Received: Added at production | Revised: Added at production | Accepted: Added at production

DOI: xxx/xxxx

ARTICLE TYPE

A Systematic Mapping Study of the Metrics, Uses, and Subjects of Diversity-Based Testing Techniques

Islam T. Elgendy | **Robert M. Hierons** | **Phil McMinn**

School of Computer Science, The University of Sheffield, Sheffield, UK

Correspondence
Islam Elgendy, School of Computer Science, The University of Sheffield, Sheffield, UK.
Email: i.elgendy@sheffield.ac.uk

Abstract

There has been a significant amount of interest regarding the use of diversity-based testing techniques in software testing over the past two decades. Diversity-based testing (DBT) technique uses similarity metrics to leverage the dissimilarity between software artefacts — such as requirements, abstract models, program structures, or inputs — in order to address a software testing problem. DBT techniques have been used to assist in finding solutions to several different types of problems including generating test cases, prioritising them, and reducing very large test suites. This paper is a systematic mapping study of DBT techniques that summarises 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 have been used by researchers to tackle 11 different types of software testing problems. We further present an analysis of the recent trends in DBT techniques and review the different application domains to which the techniques have been applied, giving an overview of the tools developed by researchers in order to do so. Finally, the paper identifies some DBT challenges that are potential topics for future work, such as exploring other diversity artefacts and measuring diversity for complex data input.

KEYWORDS

Diversity-based testing, Test data generation, Test case prioritisation, Similarity-based testing, Diversity artefacts, Similarity metrics, Test case selection, Test suite reduction, Fault localisation, Diversity-based tools

1 | INTRODUCTION

One of the key challenges associated with finding software failures 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, risk, usage and complexity analysis, and history-based testing 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[80] — 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*. Different code coverage metrics, for example, dictate all statements or all branches of the software are executed, ensuring the diversity of code structures exercised by tests[1,2]. Model-based techniques ensure diversity of tests by ensuring distinct abstract states of the system are tested along with the transitions between them[1,3]. Even boundary-value analysis[1,4], 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 we refer to in this systematic mapping study as *diversity-based testing (DBT) techniques*.

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[5,6,7,8,9], the outputs it obtains given its inputs[10,11,12,13,14], the program structures it executes[15,16,17,18]; or on the basis of more abstract artefacts, such as the requirements the test exercises[19,20], or parts of a model representation that it covers[21,22,23,24,25,26]. Examples of similarity metrics used by researchers include, for example, the Euclidean distance between two numerical inputs[27,5] or signals of a Simulink model[13,14], or the Edit distance between two regular expressions[28], or the Jaccard distance between two sets of selected product line features[29,30]. More formally, we 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 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[10,31,13,14], test case prioritisation[19,32,14,33,34], test suite reduction[35,22,36,37], and test suite quality evaluation[38,39,40,41]. They have also been applied to a wide range of subject domains, including deep learning models[42,43,44], compilers[45,46,47], web applications[10,48,49], as well as general code libraries[50,6,51,33]. The advantages of DBT techniques have been reported in several studies on test case analysis, generation, and optimisation[52,53,51,6,22,54,33]. 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[11,48,8,6]. 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[19,53,21]. Finally, DBT techniques have been applied to different types of testing at different levels, including unit testing[52,51], integration testing[55] and system testing[56,57].

DBT appears to have originated out of the work on Adaptive Random Testing (ART). ART was recently surveyed by Huang et al.[58], 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 our 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 our definition, because they partition the input domain instead — that is, *without* the use of a similarity metric[59,60,61], and potentially with the help of a human expert instead[62].

A systematic mapping study provides a structure of the type of research reports and results that have been published by categorising them[63]. 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[64]. Given the wide range of types of DBT techniques, and the different problems and application domains to which they have been applied, we conducted a systematic mapping study of the field, collecting results from multiple search engines including IEEE Xplore, the ACM Digital Library, Scopus, and SpringerLink. We conducted our study following the guidelines by Petersen et al.[63] and Kitchenham[64]. The purpose of this study is to explore how DBT techniques in general have developed and broadened since its beginnings with ART. We do not, therefore, include ART itself as part of our review, referring the reader to the comprehensive survey of 140 ART papers by Huang et al.[58] 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, our 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 paper, the first to define and survey the wider area of DBT techniques, summarises the work of 167 articles, which we list in a BibTeX file and make available for other researchers to use via a public GitHub repository and a replication package[65]. Our 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 3).
3. A compilation of the different metrics used to measure the similarity between testing artefacts (Section 4).
4. A summary of the artefacts used as a basis for DBT techniques (Section 5).
5. A summary of the different software testing problems where diversity has been applied (Section 6).

6. An overview of the different subject domains in which diversity-based testing has been utilised (Section 7), and the tools available for addressing them (Section 8).
7. A discussion of possible future research directions for DBT techniques (Section 9).

2 | METHODOLOGY

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

RQ1: ORIGINS, TRENDS, AND PROMINENT PAPERS (Section 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 4). What similarity metrics have been used in the literature? Which ones have been used the most, and why?

RQ3: ARTEFACTS (Section 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 6). What are the software testing problems to which DBT techniques have been applied?

RQ5: SUBJECT DOMAINS (Section 7). To what subject domains have DBT techniques been applied?

RQ6: TOOLS (Section 8). What DBT tools have been developed by researchers?

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.[63]. 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[66], 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 our research questions was made during our initial set of previously known papers that used DBT in software testing[52,10,32,67,37,8,48]. The authors discussed the research questions and reformulated them again with the additional papers found from the search, and again during information extraction until we reached our final decision about the research questions reported earlier.

We collected papers for our study on the 24th of June, 2024, using searches on IEEE Xplore[†], the ACM Digital Library[‡], Scopus[§], and SpringerLink[¶]. We did not limit our searches to any particular year range. The search query we 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, our search query contained only "`divers*`" in title or keywords, but we refined the search to include "`similar*`" before the screening process. We 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 our search results without this constraint. SpringerLink did not allow us to search for certain terms in specific places like the title and keywords, so we used the closest search query possible, which was {"`software test*`" AND ("`similar*`" OR "`divers*`")} anywhere in the paper. We provide the exact search queries used for each search engine and the search operationalisation in our replication package[65]. We validated our search string against our initial set of primary studies[52,10,32,67,37,8,48]. Also, we validated the search against a

[†] https://ieeexplore.ieee.org/Xplore
[‡] https://dl.acm.org
[§] https://www.scopus.com/
[¶] https://link.springer.com

suggested list from one of the reviewers of this paper[56,55,39,68,52,6,32,69,70,21,71]. The combined lists contain 16 papers and 15 out of them were among our search results, while we identified the last one[32] during the snowballing phase.

Figure 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 we removed duplicate papers returned by more than one search engine, the set of papers collated numbered 3,569 in size, that we label the **Initial** set of papers in the figure.

We screened the papers by having the first and last authors manually review them to ensure they discussed or evaluated at least one approach fitting Definition 1, and thereby fell in the scope of this study. Each author independently decided whether to include or exclude papers based on titles, abstracts, and detailed content if needed. They then reviewed and discussed their decisions together to agree on each paper. Most decisions were obvious, and most "disagreements" were just minor mistakes by one of the authors due to the large number of papers to go through. We 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 our definition of a DBT given in Definition 1; for example, the technique did not use a similarity metric, or, we could not find whether the technique made use of a similarity metric in the text of the paper. Following these decisions, we then removed papers that did not appear in journals or conferences ranked "C" or above by the CORE Rankings Portal[72], with the aim of imposing a level of quality on the papers considered by our study. We chose not to do a quality assessment of our own since this could potentially introduce our 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[73], a systematic literature review appearing at SIGCSE[74], and other reviews and mapping studies appearing in journals in related fields[75,76,77]. We 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 we wanted to consider emerging novel ideas or results that are often presented at these venues. We 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. We removed a further 66 papers because they did not fulfil our CORE requirements. Also, we added four papers recommended by the reviewers. Furthermore, when we found duplicate extension papers that presented the same topic, we included only the most complete paper (i.e. the journal version). This left 205 papers, which we mark as the **Intermediate** set of papers in Figure 1.

We 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 1. Finally, in case any DBT papers were missed by our original searches, we applied backward snowballing[66], whereupon we 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 our aforementioned criteria — we added it to the set of papers featured in this study. We found 20 additional papers via this method which we added to form the collection of papers marked as the **Final** set of papers in Figure 1. These papers did not include the search terms or were not in the database of the search engines. Each paper in the final set was read by the first author, who collated the statistics and information needed to answer our 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 our study are regular and short papers, with the majority of the papers being regular (93.4% of the papers are regular), and we did not include position papers. According to the guidelines by Kitchenham[64], the first author has applied a test re-test approach, where he 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 third author performed an independent check on the same random sample of the papers and cross-checked it with the first author to confirm the analysis results.

2.2 | Threats to Validity

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

1. **Some relevant papers might not have been included in the searches.** We mitigated this risk as much as possible by using a variety of search engines and we picked our search terms carefully to include different variations of the terms "diverse" or "similar". Also, we used backward snowballing to find papers that the search engines may not have indexed. However, we validated our search results against our initial set of primary studies and also against a list of papers suggested by one of the reviewers (both lists mentioned in Section 2.1). The combined lists consisted of 16 papers, where 15 out

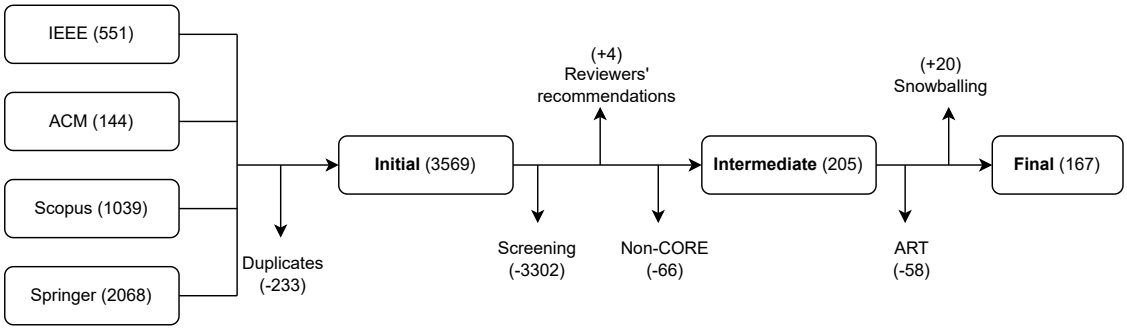


FIGURE 1 Overview of our paper collection methodology, with the number of pages collected at each stage

of them were among our search results, while we identified the last one during our 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), we limited our search and relied on backward snowballing to include the papers that might have used a variation of diversity. To further mitigate this risk, we 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 our original searches, we had a sample of 304 candidates. We did not find a single additional paper that fitted our scope and DBT definition. Therefore, it would not affect our study if such proxy words were included in the search. Finally, we did not perform forward snowballing as the guidelines[64,63] did not prescribe it for mapping studies.

2. **The manual screening process may have eliminated some relevant papers.** We mitigated this risk by having two authors do the screening process separately and independently, then coming together to have a final decision on the papers where they had different opinions.
3. **Some papers may be of low quality and their results might be questionable**. We 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**. We attempted to mitigate this risk by applying a test-retest approach, where the first author made a second data extraction of a random selection of primary studies to check data extraction consistency. Also, the third author performed an independent check on the same random sample of the papers and cross-checked it with the first author to confirm the analysis results.
5. **Reproducibility of the study**. We attempted to mitigate this risk by explaining our methodology as rigorously as possible and providing all search results and data in a replication package[65].

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

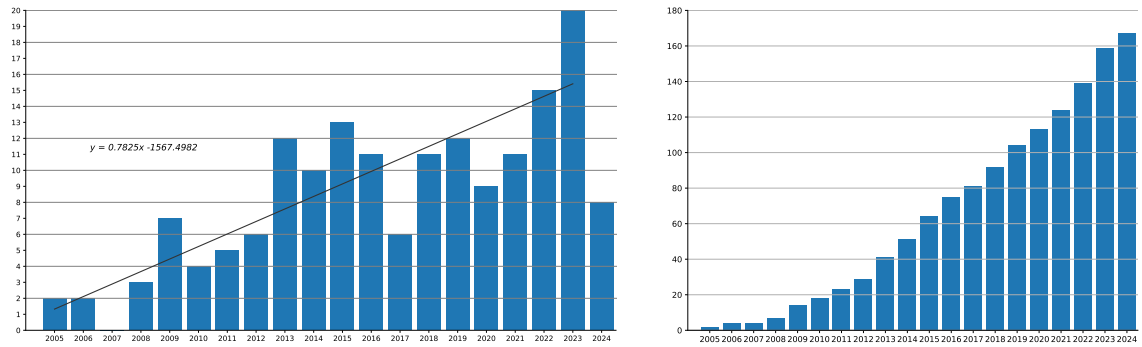
3 | RQ1: ORIGINS, TRENDS, AND PROMINENT PAPERS

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

3.1 | First DBT Papers

The first DBT papers in our final collected set of papers appeared in 2005, are two papers by Xie et al.[78] and Hao et al.[79]. These first two papers are notable in that they apply the principles of DBT to assist two existing approaches, namely 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[78]. 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.



