings of the Symposium on Applied Computing (SAC)*, pages 809–814, 2008. doi: 10.1145/1363686.1363871.

[46] P. H. Bugatti, A. JM Traina, and C. Traina Jr. Assessing the best integration between distance-function and image-feature to answer similarity queries. In *Proceedings of the Symposium on Applied Computing (SAC)*, pages 1225–1230, 2008. doi: 10.1145/1363686.1363969.

[47] J. Bürdek, M. Lochau, S. Bauregger, A. Holzer, A. von Rhein, S. Apel, and D. Beyer. Facilitating reuse in multi-goal test-suite generation for software product lines. In *Proceedings of the International Conference on Fundamental Approaches to Software Engineering (FASE)*, pages 84–99, 2015. doi: 10.1007/978-3-662-46675-9_6.

[48] D. Buttler. A short survey of document structure similarity algorithms. Technical report, Lawrence Livermore National Lab.(LLNL), Livermore, CA (United States), 2004.

[49] G. Cai, Q. Su, and Z. Hu. Binary searching iterative algorithm for generating test cases to cover paths. *Applied Soft Computing*, 113:107910, 2021. doi: 10.1016/j.asoc.2021.107910.

[50] L. Cai, W. Tong, Z. Liu, and J. Zhang. Test case reuse based on ontology. In *Proceedings of Pacific Rim International Symposium on Dependable Computing (PRDC)*, pages 103–108, 2009. doi: 10.1109/PRDC.2009.25.

[51] J. Cao, M. Li, Y. Li, M. Wen, SC Cheung, and H. Chen. SemMT: A semantic-based testing approach for machine translation systems. *ACM Transactions on Software Engineering and Methodology*, 31(2):1–36, 2022. doi: 10.1145/3490488.

[52] Y. Cao, Z. Q. Zhou, and T. Y. Chen. On the correlation between the effectiveness of metamorphic relations and dissimilarities of test case executions. In *Proceedings of the International Conference on Quality Software (QSIC)*, pages 153–162, 2013. doi: 10.1109/QSIC.2013.43.

[53] R. Carlson, H. Do, and A. Denton. A clustering approach to improving test case prioritization: An industrial case study. In *Proceedings of the International Conference on Software Maintenance (ICSM)*, pages 382–391, 2011. doi: 10.1109/ICSM.2011.6080805.

[54] E. G Cartaxo, P. DL Machado, and d F G Oliveira Neto. On the use of a similarity function for test case selection in the context of model-based testing. *Software Testing, Verification and Reliability*, 21(2):75–100, 2009. doi: 10.1002/stvr.413.

[55] E. G Cartaxo, d F G Oliveira Neto, and P. DL Machado. Automated test case selection based on a similarity function. In *Proceedings of the Model-based Testing—Workshop (MOTES07) in Conjunction with the Annual Congress of the Gesellschaft fuer Informatik (Lecture Notes in Informatics (LNI))*, volume 110, pages 381–386, 2007.

[56] KP Chan, T. Y. Chen, FC Kuo, and D. Towey. A revisit of adaptive random testing by restriction. In *Proceedings of the International Computer Software and Applications Conference (COMPSAC).*, pages 78–85, 2004. doi: 10.1109/CMPSAC.2004.1342809.

[57] KP Chan, T. Y. Chen, and D. Towey. Restricted random testing. In *Proceedings of the European Conference on Software Quality (ECSQ)*, pages 321–330, 2002. doi: 10.1007/3-540-47984-8_35.

[58] M. P. Chandra et al. On the generalised distance in statistics. *Proceedings of the National Institute of Sciences of India*, 2(1):49–55, 1936.

[59] T. T. Chekam, M. Papadakis, M. Cordy, and Y. Le Traon. Killing stubborn mutants with symbolic execution. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 30(2):1–23, 2021. doi: 10.1145/3425497.

[60] J. Chen, Y. Gu, S. Cai, H. Chen, and J. Chen. KS-TCP: An efficient test case prioritization approach based on k-medoids and similarity. In *Proceedings of the International Symposium on Software Reliability Engineering Workshops (ISSREW)*, pages 105–110, 2021. doi: 10.1109/ISSREW53611.2021.00051.

[61] J. Chen, X. Shen, and T. Menzies. Building very small test suites (with SNAP). *arXiv preprint arXiv:1905.05358*, 2019.

[62] J. Chen, X. Shen, and T. Menzies. Faster SAT solving for software with repeated structures (with case studies on software test suite minimization). *arXiv preprint arXiv:2101.02817*, 2021.

[63] J. Chen, G. Wang, D. Hao, Y. Xiong, H. Zhang, and L. Zhang. History-guided configuration diversification for compiler test-program generation. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*, pages 305–316, 2019. doi: 10.1109/ASE.2019.00037.

[64] T. Y. Chen and D. H Huang. Adaptive random testing by localization. In *Proceedings of the Asia-Pacific Software Engineering Conference (APSEC)*, pages 292–298, 2004. doi: 10.1109/APSEC.2004.17.

[65] T. Y. Chen, F-C Kuo, R. G Merkel, and S. P Ng. Mirror adaptive random testing. *Information and Software Technology*, 46(15):1001–1010, 2004. doi: 10.1016/j.infsof.2004.07.004.

[66] T. Y. Chen, H. Leung, and IK Mak. Adaptive random testing. In *Proceedings of the Annual Asian Computing Science Conference (ACSC)*, pages 320–329, 2004. doi: 10.1007/978-3-540-30502-6_23.

[67] T. Y. Chen, R Merkel, PK Wong, and G Eddy. Adaptive random testing through dynamic partitioning. In *Proceedings of the International Conference on Quality Software (QSIC)*, pages 79–86, 2004. doi: 10.1109/QSIC.2004.1357947.

[68] T. Y. Chen, TH Tse, and Y. T. Yu. Proportional sampling strategy: A compendium and some insights. *Journal of Systems and Software*, 58(1):65–81, 2001. doi: 10.1016/S0164-1212(01)00028-0.

[69] Z. Chen, J. Zhang, and B. Luo. Teaching software testing methods based on diversity principles. In *Proceedings of the Conference on Software Engineering Education and Training (CSEE&T)*, pages 391–395, 2011. doi: 10.1109/CSEET.2011.5876111.

[70] M. Cheng, Y. Zhou, and X. Xie. Behavexplor: Behavior diversity guided testing for autonomous driving systems. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, pages 488–500, 2023. doi: 10.1145/3597926.3598072.

[71] N. Chetouane, F. Wotawa, H. Felbinger, and M. Nica. On using k-means clustering for test suite reduction. In *Proceedings of the Workshop on Testing: Academia-Industry Collaboration, Practice and Research Techniques (TAIC)*, pages 380–385, 2020. doi: 10.1109/ICSTW50294.2020.00068.

[72] S. R Chidamber and C. F Kemerer. A metrics suite for object oriented design. *IEEE Transactions on software engineering*, 20(6):476–493, 1994. doi: 10.1109/32.295895.

[73] R. Cilibrasi and P. MB Vitányi. Clustering by compression. *IEEE Transactions on Information theory*, 51(4):1523–1545, 2005. doi: 10.1109/TIT.2005.844059.

[74] I. Ciupa, A. Leitner, M. Oriol, and B. Meyer. ARTOO: Adaptive random testing for object-oriented software. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 71–80, 2008. doi: 10.1145/1368088.136809.

[75] James Clark. Trang. `https://relaxng.org/jclark/trang.html`, Retrived July. 3, 2024.

[76] H. Coles, T. Laurent, C. Henard, M. Papadakis, and A. Ventresque. PIT: A practical mutation testing tool for Java. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, pages 449–452, 2016. doi: 10.1145/2931037.2948707.

[77] A. V. B. Coutinho, E. G. Cartaxo, and P. D. de Lima Machado. Test suite reduction based on similarity of test cases. In *Brazilian workshop on systematic and automated software testing—CBSoft*, volume 2013, 2013.

[78] A. V. B. Coutinho, E. G. Cartaxo, and P. D. de Lima Machado. Analysis of distance functions for similarity-based test suite reduction in the context of model-based testing. *Software Quality Journal*, 24(2):407–445, 2016. doi: 10.1007/s11219-014-9265-z.

[79] C. Coviello, S. Romano, and G. Scanniello. An empirical study of inadequate and adequate test suite reduction approaches. In *Proceedings of the International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 1–10, 2018. doi: 10.1145/3239235.3240497.

[80] C. Coviello, S. Romano, G. Scanniello, A. Marchetto, G. Antoniol, and A. Corazza. Clustering support for inadequate test suite reduction. In *Proceedings of the International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 95–105, 2018. doi: 10.1109/SANER.2018.8330200.

[81] H. Cramér. *Mathematical methods of statistics*, volume 9. Princeton University Press, 1946. doi: 10.1515/9781400883868.

[82] E. Cruciani, B. Miranda, R. Verdecchia, and A. Bertolino. Scalable approaches for test suite reduction. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 419–429, 2019. doi: 10.1109/ICSE.2019.00055.

[83] H. Dai, CA Sun, H. Liu, and X. Zhang. DFuzzer: Diversity-driven seed queue construction of fuzzing for deep learning models. *IEEE Transactions on Reliability*, 73(2):1075–1089, 2024. doi: 10.1109/TR.2023.3322406.

[84] X. Dang, X. Yao, D. Gong, and T. Tian. Efficiently generating test data to kill stubborn mutants by dynamically reducing the search domain. *IEEE Transactions on Reliability*, 69(1):334–348, 2020. doi: 10.1109/TR.2019.2922684.

[85] A. Dann, B. Hermann, and E. Bodden. SootDiff: Bytecode comparison across different Java compilers. In *Proceedings of the International Workshop on State Of the Art in Program Analysis*, page 14–19, 2019. doi: 10.1145/3315568.3329966.

[86] A. De Lucia, M. Di Penta, R. Oliveto, and A. Panichella. On the role of diversity measures for multi-objective test case selection. In *Proceedings of the International Workshop on Automation of Software Test (AST)*, pages 145–151, 2012. doi: 10.1109/IWAST.2012.6228983.

[87] R. A DeMillo, R. J Lipton, and F. G Sayward. Hints on test data selection: Help for the practicing programmer. *Computer*, 11(4):34–41, 1978.

[88] W. Dickinson, D. Leon, and A Fodgurski. Finding failures by cluster analysis of execution profiles. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 339–348, 2001. doi: 10.1109/ICSE.2001.919107.

[89] H. Do, S. Elbaum, and G. Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Software Engineering*, 10:405–435, 2005. doi: 10.1007/s10664-005-3861-2.

[90] F. Dobslaw, Francisco G. de Oliveira N., and R. Feldt. Boundary value exploration for software analysis. In *Proceedings of the Workshop on NEXt Level of Test Automation (NEXTA)*, pages 346–353, 2020. doi: 10.1109/ICSTW50294.2020.00062.

[91] S. Dola, R. McDaniel, M. B Dwyer, and M. L. Soffa. CIT4DNN: Generating diverse and rare inputs for neural networks using latent space combinatorial testing. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 1–13, 2024. doi: 10.1145/3597503.3639106.

[92] H. Du, V.K. Palepu, and J.A. Jones. To kill a mutant: An empirical study of mutation testing kills. In *Proceedings of the International Symposium on Software Testing and Analysis (SIGSOFT)*, pages 715–726, 2023. doi: 10.1145/3597926.3598090.

[93] S. Elbaum, A. Malishevsky, and G. Rothermel. Incorporating varying test costs and fault severities into test case prioritization. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 329–338, 2001. doi: 10.1109/ICSE.2001.919106.

[94] S. Elbaum, A.G. Malishevsky, and G. Rothermel. Test case prioritization: A family of empirical studies. *IEEE Transactions on Software Engineering (TSE)*, 28(2):159–182, 2002. doi: 10.1109/32.988497.

[95] I. Elgendy. A systematic mapping study of diversity-based approached in software testing: Bibliography. `https://github.com/islamelgendy/DBT-bibliography`, 2024. [Online; accessed 27-June-2024].

[96] I. Elgendy. Replication package. `https://github.com/islamelgendy/Diversity-test-suite-reduction/tree/main`, 2024. [Online; accessed 15-Janurary-2024].

[97] I. Elgendy, R. Hierons, and P. McMinn. Evaluating string distance metrics for reducing automatically generated test suites. In *Proceedings of the International Conference on Automation of Software Test (AST)*, pages 171–181, 2024. doi: 10.1145/3644032.3644455.

[98] I. Elgendy, R. Hierons, and P. McMinn. Empirically evaluating the use of bytecode for diversity-based test case prioritisation. In *Proceedings of the International Conference on Evaluation and Assessment in Software Engineering (EASE)*, 2025. doi: 10.1145/3756681.3756969.

[99] I. Elgendy, R. Hierons, and P. McMinn. Replication Package for Empirically Evaluating the Use of Bytecode for Diversity-Based Test Case Prioritisation. `https://github.com/islamelgendy/Replication-Package-Evaluating-Bytecode-Diversity`, 2025. [Online; accessed 25-March-2025].

[100] I. Elgendy, R. Hierons, and P. McMinn. A systematic mapping study of the metrics, uses and subjects of diversity-based testing techniques. *Software Testing, Verification and Reliability*, 35(2):e1914, 2025. doi: 10.1002/stvr.1914.

[101] I. Elgendy, R. Hierons, and P. McMinn. How effectively do test selection techniques kill mutants of varying stubbornness? Manuscript in preparation for submission to the International Conference on Software Testing Verification and Validation (ICST), 2026.

[102] B. Eric, L. Romain, and C. Thierry. ASM: A code manipulation tool for the java virtual machine, 2002. Available: `https://asm.ow2.io`.

[103] H. Fahmy, F. Pastore, L. Briand, and T. Stifter. Simulator-based explanation and debugging of hazard-triggering events in DNN-based safety-critical systems. *ACM Transactions on Software Engineering and Methodology*, 32(4), 2023. doi: 10.1145/3569935.

[104] C. Fang, Z. Chen, K. Wu, and Z. Zhao. Similarity-based test case prioritization using ordered sequences of program entities. *Software Quality Journal*, 22(2):335–361, 2014. doi: 10.1007/s11219-013-9224-0.

[105] F. de_A Farzat. Test case selection method for emergency changes. In *Proceedings of the International Symposium on Search Based Software Engineering (SSBSE)*, pages 31–35, 2010. doi: 10.1109/SSBSE.2010.13.

[106] M. Felderer and I. Schieferdecker. A taxonomy of risk-based testing. *International Journal on Software Tools for Technology Transfer*, 16:559–568, 2014. doi: 10.1007/s10009-014-0332-3.

[107] R. Feldt. Personal web page. `http://www.robertfeldt.net/`, Retrived Sep. 19, 2024.

[108] R. Feldt and F. Dobslaw. Towards automated boundary value testing with program derivatives and search. In *Proceedings of the International Symposium on Search Based Software Engineering (SSBSE)*, pages 155–163, 2019. doi: 10.1007/978-3-030-27455-9_11.

[109] R. Feldt, S. Poulding, D. Clark, and S. Yoo. Test set diameter: Quantifying the diversity of sets of test cases. In *Proceedings of the International Conference on Software Testing, Verification and Validation (ICST)*, pages 223–233, 2016. doi: 10.1109/ICST.2016.33.

[110] R. Feldt, R. Torkar, T. Gorschek, and W. Afzal. Searching for cognitively diverse tests: Towards universal test diversity metrics. In *Proceedings of the International Conference on Software Testing Verification and Validation Workshop (ICSTW)*, pages 178–186, 2008. doi: 10.1109/ICSTW.2008.36.

[111] J. Feng, B-B Yin, K-Y Cai, and Z-X Yu. 3-way GUI test cases generation based on event-wise partitioning. In *Proceedings of the International Conference on Quality Software (QSIC)*, pages 89–97, 2012. doi: 10.1109/QSIC.2012.42.

[112] Y. Feng, Z. Chen, J. A Jones, C. Fang, and B. Xu. Test report prioritization to assist crowdsourced testing. In *Proceedings of the Joint Meeting on Foundations of Software Engineering*, pages 225–236, 2015. doi: 10.1145/2786805.2786862.

[113] J. Ferrer, F. Chicano, and E. Alba. Estimating software testing complexity. *Information and Software Technology*, 55(12):2125–2139, 2013. doi: 10.1016/j.infsof.2013.07.007.

[114] S. Fischer, R. E Lopez-Herrejon, R. Ramler, and A. Egyed. A preliminary empirical assessment of similarity for combinatorial iteraction testing of software product lines. In *Proceedings of the International Workshop on Search-Based Software Testing (SBST)*, pages 15–18, 2016. doi: 10.1145/2897010.289701.

[115] D. Flemström, W. Afzal, and D. Sundmark. Exploring test overlap in system integration: An industrial case study. In *Proceedings of the Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, pages 303–312, 2016. doi: 10.1109/SEAA.2016.34.

[116] D. Flemström, P. Potena, D. Sundmark, W. Afzal, and M. Bohlin. Similarity-based prioritization of test case automation. *Software quality journal*, 26(4):1421–1449, 2018. doi: 10.1007/s11219-017-9401-7.

[117] G. Fraser and A. Arcuri. EvoSuite: Automatic test suite generation for object-oriented software. In *Proceedings of the SIGSOFT symposium and the European*

conference on Foundations of software engineering, pages 416–419, 2011. doi: 10.1145/2025113.2025179.

[118] G. Fraser and A. Zeller. Mutation-driven generation of unit tests and oracles. *IEEE Transactions on Software Engineering*, 38(2):278–292, 2011. doi: 10.1145/1831708.183172.