(a) Publications per year

(b) Cumulative sum of publications over the years

FIGURE 2 Papers published per year and the cumulative sum of papers found over the years for the final set of papers collected using our methodology. Since the data was collected partway through 2024, the bar for that year is incomplete and as such we plot the trendline through 2005–2023 only.

TABLE 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

We discuss this paper further in Section 5.1.2 (p.13). Hao et al.'s work[79] was a short paper proposing to detect similarity in test cases to remove bias from suspiciousness scores for fault localisation techniques. We discuss this paper in more detail in Section 6.6 (p.31).

However, peering into the intermediate set of papers collected by our methodology (Section 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[80,81,60,82]. ART itself, however, can be traced back earlier to 2001 by a paper by Chen et al.[83], 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.[80,84,82] in 2004, which were built on partition strategies.

3.2 | Statistics of DBT Papers

Figure 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 we performed the search in June of 2024.

Table 1 lists the top ten publishing venues, highlighting that DBT papers frequently 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.

We collected citation counts from Google Scholar[85], with counts correct to 3rd July 2024. The paper with the second most citations (266) is that of Yoo et al.[34] 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, we calculated the mean number of citations per year and list the top ten papers in terms of this metric in Table 2.

The papers with the highest mean citations per year are by Kim et al.[86], Henard et al.[31] and Hemmati et al.[26]. Kim et al.[86] introduce a new metric for testing Deep Learning systems that leverage on diversity. This paper is the most cited paper (431) in

TABLE 2 Papers with the highest numbers of citations

Rank / Authors	Title	Year	Section (Page)	# Cites	Mean cites per year
1 Kim et al.[86]	Guiding Deep Learning System Testing Using Surprise Adequacy	2019	5.1 (p.12)	431	71.8
2 Henard et al.[31]	Bypassing the combinatorial explosion: Using similarity to generate and prioritise t-wise test configurations for software product lines	2014	5.16 (p.19)	242	22.0
3 Hemmati et al.[26]	Achieving Scalable Model-Based Testing through Test Case Diversity	2013	6.4.2 (p.30)	259	21.6
4 Miranda et al.[33]	FAST Approaches to Scalable Similarity-Based Test Case Prioritisation	2018	6.2 (p.26), 5.2 (p.14)	121	17.3
5 Yoo et al.[34]	Clustering test cases to achieve effective and scalable prioritisation incorporating expert knowledge	2009	6.2.7 (p.28), 5.6 (p.16)	266	16.6
6 Arafeen et al.[19]	Test case prioritisation using requirements-based clustering	2013	6.2.7 (p.28), 5.4 (p.15)	194	16.2
7 Feldt et al.[6]	Test Set Diameter: Quantifying the Diversity of Sets of Test Cases	2016	5.1 (p.12)	143	15.9
8 Matinnejad et al.[14]	Test Generation and Test prioritisation for Simulink Models with Dynamic Behaviour	2019	6.1.3 (p.24), 5.3 (p.14)	87	14.5
9 Zohdinasab et al.[87]	Efficient and effective feature space exploration for testing deep learning systems	2023	5.13 (p.18)	29	14.5
10 Zohdinasab et al.[88]	Deephyperion: exploring the feature space of deep learning-based systems through illumination search	2021	5.13 (p.18)	57	14.3

our 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.[31] 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. We discuss it in more detail in Section 5.16 (p.19). Hemmati et al.[26] focussed on test case selection in the context of Model-Based Testing. We discuss this paper in Section 6.4.2 (p.30).

Test case prioritisation is the focus of four of the ten papers and in Table 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. We discuss the prominent problems to which DBT has been applied in Section 6.

3.3 | Prominent Authors

The top five most prolific authors in the papers collected in our study are listed in Table 3. The author publishing the most papers is Lionel Briand[89], with 15 publications, focussed on applying DBT in model-based testing, and Simulink models in particular (see Section 6.1.3, p.24; and Section 6.4.2, p.30).

The second in the list is Robert Feldt[90] with 10 papers. Robert Feldt applied his work mostly in Search-Based Software Testing to diversity. This work is discussed in Section 6.1 (p.22). 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 5.1, p.12), and proposed a new metric called "Surprise Adequacy for Deep Learning Systems" (SADL) for testing deep learning system (discussed in Section 5.1, p.12).

Ranked third in the list with nine papers is Francisco Gomes de Oliveria Neto[91], who has mainly used DBT approaches to solve the problem of test case selection, discussed in Section 6.4.2 (p.30).

The fourth in the list are Hadi Hemmati[92] and Paolo Tonella 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 6.2, p.26; and Section 6.4.2, p.30). Paolo Tonella worked on a handful of topics in DBT including deep learning testing (discussed in Section 5.13, p.18) and web testing (discussed in Section 6.1.5, p.25).

The most significant collaboration was between Lionel Briand and Hadi Hemmati, who co-authored five papers[23,24,25,26,93] 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[39,68,94] on applying diversity for boundary value exploration[94], visualising test diversity[39], and measuring behavioural diversity with mutation testing[68].

TABLE 3 Authors with the highest number of papers

Rank / Author	# Papers	
1 Lionel Briand (https://www.lbriand.info)	15	25,23,24,93,26,12,13,14,42,95,96,97,98,99,43
2 Robert Feldt (http://www.robertfeldt.net)	10	52,6,39,68,94,100,101,102,86,103
3 Francisco Gomes de Oliveira Neto (https://www.gu.se/en/about/find-staff/franciscodeoliveiraneto)	9	21,56,55,68,39,94,44,71,69
=4 Hadi Hemmati (https://lassonde.yorku.ca/users/hhemmati)	8	23,24,25,93,26,53,51,104
=4 Paolo Tonella (https://www.inf.usi.ch/faculty/tonella)	8	49,105,48,106,34,107,88,87

Conclusions — RQ1: ORIGINS, TRENDS, AND PROMINENT PAPERS

The first paper on DBT techniques in our 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.[86] 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.[31], on diversity for configurations of software product lines for testing, published in 2014; and Hemmati et al.[26] 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.

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. We identified 79 similarity metrics in the literature, including well-known ones like Euclidean and Hamming distances, as well as domain-specific and novel metrics. We categorised these into two groups: generic metrics from other fields and specialised metrics for Software Engineering. Figure 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[108]. Therefore, we merged the counts of these under the more common heading.

Table 4 lists the top five generic similarity metrics, while Table 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 5 references the introducing paper. Extended tables with all 79 metrics are available in the replication package[65]. Figure 3 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.

TABLE 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 [109]	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	110,19,111,99,5,112,113 35,67,36,37,114,115,116 117,118,53,86,119,32,120 12,13,14,102,121,122,123 107,95,8,124,16,125,126
G2	Edit distance [127]	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	128,129,130,48,131,28 67,36,22,55,114,50,132 117,23,24,25,26,71,32 133,134,104,51,102,8 135,136,137,138
G3	Jaccard distance [139]	The ratio of intersection over union between two sets A and B of values. The formula is: $Jac(A,B) = 1 - \frac{\lvert A \cap B \rvert}{\lvert A \cup B \rvert} = 1 - \frac{\lvert A \cap B \rvert}{\lvert A \rvert + \lvert B \rvert - \lvert A \cap B \rvert}$ Sometimes expressed [16] as: $Jac(A,B) = 1 - \frac{A.B}{A.B + \omega(\lVert A \rVert^2 + \lVert B \rVert^2 - 2(A.B))}$ with with $\omega = 1$.	Sets and strings	31	140,28,22,67,36,55,39 141,26,142,117,143,24 93,30,31,71,144,96,97 134,145,146,33,46,147 16,148,17,18,149
G4	Hamming distance [150]	The number of times when the corresponding characters in two strings are different. Huang et al. [108] referred to this as "Overlap" distance.	Vectors	19	29,38,151,67,36,114,26 152,24,108,32,144,104 51,8,153,154,155,34
G5	Manhattan distance [156]	The sum of the absolute differences between two vectors X and Y. The formula is: $Man(X,Y) = \sum_{i=1}^{n} \lvert x_i - y_i \rvert$	Numbers	18	131,38,45,157,114,117 143,53,32,47,96,97 120,122,95,8,88,87

TABLE 5 A list of the specialised Software Engineering similarity metrics.

ID	Metric & Source	Description	Total	Papers
S1	Identical transition distance [21]	The number of identical transitions between two finite state machines divided by the average length of paths.	6	21,56,23,24,25,26
S2	Test set diameter (TSDm) [6]	An extension of the pairwise normalised compression distance (G6) to multisets.	6	6,143,118,7,9,158
S3	Identical state distance [23]	The number of identical states between two paths of finite state machines divided by their average number of states.	3	23,25,26
S4	Trigger-based distance [23]	An extension of identical transition similarity (S1) to account for triggers in the transitions.	3	23,25,26
S5	Average population diameter [159]	The average distance between all vectors in a population, where the distance between two vectors is the difference of their lengths.	2	159,160

Conclusions — RQ2: SIMILARITY METRICS

Many similarity metrics were used in the literature, and we 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 5 and Section 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.

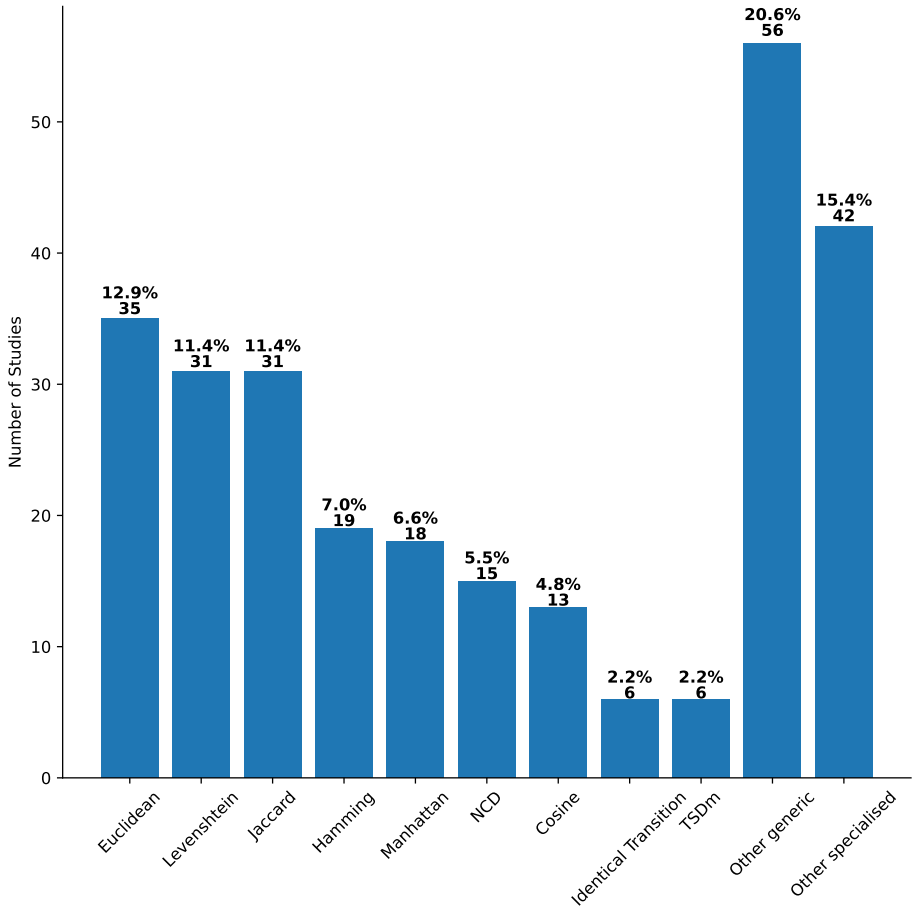


FIGURE 3 Distribution of similarity metrics used in DBT techniques.

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. We discuss how these artefacts were used in the DBT techniques.

Table 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 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 our study. Also, DBT papers that use a hybrid of diversity artefacts form 7% of the collected papers.

Finally, all the papers in our study use only one of the following categories, except four papers[23,24,25,26] that apply their experiments on transitions and triggers artefacts separately, and another paper[158] 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 5.20).

TABLE 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	161,129,162,131,5,112,163,158 164,6,86,105,100,145,106,165,44 166,123,101,167,107,168,7,8,9 135,159,160,136,169,78,170,171

2	Test Script	Information from the actual text of the test cases which may include for example setup, inputs, and assertions.	32	69,140,172,173,174,113,157,35,67,36 37,39,114,175,116,143,118,79,93,51,53 108,71,32,33,104,98,95,40,46,176,177
3	Feature	The specific capabilities, or characteristics that test cases has in the software under test.	21	29,128,42,43,99,45,151,158,115,30,178 179,119,47,121,122,180,181,125,88,87
4	Hybrid	Approaches that use a combination of different artefacts as targets for applying diversity.	12	182,110,111,48,52,144 96,97,133,49,183,184
=5	Execution	The actual runtime execution path of the software under test after running the test cases.	9	38,50,117,185,186 187,34,138,149
=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	130,21,22,56,23 24,25,26,137
=5	Output	The results produced by the software under test from running the test cases.	9	10,11,94,103,142 188,12,13,14
=5	Coverage	Information about statement or branch coverage achieved from running the test cases.	9	15,55,120,189,16 17,18,126,190
9	GUI	The visual components and their properties' values that represent the interface of the software.	6	191,192,193,134,102,41
10	Semantic	Statements that have different wordings but carry similar meanings are said to be *semantically* similar.	5	194,28,68,195,196
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	23,24,25,26
=12	Path	The possible flow of control and data within a program without executing it.	3	197,198,155
=12	Context	Data from software and environment to guide the responses and actions of the system.	3	199,153,154
=12	Test Steps	A test step corresponds to some manual action and consists of a stimulus and an expected reaction.	3	132,200,147
=12	Software product line configuration	A combination of specific features or components from a software product line to create a customized software product.	3	141,31,148
=16	Mutant	Small variations introduced to a software program using mutation operators that represent faults in the software.	2	201,202
=16	Requirements	The system specifications and requirements, where test cases linked to requirements are specified through traceability matrix.	2	19,20
=16	Graph	Graph models represent and analyze relationships between variables in a system, where the nodes of the graph are the variables and the edges are the relationships or dependencies.	2	203,204
=19	Test Report	Reports written in natural language and contain screenshots that show the results of tests.	1	152
=19	Social	Coverage data collected from similar users who utilise a web service.	1	146

=19	Function	A process to perform a certain functionality given some inputs and possibly producing an output.	1	[205]
=19	Running Time	The runtime execution of test cases.	1	[124]

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[5,8,131,170,129,48].

Input diversity is used in search-based software testing techniques to guide the search process. Bueno et al.[5] 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.[131] 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 and Miller[8] 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.[170] 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.[86] 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.[107] 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.[44] 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.[171] 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.[158] 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.[6], 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[123], to generate inputs that exercise different traces[167], or to create a diverse set of tests[165]. Also, input diversity was employed for multiple dispatch languages. Poulding and Feldt[101] proposed use of diverse test inputs in terms of values and types.

5.1.1 | 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[7] 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.[9] proposed a black-box test case generation technique of tree-structured programs (e.g. XML parsers or web browsers) based on

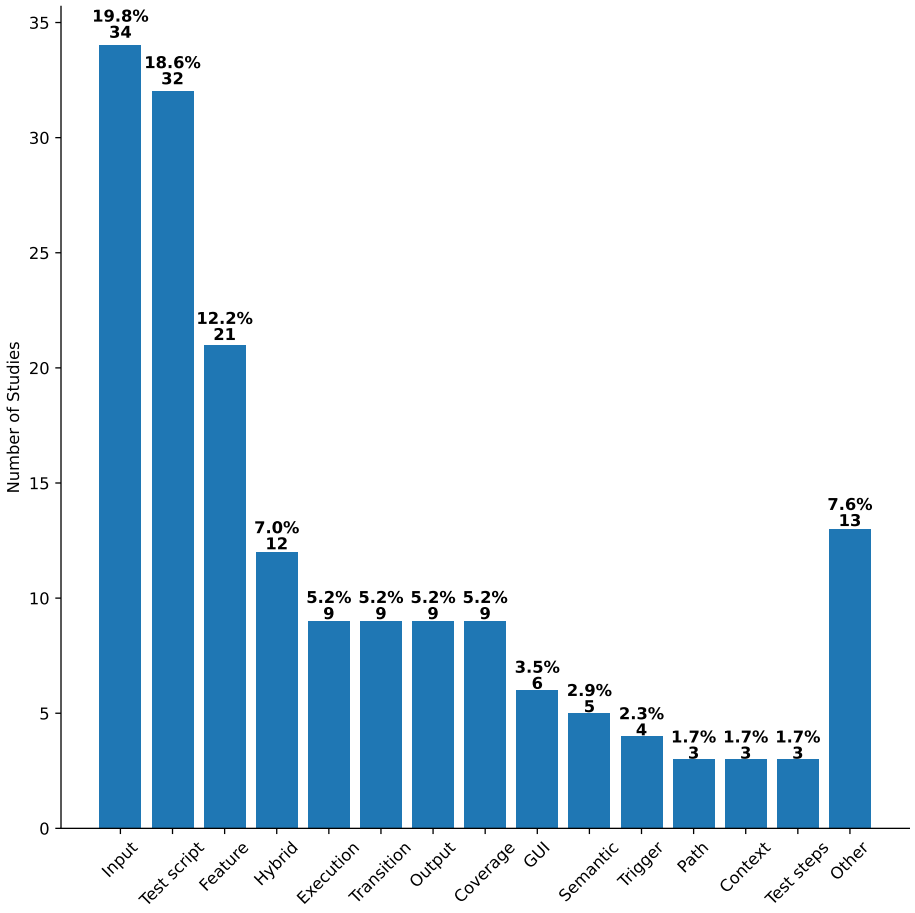


FIGURE 4 Distribution of testing artefacts used in DBT techniques.

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 [135] 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 [206], then used another tool called Triang [207] 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. [161] 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.

5.1.2 | 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. [78] 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 analyzes 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.[162] 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.[105] 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.[166] 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[129] 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.[136] 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.

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[32,53,172,33,116,108,113,173,157,143,118] or test suite reduction[67,36,37,35,114]. 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.[32] 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.[53] implemented the same algorithm used by Ledru et al.[32], using the Euclidean and Manhattan distances, to work on manual black-box system testing. Rogstad et al.[95] 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.[174] 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.[104] used a few similarity measures in diversity-based test case selection, such as Hamming distance, Levenshtein distance, and Dice diversity formula. Khojah et al.[71] 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.[46], 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.[33] 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.[37] 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.[40] 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.[114] 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.[98] 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.

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[10] 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.[11] 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. Then 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.[142] 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[188] 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.[208], 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.[12] 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. In a later study, Matinnejad et al.[13] 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. Later, Matinnejad et al.[14] used normalised Euclidean distance to calculate the diversity between output signals in terms of vector-based and feature-based diversity.

Furthermore, output diversity is used in the automatic detection of boundaries. Dobslaw et al.[94] 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.

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.[19] used requirements' diversity by using k-means clustering[209] 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 5) to prioritise the test case in each cluster. Masuda et al.[20] 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.

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.[185] 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.[117] 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[189]. 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.[36] 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.[67] 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 [18] 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. [120] 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. [155] proposed three algorithms for calculating the similarity between the test path and the target path for path-oriented testing. Panchapakesan et al. [198] 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. [197] 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.

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. [34] 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. [186,50] 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. [138] 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. [38] 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. [187] 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).

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 the Cosine similarity metric.

Two tests that fail or pass together are said to have similar semantic [196]. Ojdanic et al. [196] 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. [68] 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. [194] 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 textual (e.g., Levenshtein distance) or syntactic similarities between the original sentence and the translated sentence, but ignoring the semantic similarity. Cao et al.[28] 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.

5.8 | Context

Context-aware Pervasive Software (CPS) collects and analyzes 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[154]. 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[153] 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.[154] 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[199] 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.

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.[191] 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.[134] 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[41] 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.[102] 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.

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)[210]. An LTS is a directed graph defined in terms of states and labelled transitions between states to describe system behaviour. Coutinho et al.[22] 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.[21] and Gomes de Oliveira Neto et al.[56] 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 5. In a later study, Hemmati et al.[23] 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 5.

Another paper used transition diversity in search-based software testing. Asoudeh and Labiche[130] 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.[137] 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.

5.11 | Graph

Graph models represent and analyze 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.[204] 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.

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.[201] 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.[202] made a more comprehensive empirical study using 352 real faults from *Defects4J* on the distinguishing mutation adequacy criterion.

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)[122], constructing meaningful search spaces[179], compiler testing[45,47], and SPL[30,29,181].

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.[122] 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.[42,43] 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.[178] 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.[45] 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.

Constructing a meaningful search space (fitness landscape) refers to the process of defining and organising a set of possible solutions or configurations in a way that effectively represents the problem domain and facilitates the search or exploration of potential solutions. Joffe and Clark[179] 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.[88,87] 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[121] 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.[180] 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.[151] 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.[158] 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.[128] 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.[30] 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.[29] 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.[181] 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.

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.[152] 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.

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[200]. The similarity between test steps can be calculated using any of the similarity distance functions. Flemstrom et al.[132] used the Levenshtein distance to measure the overlap between test steps. In a later study, Flemstrom et al.[200] used similarities between test steps to guide the ordering of test cases. Viggiato et al.[147] 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).

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[31,141,148].

5.17 | Social

Miranda et al.[146] 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.

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.[205] 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.

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.[124]. 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.

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.[52] 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.[182,110] 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.[96] 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.[97] 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.[111] 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.[133] 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[49] 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 events. TCov is based on the FSM coverage by the test cases. Biagiola et al.[48] 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.

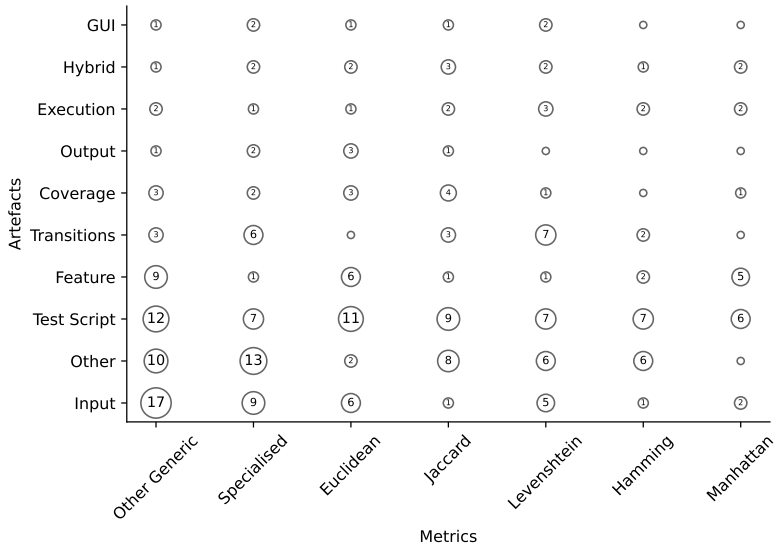


FIGURE 5 Relationship between similarity metrics and diversity artefacts used in DBT techniques.

5.21 | Relationship between Similarity Metrics and Diversity Artefacts

Figure 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.

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. We explore the various studies made regarding these problems and discuss the impact of DBT techniques upon them.

Table 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 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[6,146,86,121], investigated test reuse in software product lines[194,174], utilised output diversity in boundary value testing[103,94], used DBT in test adaptation[205], and argued for teaching diversity principles in software testing[163]. The paper by Matinnejad et al.[14] addressed both test data generation and test case prioritisation where it is cited in both problems in Table 7. All other papers in our study address only one type of problem in software testing.

TABLE 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	128,182,110,129,10,162,130,11,48,131,5,112,197,28,45,151 158,52,191,141,30,31,178,179,105,188,119,144,47,195,133 49,100,134,20,13,14,106,165,189,198,101,167,107,168,7,8,9 135,46,159,160,153,154,136,169,180,137,181,78,155,170 171,138,88,87
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	29,69,161,19,172,173,113,157,68 50,152,200,116,143,118,192,53 108,71,32,120,14,145,33,44 51,204,16,186,187,34,125,126
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	42,43,111,21,164,56,55 115,142,23,24,25,93,26,12 104,166,95,124,201,202,190
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	140,99,38,39,132,117,193,183,122 102,185,196,40,184,148,41,18
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	15,199,35,22,67,36 37,114,203,98,123,147
6 Fault localisation	The process of identifying the locations of faults in a program.	8	175,79,176,96 97,177,17,149
7 New Metric	Introducing a new similarity metric that can be used to solve any of the software testing problems.	4	6,146,86,121
=8 Test Reuse	Resuing part or the whole of a test suite on other systems.	2	174,194
=8 Boundary value analysis	A software testing technique in which tests are designed to include representatives of boundary values in a range.	2	94,103
=10 Test Adaptation	The process of identifying a test suite from another program that can be adapted to be used on the SUT.	1	205
=10 Teach DBT	Teaching software testing methods based on diversity principles.	1	163

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.

6.1.1 | 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.[5] 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.[112] 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.[131] 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[8] investigated black-box testing methods where the input value is a string and used two fitness

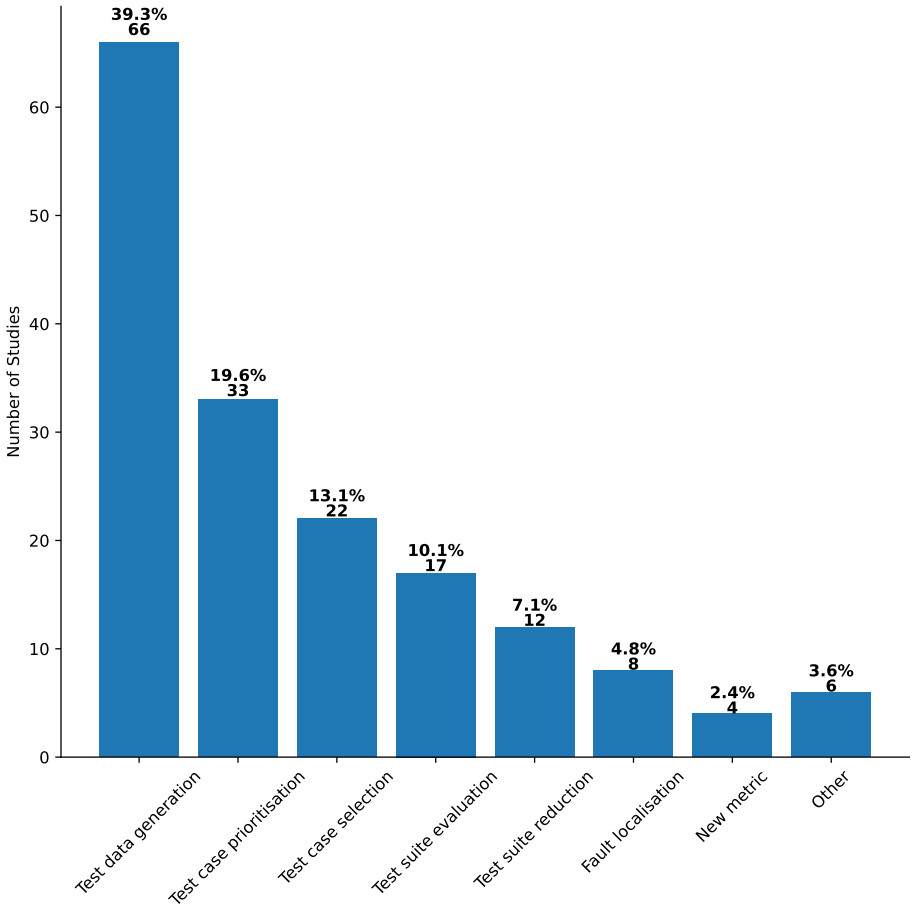


FIGURE 6 Distribution of DBT papers addressing problems in software testing.