[119] X. Gao, Y. Feng, Y. Yin, Zixi Liu, Zhenyu Chen, and Baowen Xu. Adaptive test selection for deep neural networks. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 73–85, 2022. doi: 10.1145/3510003.3510232.

[120] M. Gligoric, L. Eloussi, and D. Marinov. Practical regression test selection with dynamic file dependencies. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, pages 211–222, 2015. doi: 10.1145/2771783.2771784.

[121] O. Goldreich. *On testing expansion in bounded-degree graphs*, volume 20. 2000.

[122] L. Gong, D. Lo, L. Jiang, and H. Zhang. Diversity maximization speedup for fault localization. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*, pages 30–39, 2012. doi: 10.1145/2351676.2351682.

[123] L. Gonzalez-Hernandez, B. Lindström, J. Offutt, S.F. Andler, P. Potena, and M. Bohlin. Using mutant stubbornness to create minimal and prioritized test sets. In *Proceedings of the International Conference on Software Quality, Reliability and Security (QRS)*, pages 446–457, 2018. doi: 10.1109/QRS.2018.00058.

[124] Google. Google scholar. `https://scholar.google.com/`, Retrived Sep. 19, 2024.

[125] J. C Gower and P. Legendre. Metric and euclidean properties of dissimilarity coefficients. *Journal of classification*, 3:5–48, 1986. doi: 10.1007/BF01896809.

[126] R. Greca, B. Miranda, M. Gligoric, and A. Bertolino. Comparing and combining file-based selection and similarity-based prioritization towards regression test orchestration. In *Proceedings of the International Conference on Automation of Software Test (AST)*, pages 115–125, 2022. doi: 10.1145/3524481.3527223.

[127] G. F Guarnieri, A. V Pizzoleto, and F. C Ferrari. An automated framework for cost reduction of mutation testing based on program similarity. In *Proceedings of the International Workshop on Mutation Analysis (Mutation)*, pages 179–188, 2022. doi: 10.1109/ICSTW55395.2022.00041.

[128] A Haghighatkhah, M. Mäntylä, M. Oivo, and P. Kuvaja. Test case prioritization using test similarities. In *Proceedings of the International Conference on Product-Focused Software Process Improvement (PROFES)*, pages 243–259, 2018. doi: 10.1007/978-3-030-03673-7_18.

[129] A. Haghighatkhah, M. Mäntylä, M. Oivo, and P. Kuvaja. Test prioritization in continuous integration environments. *Journal of Systems and Software*, 146:80–98, 2018. doi: 10.1016/j.jss.2018.08.061.

[130] D. Hamlet and R. Taylor. Partition testing does not inspire confidence (program testing). *IEEE Transactions on Software engineering*, 16(12):1402–1411, 1990. doi: 10.1109/32.62448.

[131] R. W. Hamming. Error detecting and error correcting codes. *The Bell System Technical Journal*, 29(2):147–160, 1950. doi: 10.1002/j.1538-7305.1950.tb00463.x.

[132] D. Hao, Y. Pan, L. Zhang, W. Zhao, H. Mei, and J. Sun. A similarity-aware approach to testing based fault localization. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*, pages 291–294, 2005. doi: 10.1145/1101908.1101953.

[133] D. Hao, L. Zhang, Y. Pan, H. Mei, and J. Sun. On similarity-awareness in testing-based fault localization. *Automated Software Engineering*, 15(2):207–249, 2008. doi: 10.1007/s10515-008-0025-9.

[134] M. Harman, Y. Jia, P. Reales Mateo, and M. Polo. Angels and monsters: An empirical investigation of potential test effectiveness and efficiency improvement from strongly subsuming higher order mutation. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*, pages 397–408, 2014. doi: 10.1145/2642937.2643008.

[135] J. A Hartigan and M. A Wong. Algorithm as 136: A k-means clustering algorithm. *Journal of the royal statistical society. booktitle c (applied statistics)*, 28(1):100–108, 1979. doi: 10.2307/2346830.

[136] ZW He and CG Bai. GUI test case prioritization by state-coverage criterion. In *Proceedings of the International Workshop on Automation of Software Test (AST)*, pages 18–22, 2015. doi: 10.1109/AST.2015.11.

[137] E. Hellinger. Neue begründung der theorie quadratischer formen von unendlichvielen veränderlichen. *Journal für die reine und angewandte Mathematik*, 1909(136):210–271, 1909. doi: 10.1515/crll.1909.136.210.

[138] H. Hemmati. Personal web page. `https://lassonde.yorku.ca/users/hhemmati`, Retrived Sep. 19, 2024.

[139] H. Hemmati, A. Arcuri, and L. Briand. Reducing the cost of model-based testing through test case diversity. In *Proceedings of the International Conference on Testing Software and Systems (ICTSS)*, pages 63–78, 2010. doi: 10.1007/978-3-642-16573-3_6.

[140] H. Hemmati, A. Arcuri, and L. Briand. Empirical investigation of the effects of test suite properties on similarity-based test case selection. In *Proceedings of the International Conference on Software Testing, Verification and Validation (ICST)*, pages 327–336, 2011. doi: 10.1109/ICST.2011.12.

[141] H. Hemmati, A. Arcuri, and L. Briand. Achieving scalable model-based testing through test case diversity. *ACM Transactions on Software Engineering and Methodology*, 22(1):1–42, 2013. doi: 10.1145/2430536.2430540.

[142] H. Hemmati and L. Briand. An industrial investigation of similarity measures for model-based test case selection. In *Proceedings of the International Symposium on Software Reliability Engineering (ISSRE)*, pages 141–150, 2010. doi: 10.1109/ISSRE.2010.9.

[143] H. Hemmati, L. Briand, A. Arcuri, and S. Ali. An enhanced test case selection approach for model-based testing: An industrial case study. In *Proceedings of the International Symposium on Foundations of Software Engineering (FSE)*, pages 267–276, 2010. doi: 10.1145/1882291.1882331.

[144] H. Hemmati, Z. Fang, and M. V Mantyla. Prioritizing manual test cases in traditional and rapid release environments. In *Proceedings of the International Conference on Software Testing, Verification and Validation (ICST)*, pages 1–10, 2015. doi: 10.1109/ICST.2015.7102602.

[145] C. Henard, M. Papadakis, M. Harman, Y. Jia, and Y. Le Traon. Comparing white-box and black-box test prioritization. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 523–534, 2016. doi: 10.1145/2884781.2884791.

[146] C. Henard, M. Papadakis, G. Perrouin, J. Klein, P. Heymans, and Y. Le Traon. Bypassing the combinatorial explosion: Using similarity to generate and prioritize t-wise test configurations for software product lines. *IEEE Transactions on Software Engineering*, 40(7):650–670, 2014. doi: 10.1109/TSE.2014.2327020.

[147] C. Henard, M. Papadakis, G. Perrouin, J. Klein, and Y. Le Traon. Assessing software product line testing via model-based mutation: An application to similarity testing. In *Proceedings of the Advances in Model Based Testing Workshop (A-MOST)*, pages 188–197, 2013. doi: 10.1109/ICSTW.2013.30.

[148] M. O Hill. Diversity and evenness: a unifying notation and its consequences. *Ecology*, 54(2):427–432, 1973. doi: 10.2307/1934352.

[149] D. Huang, Q. Bu, Y. Qing, Y. Fu, and H. Cui. Feature map testing for deep neural networks. *arXiv preprint arXiv:2307.11563*, 2023. doi: 10.48550/arXiv.2307.11563.

[150] R. Huang, W. Sun, Y. Xu, H. Chen, D. Towey, and X. Xia. A survey on adaptive random testing. *IEEE Transactions on Software Engineering*, 47(10):2052–2083, 2019. doi: 10.1109/TSE.2019.2942921.

[151] R. Huang, Y. Zhou, W. Zong, D. Towey, and J. Chen. An empirical comparison of similarity measures for abstract test case prioritization. In *Proceedings of the Annual Computer Software and Applications Conference (COMPSAC)*, volume 1, pages 3–12, 2017. doi: 10.1109/COMPSAC.2017.271.

[152] A. Ibias and M. Núñez. Using a swarm to detect hard-to-kill mutants. In *International Conference on Systems, Man, and Cybernetics (SMC)*, pages 2190–2195, 2020. doi: 10.1109/SMC42975.2020.9282883.

[153] A. Ibias, M. Núñez, and R. M Hierons. Using mutual information to test from finite state machines: Test suite selection. *Information and Software Technology*, 132:106498, 2021. doi: 10.1016/j.infsof.2020.106498.

[154] MathWorks Inc. Simulink. `https://uk.mathworks.com/help/simulink/examples/`, Retrived July. 29, 2024.

[155] MathWorks Inc. Simulink design verifier. `https://uk.mathworks.com/help/sldv/examples/`, Retrived July. 29, 2024.

[156] P. Jaccard. Étude comparative de la distribution florale dans une portion des alpes et des jura. *Bulletin de la Société Vaudoise des Sciences Naturelles*, 37:547–579, 1901.

[157] JaCoCo. Jacoco implementation design. `http://www.jacoco.org/jacoco/trunk/doc/implementation.html`, 2025. [Last accessed: 25-March-2025].

[158] M. A Jaro. Advances in record-linkage methodology as applied to matching the 1985 census of tampa, florida. *Journal of the American Statistical Association*, 84(406):414–420, 1989. doi: 10.1080/01621459.1989.10478785.

[159] H. Jeffreys. *The theory of probability.* OuP Oxford, 1998.

[160] Y. Jia and M. Harman. An analysis and survey of the development of mutation testing. *IEEE Transactions on Software Engineering*, 37(5):649–678, 2010. doi: 10.1109/TSE.2010.62.

[161] Z. Jiang, H. Li, and R. Wang. Efficient generation of valid test inputs for deep neural networks via gradient search. *Journal of Software: Evolution and Process*, n/a(n/a):e2550, 2023. doi: 10.1002/smr.2550.

[162] L. Joffe and D. Clark. Constructing search spaces for search-based software testing using neural networks. In *Proceedings of the International Symposium on Search*

Based Software Engineering (SSBSE), pages 27–41, 2019. doi: 10.1007/978-3-030-27455-9_3.

[163] JS-frameworks. Front-end JavaScript frameworks. `https://github.com/collections/front-end-javascript-frameworks`, Retrived July. 29, 2024.

[164] R. Just, D. Jalali, and M. D Ernst. Defects4J: A database of existing faults to enable controlled testing studies for Java programs. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, pages 437–440, 2014. doi: 10.1145/2610384.2628055.

[165] R. Just, B. Kurtz, and P. Ammann. Inferring mutant utility from program context. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 284–294, 2017. doi: 10.1145/3092703.309273.

[166] R. M Keller. Formal verification of parallel programs. *Communications of the ACM*, 19(7):371–384, 1976. doi: 10.1145/360248.360251.

[167] M. Khatibsyarbini, M A. Isa, D. NA Jawawi, and R. Tumeng. Test case prioritization approaches in regression testing: A systematic literature review. *Information and Software Technology*, 93:74–93, 2018. doi: 10.1016/j.infsof.2017.08.014.

[168] R. Khojah, C. H. Chao, and d F G Oliveira Neto. Evaluating the trade-offs of text-based diversity in test prioritisation. In *Proceedings of the International Conference on Automation of Software Test (AST)*, pages 168–178, 2023. doi: 10.1109/AST58925.2023.00021.

[169] D. Y. Kichigin. A method of test-suite reduction for regression integration testing. *Programming and Computer Software*, 35(5):282–290, 2009. doi: 10.1134/S0361768809050041.

[170] F. M Kifetew, A. Panichella, A. De Lucia, R. Oliveto, and P. Tonella. Orthogonal exploration of the search space in evolutionary test case generation. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, pages 257–267, 2013. doi: 10.1145/2483760.2483789.

[171] J. Kim, R. Feldt, and S. Yoo. Guiding deep learning system testing using surprise adequacy. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 1039–1049, 2019. doi: 10.1109/ICSE.2019.00108.

[172] J-M Kim and A. Porter. A history-based test prioritization technique for regression testing in resource constrained environments. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 119–129, 2002. doi: 10.1145/581339.581357.

[173] Barbara Ann Kitchenham and Stuart Charters. Guidelines for performing systematic literature reviews in software engineering. *Technical report, Ver. 2.3 EBSE technical report. EBSE*, 2007.

[174] G. Klees, A. Ruef, B. Cooper, S. Wei, and M. Hicks. Evaluating fuzz testing. In *Proceedings of the Conference on Computer and Communications Security (CCS)*, pages 2123–2138, 2018. doi: 10.1145/3243734.3243804.

[175] W. Kong. Transcend adversarial examples: Diversified adversarial attacks to test deep learning model. In *Proceedings of the International Conference on Computer Design (ICCD)*, pages 13–20, 2023. doi: 10.1109/ICCD58817.2023.00013.

[176] A. Krizhevsky and G. Hinton. Learning multiple layers of features from tiny images. *Toronto, ON, Canada*, 2009.

[177] A. Krizhevsky, I. Sutskever, and G. Hinton. ImageNet classification with deep convolutional neural networks. *Advances in neural information processing systems*, 25, 2012. doi: 10.1145/3065386.

[178] A. Kulesza and B. Taskar. Determinantal point processes for machine learning. *Foundations and Trends® in Machine Learning*, 5(2–3):123–286, 2012. doi: 10.1561/2200000044.

[179] S. Kullback and R. Leibler. On information and sufficiencyannals of mathematical statistics. *MathSciNet MATH*, 22(n/a):79–86, 1951.

[180] B. Kurtz, P. Ammann, M. E Delamaro, J. Offutt, and L. Deng. Mutant subsumption graphs. In *Proceedings of the International Conference on Software Testing, Verification and Validation Workshops*, pages 176–185, 2014.

[181] M. Kusner, Y. Sun, N. Kolkin, and K. Weinberger. From word embeddings to document distances. In *Proceedings of the International Conference on Machine Learning (ICML)*, pages 957–966, 2015.

[182] W. Lan, Z. Cui, J. Zhang, J. Yang, and X. Gu. Fuzz testing based on seed diversity analysis. In *Proceedings of the International Conference on Systems, Man, and Cybernetics (SMC)*, pages 145–150, 2023. doi: 10.1109/SMC53992.2023.10394486.

[183] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998. doi: 10.1109/5.726791.

[184] Y. Ledru, A. Petrenko, S. Boroday, and N. Mandran. Prioritizing test cases with string distances. *Automated Software Engineering*, 19(1):65–95, 2012. doi: 10.1007/s10515-011-0093-0.

[185] M. Lee, S. Cha, and H. Oh. Learning seed-adaptive mutation strategies for greybox fuzzing. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 384–396, 2023. doi: 10.1109/ICSE48619.2023.00043.

[186] C. Leslie, E. Eskin, J. Weston, and W. S. Noble. Mismatch string kernels for svm protein classification. *Advances in neural information processing systems*, 5(n/a):1441–1448, 2004.

[187] J. Leveau, X. Blanc, L. Réveillére, J. Falleri, and R. Rouvoy. Fostering the diversity of exploratory testing in web applications. In *Proceedings of the International Conference on Software Testing, Validation and Verification (ICST)*, pages 164–174, 2020. doi: 10.1109/ICST46399.2020.00026.

[188] J. Leveau, X. Blanc, L. Réveillère, J. Falleri, and R. Rouvoy. Fostering the diversity of exploratory testing in web applications. *Software Testing, Verification and Reliability*, 32(5):e1827, 2022. doi: 10.1002/stvr.1827.

[189] V. I. Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. *Soviet Physics Doklady*, 10(8):707–710, 1966.

[190] M. Li, X0 Chen, X. Li, B. Ma, and P. MB Vitányi. The similarity metric. *IEEE transactions on Information Theory*, 50(12):3250–3264, 2004.

[191] M. Li, P. Vitányi, et al. *An introduction to Kolmogorov complexity and its applications*, volume 3. Springer Publishing Company, Incorporated, 3 edition, 2008. doi: 10.5555/1478784.

[192] X. Li, S. Guo, H. Cheng, and H. Jiang. Simulink compiler testing via configuration diversification with reinforcement learning. *IEEE Transactions on Reliability*, 73(2):1060–1074, 2024. doi: 10.1109/TR.2023.3317643.

[193] Z. Li, K. Hou, and H. Li. Similarity measurement based on trigonometric function distance. In *International Symposium on Pervasive Computing and Applications*, pages 227–231, 2006. doi: 10.1109/SPCA.2006.297573.

[194] H. Lin, Y. Wang, Y. Gong, and D. Jin. Domain-RIP analysis: A technique for analyzing mutation stubbornness. *IEEE Access*, 7:4006–4023, 2019. doi: 10.1109/ACCESS.2018.2883776.

[195] J. Lin, F. Wang, and P. Chu. Using semantic similarity in crawling-based web application testing. In *Proceedings of the International Conference on Software Testing, Verification and Validation (ICST)*, pages 138–148, 2017. doi: 10.1109/ICST.2017.20.

[196] P. Lin. CORE Rankings Portal. `https://www.core.edu.au/conference-portal`, 2024. [Online; accessed 2-July-2024].

[197] B. Lindström, J. Offutt, L. Gonzalez-Hernandez, and S. F Andler. Identifying useful mutants to test time properties. In *International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 69–76, 2018. doi: 10.1109/ICSTW.2018.00030.

[198] B. Liu, Lucia, S. Nejati, and L. Briand. Improving fault localization for Simulink models using search-based testing and prediction models. In *International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 359–370, 2017. doi: 10.1109/SANER.2017.7884636.

[199] B. Liu, Lucia, S. Nejati, L. Briand, and T. Bruckmann. Simulink fault localization: An iterative statistical debugging approach. *Software Testing, Verification and Reliability*, 26(6):431–459, 2016. doi: 10.1002/stvr.1605.

[200] B. Liu, S. Nejati, Lucia, and L.. Briand. Effective fault localization of automotive Simulink models: Achieving the trade-off between test oracle effort and fault localization accuracy. *Empirical Software Engineering*, 24(1):444–490, 2019. doi: 10.1007/s10664-018-9611-z.

[201] J. Liu, J. Tian, X. Kong, I. Lee, and F. Xia. Two decades of information systems: A bibliometric review. *Scientometrics*, 118:617–643, 2019. doi: 10.1007/s11192-018-2974-5.

[202] Q. Liu, K. Chan, and R. Chimhundu. Fintech research: Systematic mapping, classification, and future directions. *Financial Innovation*, 10(1):24, 2024. doi: 10.1186/s40854-023-00524-z.

[203] N. Locascio, K. Narasimhan, E. DeLeon, N. Kushman, and R. Barzilay. Neural generation of regular expressions from natural language with minimal domain knowledge. *arXiv preprint arXiv:1608.03000*, 2016. doi: 10.48550/arXiv.1608.03000.

[204] Y. Lou, J. Chen, L. Zhang, and D. Hao. A survey on regression test-case prioritization. In *Advances in Computers*, volume 113, pages 1–46. Elsevier, 2019. doi: 10.1016/bs.adcom.2018.10.001.

[205] L. Ma, P. Wu, and T. Y. Chen. Diversity driven adaptive test generation for concurrent data structures. *Information and software technology*, 103:162–173, 2018. doi: 10.1016/j.infsof.2018.07.001.

[206] M. Mahdieh, S. Mirian-Hosseinabadi, and M. Mahdieh. Test case prioritization using test case diversification and fault-proneness estimations. *Automated Software Engineering*, 29(2):50, 2022. doi: 10.1007/s10515-022-00344-y.

[207] S. W. Mahfoud. *Niching methods for genetic algorithms*. University of Illinois at Urbana-Champaign, 1995.

[208] A. Marchetto and P. Tonella. Search-based testing of Ajax web applications. In *Proceedings of the International Symposium on Search Based Software Engineering (SSBSE)*, pages 3–12, 2009. doi: 10.1109/SSBSE.2009.13.

[209] B. Marculescu, R. Feldt, and R. Torkar. Using exploration focused techniques to augment search-based software testing: An experimental evaluation. In *Proceedings of the International Conference on Software Testing, Verification and Validation (ICST)*, pages 69–79, 2016. doi: 10.1109/ICST.2016.26.

[210] L. Mariani, A. Mohebbi, M. Pezzè, and V. Terragni. Semantic matching of GUI events for test reuse: Are we there yet? In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, pages 177–190, 2021. doi: 10.1145/3460319.3464827.

[211] S. Masuda, T. Matsuodani, and K. Tsuda. Syntax-tree similarity for test-case derivability in software requirements. In *Proceedings of the International Workshop on Software Test Architecture (InSTA)*, pages 162–172, 2021. doi: 10.1109/ICSTW52544.2021.00037.

[212] R. Matinnejad, S. Nejati, L. Briand, and T. Bruckmann. Effective test suites for mixed discrete-continuous stateflow controllers. In *Proceedings of the Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*, pages 84–95, 2015. doi: 10.1145/2786805.2786818.

[213] R. Matinnejad, S. Nejati, L. Briand, and T. Bruckmann. Automated test suite generation for time-continuous Simulink models. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 595–606, 2016. doi: 10.1145/2884781.2884797.

[214] R. Matinnejad, S. Nejati, L. Briand, and T. Bruckmann. SimCoTest: A test suite generation tool for Simulink/Stateflow controllers. In *Proceedings of International Conference on Software Engineering Companion (ICSE-C)*, pages 585–588, 2016. doi: 10.1145/2889160.2889162.

[215] R. Matinnejad, S. Nejati, L. Briand, and T. Bruckmann. Test generation and test prioritization for Simulink models with dynamic behavior. *IEEE Transactions on Software Engineering*, 45(9):919–944, 2019. doi: 10.1109/TSE.2018.2811489.

[216] B. W Matthews. Comparison of the predicted and observed secondary structure of t4 phage lysozyme. *Biochimica et Biophysica Acta (BBA)-Protein Structure*, 405(2):442–451, 1975. doi: 10.1016/0005-2795(75)90109-9.

[217] J. Mayer, T. Y. Chen, and D. H. Huang. Adaptive random testing through iterative partitioning revisited. In *Proceedings of the International Workshop on Software Quality Assurance (WQA)*, pages 22–29, 2006. doi: 10.1145/1188895.1188903.

[218] P. McMinn. Search-based software test data generation: a survey. *Software testing, Verification and reliability*, 14(2):105–156, 2004. doi: 10.1002/stvr.294.

[219] L. Mei, Y. Cai, C. Jia, B. Jiang, and WK Chan. Test pair selection for test case prioritization in regression testing for WS-BPEL programs. *International Journal of Web Services Research (IJWSR)*, 10(1):73–102, 2013. doi: 10.4018/jwsr.2013010104.

[220] H. D. Menendez and D. Clark. Hashing fuzzing: Introducing input diversity to improve crash detection. *IEEE Transactions on Software Engineering*, 48(09):3540–3553, 2022. doi: 10.1109/TSE.2021.3100858.

[221] H. D Menéndez, G. Jahangirova, F. Sarro, P. Tonella, and D. Clark. Diversifying focused testing for unit testing. *ACM Transactions on Software Engineering and Methodology*, 30(4):1–24, 2021. doi: 10.1145/3447265.

[222] M.L. Menéndez, J.A. Pardo, L. Pardo, and M.C. Pardo. The jensen-shannon divergence. *Journal of the Franklin Institute*, 334(2):307–318, 1997. doi: 10.1016/S0016-0032(96)00063-4.

[223] C. E Metz. Basic principles of roc analysis. *Seminars in nuclear medicine*, 8(4):283–298, 1978. doi: 10.1016/S0001-2998(78)80014-2.

[224] Collard Michael, Maletic Jonathan, Decker Michael, and Newman Christian. srcml. `https://www.srcml.org/#home`, Retrived July. 3, 2024.

[225] B. Miranda and A. Bertolino. Social coverage for customized test adequacy and selection criteria. In *Proceedings of the International Workshop on Automation of Software Test (AST)*, pages 22–28, 2014. doi: 10.1145/2593501.2593505.

[226] B. Miranda, E. Cruciani, R. Verdecchia, and A. Bertolino. FAST approaches to scalable similarity-based test case prioritization. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 222–232, 2018. doi: 10.1145/3180155.3180210.

[227] B. Miranda, E. Cruciani, R. Verdecchia, and A. Bertolino. FAST replication package. `https://github.com/icse18-FAST/FAST`, 2025. [Last accessed: 25-March-2025].

[228] D. Mondal, H. Hemmati, and S. Durocher. Exploring test suite diversification and code coverage in multi-objective test case selection. In *Proceedings of the International Conference on Software Testing, Verification and Validation (ICST)*, pages 1–10, 2015. doi: 10.1109/ICST.2015.7102588.

[229] V. Mosin, M. Staron, D. Durisic, d F G Oliveira Neto, S. K. Pandey, and A. C. Koppisetty. Comparing input prioritization techniques for testing deep

learning algorithms. In *Proceedings of the Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, pages 76–83, 2022. doi: 10.1109/SEAA56994.2022.00020.

[230] G. J Myers, C. Sandler, and T. Badgett. *The art of software testing.* John Wiley & Sons, 2011.

[231] M. Nass, E. Alégroth, R. Feldt, M. Leotta, and F. Ricca. Similarity-based web element localization for robust test automation. *ACM Transactions on Software Engineering and Methodology*, 32(3):1–30, 2023. doi: 10.1145/3571855.

[232] S. B Needleman and C. D Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of molecular biology*, 48(3):443–453, 1970. doi: 10.1016/0022-2836(70)90057-4.

[233] N. Neelofar and A. Aleti. Towards reliable AI: Adequacy metrics for ensuring the quality of system-level testing of autonomous vehicles. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 1–12, 2024. doi: 10.1145/3597503.3623314.

[234] Y. Netzer, T. Wang, A. Coates, A. Bissacco, B. Wu, and A. Y Ng. Reading digits in natural images with unsupervised feature learning. In *Proceedings of the Workshop on Deep Learning and Unsupervised Feature Learning (DL-UFL)*, page 4, 2011.

[235] H. L. Nguyen and L. Grunske. BEDIVFUZZ: Integrating behavioral diversity into generator-based fuzzing. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 249–261, 2022. doi: 10.1145/3510003.3510182.

[236] B. Nikolik. Test diversity. *Information and Software Technology*, 48(11):1083–1094, 2006. doi: 10.1016/j.infsof.2006.02.001.

[237] B. Nikolik. Test suite oscillations. *Information processing letters*, 98(2):47–55, 2006. doi: 10.1016/j.ipl.2005.11.016.

[238] B. Nikolik. The π measure. *IET software*, 2(5):404–416, 2008. doi: 10.1049/iet-sen:20070094.

[239] B. Nikolik. Software quality assurance economics. *Information and Software Technology*, 54(11):1229–1238, 2012. doi: 10.1016/j.infsof.2012.06.003.

[240] T. B. Noor and H. Hemmati. A similarity-based approach for test case prioritization using historical failure data. In *Proceedings of the International Symposium on Software Reliability Engineering (ISSRE)*, pages 58–68, 2015. doi: 10.1109/ISSRE.2015.7381799.

[241] F. LS Nunes, M. E. Delamaro, V. M. Gonçalves, and M. De Souza Lauretto. CBIR based testing oracles: An experimental evaluation of similarity functions. *International Journal of Software Engineering and Knowledge Engineering*, 25(08):1271–1306, 2015. doi: 10.1142/S0218194015500254.

[242] Encyclopedia of mathematics. Hellinger distance - encyclopedia of mathematics. `https://encyclopediaofmath.org/index.php?title=Hellinger_distance`, Retrived Sep. 19, 2024.

[243] A. J. Offutt. Investigations of the software testing coupling effect. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 1(1):5–20, 1992. doi: 10.1145/125489.125473.

[244] M. Ojdanic, A. Garg, A. Khanfir, R. Degiovanni, M. Papadakis, and Y. Le Traon. Syntactic versus semantic similarity of artificial and real faults in mutation testing studies. *IEEE Transactions on Software Engineering*, 49(7):3922–3938, 2023. doi: 10.1109/TSE.2023.3277564.

[245] d F G Oliveira Neto. Personal web page. `https://www.franciscogon.com/`, Retrived Sep. 19, 2024.

[246] d F G Oliveira Neto, A. Ahmad, O. Leifler, K. Sandahl, and E. Enoiu. Improving continuous integration with similarity-based test case selection. In *Proceedings of the International Workshop on Automation of Software Test (AST)*, pages 39–45, 2018. doi: 10.1145/3194733.3194744.

[247] d F G Oliveira Neto, F. Dobslaw, and R. Feldt. Using mutation testing to measure behavioural test diversity. In *Proceedings of the International Workshop on Mutation Analysis (Mutation)*, pages 254–263, 2020. doi: 10.1109/ICSTW50294.2020.00051.

[248] d F G Oliveira Neto, R. Feldt, L. Erlenhov, and J. B. De S. Nunes. Visualizing test diversity to support test optimisation. In *Proceedings of the Asia-Pacific Software Engineering Conference (APSEC)*, pages 149–158, 2018. doi: 10.1109/APSEC.2018.00029.

[249] d F G Oliveira Neto, R. Torkar, and P. DL Machado. Full modification coverage through automatic similarity-based test case selection. *Information and Software Technology*, 80:124–137, 2016. doi: 10.1016/j.infsof.2016.08.008.

[250] C. Pacheco and M. D Ernst. Randoop: Feedback-directed random testing for Java. In *Proceedings of the Companion to the SIGPLAN Conference on Object-Oriented Programming Systems and Applications Companion (OOPSLA Companion)*, pages 815–816, 2007. doi: 10.1145/1297846.1297902.

[251] C. Pacheco, S. K Lahiri, M. D Ernst, and T. Ball. Feedback-directed random test generation. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 75–84, 2007. doi: 10.1109/ICSE.2007.37.

[252] R. Padhye, C. Lemieux, K. Sen, M. Papadakis, and Y. Le Traon. Semantic fuzzing with zest. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, pages 329–340, 2019. doi: 10.1145/3293882.3330576.

[253] R. Pan, T. A Ghaleb, and L. Briand. ATM: Black-box test case minimization based on test code similarity and evolutionary search. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 1700–1711, 2023. doi: 10.1109/ICSE48619.2023.00146.

[254] A. Panchapakesan, R. Abielmona, and E. Petriu. Dynamic white-box software testing using a recursive hybrid evolutionary strategy/genetic algorithm. In *Proceedings of the Congress on Evolutionary Computation (CEC)*, pages 2525–2532, 2013. doi: 10.1109/CEC.2013.6557873.

[255] A. Panichella, R. Oliveto, M. Di Penta, and A. De Lucia. Improving multi-objective test case selection by injecting diversity in genetic algorithms. *IEEE Transactions on Software Engineering*, 41(4):358–383, 2015. doi: 10.1109/TSE.2014.2364175.

[256] M. Papadakis, T. T. Chekam, and Y. Le Traon. Mutant quality indicators. In *Proceedings of the International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 32–39, 2018. doi :10.1109/ICSTW.2018.00025.

[257] K. Pearson. On the criterion that a given system of deviations from the probable in the case of a correlated system of variables is such that it can be reasonably supposed to have arisen from random sampling. *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*, 50(302):157–175, 1900. doi: 10.1080/14786440009463897.

[258] Y. Pei, A. Christi, X. Fern, A. Groce, and WK Wong. Taming a fuzzer using delta debugging trails. In *Proceedings of the International Conference on Data Mining Workshop (ICDMW)*, pages 840–843, 2014. doi: 10.1109/ICDMW.2014.58.

[259] K. Petersen, R. Feldt, S. Mujtaba, and M. Mattsson. Systematic mapping studies in software engineering. In *Proceedings of the International Conference on Evaluation and Assessment in Software Engineering (EASE)*, pages 68–77, 2008. doi: 10.14236/ewic/EASE2008.8.

[260] A. V Pizzoleto, F. C Ferrari, L. D Dallilo, and J. Offutt. SiMut: Exploring program similarity to support the cost reduction of mutation testing. In *Proceedings of the International Workshop on Mutation Analysis (Mutation)*, pages 264–273, 2020. doi: 10.1109/ICSTW50294.2020.00052.

[261] S. Poulding and R. Feldt. Automated random testing in multiple dispatch languages. In *Proceedings of the International Conference on Software Testing, Verification and Validation (ICST)*, pages 333–344, 2017. doi: 10.1109/ICST.2017.37.

[262] R. I. Rabin and M. A. Alipour. Configuring test generators using bug reports: A case study of GCC compiler and CSmith. In *Proceedings of the Symposium on Applied Computing (SAC)*, pages 1750–1758, 2021. doi: 10.1145/3412841.3442047.

[263] H. T. Reda, A. Anwar, and A. Mahmood. Comprehensive survey and taxonomies of false data injection attacks in smart grids: Attack models, targets, and impacts. *Renewable and Sustainable Energy Reviews*, 163:112423, 2022. doi: 10.1016/j.rser.2022.112423.

[264] S. Reddy, C. Lemieux, R. Padhye, and K. Sen. Quickly generating diverse valid test inputs with reinforcement learning. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 1410–1421, 2020. doi: 10.1145/3377811.3380399.

[265] V. Riccio, N. Humbatova, G. Jahangirova, and P. Tonella. DeepMetis: Augmenting a deep learning test set to increase its mutation score. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*, pages 355–367, 2021. doi: 10.1109/ASE51524.2021.9678764.

[266] E. Rogstad and L. Briand. Clustering deviations for black box regression testing of database applications. *IEEE Transactions on Reliability*, 65(1):4–18, 2015. doi: 10.1109/TR.2015.2437840.

[267] E. Rogstad and L. Briand. Cost-effective strategies for the regression testing of database applications: Case study and lessons learned. *Journal of Systems and Software*, 113:257–274, 2016. doi: 10.1016/j.jss.2015.12.003.

[268] E. Rogstad, L. Briand, R. Dalberg, M. Rynning, and E. Arisholm. Industrial experiences with automated regression testing of a legacy database application. In *Proceedings of the International Conference on Software Maintenance (ICSM)*, pages 362–371, 2011. doi: 10.1109/ICSM.2011.6080803.

[269] E. Rogstad, L. Briand, and R. Torkar. Test case selection for black-box regression testing of database applications. *Information and Software Technology*, 55(10):1781–1795, 2013. doi: 10.1016/j.infsof.2013.04.004.

[270] G. Rong, B. H. Shin, H. Tabatabaee, Q. Lu, S. Lemke, M. Možeiko, E. Boise, G. Uhm, M. Gerow, S. Mehta, E. Agafonov, T. H. Kim, E. Sterner, K. Ushiroda, M. Reyes, D. Zelenkovsky, and S. Kim. LGSVL simulator: A high fidelity simulator for autonomous driving. In *Proceedings of the International Conference on Intelligent Transportation Systems (ITSC)*, pages 1–6, 2020. doi: 10.1109/ITSC45102.2020.9294422.

[271] R. H Rosero, O. S Gómez, and G. Rodríguez. An approach for regression testing of database applications in incremental development settings. In *Proceedings of the International Conference on Software Process Improvement (CIMPS)*, pages 1–4, 2017. doi: 10.1109/CIMPS.2017.8169952.