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[182]. Albunian et al.[110] investigated the effect of holding the population diversity on the Many-Objective Sorting Algorithm (MOSA) for test data generation. Alshraideh et al.[162] 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.[52] proposed a model for test variability referred to as VAT (VAriability of Tests), which we discussed in Section 5.20 (p. 20). 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.[155] 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.[198] 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.[197] 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.

6.1.2 | 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[211,212], 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.[138] 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.[169] 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.[119] used seed diversity, representing seeds as byte sequences, clustering similar sequences, and selecting diverse seeds to improve coverage. Lee et al.[144] 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.[167] proposed a black-box approach for quickly generating diverse, valid test inputs in property-based testing. Nguyen and Grunske[189] focused on increasing behavioural diversity and found their method superior to fuzzers like Zest[213] and RLCheck[167]. Menendez et al.[165] introduced *HashFuzz*, which combines coverage and diversity using hash functions to create diverse tests for C programs. Wang et al.[180] 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.[158] 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.

6.1.3 | Model-based testing

Researchers have also applied diversity in model-based testing. Asoudeh and Labiche[130] 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.[13] 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[13]. In a later study, Matinnejad et al.[14] 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.[178] proposed an approach for generating valid and diverse test inputs for a testing task-specific DNN.

6.1.4 | 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, we 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.[106] 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*.

6.1.5 | 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[49] 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[10] proposed a black-box test criterion based on Output Uniqueness and evaluated it on six web applications. Later, Benito et al.[11] 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.[48] 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.[136] 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.

6.1.6 | 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.[159] 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.[160] 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.

6.1.7 | Directed random testing

Pacheco et al.[214] proposed a technique called feedback-directed random test generation and implemented a tool called "Randoop"[215,216] 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[170]. Yatoh et al.[170] 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.

6.1.8 | 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[101] 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.

6.1.9 | 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.[191] 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.[134] 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.

6.1.10 | 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.[45] 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.[47] 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.[46] 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.

6.1.11 | 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[135] 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.

6.1.12 | 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[5,112,131,110,162,155].
- **Fuzzing**: DBT was applied in fuzzing to improve coverage using behavioural diversity[189] or seed diversity and clustering based on similarity measures[180,158].
- **Model-based testing**: DBT was used for testing extended finite state machines[130] and Simulink models[13,13,14] to enhance coverage and fault detection.
- **Web and mobile testing**: DBT was used to enhance web fault detection using output diversity[10,11], and population diversity[136] to reduce test generation time. Also, diversity was utilised for testing mobile apps to improve coverage and fault detection et al.[159,160].
- **Compiler testing**: Chen et al.[45] proposed HiCOND for test-program generation based on historical bug data. Li et al.[47] introduced RECORD, and Tang et al.[46] developed DIPROM, both finding more bugs than HiCOND.

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[217]. DBT techniques have been used in many test case prioritisation papers.

6.2.1 | Prioritisation based on test cases

Some techniques use test cases themselves to guide prioritisation. Ledru et al.[32] 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.[186] aimed to improve similarity-based prioritisation by considering the execution times of program elements in ordering test cases. Fang et al.[50] focused on ordered execution sequences for prioritisation. Wang et al.[16] proposed a branch coverage similarity-based approach using six similarity measures, finding Euclidean distance most effective. Khojah et al.[71] 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.[161] 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.[33] 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.[116] later compared test case selection using Ekstazi[218] with test case prioritisation using FAST[33], 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.[53] modified existing test case prioritisation techniques to work on manual black-box system testing. They implemented three different

Some empirical studies have evaluated the effectiveness of DBT test case prioritisation (TCP) techniques. Huang et al.[108] 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.[143] 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[32] 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[51] 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.

6.2.2 | 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.[200] 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.

6.2.3 | Prioritisation based on program executions

Some researchers designed their prioritisation techniques based on program execution diversity. Gomes de Oliveira Neto et al.[68] 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.

6.2.4 | 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[44,187]. Some prioritisation approaches involve DBT techniques. Mosin et al.[44] 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.[187] proposed a DBT prioritisation technique for MPs, suggesting that diverse execution patterns between source and follow-up cases increase the likelihood of detecting violations.

6.2.5 | 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.[145] 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.[173] 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.

6.2.6 | Prioritisation in CI environments

Some studies investigated diversity-based test case prioritisation in Continuous Integration (CI) environments. Haghighatkhah et al.[118] 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[172] 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.

6.2.7 | 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.[34] proposed clustering based on dynamic runtime behavior 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.[113] 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.[126] 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[157] 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.[125] 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.[19] 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.[120] 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.

6.2.8 | 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)[32,50,71,186].
- **Clustering based**: DBT was applied in clustering-based TCP based on runtime behaviour[34], based on code complexity and fault history[113], based on coverage and fault-proneness[120], based on log analysis[125], or based on requirements and code modifications[19].
- **Empirical studies and comparisons**: Huang et al.[108] compared 14 similarity measures and found global TCP more effective than local TCP for coverage and fault detection. Haghighatkhah et al.[143] evaluated various similarity-based TCP methods to determine their effectiveness in fault location using greedy algorithms based on Ledru et al.'s[32] work. Mosin et al.[44] 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.[33] introduced the FAST family of prioritisation techniques, using big data methods for efficient similarity detection in large test suites. Greca et al.[116] compared FAST[33] with Ekstazi[218] 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[118], or based on textual similarity to the changed parts of the code[172].

6.3 | Test Suite Reduction

Test suite reduction (TSR) techniques eliminate redundant test cases to reduce test suite size[217]. Coviello et al.[67] 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.[37] 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.[114] maximised textual diversity between test cases to reach a desired test suite size. Pan et al.[98] 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.[36] proposed a method where similar test cases are clustered based on statement coverage, selecting the most comprehensive case from each cluster. Beena and Sarala[15] developed a multi-objective approach focusing on maximising coverage and minimising execution time using a similarity matrix for clustering. Chetouane et al.[35] combined k-means clustering with binary search to optimise cluster numbers while maintaining test suite effectiveness. Viggiato et al.[147] 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.[22] 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.[123] 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[203], or based on block coverage criteria, control flow of a program, variable definition-usage, and data values[184].

6.4 | Test Case Selection

Test case selection involves choosing a subset of available tests relevant to recent changes[217]. Rogstad et al.[95] 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[115] suggested a heuristic method to maximise coverage and diversity, reducing the risk of partial testing. Gomes de Oliveira Neto et al.[55] 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.[69] later expanded on using DBT techniques for prioritising CI pipelines.

In a recent study, Shimari et al.[124] 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.

6.4.1 | Multi-objective algorithms

Multi-objective algorithms have been used for test case selection in a number of studies[164,104,166]. De Lucia et al.[164] 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.[104] 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.[166] 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.

6.4.2 | 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.[21] and Gomes de Oliveira Neto et al.[56] proposed similarity-based approaches for MBTS, detecting code modifications through test case similarities. Hemmati et al.[23,25] 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.[111] 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[24], and Hemmati et al.[93], 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.[26] 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.

6.4.3 | 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.[190] 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.[142] 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.[42] 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.[43] developed a test case selection approach called "DeepGD".

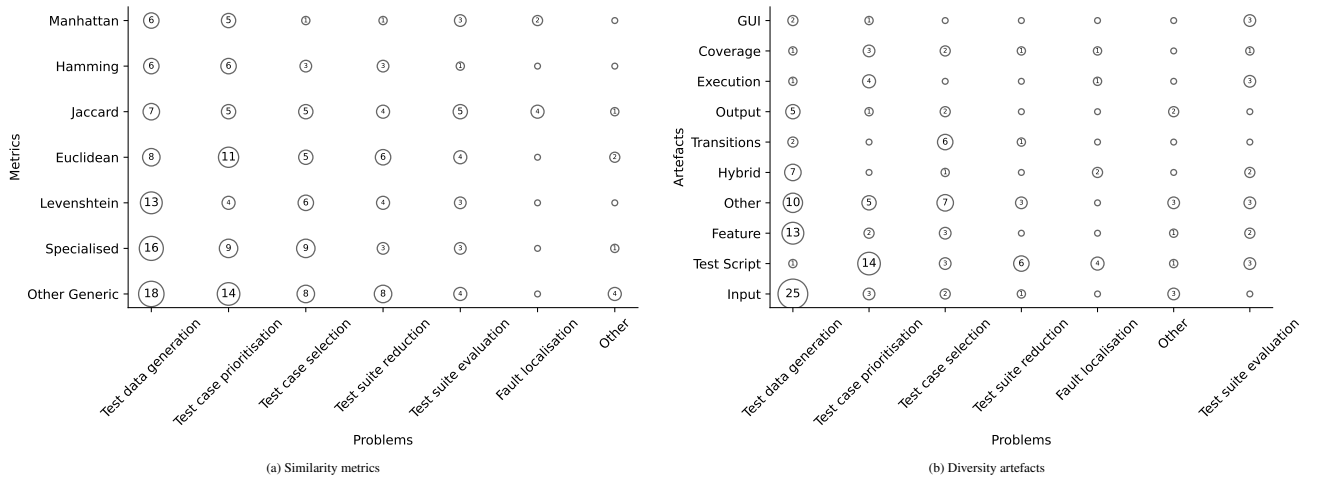
6.5 | Test Suite Quality Evaluation

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[183] 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.[40] 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[41,18,132] and some of its challenges[39]. Xie and Memon[41] 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[18] 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.[132] 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.[39] 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.[140] 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[219], 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.[193,219] proposed an approach to help testers explore any web application using Exploratory Testing. They used



(a) Similarity metrics (b) Diversity artefacts

FIGURE 7 Relationship between DBT problems and similarity metrics (a) and diversity artefacts (b), respectively

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.[102] 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.[99] 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.

6.6 | Fault Localisation

Fault localisation (FL) is the process of identifying the locations of faults in a program[220]. In this section, we cover testing-based fault localisation, which utilises the testing information to localise the faults[79], and in particular DBT techniques. Hao et al.[79,176] 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.[149] proposed a fault localisation framework to make use of execution similarities of test cases to improve coverage-based fault localisation techniques. Gong et al.[175] 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.[177] 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.[17] 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.[96] 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.[97] extended this work by adding a new test objective based on output diversity.

6.7 | Relationship between DBT Problems and Diversity Artefacts and Similarity Metrics

Figure 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.

TABLE 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	29,69,182,110,129,162,161,140,19,172,15,11,173,131,199,5 112,174,194,197,38,28,113,163,157,35,67,36,37,164,55,39 68,94,114,50,115,52,6,103,152,141,132,200,175,116,117,143 118,79,176,93,30,31,108,203,105,71,119,32,144,96,97,133,120 100,20,145,106,165,146,33,104,189,183,51,122,196,198,166,98 123,185,101,167,204,7,8,9,40,124,201,202,135,184,205,147,153 154,16,186,177,148,181,78,155,170,34,17,125,138,149,126
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	42,43,128,99,158,142,178,179,86 188,44,107,187,171,190,18,88,87
3 Model-Based	Systems represented as models like FSM or Simulink models.	12	111,130,21,22,56,23 24,25,26,12,13,14
4 Web	Applications that run on the web browser.	10	10,48,53,193,219 195,49,102,136,169
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	191,192,134,41
=6 Autonomous Vehicles	Systems running autonomous vehicles, which perform various driving tasks, such as perception, localisation, planning, and control.	3	151,121,180
=6 Compilers	A program that translates a programming language's source code into machine code, bytecode or another programming language.	3	45,47,46
=6 Mobile	Applications designed to run on a mobile device.	3	159,160,137
9 Database	Applications that process and manipulate data stored and work as a database management system.	1	95

Figure 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[8] with an increase of up to 30%[112], assisted in preventing the search process from stagnating in search-based approaches[162,110,136], showed a significant increase in crash detection of between 28% to 97%[165], and decreased the average generation time of valid tests in deep learning systems by 15.7% to 47%[178]. Furthermore, DBT techniques increased the overall average percentage of fault detection compared to other test case prioritisation techniques[33,53,143], improved fault-detection rate by up to 45% over the coverage-based selection and 110% over the random selection[26], and achieved 82.2% test suite reduction maintaining both coverage and mutation score of the original test suites[35].