[272] G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold. Test case prioritization: An empirical study. In *Proceedings of the International Conference on Software Maintenance (ICSM)*, pages 179–188, 1999. doi: 10.1109/ICSM.1999.792604.

[273] G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold. Prioritizing test cases for regression testing. *IEEE Transactions on Software Engineering*, 27(10):929–948, 2001. doi: 10.1109/32.962562.

[274] K. K. Sabor and S. Thiel. Adaptive random testing by static partitioning. In *Proceedings of the International Workshop on Automation of Software Test (AST)*, pages 28–32, 2015. doi: 10.1109/AST.2015.13.

[275] B. Sadeghiyan and M. Mouzarani. A new view on classification of software vulnerability mitigation methods. *Global Journal of Computer Science and Technology*, 17(C1):41–61, 2017.

[276] A. Sanfeliu and K-S Fu. A distance measure between attributed relational graphs for pattern recognition. *IEEE transactions on systems, man, and cybernetics*, SMC-13(3):353–362, 1983. doi: 10.1109/TSMC.1983.6313167.

[277] S. Scalabrino, G. Grano, D. Di Nucci, R. Oliveto, and A. De Lucia. Search-based testing of procedural programs: Iterative single-target or multi-target approach? In *Proceedings of the International Symposium on Search Based Software Engineering (SSBSE)*, pages 64–79, 2016. doi: 10.1007/978-3-319-47106-8_5.

[278] S. Schott, S. E. Ponta, W. Fischer, J. Klauke, and E. Bodden. Java bytecode normalization for code similarity analysis. In *European Conference on Object-Oriented Programming (ECOOP)*, volume 313, pages 37:1–37:29, 2024. doi: 10.4230/LIPIcs.ECOOP.2024.37.

[279] 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*, 28(10):1399–1428, 2018. doi: 10.1142/S0218194018500407.

[280] P. H Sellers. The theory and computation of evolutionary distances: pattern recognition. *Journal of algorithms*, 1(4):359–373, 1980. doi: 10.1016/0196-6774(80)90016-4.

[281] O. Semeráth, R. Farkas, G. Bergmann, and D. Varró. Diversity of graph models and graph generators in mutation testing. *International Journal on Software Tools for Technology Transfer*, 22(1):57–78, 2020. doi: 10.1007/s10009-019-00530-6.

[282] A. Shahbazi and J. Miller. Extended subtree: A new similarity function for tree structured data. *Transactions on Knowledge and Data Engineering*, 26(4):864 – 877, 2014. doi: 10.1109/TKDE.2013.53.

[283] A. Shahbazi and J. Miller. Black-box string test case generation through a multi-objective optimization. *IEEE Transactions on Software Engineering*, 42(4):361–378, 2015. doi: 10.1109/TSE.2015.2487958.

[284] A. Shahbazi, M. Panahandeh, and J. Miller. Black-box tree test case generation through diversity. *Automated Software Engineering*, 25(3):531–568, 2018. doi: 10.1007/s10515-018-0232-y.

[285] G. Shakhnarovich, T. Darrell, and P. Indyk. *Nearest-neighbor methods in learning and vision: theory and practice (neural information processing)*. The MIT press, 2006.

[286] Q. Shi, Z. Chen, C. Fang, Y. Feng, and B. Xu. Measuring the diversity of a test set with distance entropy. *Transactions on Reliability*, 65(1):19–27, 2015. doi: 10.1109/TR.2015.2434953.

[287] K. Shimari, M. Tanaka, T. Ishio, M. Matsushita, K. Inoue, and S. Takanezawa. Selecting test cases based on similarity of runtime information: A case study of an industrial simulator. In *Proceedings of the International Conference on Software Maintenance and Evolution (ICSME)*, pages 564–567, 2022. doi: 10.1109/ICSME55016.2022.00077.

[288] S. Shimmi and M. Rahimi. Leveraging code-test co-evolution patterns for automated test case recommendation. In *Proceedings of the International Conference on Automation of Software Test (AST)*, pages 65–76, 2022. doi: 10.1145/3524481.3527222.

[289] D. Shin, S. Yoo, and D. Bae. Diversity-aware mutation adequacy criterion for improving fault detection capability. In *Proceedings of the International Workshop on Mutation Analysis (Mutation)*, pages 122–131, 2016. doi: 10.1109/ICSTW.2016.37.

[290] D. Shin, S. Yoo, and D. Bae. A theoretical and empirical study of diversity-aware mutation adequacy criterion. *IEEE Transactions on Software Engineering*, 44(10):914–931, 2018. doi: 10.1109/TSE.2017.2732347.

[291] S. Singh, C. Sharma, and U. Singh. A simple technique to find diverse test cases. *International Journal of Computer Applications*, 975(1):88–87, 2013.

[292] T. F Smith and M. S Waterman. Identification of common molecular subsequences. *Journal of molecular biology*, 147(1):195–197, 1981. doi: 10.1016/0022-2836(81)90087-5.

[293] D. Sondhi, M. Jobanputra, D. Rani, S. Purandare, S. Sharma, and R. Purandare. Mining similar methods for test adaptation. *IEEE Transactions on Software Engineering*, 48(07):2262–2276, 2022. doi: 10.1109/TSE.2021.3057163.

[294] H. Srikanth, L. Williams, and J. Osborne. System test case prioritization of new and regression test cases. In *International Symposium on Empirical Software Engineering*, pages 10–pp, 2005. doi: 10.1109/ISESE.2005.1541815.

[295] G. W Stewart. On the early history of the singular value decomposition. *SIAM review*, 35(4):551–566, 1993. doi: 10.1137/1035134.

[296] V. Švábenský̌, J. Vykopal, and P. Čeleda. What are cybersecurity education papers about? a systematic literature review of SIGCSE and ITiCSE conferences. In *Proceedings of the ACM technical symposium on computer science education*, pages 2–8, 2020. doi: 10.1145/3328778.3366816.

[297] S. Tahvili, L. Hatvani, M. Felderer, W. Afzal, M. Saadatmand, and M. Bohlin. Cluster-based test scheduling strategies using semantic relationships between test specifications. In *Proceedings of the International Workshop on Requirements Engineering and Testing (RET)*, pages 1–4, 2018. doi: 10.1145/3195538.3195540.

[298] E. Tanaka and K. Tanaka. The tree-to-tree editing problem. *International Journal of pattern recognition and artificial intelligence*, 2(02):221–240, 1988. doi: 10.1016/0020-0190(77)90064-3.

[299] Y. Tang, H. Jiang, Z. Zhou, X. Li, Z. Ren, and W. Kong. Detecting compiler warning defects via diversity-guided program mutation. *IEEE Transactions on Software Engineering*, 48(11):4411–4432, 2022. doi: 10.1109/TSE.2021.3119186.

[300] Y. Tashiro, Y. Song, and S. Ermon. Diversity can be transferred: Output diversification for white-and black-box attacks. *Advances in neural information processing systems*, 33:4536–4548, 2020.

[301] P. Tonella. Personal web page. `https://www.inf.usi.ch/faculty/tonella/#/`, Retrived Sep. 19, 2024.

[302] C-Y Tsai and G. W Taylor. DeepRNG: Towards deep reinforcement learning-assisted generative testing of software, 2022. doi: 10.48550/arXiv.2201.12602.

[303] P. Tsankov, M. T. Dashti, and D. Basin. SECFUZZ: Fuzz-testing security protocols. In *Proceedings of the International Workshop on Automation of Software Test (AST)*, pages 1–7, 2012. doi: 10.1109/IWAST.2012.6228985.

[304] M. Utting and B. Legeard. *Practical model-based testing: A tools approach*. Elsevier, 2010.

[305] A. Vargha and H. D. Delaney. A critique and improvement of the cl common language effect size statistics of mcgraw and wong. *Journal of Educational and Behavioral Statistics*, 25(2):101–132, 2000.

[306] M. Viggiato, D. Paas, C. Buzon, and CP Bezemer. Identifying similar test cases that are specified in natural language. *IEEE Transactions on Software Engineering*, 49(3):1027–1043, 2023. doi: 10.1109/TSE.2022.3170272.

[307] T. Vogel, C. Tran, and L. Grunske. Does diversity improve the test suite generation for mobile applications? In *Proceedings of the International Symposium on Search Based Software Engineering (SSBSE)*, pages 58–74, 2019. doi: 10.1007/978-3-030-27455-9_5.

[308] T. Vogel, C. Tran, and L. Grunske. A comprehensive empirical evaluation of generating test suites for mobile applications with diversity. *Information and Software Technology*, 130:106436, 2021. doi: 10.1016/j.infsof.2020.106436.

[309] H. Wang and WK Chan. Weaving context sensitivity into test suite construction. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*, pages 610–614, 2009. doi: 10.1109/ASE.2009.79.

[310] H. Wang, WK Chan, and TH Tse. Improving the effectiveness of testing pervasive software via context diversity. *Transactions on Autonomous and Adaptive Systems*, 9(2):1–28, 2014. doi: 10.1145/2620000.

[311] K. Wang, Y. Wang, J. Wang, and Q. Wang. Fuzzing with sequence diversity inference for sequential decision-making model testing. In *Proceedings of the International Symposium on Software Reliability Engineering (ISSRE)*, pages 706–717, 2023. doi: 10.1109/ISSRE59848.2023.00041.

[312] R. Wang, S. Jiang, and D. Chen. Similarity-based regression test case prioritization. In *Proceedings of the International Conference on Software Engineering and Knowledge Engineering (SEKE)*, pages 358–363, 2015. doi: 10.18293/seke2015-115.

[313] W. Wang, Z. Chen, Z. Zheng, and H. Wang. An adaptive fuzzing method based on transformer and protocol similarity mutation. *Computers & Security*, 129(C):103197, 2023. doi: 10.1016/j.cose.2023.103197.

[314] W. Wang, J. Guo, B. Li, Y. Shang, and R. Zhao. EFSM model-based testing for android applications. *International Journal of Software Engineering and Knowledge Engineering*, 34(04):597–621, 2024. doi: 10.1142/S0218194023500638.

[315] W. Wang, S. Wu, Z. Li, and R. Zhao. Parallel evolutionary test case generation for web applications. *Information and Software Technology*, 155:107113, 2023. doi: 10.1016/j.infsof.2022.107113.

[316] X. Wang, S. Jiang, P. Gao, X. Ju, R. Wang, and Y. Zhang. Cost-effective testing based fault localization with distance based test-suite reduction. *Science China Information Sciences*, 60(9):1–15, 2017.

[317] C. Watson, N. Cooper, D. N. Palacio, K. Moran, and D. Poshyvanyk. A systematic literature review on the use of deep learning in software engineering research. *ACM Transactions on Software Engineering and Methodology*, 31(2):1–58, 2022. doi: 10.1145/3485275.

[318] L. J White and E. I Cohen. A domain strategy for computer program testing. *IEEE Transactions on Software Engineering*, SE-6(3):247–257, 1980. doi: 10.1109/TSE.1980.234486.

[319] R. Widyasari, S. Q. Sim, C. Lok, H. Qi, J. Phan, Q. Tay, C. Tan, F. Wee, J. E. Tan, Y. Yieh, B. Goh, F. Thung, H. J. Kang, T. Hoang, D. Lo, and E. L. Ouh. BugsInPy: A database of existing bugs in python programs to enable controlled testing and debugging studies. In *Proceedings of the Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 1556–1560, 2020. doi: 10.1145/3368089.3417943.

[320] W. E Winkler. The state of record linkage and current research problems. In *Statistical Research Division, US Census Bureau*, pages 1–15, 1999.

[321] C. Wohlin. Guidelines for snowballing in systematic literature studies and a replication in software engineering. In *Proceedings of the International Conference on Evaluation and Assessment in Software Engineering (EASE)*, pages 1–10, 2014. doi: 10.1145/2601248.2601268.

[322] W E. Wong, R. Gao, Y. Li, R. Abreu, and F. Wotawa. A survey on software fault localization. *IEEE Transactions on Software Engineering*, 42(8):707–740, 2016. doi: 10.1109/TSE.2016.2521368.

[323] K. Wu, C. Fang, Z. Chen, and Z. Zhao. Test case prioritization incorporating ordered sequence of program elements. In *Proceedings of the International Workshop on Automation of Software Test (AST)*, pages 124–130, 2012. doi: 10.1109/IWAST.2012.6228980.

[324] X. Xia, L. Gong, T-D B Le, D. Lo, L. Jiang, and H. Zhang. Diversity maximization speedup for localizing faults in single-fault and multi-fault programs. *Automated Software Engineering*, 23(1):43–75, 2016. doi: 10.1007/s10515-014-0165-z.

[325] Y. Xiang, H. Huang, M. Li, S. Li, and X. Yang. Looking for novelty in search-based software product line testing. *IEEE Transactions on Software Engineering*, 1(1):1–1, 2021. doi: 10.1109/TSE.2021.3057853.

[326] Y. Xiang, H. Huang, S. Li, M. Li, C. Luo, and X. Yang. Automated test suite generation for software product lines based on quality-diversity optimization. *ACM Transactions on Software Engineering and Methodology*, 33(2):1–52, 2023. doi: 10.1145/3628158.

[327] Q. Xie and A. M Memon. Studying the characteristics of a "Good" GUI test suite. In *Proceedings of the International Symposium on Software Reliability Engineering (ISSRE)*, pages 159–168, 2006. doi: 10.1109/ISSRE.2006.45.

[328] X. Xie, B. Xu, L. Shi, C. Nie, and Y. He. A dynamic optimization strategy for evolutionary testing. In *Proceedings of the Asia-Pacific Software Engineering Conference (APSEC)*, pages 568–575, 2005. doi: 10.1109/APSEC.2005.6.

[329] X. Xie, P. Yin, and S. Chen. Boosting the revealing of detected violations in deep learning testing: A diversity-guided method. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*, pages 1–13, 2022. doi: 10.1145/3551349.3556919.

[330] XY Xie, BW Xu, L Shi, and CH Nie. Genetic test case generation for path-oriented testing. *Journal of Software*, 20(12):3117–3136, 2009. doi: 10.3724/SP.J.1001.2009.00580.

[331] Y. Yang. Improve model testing by integrating bounded model checking and coverage guided fuzzing. *arXiv preprint arXiv:2211.04712*, n/a(n/a):arXiv:2211.04712, 2022. doi: 10.48550/arXiv.2211.04712.

[332] X. Yao, M. Harman, and Y. Jia. A study of equivalent and stubborn mutation operators using human analysis of equivalence. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 919–930, 2014. doi: 10.1145/2568225.256826.

[333] K. Yatoh, K. Sakamoto, F. Ishikawa, and S. Honiden. Feedback-controlled random test generation. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, pages 316–326, 2015. doi: 10.1145/2771783.2771805.

[334] S. Yoo and M. Harman. Regression testing minimization, selection and prioritization: A survey. *Software testing, verification and reliability*, 22(2):67–120, 2012. doi: 10.1002/stvr.430.

[335] S. Yoo, M. Harman, P. Tonella, and A. Susi. Clustering test cases to achieve effective and scalable prioritisation incorporating expert knowledge. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, pages 201–212, 2009. doi: 10.1145/1572272.1572296.

[336] Y-S You, C-Y Huang, K-L Peng, and C-J Hsu. Evaluation and analysis of spectrum-based fault localization with modified similarity coefficients for software debugging. In *Proceedings of the Annual Computer Software and Applications Conference (COMPSAC)*, pages 180–189, 2013. doi: 10.1109/COMPSAC.2013.32.

[337] X. Yu, K. Jia, W. Hu, J. Tian, and J. Xiang. Black-box test case prioritization using log analysis and test case diversity. In *Proceedings of the International Symposium on Software Reliability Engineering Workshops (ISSREW)*, pages 186–191, 2023. doi: 10.1109/ISSREW60843.2023.00072.

[338] Y. Yuan, Q. Pang, and S. Wang. Provably valid and diverse mutations of real-world media data for dnn testing. *IEEE Transactions on Software Engineering*, 50(5), 2024. doi: 10.1109/TSE.2024.3370807.

[339] M. Zalewski. American fuzzy lop, 2014.

[340] D. Zhang, D. Liu, Y. Lei, D. Kung, C. Csallner, N. Nystrom, and W. Wang. SimFuzz: Test case similarity directed deep fuzzing. *Journal of Systems and Software*, 85(1):102–111, 2012. doi: 10.1016/j.jss.2011.07.028.

[341] K. Zhang and D. Shasha. Simple fast algorithms for the editing distance between trees and related problems. *SIAM journal on computing*, 18(6):1245–1262, 1989. doi: 10.1137/0218082.

[342] Z. Zhang and X. Xie. On the investigation of essential diversities for deep learning testing criteria. In *Proceedings of the International Conference on Software Quality, Reliability and Security (QRS)*, pages 394–405, 2019. doi: 10.1109/QRS.2019.00056.

[343] C. Zhao, Y. Mu, X. Chen, J. Zhao, X. Ju, and G. Wang. Can test input selection methods for deep neural network guarantee test diversity? A large-scale empirical study. *Information and Software Technology*, 150:106982, 2022. doi: 10.1016/j.infsof.2022.106982.

[344] L. Zhao, Z. Zhang, L. Wang, and X. Yin. A fault localization framework to alleviate the impact of execution similarity. *International Journal of Software Engineering and Knowledge Engineering*, 23(07):963–998, 2013. doi: 10.1142/S0218194013500289.

[345] X. Zhao, Z. Wang, X. Fan, and Z. Wang. A clustering-bayesian network based approach for test case prioritization. In *Proceedings of the Annual Computer Software and Applications Conference (COMPSAC)*, volume 3, pages 542–547, 2015. doi: 10.1109/COMPSAC.2015.154.