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 8 summarises these domains, including descriptions, paper counts, and citations. Figure 8 shows the distribution of DBT applications across these domains. All the papers in our study fall under one of the following subject domains.

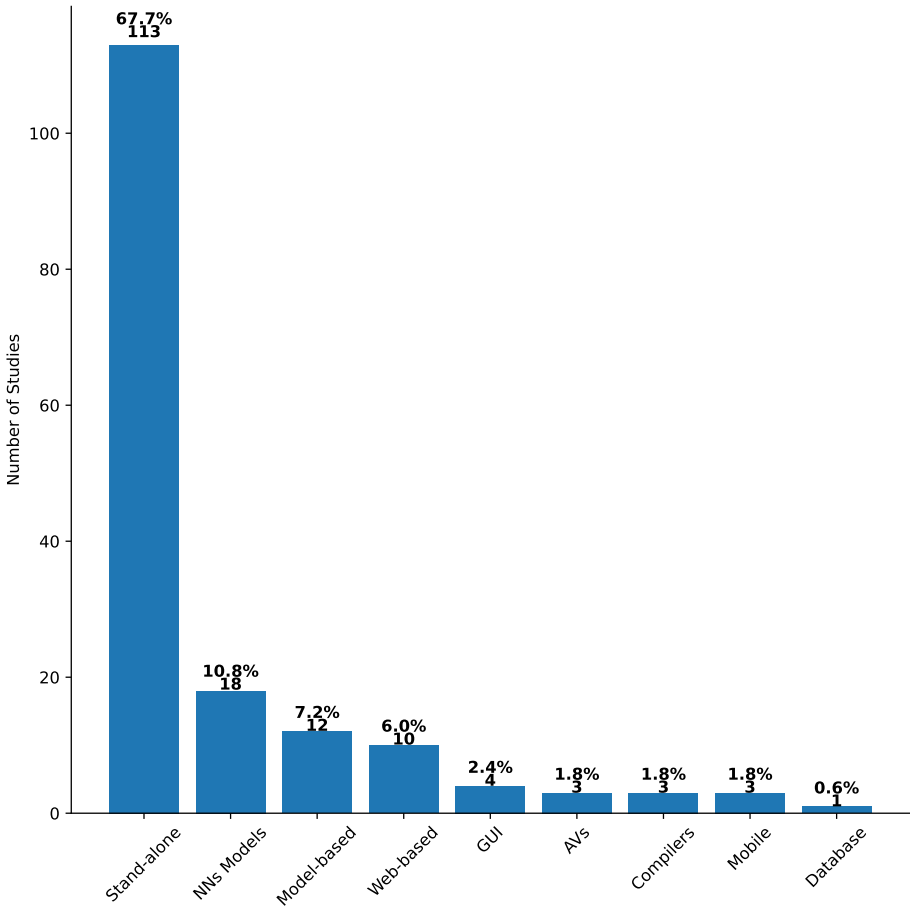


FIGURE 8 Distribution of the subject domains of the DBT papers.

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 in search-based software testing [5,112,52,155,8,110,131,197,198], mutation testing [201,202,204,68,185], continuous-integration environments [55,118,172], test case recommendation [135,140], and machine translation systems [28].

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 [167], used feedback-driven fuzzing [189], and exploration of deep program semantics [138]. Other approaches have aimed to enhance the diversity of tests in general [165], or reduce the number of generated tests [123].

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 [216] (Java), Software-artifact Infrastructure Repository [221] (C), TravisTorrent [222] (continuous integration platforms), NL-RX-Synth [223] (regular expressions for natural language), and BugsInPy [224] (Python).

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 [130], test suite reduction for MBT [22], test case selection for MBT [21,23,24,25,26,56], and the application of DBT techniques to Simulink models [12,13,14,111,96].

Most papers used case studies for evaluation but did not provide them. For example, Hemmati et al. [23,24,25,26] used a C++ safety-critical control system that is not available. Public domain Simulink models, such as the Cruise Controller [225] and Clutch

Lockup Controller from Mathworks[226], were used. Additionally, industrial Simulink models, Clutch Position Controller and Flap Position Controller, developed by Delphi Automotive Systems, were used[12,13].

7.3 | Web Applications

The realm of web testing has witnessed the utilisation of DBT techniques, as discussed in Section 6 and Section 5. Certain characteristics specific to web applications, such as the document object model (DOM)[48,102], HTML tag structure[10], long interacting sequences[49], exploration of web applications[193,219], 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[136], and an adaptive fuzzing method for testing Modbus TCP/IP[169], which is an application layer messaging protocol.

Popular JavaScript frameworks from GitHub[227] and web applications used in web testing[228,10] include FAQForge, Schoolmate, Webchess, PHPSysInfo, Timeclock, and PHPBB2.

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[95]. The regression test suite was based on a legacy database application[229].

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[41], generate GUI test cases[191,192], or investigated the reuse of GUI tests based on semantic similarities[134].

7.6 | Mobile Applications

Although limited, DBT techniques have also been applied to mobile application testing. Notably, Vogel et al.[159,160] 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.[137] 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 CRAFTDROID[134].

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[179]. Additionally, NNs have been targeted for testing using neuron coverage diversity for deep learning (DL) models[18,190] or through population diversity to generate diverse test cases[230]. Aghababaeyan et al.[42,43] and Jiang et al.[178] 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.[86] proposed a new test adequacy criterion for DL systems based on input diversity.

For test input prioritisation in DL models, Xie et al.[187] 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.[44] 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[231], MNIST[232], CIFAR10[233], and SVHN[234].

7.8 | Autonomous Driving Systems

Autonomous Driving Systems (ADSs) replace human drivers, which perform various driving tasks, such as perception, localisation, planning, and control[151]. Applying DBT techniques for testing ADSs is playing a major role recently. Neelofar et al.[121]

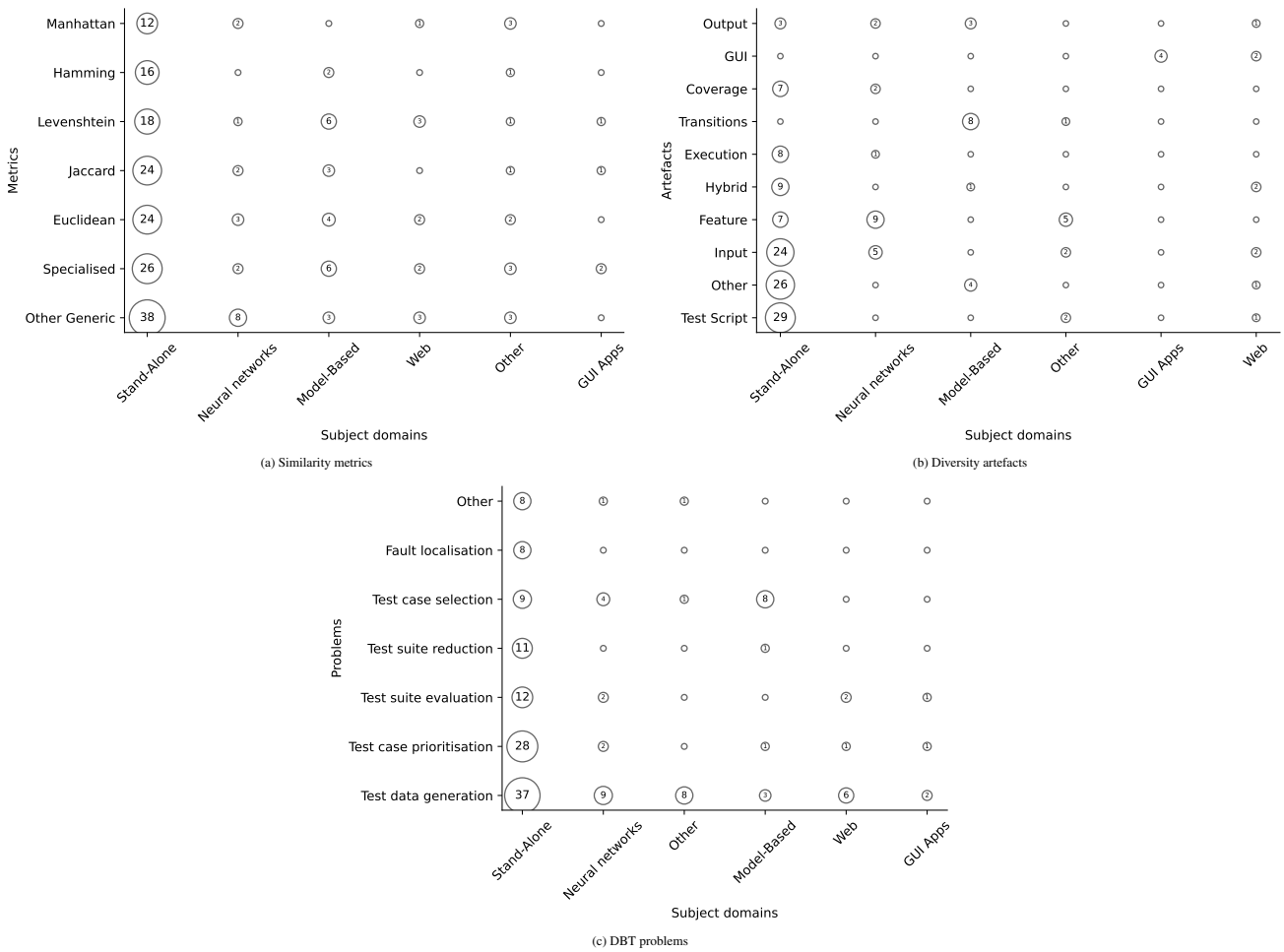


FIGURE 9 Relationship between subject domains and similarity metrics (a), diversity artefacts (b), and DBT problems (c), respectively

proposed new test adequacy metrics for autonomous vehicles based on coverage and diversity. Cheng et al.[151] developed a behaviour diversity-based technique for generating diverse scenarios to test autonomous vehicles. Wang et al.[180] 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[235] and LGSVL[236]. Some works in testing DL systems, mentioned in Section 7.7, used autonomous vehicles in their testing subjects[86,88,87,107].

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[45], testing Simulink compilers[47], and the detection of compiler warning defects[46]. 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.

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

Figure 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

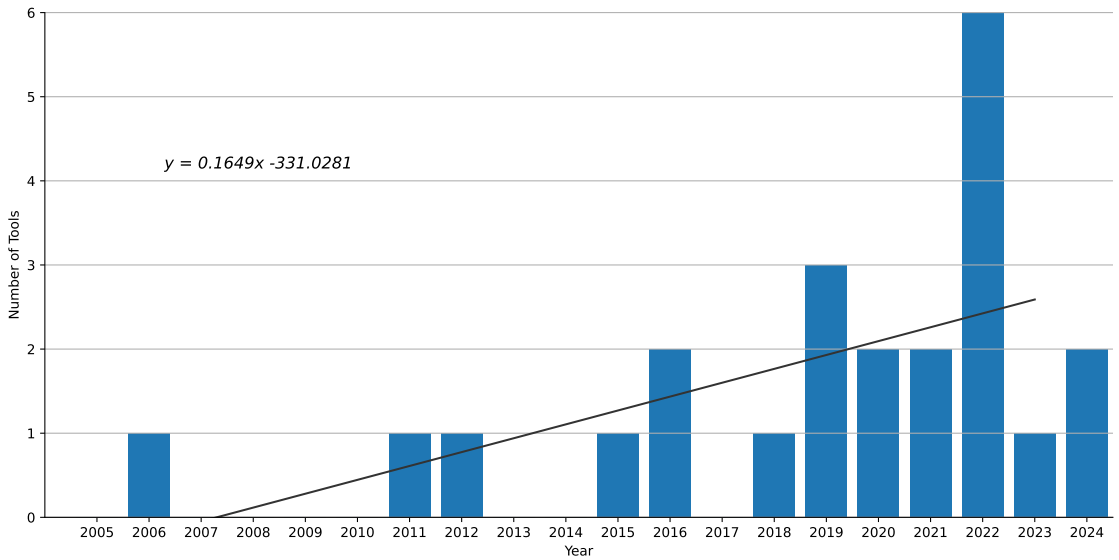


FIGURE 10 The number of diversity-based tools developed per year.

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 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 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.

8 | RQ6: TOOLS

What DBT tools have been developed by researchers?

In this section, we present the tools that were developed based on DBT techniques. Table 9 lists the tools encountered in the study. Figure 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"[138], "RLCheck"[167], and "BeDivFuzz"[189].