[346] Y. Zheng, Y. Liu, X. Xie, Y. Liu, L. Ma, J. Hao, and Y. Liu. Automatic web testing using curiosity-driven reinforcement learning. In *Proceedings of the*

International Conference on Software Engineering (ICSE), pages 423–435, 2021. doi: 10.1109/ICSE43902.2021.00048.

[347] Y. Zhi, X. Xie, C. Shen, J. Sun, X. Zhang, and X. Guan. Seed selection for testing deep neural networks. *ACM Transactions on Software Engineering and Methodology*, 33(1), 2023. doi: 10.1145/3607190.

[348] Q. Zhu, A. Zaidman, and A. Panichella. How to kill them all: An exploratory study on the impact of code observability on mutation testing. *Journal of Systems and Software*, 173:110864, 2021. doi: 10.1016/j.jss.2020.110864.

[349] T. Zohdinasab, V. Riccio, A. Gambi, and P. Tonella. DeepHyperion: Exploring the feature space of deep learning-based systems through illumination search. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, pages 79–90, 2021. doi: 10.1145/3460319.3464811.

[350] T. Zohdinasab, V. Riccio, A. Gambi, and P. Tonella. Efficient and effective feature space exploration for testing deep learning systems. *ACM Transactions on Software Engineering and Methodology*, 32(2):1–38, 2023. doi: 10.1145/3544792.

[351] T. Zohdinasab, V. Riccio, and P. Tonella. DeepAtash: Focused test generation for deep learning systems. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, pages 954–966, 2023. doi: 10.1145/3597926.3598109.

[352] T. Zohdinasab, V. Riccio, and P. Tonella. An empirical study on low-and high-level explanations of deep learning misbehaviours. In *Proceedings of the International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 1–11, 2023. doi: 10.1109/ESEM56168.2023.10304866.

[353] T. Zohdinasab, V. Riccio, and P. Tonella. Focused test generation for autonomous driving systems. *ACM Transactions on Software Engineering and Methodology*, 33(6), 2024. doi: 10.1145/3664605.

Appendix A

A Systematic Mapping Study of The Metrics, Uses and Subjects of Diversity-Based Testing Techniques

Table A.1: *A list of the generic similarity metrics with a citation of the source, a brief description, the number of papers using them, and citation these papers.*

ID	Metric & Source	Description	Total	Papers
G1	Euclidean distance [14]	The square root of the sum of the squared differences between the vectors X and Y. The formula is: $$Euc(X, Y) = \sqrt{\sum_{i=1}^{n}(x_i - y_i)^2}$$	27	[7, 24, 26, 45, 43, 53, 71, 79, 80, 82, 105, 126, 127, 129, 144, 184, 206, 212, 213, 215, 241, 258, 269, 283, 287, 312, 345]

G2	Jaccard distance [156]	The ratio of intersection over union between two sets A and B of values. The formula is: $Jac(A,B) = 1 - \frac{\|A \cap B\|}{\|A \cup B\|} = 1 - \frac{\|A \cap B\|}{\|A\| + \|B\| - \|A \cap B\|}$ Sometimes expressed, (e.g. [312]) as: $Jac(A,B) = 1 - \frac{A.B}{A.B + \omega(\parallel A \parallel^2 + \parallel B \parallel^2 - 2(A.B))}$ with with $\omega = 1$.	26	[51, 78, 79, 80**?** , 114, 127, 128, 142, 140, 147, 146, 198, 200, 210, 219, 225, 226**?** , 299, 306, 312, 325, 336, 342, 344]
G3	Edit distance [189]	The minimum number of edits *(insertions, deletions or substitutions)* required to change one string into the other. It takes into consideration that parts of the strings can be similar even if not in corresponding places, and can work with strings of different sizes.	26	[10, 27, 35, 37, 51, 78, 79, 80**?** , 104, 115, 127, 143, 142, 139, 141, 184, 205, 210, 228, 240, 283, 315, 323, 340, 288]
G4	Hamming distance [131]	The number of times when the corresponding characters in two strings are different. Some (e.g. [151]) refer to this as "Overlap" distance.	15	[1, 52, 79, 80, 112, 142, 151, 184, 228, 240, 283, 309, 310, 330, 335]
G5	Manhattan distance [16]	The sum of the absolute differences between two vectors X and Y. The formula is: $Man(X,Y) = \sum_{i=1}^{n} \|x_i - y_i\|$	14	[37, 52, 63, 60, 127, 128, 144, 184, 198, 200, 206, 241, 269, 283]

G6	Cosine similarity [13]	The cosine of the angle of two vectors X and Y. The formula is: $Cosine(X,Y) = \frac{\sum_{i=1}^{n} x_i \times y_i}{\sqrt{\sum_{i=1}^{n} x_i^2} \times \sqrt{\sum_{i=1}^{n} y_i^2}}$	11	[29, 42, 79, 80, 195, 206, 281, 283, 293, 306, 312]
G7	Normalised compression distance [73]	An approximation of the Kolmogorov complexity using real-world compressors.	9	[4**?** , 247, 90, 110, 109, 127, 128, 269]
G8	Tree edit distance [341]	The minimum number of edit operations required to change one tree into the other.	3	[219, 282, 284]
G9	Locality-sensitive hashing [285]	A technique that maps similar strings or inputs to the same hash code with high probability to get a fast estimation of the dissimilarity between two subjects.	3	[128, 226, 283]
G10	Crowding distance [207]	A measure of how far a chromosome or an individual is from the rest of the population.	2	[255, 277]
G11	Geometric diversity [178]	The measurement of feature similarity between two feature vectors given an input sample.	2	[4, 161]
G12	Gower-Legendre distance [125]	A variant of the Jaccard Index (G2) where the weight ω is 1/2.	2	[228, 312]
G13	Isolated subTree distance [298]	A variation of tree edit distance (G8), where disjoint subtrees are mapped to similar disjoint subtrees of another set.	2	[282, 284]
G14	Jaro-Winkler distance [320]	A variation of the Jaro distance (G23) that adds more weight in strings starting with the exact match characters.	2	[1, 78]
G15	L2-test [121]	The distance between a uniform distribution and a sampled distribution by checking if the sampled distribution is ϵ-far from uniformity.	2	[221, 220]
G16	Mahalanobis distance [58]	The distance between a point and a distribution.	2	[241, 269]

G17	Needleman-Wunsch distance [232]	Originally used in bioinformatics to align protein or nucleotide sequences, and can be used to identify similarities between two test cases by encoding them.	2	[142, 140]
G18	Canberra distance [11]	A weighted version of the Manhattan distance (G5), in which each term in the sum is normalised.	1	[241]
G19	Chebyshev distance [12]	The greatest difference between two points in two vectors along any coordinate dimension.	1	[241]
G20	Fractional distance [3]	A variation of the Euclidean distance (G1) to deal with multi-dimensional space.	1	[209]
G21	Hellinger distance [242]	The difference between two distributions with Hellinger integral [137].	1	[329]
G22	Hill-numbers [148]	A measure originally used in ecology that considers both species richness and species abundances in a sample.	1	[235]
G23	Jaro distance [158]	The number of matching characters and the number of transpositions (i.e. matching characters but not in order) between two strings.	1	[78]
G24	Jeffrey divergence [159]	A derived distribution from the Kullback-Leibler Divergence (G27) that is symmetric and more robust to noises.	1	[241]
G25	Jensen-Shannon distance [222]	An improved version of Kullback-Leibler Divergence (G27) to measure the similarity of two probability distributions. The metric is symmetric and always has a finite value.	1	[329]
G26	Kronecker delta [15]	A discrete function of two variables that is one if they are equal, 0 otherwise.	1	[169]
G27	Kullback-Leibler divergence [179]	The expected value of the logarithmic difference between two probability distributions, but it is not symmetric.	1	[329]

G28	Mean-square-error [40]	The average squared difference between the values predicted from a model and the actual values.	1	[229]
G29	Modified trigonometric distance [193]	A modified version of the trigonometric distance (G38) with a greater degree of accuracy for points of larger magnitude of values.	1	[241]
G30	N-gram models [41]	A contiguous sequence of n items from a given sample of text or speech.	1	[187]
G31	Proportional distance [88]	The sum of squares of the difference between two vectors over the difference between the maximum and minimum values.	1	[312]
G32	Singular value decomposition [295]	An estimate of where the evolution is going in search-based approaches, by monitoring the movements of individuals across different generations.	1	[170]
G33	Smith-Waterman distance [292]	The alignment of local sequences for determining similar regions between two strings of nucleic acid sequences or protein sequences.	1	[142]
G34	Sokal-Sneath distance [339]	A variant of the Jaccard Index (G2) where the weight ω is 2.	1	[312]
G35	Statistic salue X^2 [46]	A distance function that emphasizes large absolute difference existing between the feature values.	1	[241]
G36	String-Kernels [186]	The inner product between two strings by counting the occurrences of common substrings in the two strings.	1	[312]
G37	Sellers algorithm [280]	A variation of the edit distance (G3) to find a sub-string in another string with at most k edit operations.	1	[78]
G38	Trigonometric distance [193]	A normalised distance between two points used in image matching. The distance between two vectors X and Y is $\sum_{i=1}^{n} \sin(\arctan \lvert x_i - y_i \rvert)$	1	[241]
G39	Wasserstein distance [25]	The difference between two frequency distributions over a region, which is also known as the earth mover's distance.	1	[329]

G40	Word mover's distance [181]	The minimum amount of distance that the embedded words of one document need to be moved to reach the embedded words of another document.	1	[306]

Table A.2: *A list of the specialised Software Engineering similarity metrics.*

ID	Metric & Source	Description	Total	Papers
S1	Identical transition distance [54]	The number of identical transitions between two finite state machines divided by the average length of paths.	6	[54, 249, 143, 142, 139, 141]
S2	Test set diameter (TSDm) [109]	An extension of the pairwise normalised compression distance (G7) to multisets.	5	[109, 128, 129, 282, 284]
S3	Approach level [218]	The number of mismatched branch predicates to reach the target branch.	3	[20, 49, 254]
S4	Identical state distance [143]	The number of identical states between two paths of finite state machines divided by their average number of states.	3	[143, 139, 141]
S5	Trigger-based distance [143]	An extension of identical transition similarity (S1) to account for triggers in the transitions.	3	[143, 139, 141]
S6	Average population diameter [307]	The average distance between all vectors in a population, where the distance between two vectors is the difference of their lengths.	2	[307, 308]
S7	Distinguishing mutation adequacy [289]	An assessment of the diversity of mutants' behaviour based on the mutants' killing information.	2	[289, 290]
S8	Extended subTree distance [282]	A variation of isolated subtree distance (G12) with different mapping conditions.	2	[282, 284]
S9	Path distance [48]	The size of the intersection between the two paths of multisets of trees.	2	[282, 284]
S10	[GUI] State similarity [111]	The difference between the values of two GUI states using the widgets of the GUI.	2	[111, 136]

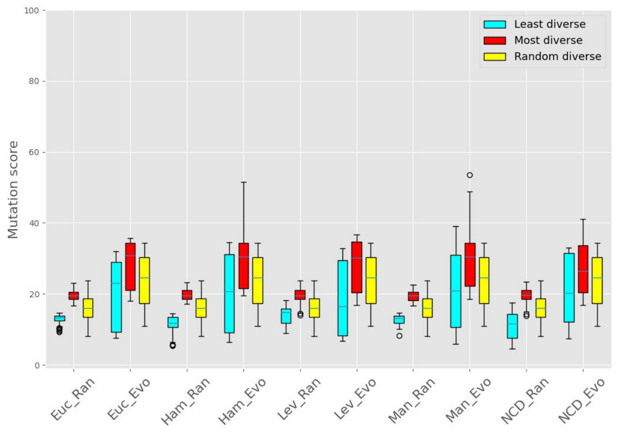
S11	Test diversity [236]	A hybrid measure calculating the difference between two test cases in terms of branches covered, variation of the data inputs, and standard deviation between conditions covered.	2	[236, 208]
S12	[Graph model diversity] Symmetric distance [281]	The difference between two models in a domain-specific language, where it is calculated as the number of "shapes" contained exclusively in one of the models but not both.	1	[281]
S13	Text uniqueness [18]	Text matching between two strings, where a string is unique if no other string matches it.	1	[18]
S14	Achieved coverage of pools [333]	The number of items selected from a pool of values for a program's variables over the time spent using that pool of values.	1	[333]
S15	Accuracy-based performance measure [343]	The proportion of correctly predicted test inputs to all the test inputs for a DNN.	1	[343]
S16	[Test behavioural similarity] Accuracy (acc) [223]	The percentage of tests that fail or pass together, calculated as the number of correct predictions divided by the total number of predictions in a confusion matrix.	1	[247]
S17	Average cyclomatic complexity per method (ACCM) [72]	Cyclomatic complexity is the number of independent paths in a program or method. ACCM is calculated by computing the number of independent paths within each method and then taking the sum, over all methods, of these values.	1	[260]
S18	Basic counting [240]	The overlapping occurrences of method calls between two failing sequences of method calls extracted from execution traces of tests.	1	[240]
S19	Code complexity (cm) [24]	Consists of three types of information (Lines of code, Nested Block Depth, and Cyclomatic Complexity) derived from the source code to measure similarity.	1	[24]

S20	[GUI similarity] $CONTeSSi(n)$ [42]	The differences of the frequencies between the past n executed events of one test suite to another test suite.	1	[42]
S21	Distance entropy [286]	The distribution of tests in a set represented in a graph using the minimum weight set (i.e. the set of vertices or edges in a weighted graph that collectively has the smallest sum of weights).	1	[286]
S22	Enhanced Jaro-Winkler [1]	A hybrid metric between Jaro-Winkler (G14) and Hamming Distance (G4) that considers the deselected features from Hamming distance combined into the Jaro distance equation.	1	[1]
S23	Graph edit distance [276]	The minimum number of edit operations required to make two graphs identical.	1	[299]
S24	Matthew's correlation coefficient (MCC) [216]	A more accurate measure of tests behavioural similarity than accuracy (S16) that accounts for both true positives and true negatives.	1	[247]
S25	Probabilistic type tree [261]	A tree structure to represent a probability distribution over the types.	1	[261]
S26	Response for class (RFC) [72]	The sum of the number of methods inside the class and the number of external methods used by the class.	1	[260]
S27	Syntax-tree similarity [211]	The structural similarity between two sentences represented as "syntax trees" by comparing the tree topologies, node positions, and the types of grammatical relationships.	1	[211]
S28	Traces [339]	The difference between two execution paths, that takes into account the branches covered and the number of times these branches were covered.	1	[264]
S29	Weighted distance function [31]	The number of statements covered by one test case but not the other, and the difference of the execution times between the two test cases.	1	[31]

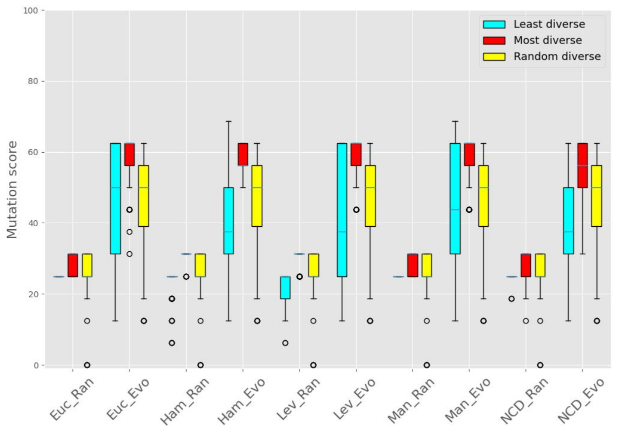
S30	Extensible access control markup language (XACML) Similarity [34]	The distance between the requests attributes' values of two XACML test cases and the difference between their policies.	1	[34]

Appendix B

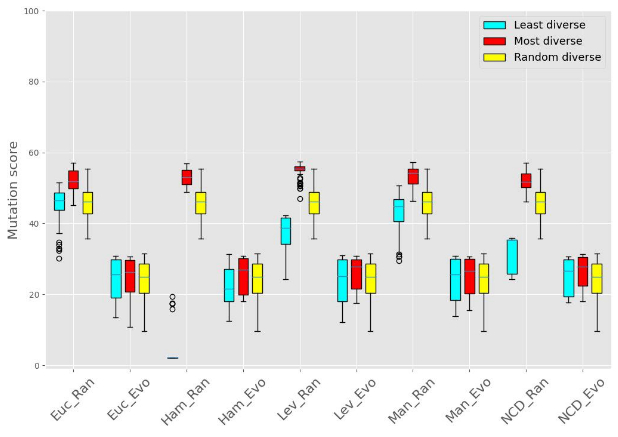
Evaluating String Distance Metrics for Reducing Automatically Generated Test Suites



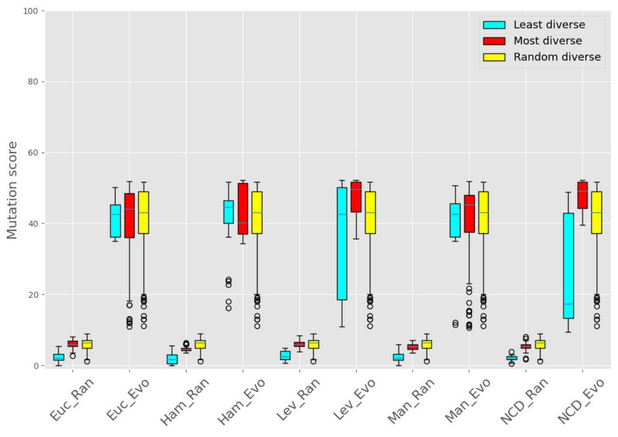
(a) Analysis on the “`Chart`” project.



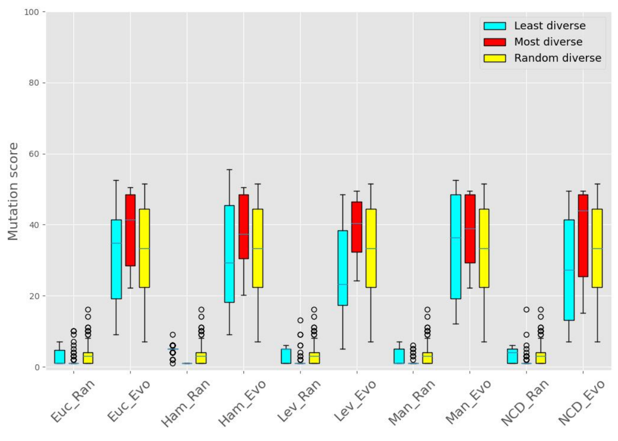
(b) Analysis on the “`Cli`” project.



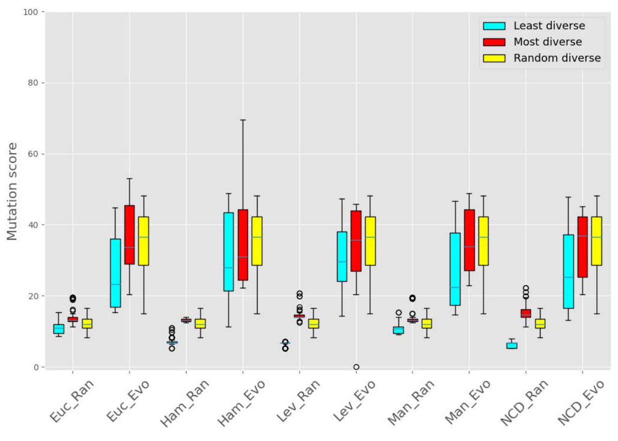
(c) Analysis on the “`Codec`” project.



(d) Analysis on the “`Compress`” project.

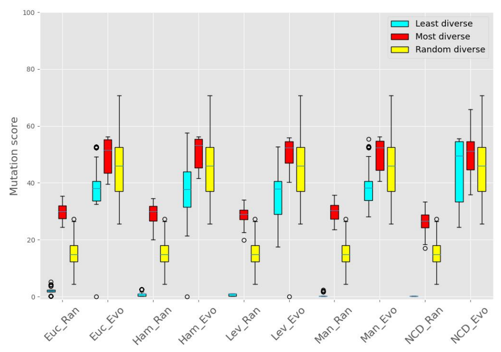


(e) Analysis on the “`Csv`” project.

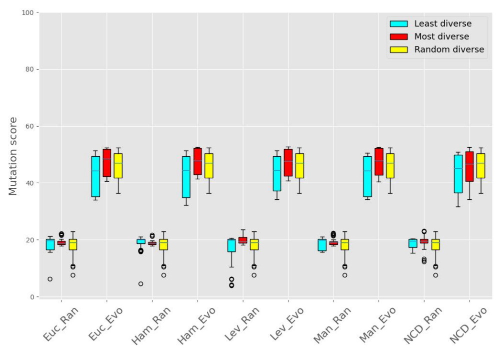


(f) Analysis on the “`Gson`” project.

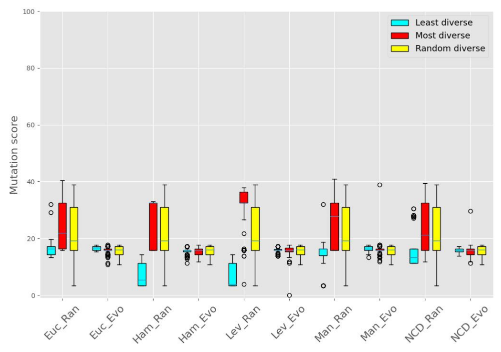
Fig. 1: The reduction analysis for “`Chart`”, “`Chart`”, “`Chart`”, “`Chart`”, “`Chart`”, and “`Gson`”.



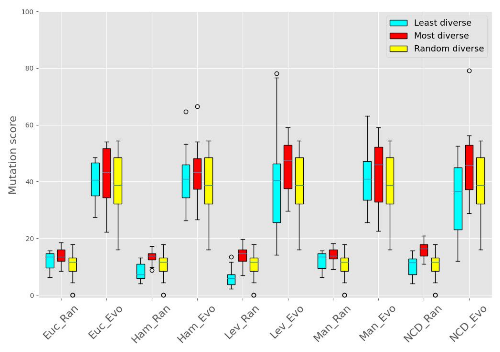
(a) Analysis on the “`JacksonCore`” project.



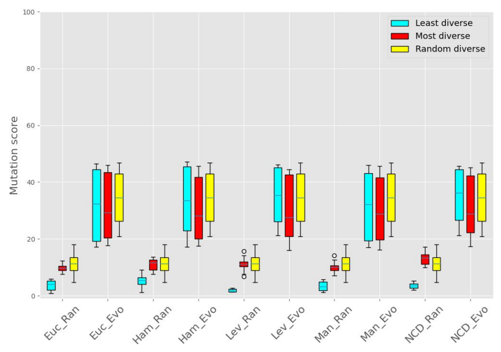
(b) Analysis on the “`JacksonDatabind`” project.



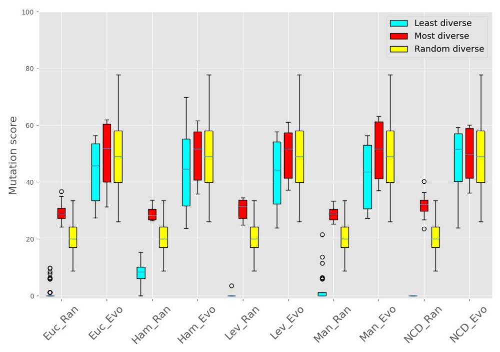
(c) Analysis on the “`Jsoup`” project.



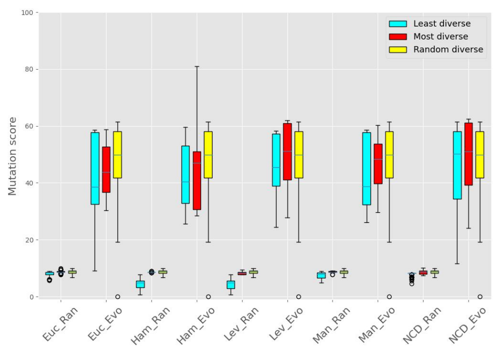
(d) Analysis on the “`JxPath`” project.



(e) Analysis on the “`Lang`” project.



(f) Analysis on the “`Math`” project.



(g) Analysis on the “`Time`” project.

Fig. 2: The reduction analysis for “`JacksonCore`”, “`JacksonDatabind`”, “`Jsoup`”, “`JxPath`”, “`Lang`”, “`Math`”, and “`Time`”.

TABLE I: The reduction analysis for the Chart project.

SM	TS	Least diverse				Most diverse				Random diverse				p-Value				Effect size			
		Rand		Evo		Rand		Evo		Rand		Evo		Rand		Evo		Rand		Evo	
		Avg	SD	Avg	SD	Avg	SD	Avg	SD	Avg	SD	Avg	SD	LR	MR	LR	MR	LR	MR	LR	MR
Euc	37	11.6	0.96	8.9	1.6	17.9	0.73	20.5	1.2	12.5	2.28	16.1	2.5	0.055	α	α	α	0.36	0.99	0.01	0.92
	63	13.2	0.58	22.8	1.13	20.0	0.73	30.8	0.34	16.5	2.00	24.1	2.18	α	α	α	α	0.06	0.96	0.26	1.00
	90	14.2	0.49	30.6	1.14	20.8	0.83	34.7	0.46	18.6	2.36	31.3	1.72	α	α	α	α	0.05	0.86	0.41	0.99
Ham	37	10.0	2.63	9.4	2.52	18.3	0.54	21.0	0.67	12.5	2.28	16.1	2.52	α	α	α	α	0.24	1.00	0.06	0.95
	63	12.0	0.58	20.9	1.13	19.7	0.73	30.8	0.34	16.5	2.00	24.1	2.18	α	α	α	α	0.04	0.94	0.11	1.00
	90	13.0	0.49	32.0	1.14	21.3	0.83	37.4	0.46	18.6	2.36	31.3	1.72	α	α	0.074	α	0.04	0.92	0.63	0.97
Lev	37	11.6	1.79	7.8	0.73	17.3	1.59	19.5	2.53	12.5	2.28	16.1	2.52	0.09	α	α	α	0.37	0.96	0.00	0.85
	63	14.2	1.76	17.3	3.28	19.8	1.17	30.6	2.99	16.5	2.00	24.1	2.18	α	α	α	α	0.19	0.93	0.05	0.99
	90	16.3	1.12	30.8	1.35	21.0	0.84	34.9	0.1	18.6	2.36	31.3	1.72	α	α	0.231	α	0.16	0.89	0.41	1.00
Man	37	11.4	0.93	8.8	2.03	18.0	0.57	21.6	1.08	12.5	2.28	16.1	2.52	0.09	α	α	α	0.32	1.00	0.02	0.97
	63	13.0	0.97	21.7	1.85	19.7	0.65	31.1	1.52	16.5	2.00	24.1	2.18	α	α	α	α	0.06	0.94	0.19	1.00
	90	14.1	0.51	32.0	1.83	20.9	0.77	37.2	4.66	18.6	2.36	31.3	1.72	α	α	0.193	α	0.05	0.88	0.60	0.96
NCD	37	7.5	2.42	10.2	2.02	17.6	1.41	19.5	1.41	12.5	2.28	16.1	2.52	0.09	α	α	α	0.08	0.97	0.03	0.87
	63	10.9	2.51	20.8	1.77	20.0	1.02	26.7	1.01	16.5	2.00	24.1	2.18	α	α	α	α	0.03	0.95	0.12	0.86
	90	15.0	2.24	31.9	1.11	21.3	1.08	34.5	1.35	18.6	2.36	31.3	1.72	α	α	0.108	α	0.11	0.90	0.62	0.94

TABLE II: The reduction analysis for the Cli project.

SM	TS	Least diverse				Most diverse				Random diverse				p-Value				Effect size			
		Rand		Evo		Rand		Evo		Rand		Evo		Rand		Evo		Rand		Evo	
		Avg	SD	Avg	SD	Avg	SD	Avg	SD	Avg	SD	Avg	SD	LR	MR	LR	MR	LR	MR	LR	MR
Euc	5	25.0	0.00	30.2	15.90	26.9	2.86	51.5	6.79	23.1	7.24	32.9	14.43	0.65	0.24	0.42	α	0.14	0.39	0.41	0.84
	8	25.0	0.00	40.4	14.32	29.8	2.64	60.6	2.86	24.2	7.35	49.2	10.17	0.18	α	0.01	α	0.14	0.69	0.30	0.86
	12	25.0	0.00	59.6	5.29	31.0	1.12	62.5	0.00	27.3	5.93	58.3	5.43	α	α	0.30	α	0.14	0.69	0.30	0.86
Ham	5	21.9	5.53	26.9	14.08	30.2	2.33	54.6	3.20	23.1	7.24	32.9	14.43	0.03	α	0.11	α	0.11	0.78	0.35	0.92
	8	22.1	5.53	37.5	8.54	31.3	0.00	59.0	3.10	24.2	7.35	49.2	10.17	α	α	α	α	0.11	0.81	0.18	0.79
	12	24.2	2.12	58.5	6.14	31.3	0.00	62.5	0.00	27.3	5.93	58.3	5.43	α	α	0.72	α	0.00	0.56	0.46	0.58
Lev	5	18.1	5.19	22.7	10.76	28.1	3.12	53.5	6.39	23.1	7.24	32.9	14.43	α	0.02	α	α	0.06	0.52	0.26	0.88
	8	21.3	4.45	40.8	12.15	30.0	2.50	61.0	2.64	24.2	7.35	49.2	10.17	α	α	α	α	0.09	0.70	0.27	0.87
	12	23.5	2.64	60.0	5.00	31.0	1.12	62.5	0.00	27.3	5.93	58.3	5.43	α	α	0.19	α	0.00	0.55	0.50	0.58
Man	5	25.0	0.00	28.1	14.77	28.1	3.12	50.0	6.04	23.1	7.24	32.9	14.43	0.65	0.02	0.18	α	0.14	0.52	0.37	0.80
	8	25.0	0.00	39.4	14.08	30.0	2.50	61.3	2.50	24.2	7.35	49.2	10.17	0.18	α	α	α	0.14	0.69	0.28	0.90
	12	25.0	0.00	60.6	4.88	30.6	1.88	62.5	0.00	27.3	5.93	58.3	5.43	α	0.02	0.09	α	0.00	0.52	0.53	0.58
NCD	5	24.4	1.88	25.8	10.42	25.8	5.53	46.7	8.50	23.1	7.24	32.9	14.43	0.34	0.63	0.07	α	0.13	0.45	0.33	0.76
	8	25.0	0.00	44.4	14.19	27.3	4.41	56.5	5.22	24.2	7.35	49.2	10.17	0.18	0.41	0.34	α	0.14	0.48	0.41	0.71
	12	25.0	0.00	50.8	7.86	31.0	1.12	62.5	0.00	27.3	5.93	58.3	5.43	α	α	α	α	0.00	0.55	0.20	0.58

TABLE III: The reduction analysis for the Codec project.

SM	TS	Least diverse				Most diverse				Random diverse				p-Value				Effect size			
		Rand		Evo		Rand		Evo		Rand		Evo		Rand		Evo		Rand		Evo	
		Avg	SD	Avg	SD	Avg	SD	Avg	SD	Avg	SD	Avg	SD	LR	MR	LR	MR	LR	MR	LR	MR
Euc	12	41.0	4.99	17.8	1.67	48.8	2.42	17.5	3.77	43.3	4.01	18.5	2.89	0.10	α	0.15	0.48	0.38	0.86	0.39	0.45
	22	46.6	2.70	25.2	1.47	52.8	2.07	25.9	1.55	46.3	3.34	24.8	2.46	0.73	α	0.59	0.14	0.52	0.97	0.54	0.61
	31	47.8	2.55	30.2	0.68	54.4	2.01	29.9	0.36	47.9	3.48	29.1	1.9	0.83	α	0.03	0.35	0.48	0.94	0.66	0.57
Ham	12	2.1	0.07	16.3	2.22	51.3	1.46	19.5	1.09	43.3	4.01	18.5	2.89	α	α	α	0.21	0.00	0.98	0.24	0.59
	22	2.6	2.47	22.5	1.98	52.6	1.74	26.4	1.56	46.3	3.34	24.8	2.46	α	α	α	0.02	0.00	0.97	0.26	0.68
	31	4.2	5.36	28.5	1.67	54.9	1.78	30.4	0.35	47.9	3.48	29.1	1.9	α	α	0.07	α	0.00	0.96	0.37	0.72
Lev	12	31.5	3.66	15.9	2.32	53.3	2.25	20.1	2.32	43.3	4.01	18.5	2.89	α	α	α	0.03	0.01	0.99	0.21	0.66
	22	38.9	3.28	24.8	2.40	55.5	0.47	27.6	0.61	46.3	3.34	24.8	2.46	α	α	0.93	α	0.06	1.00	0.50	0.86
	31	40.8	1.62	30.1	0.53	56.2	0.55	30.0	0.34	47.9	3.48	29.1	1.90	α	α	0.05	0.23	0.03	1.00	0.64	0.59
Man	12	37.0	5.00	17.7	1.20	50.3	2.38	19.3	1.63	43.3	4.01	18.5	2.89	α	α	0.05	0.34	0.19	0.92	0.36	0.57
	22	45.3	2.57	25.1	1.47	54.1	1.98	26.0	1.38	46.3	3.34	24.8	2.46	0.12	α	0.75	0.07	0.38	0.99	0.51	0.63
	31	47.5	2.26	30.2	0.55	55.6	0.82	30.3	0.30	47.9	3.48	29.1	1.90	0.48	α	0.03	α	0.44	0.98	0.67	0.69
NCD	12	37.0	4.16	17.7	0.68	50.3	2.14	19.3	1.62	43.3	4.01	18.5	2.89	α	α	0.91	α	0.00	0.94	0.49	0.77
	22	45.3	2.80	25.1	2.42	54.1	2.11	26.0	1.12	46.3	3.34	24.8	2.46	α	α	0.11	α	0.00	0.97	0.62	0.80
	31	47.5	0.54	30.2	1.27	55.6	1.87	30.3	0.27	47.9	3.48	29.1	1.90	α	α	0.27	α	0.00	0.90	0.58	0.81

TABLE IV: The reduction analysis for the JacksonCore project.