TABLE 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	183	237,238,239	✗
DART	Test case selection	Database Application	229	240,241,242	✗
SimFuzz	Test data generation	General-purpose Applications	138	243,244	✗
SIMTAC	Test case prioritisation	C & Access Control	173	-	✓
SimFL	Fault localisation	Simulink Models	245	97	✗
SimCoTest	Test data generation & Test case prioritisation	Simulink Models	246	14,247	✓
FAST	Test case prioritisation	C & Java programs	33	37,116	✓
HiCOND	Test data generation	Compilers	45	248,46,47	✓
DIG	Test data generation	Web Applications	48	249	✓
SAPIENZ^{DIV}	Test data generation	Mobile Applications	159	160	✓
SiMut	Test suite evaluation	Java programs	185	117	✗
RLCheck	Test data generation	Java programs	167	189,250	✓
DEEPMETIS	Test data generation	Deep Learning Systems	107	251,252	✓
DEEPHYPERION	Test data generation	Deep Learning Systems	88	252,87,253,254,255,256	✓
OutGen	Test data generation	General-purpose Applications	11	-	✓
DIPROM	Test data generation	Compilers	46	-	✓
BEDIVFUZZ	Test data generation	Java programs	189	-	✓
TSE	Test data generation	Java programs	135	-	✓
FASTAZI	Test case prioritisation	Java programs	116	-	✗
METALLICUS	Test adaptation	Python programs	205	-	✓
DEEPHYPERION-CS	Test data generation	Deep Learning Systems	87	257,258	✓
RECORD	Test data generation	Compilers	47	-	✓
DEEPWALK	Test data generation	Deep Learning Systems	171	-	✓

Furthermore, "OutGen"[11] caters for a broad range of programs and software systems, focusing on the generation of diverse output data. In compiler testing, there is "HiCOND"[45] a test-program generation tool that leverages historical data for compiler testing, "DIPROM"[46], which constructs warning-sensitive programs to detect compiler warning defects, "RECORD"[47] for testing Simulink compilers. Moreover, in test case recommendation, Test Suite Evolver (TSE)[135] 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"[48] (**DI**versity-based **G**enerator) has been introduced for generating test cases. For mobile applications, SAPIENZ^{DIV} [159] 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"[246], is a tool for generating tests for Simulink models. Additionally, "SiMut"[185] is notable for its ability to generate a reduced set of mutants, offering efficiency gains in mutation testing. Furthermore, DEEPHYPERION[88] and DEEPHYPERION-CS[87] are tools that are capable of generating failure-inducing inputs for deep learning systems. DEEPMETIS[107] is another tool capable of augmenting the test set of a deep learning system to improve ability to kill mutants. DEEPWALK[171] 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)[173], focuses on reordering test cases for Access Control systems based on their similarity. Similarly, "SimCoTest"[246] 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)[33] and "Fastazi"[116], 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[183] 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)[229] employs a similarity-based regression test reduction approach for large-scale database applications. Another tool, "SimFL" (Simulink Fault Localisation)[245], focuses

on fault localisation within Simulink models. Lastly, METALLICUS [205] 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.

9 | FUTURE RESEARCH DIRECTIONS

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

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.

, the inclusion of other types of features, such as color and texture features.

One of the challenges with DBT for test generation is managing complex data inputs (such as images [122] and abstract tree models [8]). 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 [8]. 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 [122]. 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.

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 [114,71] and others proposed solutions [33]. Generating and running numerous diverse test cases can be resource-intensive, demanding significant computational power and time. Additionally, analyzing the results from a wide range of test cases can be particularly time-consuming for complex systems.

9.3 | 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 7, and we 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 7.3, all of these address test data generation [49,10,195,259,48,193,219,136], 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. [160]. 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. [33], may be useful. Further investigations are needed to apply DBT techniques in mobile applications.

9.4 | Tool Support

As discussed in Section 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 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[246], test case prioritisation[246], and FL[245]. 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[48], and only a single tool for Mobile applications was developed[159]. 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[229] is for test case selection. There is no available DBT techniques or tools to generate test cases for such applications.

9.5 | Exploring Other Diversity Artefacts

Most current works focused on input and test script diversity as shown in Section 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.[52], 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.[124] 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 works 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.

10 | CONCLUSIONS

In this study we covered 167 papers that described the use of DBT techniques to solve a software testing problem. Our systematic mapping study discussed DBT publication trends showing the number of DBT publications per year, and we presented the prominent venues, papers and authors in the field. We 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, we 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, we 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, we 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, we 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.

REFERENCES

1. Ammann P, Offutt J. *Introduction to software testing*. Cambridge University Press, 2016.
2. Myers GJ, Sandler C, Badgett T. *The art of software testing*. John Wiley & Sons, 2011.
3. Utting M, Legeard B. *Practical model-based testing: a tools approach*. Elsevier, 2010.
4. Hamlet D, Taylor R. Partition testing does not inspire confidence (program testing). *IEEE Transactions on Software engineering.* 1990;16(12):1402–1411. 10.1109/32.62448.
5. Bueno PM, Wong WE, Jino M. Automatic test data generation using particle systems. In: *Proceedings of the symposium on Applied computing* 2008:809–814. 10.1145/1363686.1363871.
6. Feldt R, Poulding S, Clark D, Yoo S. Test set diameter: Quantifying the diversity of sets of test cases. In: *International Conference on Software Testing, Verification and Validation (ICST)* 2016:223–233. 10.1109/ICST.2016.33.
7. Shahbazi A, Miller J. Extended Subtree: A New Similarity Function for Tree Structured Data. *Transactions on Knowledge and Data Engineering.* 2014;26(4):864 - 877. 10.1109/TKDE.2013.53.
8. Shahbazi A, Miller J. Black-box string test case generation through a multi-objective optimization. *Transactions on Software Engineering.* 2015;42(4):361–378. 10.1109/TSE.2015.2487958.
9. Shahbazi A, Panahandeh M, Miller J. Black-box tree test case generation through diversity. *Automated Software Engineering.* 2018;25(3):531–568. 10.1007/s10515-018-0232-y.
10. Alshahwan N, Harman M. Coverage and fault detection of the output-uniqueness test selection criteria. In: *Proceedings of the International Symposium on Software Testing and Analysis* 2014:181–192. 10.1145/2610384.2610413.
11. Benito HM, Boreale M, Gorla D, Clark D. Output sampling for output diversity in automatic unit test generation. *Transactions on Software Engineering.* 2020;48(1):295-308. 10.1109/TSE.2020.2987377.
12. Matinnejad R, Nejati S, Briand LC, Bruckmann T. Effective test suites for mixed discrete-continuous stateflow controllers. In: *Proceedings of the Joint Meeting on Foundations of Software Engineering* 2015:84–95. 10.1145/2786805.2786818.
13. Matinnejad R, Nejati S, Briand LC, Bruckmann T. Automated test suite generation for time-continuous simulink models. In: *proceedings of the International Conference on Software Engineering (ICSE)* 2016:595–606. 10.1145/2884781.2884797.
14. Matinnejad R, Nejati S, Briand LC, Bruckmann T. Test generation and test prioritization for simulink models with dynamic behavior. *Transactions on Software Engineering.* 2019;45(9):919–944. 10.1109/TSE.2018.2811489.
15. Beena R, Sarala S. Multi objective test case minimization collaborated with clustering and minimal hitting set. *Journal of Theoretical and Applied Information Technology.* 2014;69(1):200–210.
16. Wang R, Jiang S, Chen D. Similarity-based regression test case prioritization. In: *International Conference on Software Engineering and Knowledge Engineering (SEKE)* 2015:358–363. 10.18293/seke2015-115.
17. You YS, Huang CY, Peng KL, Hsu CJ. Evaluation and analysis of spectrum-based fault localization with modified similarity coefficients for software debugging. In: *Annual Computer Software and Applications Conference* 2013:180–189. 10.1109/COMPSAC.2013.32.
18. Zhang Z, Xie X. On the investigation of essential diversities for deep learning testing criteria. In: *International Conference on Software Quality, Reliability and Security (QRS)* 2019:394–405. 10.1109/QRS.2019.00056.
19. Arafeen MJ, Do H. Test case prioritization using requirements-based clustering. In: *International Conference on Software Testing, Validation and Verification (ICST)* 2013:312–321. 10.1109/ICST.2013.12.
20. Masuda S, Matsuodani T, Tsuda K. Syntax-Tree Similarity for Test-Case Derivability in Software Requirements. In: *International Workshop on Software Test Architecture (InSTA)* 2021:162–172. 10.1109/ICSTW52544.2021.00037.
21. Cartaxo EG, Machado PD, Oliveira Neto dFG. On the use of a similarity function for test case selection in the context of model-based testing. *Software Testing, Verification and Reliability.* 2009;21(2):75–100. 10.1002/stvr.413.
22. Coutinho AVB, Cartaxo EG, Lima Machado dPD. Analysis of distance functions for similarity-based test suite reduction in the context of model-based testing. *Software Quality Journal.* 2016;24(2):407–445. 10.1007/s11219-014-9265-z.
23. Hemmati H, Briand L, Arcuri A, Ali S. An enhanced test case selection approach for model-based testing: an industrial case study. In: *Proceedings of the SIGSOFT international symposium on Foundations of software engineering* 2010:267–276. 10.1145/1882291.1882331.
24. Hemmati H, Briand L. An industrial investigation of similarity measures for model-based test case selection. In: *International Symposium on Software Reliability Engineering* 2010:141–150. 10.1109/ISSRE.2010.9.
25. Hemmati H, Arcuri A, Briand L. Reducing the cost of model-based testing through test case diversity. In: *IFIP International Conference on Testing Software and Systems* 2010:63–78. 10.1007/978-3-642-16573-3_6.
26. Hemmati H, Arcuri A, Briand L. Achieving scalable model-based testing through test case diversity. *Transactions on Software Engineering and Methodology (TOSEM).* 2013;22(1):1–42. 10.1145/2430536.2430540.
27. Bueno PM, Wong WE, Jino M. Improving random test sets using the diversity oriented test data generation. In: *Proceedings of the international workshop on Random testing* 2007:10–17. 10.1145/1292414.1292419.
28. Cao J, Li M, Li Y, Wen M, Cheung S, Chen H. SemMT: a semantic-based testing approach for machine translation systems. *ACM Transactions on Software Engineering and Methodology (TOSEM).* 2022;31(2):1–36. 10.1145/3490488.
29. Abd Halim S, Jawawi DNA, Sahak M. Similarity distance measure and prioritization algorithm for test case prioritization in software product line testing. *Journal of Information and Communication Technology.* 2019;18(1):57–75.
30. Henard C, Papadakis M, Perrouin G, Klein J, Le Traon Y. Assessing software product line testing via model-based mutation: An application to similarity testing. In: *Advances in Model Based Testing workshop (A-MOST)* 2013:188–197. 10.1109/ICSTW.2013.30.
31. Henard C, Papadakis M, Perrouin G, Klein J, Heymans P, Le Traon Y. Bypassing the combinatorial explosion: Using similarity to generate and prioritize t-wise test configurations for software product lines. *Transactions on Software Engineering.* 2014;40(7):650–670. 10.1109/TSE.2014.2327020.
32. Ledru Y, Petrenko A, Boroday S, Mandran N. Prioritizing test cases with string distances. *Automated Software Engineering.* 2012;19(1):65–95. 10.1007/s10515-011-0093-0.
33. Miranda B, Cruciani E, Verdecchia R, Bertolino A. FAST approaches to scalable similarity-based test case prioritization. In: *International Conference on Software Engineering (ICSE)* 2018:222–232. 10.1145/3180155.3180210.
34. Yoo S, Harman M, Tonella P, Susi A. Clustering test cases to achieve effective and scalable prioritisation incorporating expert knowledge. In: *Proceedings of the international symposium on Software testing and analysis* 2009:201–212. 10.1145/1572272.1572296.