SM	TS	Least diverse				Most diverse				Random diverse				p-Value				Effect size			
		Rand		Evo		Rand		Evo		Rand		Evo		Rand		Evo		Rand		Evo	
		Avg	SD	Avg	SD	Avg	SD	Avg	SD	Avg	SD	Avg	SD	LR	MR	LR	MR	LR	MR	LR	MR
	18	0.9	0.95	33.4	0.52	26.7	1.55	42.4	1.44	11.4	3.23	35.2	3.89	α	α	0.01	α	0.00	1.00	0.31	0.99
Euc	31	2.1	0.74	38.0	1.03	29.7	1.29	51.4	1.26	15.9	3.21	45.3	4.23	α	α	α	α	0.00	1.00	0.07	0.95
	45	2.7	0.94	46.8	9.84	33.0	1.13	55.6	0.42	18.4	4.33	53.6	3.88	α	α	α	α	0.00	1.00	0.12	0.92
	18	0.7	0.65	29.2	5.75	24.9	2.55	44.5	1.40	11.4	3.23	35.2	3.89	α	α	α	α	0.00	1.00	0.14	1.00
Ham	31	1.0	0.60	37.9	3.74	29.7	1.64	52.8	0.96	15.9	3.21	45.3	4.23	α	α	α	α	0.00	1.00	0.09	0.99
	45	1.5	0.94	45.9	9.79	32.6	1.02	55.6	0.30	18.4	4.33	53.6	3.88	α	α	α	α	0.00	1.00	0.11	0.94
	18	0.5	0.40	25.7	3.77	26.1	2.15	45.6	2.09	11.4	3.23	35.2	3.89	α	α	α	α	0.00	1.00	0.04	0.99
Lev	31	0.6	0.41	37.7	2.13	29.4	1.33	50.4	9.39	15.9	3.21	45.3	4.23	α	α	α	α	0.00	1.00	0.11	0.94
	45	0.9	0.33	44.4	4.17	30.5	1.45	54.9	0.58	18.4	4.33	53.6	3.88	α	α	α	α	0.00	1.00	0.04	0.74
	18	0.2	0.00	32.7	1.69	26.5	1.36	43.3	1.46	11.4	3.23	35.2	3.89	α	α	α	α	0.00	1.00	0.28	0.99
Man	31	0.2	0.00	38.3	0.96	30.2	1.48	52.4	0.76	15.9	3.21	45.3	4.23	α	α	α	α	0.00	1.00	0.08	0.98
	45	0.6	0.71	47.8	5.12	32.8	1.01	55.2	0.50	18.4	4.33	53.6	3.88	α	α	α	α	0.00	1.00	0.14	0.84
	18	0.2	0.00	31.3	2.42	23.4	2.92	43.7	4.63	11.4	3.23	35.2	3.89	α	α	α	α	0.00	1.00	0.20	0.97
NCD	31	0.2	0.00	48.2	3.69	26.6	2.73	51.1	3.03	15.9	3.21	45.3	4.23	α	α	α	α	0.00	0.99	0.72	0.91
	45	0.2	0.00	54.8	0.37	29.2	2.30	54.6	0.76	18.4	4.33	53.6	3.88	α	α	α	0.01	0.00	0.99	0.73	0.67

TABLE V: The reduction analysis for the JxPath project.

SM	TS	Least diverse				Most diverse				Random diverse				p-Value				Effect size			
		Rand		Evo		Rand		Evo		Rand		Evo		Rand		Evo		Rand		Evo	
		Avg	SD	Avg	SD	Avg	SD	Avg	SD	Avg	SD	Avg	SD	LR	MR	LR	MR	LR	MR	LR	MR
	19	9.6	2.40	34.0	2.54	11.6	1.48	31.8	4.30	9.1	3.31	26.6	5.68	0.77	α	α	α	0.52	0.69	0.88	0.75
Euc	33	12.9	2.10	40.6	0.75	13.9	1.75	43.3	2.84	10.8	3.23	38.8	3.99	α	α	0.14	α	0.73	0.77	0.61	0.80
	47	14.4	1.25	47.1	0.48	16.0	1.74	52.2	0.67	12.9	2.69	50.8	2.78	0.05	α	α	0.17	0.65	0.81	0.14	0.59
	19	6.0	1.87	33.1	4.73	11.6	1.22	34.0	4.87	9.1	3.31	26.6	5.68	α	0.02	α	α	0.14	0.68	0.80	0.83
Ham	33	8.4	2.50	41.6	1.15	13.5	4.43	44.4	4.88	10.8	3.23	38.8	3.99	α	α	0.06	α	0.17	0.78	0.64	0.81
	47	10.3	2.33	46.5	0.64	14.8	0.57	50.1	2.45	12.9	2.69	50.8	2.78	α	0.01	α	0.39	0.24	0.69	0.11	0.44
	19	3.3	1.01	22.6	3.70	11.5	1.71	35.4	3.33	9.1	3.31	26.6	5.68	α	0.01	α	α	0.01	0.69	0.29	0.88
Lev	33	6.3	1.79	41.0	4.09	14.7	2.09	47.4	3.49	10.8	3.23	38.8	3.99	α	α	0.09	α	0.08	0.85	0.63	0.96
	47	8.2	1.68	51.8	9.12	16.5	1.71	53.6	1.25	12.9	2.69	50.8	2.78	α	α	0.16	α	0.07	0.86	0.39	0.85
	19	9.3	2.47	32.3	3.88	11.9	1.43	30.8	5.92	9.1	3.31	26.6	5.68	0.82	α	α	α	0.48	0.72	0.79	0.69
Man	33	12.5	2.16	42.8	4.74	14.5	1.46	45.0	4.14	10.8	3.23	38.8	3.99	0.03	α	0.01	α	0.68	0.87	0.69	0.85
	47	14.7	0.33	50.2	4.89	16.5	1.49	52.7	0.70	12.9	2.69	50.8	2.78	0.02	α	0.12	α	0.67	0.87	0.37	0.72
	19	7.5	2.74	21.2	3.45	13.5	1.68	35.4	3.02	9.1	3.31	26.6	5.68	0.01	α	α	α	0.30	0.87	0.22	0.92
NCD	33	10.6	2.19	36.5	2.20	16.2	1.60	46.2	6.86	10.8	3.23	38.8	3.99	0.14	α	0.02	α	0.38	0.94	0.31	0.88
	47	12.9	0.77	45.7	1.47	18.3	1.42	53.7	1.10	12.9	2.69	50.8	2.78	0.86	α	α	α	0.48	0.96	0.08	0.83

TABLE VI: The reduction analysis for the Math project.

SM	TS	Least diverse				Most diverse				Random diverse				p-Value				Effect size			
		Rand		Evo		Rand		Evo		Rand		Evo		Rand		Evo		Rand		Evo	
		Avg	SD	Avg	SD	Avg	SD	Avg	SD	Avg	SD	Avg	SD	LR	MR	LR	MR	LR	MR	LR	MR
	39	0.0	0.00	30.9	2.74	26.5	0.97	38.0	3.17	16.2	3.60	37.1	3.88	α	α	α	0.40	0.00	1.00	0.11	0.56
Euc	67	0.6	2.02	45.3	1.34	29.2	1.14	52.0	1.62	21.0	4.91	49.8	6.20	α	α	α	α	0.00	0.93	0.15	0.76
	95	2.2	3.03	54.5	1.25	31.5	1.69	60.9	0.52	25.3	3.70	56.0	2.89	α	α	α	α	0.00	0.89	0.05	0.90
	39	6.2	3.02	29.9	2.47	26.8	0.16	39.3	1.67	16.2	3.60	37.1	3.88	α	α	α	0.03	0.01	1.00	0.07	0.66
Ham	67	8.3	2.24	44.6	1.34	28.3	0.20	51.8	1.73	21.0	4.91	49.8	6.20	α	α	α	α	0.00	0.93	0.13	0.76
	95	9.3	2.60	56.0	2.64	31.7	1.24	58.6	1.18	25.3	3.70	59.0	2.89	α	α	α	0.23	0.00	0.92	0.10	0.40
	39	0.0	0.00	30.0	2.97	26.6	0.95	40.0	1.61	16.2	3.60	37.1	3.88	α	α	α	α	0.00	1.00	0.09	0.74
Lev	67	0.0	0.00	44.2	1.95	31.4	0.93	51.7	1.53	21.0	4.91	49.8	6.20	α	α	α	α	0.00	0.93	0.12	0.76
	95	0.1	0.65	55.7	1.40	33.8	0.34	59.3	1.48	25.3	3.70	59.0	2.89	α	α	α	0.15	0.00	0.99	0.08	0.61
	39	0.0	0.00	29.4	2.29	26.5	1.37	40.2	2.32	16.2	3.60	37.1	3.88	α	α	α	α	0.00	1.00	0.07	0.73
Man	67	0.84	2.30	43.7	0.99	29.0	0.84	51.5	2.13	21.0	4.91	49.8	6.20	α	α	α	α	0.00	0.93	0.10	0.72
	95	2.5	4.57	54.6	1.39	31.2	0.94	61.3	0.95	25.3	3.70	59.0	2.89	α	α	α	α	0.00	0.89	0.06	0.91
	39	0.0	0.00	36.4	5.73	28.8	2.10	40.0	2.20	16.2	3.60	37.1	3.88	α	α	α	α	0.00	1.00	0.07	0.73
NCD	67	0.00	0.00	51.6	1.24	31.9	1.30	50.1	1.52	21.0	4.91	49.8	6.20	α	α	α	α	0.00	0.93	0.10	0.72
	95	0.0	0.00	57.6	0.76	34.3	1.50	59.1	0.32	25.3	3.70	59.0	2.89	α	α	α	α	0.00	0.89	0.06	0.91

TABLE VII: The reduction analysis for the Compress project.

SM	**TS**	**Least diverse**				**Most diverse**				**Random diverse**				**p-Value**				**Effect size**			
		Rand		Evo		Rand		Evo		Rand		Evo		Rand		Evo		Rand		Evo	
		Avg	SD	Avg	SD	Avg	SD	Avg	SD	Avg	SD	Avg	SD	LR	MR	LR	MR	LR	MR	LR	MR
Euc	7	1.4	0.93	35.8	1.20	5.1	0.85	26.6	11.21	4.4	1.63	31.9	10.82	α	0.23	0.44	0.07	0.05	0.59	0.44	0.36
	13	2.6	1.41	41.9	2.01	6.1	0.76	44.9	3.68	6.5	1.41	41.4	8.52	α	0.03	0.17	0.18	0.04	0.34	0.40	0.60
	18	3.2	1.18	47.2	2.24	6.6	0.84	48.6	3.05	6.7	1.63	48.4	10.82	α	0.39	0.01	0.32	0.03	0.42	0.31	0.57
Ham	7	0.7	0.75	36.7	7.54	4.3	0.57	36.3	1.19	4.4	1.63	31.9	10.82	α	0.43	0.05	0.96	0.01	0.43	0.64	0.50
	13	2.2	1.30	44.4	3.11	4.7	0.43	40.9	1.93	6.5	1.41	41.4	8.52	α	α	0.38	0.02	0.02	0.10	0.56	0.33
	18	2.9	1.34	47.7	2.57	4.9	0.64	51.7	0.21	6.7	1.20	48.4	3.06	α	α	0.19	α	0.03	0.10	0.39	0.96
Lev	7	2.0	0.68	16.6	6.01	5.0	0.56	41.1	2.56	4.4	1.63	31.9	10.82	α	0.28	α	α	0.08	0.57	0.15	0.79
	13	3.0	0.98	36.3	12.16	6.1	0.70	48.7	2.36	6.5	1.41	41.4	8.52	α	0.01	0.27	α	0.04	0.30	0.41	0.89
	18	3.5	0.90	48.9	3.49	6.8	0.62	51.7	0.16	6.7	1.20	48.4	3.06	α	0.64	0.07	α	0.04	0.45	0.63	0.98
Man	7	1.8	0.98	34.2	6.16	4.7	0.62	28.54	12.05	4.4	1.63	31.9	10.82	α	0.73	0.29	0.23	0.08	0.51	0.42	0.41
	13	2.7	1.20	42.4	1.64	5.4	0.67	44.7	3.65	6.5	1.41	41.4	8.52	α	α	0.28	0.17	0.03	0.19	0.42	0.60
	18	3.0	1.53	47.8	2.37	5.6	0.71	48.7	3.02	6.7	1.20	48.4	3.06	α	α	0.11	0.23	0.03	0.18	0.37	0.58
NCD	7	1.4	0.41	13.1	4.48	4.4	0.91	42.8	2.34	4.4	1.63	31.9	10.82	α	0.71	α	α	0.02	0.47	0.06	0.87
	13	2.1	0.58	25.2	11.92	5.4	0.57	49.3	0.42	6.5	1.41	41.4	8.52	α	α	α	α	0.00	0.17	0.13	0.92
	18	2.4	0.32	42.8	5.55	6.0	0.84	51.7	0.21	6.7	1.20	48.4	3.06	α	α	α	α	0.00	0.26	0.12	0.97

TABLE VIII: The reduction analysis for the Csv project.

SM	**TS**	**Least diverse**				**Most diverse**				**Random diverse**				**p-Value**				**Effect size**			
		Rand		Evo		Rand		Evo		Rand		Evo		Rand		Evo		Rand		Evo	
		Avg	SD	Avg	SD	Avg	SD	Avg	SD	Avg	SD	Avg	SD	LR	MR	LR	MR	LR	MR	LR	MR
Euc	7	2.2	2.08	16.8	2.98	2.0	2.09	26.2	3.24	2.4	2.19	18.2	6.87	0.40	0.24	0.55	α	0.24	0.22	0.45	0.84
	13	1.8	1.57	33.1	5.23	1.5	1.69	41.3	3.04	3.6	2.91	33.4	6.46	α	α	0.72	α	0.17	0.09	0.52	0.84
	18	3.1	2.42	46.5	4.76	2.0	2.32	49.0	0.57	5.1	3.67	45.2	3.75	0.04	α	0.18	α	0.32	0.12	0.58	0.83
Ham	7	4.6	1.27	16.0	3.15	1.0	0.00	28.1	3.26	2.4	2.19	18.2	6.87	α	α	0.26	α	0.83	0.00	0.41	0.88
	13	5.0	0.73	30.5	5.55	1.0	0.00	38.5	3.50	3.6	2.91	33.4	6.46	α	α	0.08	α	0.80	0.00	0.37	0.75
	18	5.3	0.88	48.8	4.60	1.0	0.00	49.1	0.56	5.1	3.67	45.2	3.75	0.02	α	α	α	0.65	0.00	0.75	0.84
Lev	7	2.1	1.75	14.4	4.32	1.0	1.50	29.3	2.89	2.4	2.19	18.2	6.87	0.49	0.02	0.04	α	0.27	0.10	0.34	0.91
	13	4.0	2.02	23.7	3.10	1.0	1.72	39.6	2.09	3.6	2.91	33.4	6.46	0.15	α	α	α	0.57	0.09	0.07	0.81
	18	4.1	2.14	40.1	2.98	1.4	1.51	48.0	1.16	5.1	3.67	45.2	3.75	0.97	α	α	α	0.48	0.05	0.16	0.73
Man	7	2.4	1.98	17.3	2.20	1.5	1.16	27.1	3.08	2.4	2.19	18.2	6.87	0.87	0.05	0.77	α	0.32	0.13	0.48	0.87
	13	2.9	2.35	36.1	3.72	1.3	0.95	39.9	3.51	3.6	2.91	33.4	6.46	0.27	α	α	α	0.32	0.07	0.67	0.82
	18	3.1	2.16	49.3	1.97	1.4	0.96	49.0	0.50	5.1	3.67	45.2	3.75	0.06	α	α	α	0.34	0.04	0.84	0.83
NCD	7	2.0	1.38	11.3	2.64	2.0	3.06	23.3	4.53	2.4	2.19	18.2	6.87	0.60	0.07	α	α	0.28	0.14	0.18	0.73
	13	3.1	1.83	27.4	2.53	1.3	0.56	44.0	2.50	3.6	2.91	33.4	6.46	0.95	α	α	α	0.44	0.07	0.20	0.90
	18	4.4	1.15	44.8	2.98	1.5	1.22	49.1	0.49	5.1	3.67	45.2	3.75	0.33	α	0.51	α	0.55	0.08	0.44	0.85

TABLE IX: The reduction analysis for the Gson project.

SM	**TS**	**Least diverse**				**Most diverse**				**Random diverse**				**p-Value**				**Effect size**			
		Rand		Evo		Rand		Evo		Rand		Evo		Rand		Evo		Rand		Evo	
		Avg	SD	Avg	SD	Avg	SD	Avg	SD	Avg	SD	Avg	SD	LR	MR	LR	MR	LR	MR	LR	MR
Euc	23	9.7	0.84	16.8	1.17	13.1	1.68	25.4	3.53	11.4	1.73	25.5	4.48	α	α	α	0.77	0.16	0.81	0.04	0.47
	39	10.7	1.27	23.9	1.65	14.3	2.11	33.9	2.80	12.3	1.47	35.2	3.81	α	α	α	0.09	0.18	0.77	0.00	0.36
	56	12.6	1.37	39.1	3.44	14.3	1.83	46.2	1.67	13.4	1.42	44.0	2.88	0.04	0.09	α	α	0.33	0.60	0.13	0.72
Ham	23	6.5	0.39	19.1	3.99	13.0	0.25	23.9	1.00	11.4	1.73	25.5	4.48	α	α	α	0.21	0.00	0.84	0.15	0.40
	39	10.0	0.51	26.5	3.21	13.1	0.33	30.7	1.18	12.3	1.47	35.2	3.81	α	0.02	α	α	0.00	0.68	0.04	0.18
	56	8.0	1.21	44.1	3.54	13.4	0.24	45.7	4.50	13.4	1.42	44.0	2.88	α	0.91	0.36	0.52	0.00	0.50	0.56	0.54
Lev	23	5.8	0.73	22.0	3.48	14.1	1.63	25.1	3.36	11.4	1.73	25.5	4.48	α	α	α	0.62	0.00	0.89	0.30	0.46
	39	6.7	0.27	29.1	2.58	14.2	0.47	36.3	1.52	12.3	1.47	35.2	3.81	α	α	α	0.71	0.00	0.87	0.13	0.53
	56	6.9	0.81	42.3	3.73	9.0	1.66	43.4	8.08	13.4	1.42	44.0	2.88	α	α	0.04	0.97	0.00	0.79	0.35	0.49
Man	23	9.6	0.58	16.7	0.97	13.6	1.98	26.1	2.61	11.4	1.73	25.5	4.48	α	α	α	0.53	0.11	0.85	0.03	0.54
	39	10.6	0.96	23.2	1.67	13.9	1.91	34.1	3.78	12.3	1.47	35.2	3.81	α	α	α	0.34	0.16	0.73	0.00	0.42
	56	12.1	1.32	41.4	3.48	13.6	1.12	45.3	1.12	13.4	1.42	44.0	2.88	α	0.81	α	0.19	0.23	0.51	0.25	0.59
NCD	23	5.3	0.28	15.9	1.77	14.9	2.56	24.1	3.18	11.4	1.73	25.5	4.48	α	α	α	0.17	0.00	0.89	0.03	0.39
	39	6.6	0.30	25.9	3.31	15.6	2.14	36.9	1.03	12.3	1.47	35.2	3.81	α	α	α	0.44	0.00	0.93	0.05	0.55
	56	6.8	0.20	42.0	4.15	15.8	2.13	43.1	1.16	13.4	1.42	44.0	2.88	α	α	0.06	α	0.00	0.84	0.35	0.30

TABLE X: The reduction analysis for the JacksonDatabind project.