35. Chetouane N, Wotawa F, Felbinger H, Nica M. On using k-means clustering for test suite reduction. In: *Workshop on Testing: Academia-Industry Collaboration, Practice and Research Techniques (TAIC)* 2020:380–385. 10.1109/ICSTW50294.2020.00068.
36. Coviello C, Romano S, Scanniello G, Marchetto A, Antoniol G, Corazza A. Clustering support for inadequate test suite reduction. In: *International Conference on Software Analysis, Evolution and Reengineering (SANER)* 2018:95–105. 10.1109/SANER.2018.8330200.
37. Cruciani E, Miranda B, Verdecchia R, Bertolino A. Scalable approaches for test suite reduction. In: *International Conference on Software Engineering (ICSE)* 2019:419–429. 10.1109/ICSE.2019.00055.
38. Cao Y, Zhou ZQ, Chen TY. On the correlation between the effectiveness of metamorphic relations and dissimilarities of test case executions. In: *International Conference on Quality Software* 2013:153–162. 10.1109/QSIC.2013.43.
39. Oliveira Neto dFG, Feldt R, Erlenhov L, Nunes JBDS. Visualizing test diversity to support test optimisation. In: *Asia-Pacific Software Engineering Conference (APSEC)* 2018:149–158. 10.1109/APSEC.2018.00029.
40. Shi Q, Chen Z, Fang C, Feng Y, Xu B. Measuring the diversity of a test set with distance entropy. *Transactions on Reliability.* 2015;65(1):19–27. 10.1109/TR.2015.2434953.
41. Xie Q, Memon AM. Studying the characteristics of a "Good" GUI test suite. In: *International Symposium on Software Reliability Engineering* 2006:159–168. 10.1109/ISSRE.2006.45.
42. Aghababaeyan Z, Abdellatif M, Briand L, Ramesh S, Bagherzadeh M. Black-box testing of deep neural networks through test case diversity. *IEEE Transactions on Software Engineering.* 2023;49(5):3182-3204. 10.1109/TSE.2023.3243522.
43. Aghababaeyan Z, Abdellatif M, Dadkhah M, Briand L. Deepgd: A multi-objective black-box test selection approach for deep neural networks. *ACM Transactions on Software Engineering and Methodology.* 2024;33(6). 10.1145/3644388.
44. Mosin V, Staron M, Durisic D, Oliveira Neto dFG, Pandey SK, Koppisetty AC. Comparing Input Prioritization Techniques for Testing Deep Learning Algorithms. In: *Euromicro Conference on Software Engineering and Advanced Applications (SEAA)* 2022:76–83. 10.1109/SEAA56994.2022.00020.
45. Chen J, Wang G, Hao D, Xiong Y, Zhang H, Zhang L. History-guided configuration diversification for compiler test-program generation. In: *International Conference on Automated Software Engineering (ASE)* 2019:305–316. 10.1109/ASE.2019.00037.
46. Tang Y, Jiang H, Zhou Z, Li X, Ren Z, Kong W. Detecting compiler warning defects via diversity-guided program mutation. *Transactions on Software Engineering.* 2022;48(11):4411-4432. 10.1109/TSE.2021.3119186.
47. Li X, Guo S, Cheng H, Jiang H. Simulink Compiler Testing via Configuration Diversification With Reinforcement Learning. *IEEE Transactions on Reliability.* 2024;73(2):1060-1074. 10.1109/TR.2023.3317643.
48. Biagiola M, Stocco A, Ricca F, Tonella P. Diversity-based web test generation. In: *Proceedings of the Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* 2019:142–153. 10.1145/3338906.3338970.
49. Marchetto A, Tonella P. Search-based testing of Ajax web applications. In: *international symposium on search based software engineering* 2009:3–12. 10.1109/SSBSE.2009.13.
50. Fang C, Chen Z, Wu K, Zhao Z. Similarity-based test case prioritization using ordered sequences of program entities. *Software Quality Journal.* 2014;22(2):335–361. 10.1007/s11219-013-9224-0.
51. Noor TB, Hemmati H. A similarity-based approach for test case prioritization using historical failure data. In: *International Symposium on Software Reliability Engineering (ISSRE)* 2015:58–68. 10.1109/ISSRE.2015.7381799.
52. Feldt R, Torkar R, Gorschek T, Afzal W. Searching for cognitively diverse tests: Towards universal test diversity metrics. In: *International Conference on Software Testing Verification and Validation Workshop* 2008:178–186. 10.1109/ICSTW.2008.36.
53. Hemmati H, Fang Z, Mantyla MV. Prioritizing manual test cases in traditional and rapid release environments. In: *International Conference on Software Testing, Verification and Validation (ICST)* 2015:1–10. 10.1109/ICST.2015.7102602.
54. Henard C, Papadakis M, Harman M, Jia Y, Le Traon Y. Comparing white-box and black-box test prioritization. In: *International Conference on Software Engineering (ICSE)* 2016:523–534. 10.1145/2884781.2884791.
55. Oliveira Neto dFG, Ahmad A, Leifler O, Sandahl K, Enoiu E. Improving continuous integration with similarity-based test case selection. In: *Proceedings of the International Workshop on Automation of Software Test* 2018:39–45. 10.1145/3194733.3194744.
56. Oliveira Neto dFG, Torkar R, Machado PD. Full modification coverage through automatic similarity-based test case selection. *Information and Software Technology.* 2016;80:124–137. 10.1016/j.infsof.2016.08.008.
57. Tahvili S, Hatvani L, Felderer M, Afzal W, Saadatmand M, Bohlin M. Cluster-based test scheduling strategies using semantic relationships between test specifications. In: *Proceedings of the International Workshop on Requirements Engineering and Testing* 2018:1–4. 10.1145/3195538.3195540.
58. Huang R, Sun W, Xu Y, Chen H, Towey D, Xia X. A survey on adaptive random testing. *Transactions on Software Engineering.* 2019;47(10):2052–2083. 10.1109/TSE.2019.2942921.
59. Chan K, Chen TY, Towey D. Restricted random testing. In: *European Conference on Software Quality* 2002:321–330. 10.1007/3-540-47984-8_35.
60. Chen TY, Merkel R, Wong P, Eddy G. Adaptive random testing through dynamic partitioning. In: *Proceedings of the International Conference on Quality Software (QSIC)* 2004:79–86. 10.1109/QSIC.2004.1357947.
61. Mayer J, Chen TY, Huang DH. Adaptive random testing through iterative partitioning revisited. In: *Proceedings of the international workshop on Software quality assurance* 2006:22–29.
62. Sabor KK, Thiel S. Adaptive random testing by static partitioning. In: *International Workshop on Automation of Software Test* 2015:28–32. 10.1109/AST.2015.13.
63. Petersen K, Feldt R, Mujtaba S, Mattsson M. Systematic Mapping Studies in Software Engineering. In: *Proceedings of the International Conference on Evaluation and Assessment in Software Engineering* 2008:68–77. 10.14236/ewic/EASE2008.8.
64. Kitchenham BA, Charters S. Guidelines for performing systematic literature reviews in software engineering. *Technical report, Ver. 2.3 EBSE technical report. EBSE.* 2007.
65. Elgendy IT. A Systematic Mapping Study of diversity-based approached in software testing: Bibliography. https://github.com/islamelgendy/DBT-bibliography; 2024. [Online; accessed 27-June-2024].
66. Wohlin C. 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* 2014:1–10. 10.1145/2601248.2601268.
67. Coviello C, Romano S, Scanniello G. An empirical study of inadequate and adequate test suite reduction approaches. In: *Proceedings of the international symposium on empirical software engineering and measurement* 2018:1–10. 10.1145/3239235.3240497.

68. Oliveira Neto dFG, Dobslaw F, Feldt R. Using mutation testing to measure behavioural test diversity. In: *International Workshop on Mutation Analysis* 2020:254–263. 10.1109/ICSTW50294.2020.00051.
69. Ahmad A, Oliveira Neto dFG, Enoiu EP, Sandahl K, Leifler O. An Industrial Study on the Challenges and Effects of Diversity-Based Testing in Continuous Integration. In: *International Conference on Software Quality, Reliability, and Security (QRS)* 2023:337-347. 10.1109/QRS60937.2023.00041.
70. Cartaxo EG, Oliveira Neto dFG, Machado PD. Automated test case selection based on a similarity function. In: *Informatik trifft Logistik*. P-110. 2007:381–386.
71. Khojah R, Chao CH, Oliveira Neto dFG. Evaluating the Trade-offs of Text-based Diversity in Test Prioritisation. In: *International Conference on Automation of Software Test (AST)* 2023:168-178. 10.1109/AST58925.2023.00021.
72. Lin P. CORE Rankings Portal. https://www.core.edu.au/conference-portal; 2024. [Online; accessed 2-July-2024].
73. Watson C, Cooper N, Palacio DN, Moran K, Poshyvanyk D. A systematic literature review on the use of deep learning in software engineering research. *ACM Transactions on Software Engineering and Methodology (TOSEM).* 2022;31(2):1–58. 10.1145/3485275.
74. Švábenskỳ V, Vykopal J, Čeleda P. 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* 2020:2–8. 10.1145/3328778.3366816.
75. Reda HT, Anwar A, Mahmood A. Comprehensive survey and taxonomies of false data injection attacks in smart grids: attack models, targets, and impacts. *Renewable and Sustainable Energy Reviews.* 2022;163:112423. 10.1016/j.rser.2022.112423.
76. Liu Q, Chan KC, Chimhundu R. Fintech research: systematic mapping, classification, and future directions. *Financial Innovation.* 2024;10(1):24. 10.1186/s40854-023-00524-z.
77. Liu J, Tian J, Kong X, Lee I, Xia F. Two decades of information systems: a bibliometric review. *Scientometrics.* 2019;118:617–643. 10.1007/s11192-018-2974-5.
78. Xie X, Xu B, Shi L, Nie C, He Y. A dynamic optimization strategy for evolutionary testing. In: *Asia-Pacific Software Engineering Conference (APSEC)* 2005:568–575. 10.1109/APSEC.2005.6.
79. Hao D, Pan Y, Zhang L, Zhao W, Mei H, Sun J. A similarity-aware approach to testing based fault localization. In: *Proceedings of the international Conference on Automated software engineering* 2005:291–294. 10.1145/1101908.1101953.
80. Chen TY, Leung H, Mak I. Adaptive random testing. In: *Annual Asian Computing Science Conference* 2004:320–329. 10.1007/978-3-540-30502-6_23.
81. Chan K, Chen TY, Kuo F, Towey D. A revisit of adaptive random testing by restriction. In: *Proceedings of the International Computer Software and Applications Conference (COMPSAC).* 2004:78–85. 10.1109/CMPSAC.2004.1342809.
82. Chen TY, Kuo FC, Merkel RG, Ng SP. Mirror adaptive random testing. *Information and Software Technology.* 2004;46(15):1001–1010. 10.1016/j.infsof.2004.07.004.
83. Chen TY, Tse T, Yu YT. Proportional sampling strategy: A compendium and some insights. *Journal of Systems and Software.* 2001;58(1):65–81. 10.1016/S0164-1212(01)00028-0.
84. Chen TY, Huang DH. Adaptive random testing by localization. In: *Asia-Pacific Software Engineering Conference* 2004:292–298. 10.1109/APSEC.2004.17.
85. Google . Google scholar. https://scholar.google.com/; 2024.
86. Kim J, Feldt R, Yoo S. Guiding deep learning system testing using surprise adequacy. In: *International Conference on Software Engineering (ICSE)* 2019:1039–1049. 10.1109/ICSE.2019.00108.
87. Zohdinasab T, Riccio V, Gambi A, Tonella P. Efficient and effective feature space exploration for testing deep learning systems. *ACM Transactions on Software Engineering and Methodology.* 2023;32(2):1–38. 10.1145/3544792.
88. Zohdinasab T, Riccio V, Gambi A, Tonella P. Deephyperion: exploring the feature space of deep learning-based systems through illumination search. In: *Proceedings of the International Symposium on Software Testing and Analysis* 2021:79–90. 10.1145/3460319.3464811.
89. Briand L. Personal web page. https://www.lbriand.info/; Retrived May. 11, 2023.
90. Feldt R. Personal web page. http://www.robertfeldt.net/; Retrived May. 11, 2023.
91. Oliveira Neto dFG. Personal web page. https://www.franciscogon.com/; Retrived May. 11, 2023.
92. Hemmati H. Personal web page. https://lassonde.yorku.ca/users/hhemmati; Retrived May. 11, 2023.
93. Hemmati H, Arcuri A, Briand L. Empirical investigation of the effects of test suite properties on similarity-based test case selection. In: *International Conference on Software Testing, Verification and Validation* 2011:327–336. 10.1109/ICST.2011.12.
94. Dobslaw F, Oliveira N. dFG, Feldt R. Boundary value exploration for software analysis. In: *Workshop on NEXt Level of Test Automation (NEXTA)* 2020:346–353. 10.1109/ICSTW50294.2020.00062.
95. Rogstad E, Briand L, Torkar R. Test case selection for black-box regression testing of database applications. *Information and Software Technology.* 2013;55(10):1781–1795. 10.1016/j.infsof.2013.04.004.
96. Liu B, Lucia , Nejati S, Briand LC. Improving fault localization for Simulink models using search-based testing and prediction models. In: *International Conference on Software Analysis, Evolution and Reengineering (SANER)* 2017:359–370. 10.1109/SANER.2017.7884636.
97. Liu B, Nejati S, Lucia , Briand LC. Effective fault localization of automotive Simulink models: achieving the trade-off between test oracle effort and fault localization accuracy. *Empirical Software Engineering.* 2019;24(1):444–490. 10.1007/s10664-018-9611-z.
98. Pan R, Ghaleb TA, Briand L. Atm: Black-box test case minimization based on test code similarity and evolutionary search. In: *International Conference on Software Engineering (ICSE)* 2023:1700–1711. 10.1109/ICSE48619.2023.00146.
99. Attaoui M, Fahmy H, Pastore F, Briand L. Black-box safety analysis and retraining of DNNs based on feature extraction and clustering. *ACM Transactions on Software Engineering and Methodology.* 2023;32(3):1–40. 10.1145/3550271.
100. Marculescu B, Feldt R, Torkar R. Using exploration focused techniques to augment search-based software testing: An experimental evaluation. In: *International Conference on Software Testing, Verification and Validation (ICST)* 2016:69–79. 10.1109/ICST.2016.26.
101. Poulding S, Feldt R. Automated random testing in multiple dispatch languages. In: *International Conference on Software Testing, Verification and Validation (ICST)* 2017:333–344. 10.1109/ICST.2017.37.
102. Nass M, Alégroth E, Feldt R, Leotta M, Ricca F. Similarity-based web element localization for robust test automation. *ACM Transactions on Software Engineering and Methodology.* 2023;32(3):1–30. 10.1145/3571855.
103. Feldt R, Dobslaw F. Towards automated boundary value testing with program derivatives and search. In: *International Symposium on Search Based Software Engineering* 2019:155–163. 10.1007/978-3-030-27455-9_11.