SM	**TS**	**Least diverse**				**Most diverse**				**Random diverse**				**p-Value**				**Effect size**			
		Rand		Evo		Rand		Evo		Rand		Evo		Rand		Evo		Rand		Evo	
		Avg	SD	Avg	SD	Avg	SD	Avg	SD	Avg	SD	Avg	SD	LR	MR	LR	MR	LR	MR	LR	MR
	44	16.5	2.32	35.3	0.78	18.2	0.21	42.0	0.94	15.1	2.26	39.7	1.65	0.24	α	α	α	0.58	0.80	0.02	0.80
Euc	76	19.4	1.08	44.6	0.65	19.1	0.57	48.4	0.56	18.7	3.23	46.8	3.89	0.51	0.67	α	α	0.55	0.46	0.09	0.82
	108	20.5	0.39	49.6	0.57	20.1	1.05	52.0	0.20	20.2	1.36	50.9	0.93	0.11	0.52	α	α	0.60	0.43	0.12	0.90
	44	16.8	2.64	34.5	1.02	18.2	0.17	42.6	0.79	15.1	2.26	39.7	1.65	0.05	α	α	α	0.64	0.80	0.01	0.90
Ham	76	19.8	0.83	44.7	0.71	19.0	0.63	47.7	0.16	18.7	3.23	46.8	3.89	0.09	0.57	α	α	0.62	0.46	0.10	0.71
	108	20.5	0.23	49.6	0.50	19.3	0.48	52.3	0.13	20.2	1.36	50.9	0.93	0.04	α	α	α	0.65	0.19	0.12	0.95
	44	13.0	4.45	35.9	1.21	18.8	0.77	42.0	0.75	15.1	2.26	39.7	1.65	0.02	α	α	α	0.33	0.85	0.09	0.81
Lev	76	19.6	0.94	44.8	0.79	19.3	0.48	47.4	0.80	18.7	3.23	46.8	3.89	0.26	0.98	α	0.14	0.58	0.50	0.11	0.61
	108	20.3	0.17	49.6	0.54	21.6	1.19	52.1	0.23	20.2	1.36	50.9	0.93	0.27	α	α	α	0.57	0.76	0.13	0.92
	44	16.6	1.24	35.2	0.87	18.2	0.19	42.1	0.93	15.1	2.26	39.7	1.65	0.28	α	α	α	0.57	0.80	0.03	0.83
Man	76	19.3	1.17	44.5	0.54	19.4	0.88	47.9	0.43	18.7	3.23	46.8	3.89	0.58	0.74	α	α	0.54	0.52	0.08	0.73
	108	20.3	0.31	49.6	0.51	19.9	1.00	52.2	0.14	20.2	1.36	50.9	0.93	0.28	0.09	α	α	0.56	0.34	0.12	0.95
	44	16.8	1.38	34.8	1.76	17.9	1.88	38.8	1.97	15.1	2.26	39.7	1.65	0.17	α	α	0.10	0.60	0.78	0.05	0.38
NCD	76	19.3	0.99	45.2	1.31	19.6	0.83	46.8	1.33	18.7	3.23	46.8	3.89	0.55	0.36	α	0.89	0.54	0.56	0.21	0.49
	108	20.3	0.06	50.3	0.45	20.8	0.97	51.3	0.59	20.2	1.36	50.9	0.93	0.30	0.02	α	0.10	0.56	0.66	0.26	0.61

TABLE XI: The reduction analysis for the Jsoup project.

SM	**TS**	**Least diverse**				**Most diverse**				**Random diverse**				**p-Value**				**Effect size**			
		Rand		Evo		Rand		Evo		Rand		Evo		Rand		Evo		Rand		Evo	
		Avg	SD	Avg	SD	Avg	SD	Avg	SD	Avg	SD	Avg	SD	LR	MR	LR	MR	LR	MR	LR	MR
	6	15.3	1.44	15.5	0.33	21.4	7.48	14.6	1.36	19.4	9.02	14.2	1.39	0.19	0.16	α	0.37	0.39	0.60	0.81	0.55
Euc	11	16.4	2.70	16.5	0.67	24.1	8.97	16.0	0.61	20.4	7.19	16.2	1.04	0.10	0.05	0.52	0.32	0.37	0.64	0.50	0.36
	16	17.6	3.10	17.3	0.15	28.5	7.33	16.6	0.74	25.5	7.86	17.0	0.87	α	0.06	0.63	α	0.22	0.63	0.44	0.27
	6	5.1	3.16	14.3	1.07	16.9	4.18	14.3	1.06	19.4	9.02	14.2	1.39	α	0.64	0.98	0.25	0.03	0.45	0.49	0.56
Ham	11	7.8	3.91	15.9	0.50	32.5	0.20	16.1	0.66	20.4	7.19	16.2	1.04	α	α	0.07	0.51	0.01	0.91	0.26	0.38
	16	10.4	3.54	16.4	0.68	32.5	0.09	16.9	0.79	25.5	7.86	17.0	0.87	α	α	α	0.28	0.00	0.78	0.21	0.40
	6	5.8	3.89	15.1	0.68	24.7	8.99	14.3	1.29	19.4	9.02	14.2	1.39	α	0.23	α	0.51	0.05	0.67	0.72	0.52
Lev	11	6.3	3.87	15.9	0.38	33.6	3.82	16.0	0.75	20.4	7.19	16.2	1.04	α	α	0.09	0.23	0.01	0.93	0.28	0.35
	16	10.7	3.23	16.3	0.50	35.1	1.72	16.4	3.15	25.5	7.86	17.0	0.87	α	α	α	0.60	0.00	0.89	0.19	0.44
	6	12.5	4.33	15.6	0.67	18.3	5.03	14.9	1.14	19.4	9.02	14.2	1.39	α	0.73	α	0.02	0.27	0.52	0.81	0.67
Man	11	15.9	1.47	16.2	0.64	23.2	7.68	15.9	0.88	20.4	7.19	16.2	1.04	0.12	0.10	0.71	0.22	0.38	0.61	0.42	0.33
	16	16.6	3.09	17.3	0.15	34.5	2.47	17.4	4.06	25.5	7.86	17.0	0.87	α	α	0.63	0.11	0.15	0.85	0.44	0.32
	6	12.1	2.96	14.7	0.65	17.0	4.77	14.3	1.11	19.4	9.02	14.2	1.39	α	0.52	0.08	0.68	0.14	0.45	0.57	0.51
NCD	11	15.3	5.93	15.8	0.25	26.4	8.51	15.8	0.97	20.4	7.19	16.2	1.04	α	α	0.05	0.08	0.19	0.71	0.27	0.32
	16	21.0	7.54	16.3	0.54	29.9	6.97	17.1	2.43	25.5	7.86	17.0	0.87	α	0.01	α	0.15	0.28	0.69	0.18	0.34

TABLE XII: The reduction analysis for the Lang project.

SM	**TS**	**Least diverse**				**Most diverse**				**Random diverse**				**p-Value**				**Effect size**			
		Rand		Evo		Rand		Evo		Rand		Evo		Rand		Evo		Rand		Evo	
		Avg	SD	Avg	SD	Avg	SD	Avg	SD	Avg	SD	Avg	SD	LR	MR	LR	MR	LR	MR	LR	MR
	43	2.1	1.13	18.8	0.60	8.7	0.73	19.6	1.03	7.9	1.74	24.5	2.24	α	0.12	α	α	0.00	0.62	0.00	0.00
Euc	73	3.6	1.17	32.4	1.75	9.9	0.67	29.6	1.29	11.4	1.90	34.7	2.73	α	α	α	α	0.00	0.21	0.26	0.03
	104	5.2	0.82	44.6	1.07	10.8	0.56	44.2	0.85	13.8	1.99	55.1	1.48	α	α	0.15	0.79	0.00	0.06	0.60	0.48
	43	4.0	1.47	21.1	2.42	8.9	0.73	19.4	1.09	7.9	1.74	24.5	2.24	α	0.05	α	α	0.05	0.65	0.15	0.01
Ham	73	5.5	1.54	33.8	2.03	11.8	0.86	28.3	0.63	11.4	1.90	34.7	2.73	α	0.70	0.34	α	0.01	0.53	0.43	0.00
	104	6.4	1.77	45.8	0.59	12.6	0.92	42.9	1.74	13.8	1.99	55.1	1.48	α	α	α	0.01	0.00	0.28	0.88	0.31
	43	1.3	0.12	25.4	1.37	9.5	1.18	19.7	1.73	7.9	1.74	24.5	2.24	α	α	0.16	α	0.00	0.76	0.60	0.04
Lev	73	2.0	0.23	35.4	0.76	11.1	0.86	27.6	1.58	11.4	1.90	34.7	2.73	α	0.16	0.13	α	0.00	0.39	0.62	0.00
	104	2.4	0.12	45.5	0.38	12.3	1.06	43.1	0.64	13.8	1.99	55.1	1.48	α	α	α	α	0.00	0.25	0.82	0.26
	43	1.9	0.68	18.9	0.91	8.5	0.76	18.7	1.23	7.9	1.74	24.5	2.24	α	0.33	α	α	0.00	0.57	0.00	0.00
Man	73	3.0	1.13	32.8	1.88	10.3	0.77	29.0	1.16	11.4	1.90	34.7	2.73	α	α	α	α	0.00	0.26	0.28	0.01
	104	4.8	0.95	44.2	1.07	10.8	0.70	42.8	1.48	13.8	1.99	55.1	1.48	α	α	0.81	α	0.00	0.07	0.52	0.27
	43	2.7	0.22	24.9	1.95	10.9	0.82	20.7	2.09	7.9	1.74	24.5	2.24	α	α	0.60	α	0.00	0.94	0.54	0.12
NCD	73	3.6	0.43	36.1	0.70	13.5	1.45	28.9	1.22	11.4	1.90	34.7	2.73	α	α	α	α	0.00	0.81	0.70	0.00
	104	4.4	0.45	44.9	0.41	14.9	1.16	43.1	0.92	13.8	1.99	55.1	1.48	α	0.03	0.03	α	0.00	0.66	0.66	0.28

TABLE XIII: The reduction analysis for the Time project.

SM	**TS**	**Least diverse**				**Most diverse**				**Random diverse**				**p-Value**				**Effect size**			
		Rand		Evo		Rand		Evo		Rand		Evo		Rand		Evo		Rand		Evo	
		Avg	SD	Avg	SD	Avg	SD	Avg	SD	Avg	SD	Avg	SD	LR	MR	LR	MR	LR	MR	LR	MR
Euc	26	7.2	0.74	28.9	4.67	8.5	0.42	35.9	4.71	8.3	0.71	36.5	9.92	α	0.24	α	0.07	0.12	0.58	0.15	0.36
	45	8.3	0.43	39.7	2.78	9.0	0.29	43.0	4.18	8.6	0.60	49.1	4.24	0.02	0.04	α	α	0.32	0.64	0.04	0.15
	63	8.7	0.22	58.1	0.56	8.9	0.27	53.4	1.45	9.1	0.43	58.9	1.64	α	α	α	α	0.18	0.26	0.27	0.02
Ham	26	2.5	1.06	31.3	3.18	8.6	0.11	30.5	2.17	8.3	0.71	36.5	9.92	α	0.03	α	α	0.00	0.66	0.19	0.18
	45	4.7	1.19	40.4	2.01	8.7	0.00	47.3	1.04	8.6	0.60	49.1	4.24	α	0.66	α	0.06	0.00	0.43	0.02	0.35
	63	5.8	0.96	54.7	2.55	8.7	0.09	54.5	6.02	9.1	0.43	58.9	1.64	α	α	α	α	0.00	0.17	0.09	0.12
Lev	26	2.5	1.41	36.6	3.39	8.0	0.31	36.2	5.23	8.3	0.71	36.5	9.92	α	0.02	0.19	0.19	0.00	0.30	0.40	0.40
	45	5.1	1.45	45.4	1.64	8.3	0.42	51.9	2.29	8.6	0.60	49.1	4.24	α	0.01	α	0.02	0.01	0.31	0.24	0.67
	63	5.9	1.30	57.5	0.79	8.6	0.46	61.0	0.43	9.1	0.43	58.9	1.64	α	α	α	α	0.00	0.21	0.18	0.93
Man	26	6.4	0.82	29.7	3.54	8.6	0.30	35.6	5.01	8.3	0.71	36.5	9.92	α	0.04	α	0.08	0.04	0.64	0.16	0.37
	45	7.6	0.97	40.7	3.75	8.7	0.12	47.9	1.95	8.6	0.60	49.1	4.24	α	0.95	α	0.11	0.17	0.48	0.08	0.38
	63	8.6	0.20	58.1	0.60	8.8	0.15	55.9	2.63	9.1	0.43	58.9	1.64	α	α	α	α	0.13	0.20	0.27	0.18
NCD	26	7.4	0.91	26.2	9.33	8.2	0.65	36.0	5.18	8.3	0.71	36.5	9.92	α	0.34	α	0.17	0.21	0.40	0.15	0.40
	45	8.1	0.44	49.2	2.51	8.6	0.57	53.0	2.80	8.6	0.60	49.1	4.24	α	0.67	0.87	α	0.22	0.45	0.48	0.74
	63	8.2	0.09	59.1	1.09	9.1	0.52	61.4	0.41	9.1	0.43	58.9	1.64	α	0.89	0.93	α	0.03	0.43	0.50	0.96

Appendix C

Empirically Evaluating the Use of Bytecode for Diversity-Based Test Case Prioritisation

Table C.1: *Projects, Bug IDs, and Corresponding Classes Under Test*

Project	Bug ID	Classes Under Test
Cli	30	Parser
	31	Option, OptionBuilder
	32	HelpFormatter
	33	HelpFormatter
	34	Option, OptionBuilder
Compress	1	CpioArchiveOutputStream
	11	ArchiveStreamFactory
	16	ArchiveStreamFactory
	22	BZip2CompressorInputStream
	24	TarUtils
	26	IOUtils
	27	TarUtils
Csv	2	CSVRecord
	3	Lexer
	4	CSVParser
	5	CSVPrinter
	7	CSVParser
	8	CSVFormat
	10	CSVPrinter
	11	CSVParser
	12	CSVFormat

	16	CSVParser
Jsoup	4	Entities
	15	TreeBuilderState
	16	DocumentType
	19	Whitelist
	20	DataUtil
	26	Cleaner
	27	DataUtil
	29	Document
	30	Cleaner
	33	HtmlTreeBuilder
	35	HtmlTreeBuilderState
	36	DataUtil
	37	Element
	38	HtmlTreeBuilderState
	39	DataUtil
	40	DocumentType
Lang	4	LookupTranslator
	6	CharSequenceTranslator
	15	TypeUtils
	16	NumberUtils
	17	CharSequenceTranslator
	19	NumericEntityUnescaper
	22	Fraction
	23	ExtendedMessageFormat
	24	NumberUtils
	25	EntityArrays
	27	NumberUtils
	28	NumericEntityUnescaper
	31	StringUtils
	33	ClassUtils
	35	ArrayUtils
Math	4	SubLine
	5	Complex
	6	NonLinearConjugateGradientOptimizer, CMAESOptimizer, PowellOptimizer, SimplexOptimizer, LevenbergMarquardtOptimizer
	9	Line
	13	AbstractLeastSquaresOptimizer
	14	AbstractLeastSquaresOptimizer, Weight
	17	Dfp

	18	CMAESOptimizer
	19	CMAESOptimizer
	20	CMAESOptimizer
	21	RectangularCholeskyDecomposition
	23	BrentOptimizer
	24	BrentOptimizer
	25	HarmonicFitter
	26	Fraction
	27	Fraction
	28	SimplexSolver
	30	MannWhitneyUTest
	32	PolygonsSet
	33	SimplexTableau
	37	Complex
	42	SimplexTableau
	47	Complex
	49	OpenMapRealVector
	50	BaseSecantSolver
	51	BaseSecantSolver
	52	Rotation
	54	Dfp
	56	MultidimensionalCounter
	58	GaussianFitter
	61	PoissonDistributionImpl
	64	LevenbergMarquardtOptimizer
	65	AbstractLeastSquaresOptimizer
	67	MultiStartUnivariateRealOptimizer
	68	LevenbergMarquardtOptimizer
	69	PearsonsCorrelation
	70	BisectionSolver
	73	BrentSolver
	76	SingularValueDecompositionImpl
	78	EventState
	80	EigenDecompositionImpl
	81	EigenDecompositionImpl
Time	11	ZoneInfoCompiler
	13	PeriodFormatterBuilder