104. Mondal D, Hemmati H, Durocher S. Exploring test suite diversification and code coverage in multi-objective test case selection. In: *International Conference on Software Testing, Verification and Validation (ICST)* 2015:1–10. 10.1109/ICST.2015.7102588.
105. Kifetew FM, Panichella A, De Lucia A, Oliveto R, Tonella P. Orthogonal exploration of the search space in evolutionary test case generation. In: *Proceedings of the International Symposium on Software Testing and Analysis* 2013:257–267. 10.1145/2483760.2483789.
106. Menéndez HD, Jahangirova G, Sarro F, Tonella P, Clark D. Diversifying focused testing for unit testing. *Transactions on Software Engineering and Methodology (TOSEM).* 2021;30(4):1–24. 10.1145/3447265.
107. Riccio V, Humbatova N, Jahangirova G, Tonella P. Deepmetis: Augmenting a deep learning test set to increase its mutation score. In: *International Conference on Automated Software Engineering (ASE)* 2021:355–367. 10.1109/ASE51524.2021.9678764.
108. Huang R, Zhou Y, Zong W, Towey D, Chen J. An empirical comparison of similarity measures for abstract test case prioritization. In: *Annual Computer Software and Applications Conference (COMPSAC)*. 1. 2017:3–12
109. Alpha W. EuclideanDistance - Wolfram: Alpha. https://mathworld.wolfram.com/EuclideanMetric.html; 2023.
110. Albunian N, Fraser G, Sudholt D. Measuring and maintaining population diversity in search-based unit test generation. In: *International Symposium on Search Based Software Engineering* 2020:153–168. 10.1007/978-3-030-59762-7_11.
111. Arrieta A, Wang S, Markiegi U, Arruabarrena A, Etxeberria L, Sagardui G. Pareto efficient multi-objective black-box test case selection for simulation-based testing. *Information and Software Technology.* 2019;114:137–154. 10.1016/j.infsof.2019.06.009.
112. Bueno PM, Jino M, Wong WE. Diversity oriented test data generation using metaheuristic search techniques. *Information sciences.* 2014;259:490–509. 10.1016/j.ins.2011.01.025.
113. Carlson R, Do H, Denton A. A clustering approach to improving test case prioritization: An industrial case study. In: *International Conference on Software Maintenance (ICSM)* 2011:382–391. 10.1109/ICSM.2011.6080805.
114. Elgendy I, Hierons R, McMinn P. Evaluating String Distance Metrics for Reducing Automatically Generated Test Suites. In: *Proceedings of the International Conference on Automation of Software Test (AST)* 2024:171–181.
115. Farzat Fd. Test case selection method for emergency changes. In: *Proceedings of the International Symposium on Search Based Software Engineering* 2010:31–35. 10.1109/SSBSE.2010.13.
116. Greca R, Miranda B, Gligoric M, Bertolino A. 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* 2022:115–125. 10.1145/3524481.3527223.
117. Guarnieri GF, Pizzoleto AV, Ferrari FC. An Automated Framework for Cost Reduction of Mutation Testing Based on Program Similarity. In: *International Workshop on Mutation Analysis* 2022:179–188. 10.1109/ICSTW55395.2022.00041.
118. Haghighatkhah A, Mäntylä M, Oivo M, Kuvaja P. Test prioritization in continuous integration environments. *Journal of Systems and Software.* 2018;146:80–98. 10.1016/j.jss.2018.08.061.
119. Lan W, Cui Z, Zhang J, Yang J, Gu X. Fuzz Testing Based on Seed Diversity Analysis. In: *IEEE International Conference on Systems, Man, and Cybernetics (SMC)* 2023:145–150. 10.1109/SMC53992.2023.10394486.
120. Mahdieh M, Mirian-Hosseinabadi S, Mahdieh M. Test case prioritization using test case diversification and fault-proneness estimations. *Automated Software Engineering.* 2022;29(2):50. 10.1007/s10515-022-00344-y.
121. Neelofar N, Aleti A. 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* 2024:1–12. 10.1145/3597503.3623314.
122. Nunes FL, Delamaro ME, Gonçalves VM, Lauretto MDS. CBIR based testing oracles: an experimental evaluation of similarity functions. *International Journal of Software Engineering and Knowledge Engineering.* 2015;25(08):1271–1306. 10.1142/S0218194015500254.
123. Pei Y, Christi A, Fern X, Groce A, Wong W. Taming a Fuzzer Using Delta Debugging Trails. In: *International Conference on Data Mining Workshop (ICDMW)* 2014:840–843. 10.1109/ICDMW.2014.58.
124. Shimari K, Tanaka M, Ishio T, Matsushita M, Inoue K, Takanezawa S. Selecting Test Cases based on Similarity of Runtime Information: A Case Study of an Industrial Simulator. In: *International Conference on Software Maintenance and Evolution (ICSME)* 2022:564–567. 10.1109/ICSME55016.2022.00077.
125. Yu X, Jia K, Hu W, Tian J, Xiang J. Black-Box Test Case Prioritization Using Log Analysis and Test Case Diversity. In: *International Symposium on Software Reliability Engineering Workshops (ISSREW)* 2023:186–191. 10.1109/ISSREW60843.2023.00072.
126. Zhao X, Wang Z, Fan X, Wang Z. A clustering-bayesian network based approach for test case prioritization. In: *Annual Computer Software and Applications Conference*. 3. 2015:542–547. 10.1109/COMPSAC.2015.154.
127. Levenshtein VI. Binary codes capable of correcting deletions, insertions, and reversals. In: *Soviet physics doklady*. 10. 1966:707–710.
128. Adigun JG, H. TP, Camilli M, Felderer M. Risk-driven Online Testing and Test Case Diversity Analysis for ML-enabled Critical Systems. In: *International Symposium on Software Reliability Engineering (ISSRE)* 2023:344–354. 10.1109/ISSRE59848.2023.00017.
129. Almulla H, Gay G. Generating diverse test suites for Gson through adaptive fitness function selection. In: *International Symposium on Search Based Software Engineering* 2020:246–252. 10.1007/978-3-030-59762-7_18.
130. Asoudeh N, Labiche Y. A multi-objective genetic algorithm for generating test suites from extended finite state machines. In: *International Symposium on Search Based Software Engineering* 2013:288–293. 10.1007/978-3-642-39742-4_26.
131. Boussaa M, Barais O, Sunyé G, Baudry B. A novelty search approach for automatic test data generation. In: *International Workshop on Search-Based Software Testing* 2015:40–43. 10.1109/SBST.2015.17.
132. Flemström D, Afzal W, Sundmark D. Exploring test overlap in system integration: An industrial case study. In: *Euromicro Conference on Software Engineering and Advanced Applications (SEAA)* 2016:303–312. 10.1109/SEAA.2016.34.
133. Ma L, Wu P, Chen TY. Diversity driven adaptive test generation for concurrent data structures. *Information and software technology.* 2018;103:162–173. 10.1016/j.infsof.2018.07.001.
134. Mariani L, Mohebbi A, Pezzè M, Terragni V. Semantic matching of gui events for test reuse: are we there yet?. In: *Proceedings of the SIGSOFT International Symposium on Software Testing and Analysis* 2021:177–190. 10.1145/3460319.3464827.
135. Shimmi S, Rahimi M. Leveraging code-test co-evolution patterns for automated test case recommendation. In: *Proceedings of the International Conference on Automation of Software Test* 2022:65–76. 10.1145/3524481.3527222.
136. Wang W, Wu S, Li Z, Zhao R. Parallel evolutionary test case generation for web applications. *Information and Software Technology.* 2023;155:107113. 10.1016/j.infsof.2022.107113.
137. Wang W, Guo J, Li B, Shang Y, Zhao R. EFSM Model-Based Testing for Android Applications. *International Journal of Software Engineering and Knowledge Engineering.* 2024;34(04):597–621. 10.1142/S0218194023500638.

138. Zhang D, Liu D, Lei Y, et al. SimFuzz: Test case similarity directed deep fuzzing. *Journal of Systems and Software.* 2012;85(1):102–111. 10.1016/j.jss.2011.07.028.
139. Jaccard P. Étude comparative de la distribution florale dans une portion des alpes et des jura. *Bulletin de la Société Vaudoise des Sciences Naturelles.* 1901;n/a(37):547–579.
140. Aman H, Nakano T, Ogasawara H, Kawahara M. A test case recommendation method based on morphological analysis, clustering and the Mahalanobis-Taguchi method. In: *Workshop on Testing: Academia-Industry Collaboration, Practice and Research Techniques (TAIC)* 2017:29–35. 10.1109/ICSTW.2017.9.
141. Fischer S, Lopez-Herrejon RE, Ramler R, Egyed A. A preliminary empirical assessment of similarity for combinatorial iteraction testing of software product lines. In: *International Workshop on Search-Based Software Testing (SBST)* 2016:15–18. 10.1145/2897010.289701.
142. Gao X, Feng Y, Yin Y, Liu Z, Chen Z, Xu B. Adaptive test selection for deep neural networks. In: *Proceedings of the International Conference on Software Engineering* 2022:73–85. 10.1145/3510003.3510232.
143. Haghighatkhah A, Mäntylä M, Oivo M, Kuvaja P. Test case prioritization using test similarities. In: *International Conference on Product-Focused Software Process Improvement* 2018:243–259. 10.1007/978-3-030-03673-7_18.
144. Lee M, Cha S, Oh H. Learning seed-adaptive mutation strategies for greybox fuzzing. In: *International Conference on Software Engineering (ICSE)* 2023:384–396. 10.1109/ICSE48619.2023.00043.
145. Mei L, Cai Y, Jia C, Jiang B, Chan W. Test pair selection for test case prioritization in regression testing for WS-BPEL programs. *International Journal of Web Services Research (IJWSR).* 2013;10(1):73–102. 10.4018/jwsr.2013010104.
146. Miranda B, Bertolino A. Social coverage for customized test adequacy and selection criteria. In: *Proceedings of the International Workshop on Automation of Software Test* 2014:22–28. 10.1145/2593501.2593505.
147. Viggiato M, Paas D, Buzon C, Bezemer C. Identifying similar test cases that are specified in natural language. *Transactions on Software Engineering.* 2023;49(3):1027-1043. 10.1109/TSE.2022.3170272.
148. Xiang Y, Huang H, Li M, Li S, Yang X. Looking for novelty in search-based software product line testing. *Transactions on Software Engineering.* 2021;1(1):1–1. 10.1109/TSE.2021.3057853.
149. Zhao L, Zhang Z, Wang L, Yin X. A fault localization framework to alleviate the impact of execution similarity. *International Journal of Software Engineering and Knowledge Engineering.* 2013;23(07):963–998. 10.1142/S0218194013500289.
150. Hamming RW. Error detecting and error correcting codes. *The Bell system technical journal.* 1950;29(2):147–160. 10.1002/j.1538-7305.1950.tb00463.x.
151. Cheng M, Zhou Y, Xie X. Behavexplor: Behavior diversity guided testing for autonomous driving systems. In: *Proceedings of the International Symposium on Software Testing and Analysis* 2023:488–500. 10.1145/3597926.3598072.
152. Feng Y, Chen Z, Jones JA, Fang C, Xu B. Test report prioritization to assist crowdsourced testing. In: *Proceedings of the Joint Meeting on Foundations of Software Engineering* 2015:225–236. 10.1145/2786805.2786862.
153. Wang H, Chan W. Weaving context sensitivity into test suite construction. In: *International Conference on Automated Software Engineering* 2009:610–614. 10.1109/ASE.2009.79.
154. Wang H, Chan W, Tse T. Improving the effectiveness of testing pervasive software via context diversity. *Transactions on Autonomous and Adaptive Systems (TAAS).* 2014;9(2):1–28. 10.1145/2620000.
155. Xie X, Xu B, Shi L, Nie C. Genetic test case generation for path-oriented testing. *Journal of Software.* 2009;20(12):3117–3136. 10.3724/SP.J.1001.2009.00580.
156. Alpha W. ManhattanDistance - Wolfram: Alpha. https://www.wolframalpha.com/input?i=ManhattanDistance; 2023.
157. Chen J, Gu Y, Cai S, Chen H, Chen J. KS-TCP: An Efficient Test Case Prioritization Approach based on K-medoids and Similarity. In: *International Symposium on Software Reliability Engineering (ISSRE) Workshops* 2021:105–110. 10.1109/ISSREW53611.2021.00051.
158. Dai H, Sun C, Liu H, Zhang X. DFuzzer: diversity-driven seed queue construction of fuzzing for deep learning models. *IEEE Transactions on Reliability.* 2024;73(2):1075-1089. 10.1109/TR.2023.3322406.
159. Vogel T, Tran C, Grunske L. Does diversity improve the test suite generation for mobile applications?. In: *International Symposium on Search Based Software Engineering* 2019:58–74. 10.1007/978-3-030-27455-9_5.
160. Vogel T, Tran C, Grunske L. A comprehensive empirical evaluation of generating test suites for mobile applications with diversity. *Information and Software Technology.* 2021;130:106436. 10.1016/j.infsof.2020.106436.
161. Altiero F, Corazza A, Di Martino S, Peron A, Libero Lucio Starace L. Regression test prioritization leveraging source code similarity with tree kernels. *Journal of Software: Evolution and Process.* 2024:e2653. 10.1002/smr.2653.
162. Alshraideh M, Mahafzah BA, Al-Sharaeh S. A multiple-population genetic algorithm for branch coverage test data generation. *Software Quality Journal.* 2011;19:489–513. 10.1007/s11219-010-9117-4.
163. Chen Z, Zhang J, Luo B. Teaching software testing methods based on diversity principles. In: *Conference on Software Engineering Education and Training (CSEE&T)* 2011:391–395. 10.1109/CSEET.2011.5876111.
164. De Lucia A, Di Penta M, Oliveto R, Panichella A. On the role of diversity measures for multi-objective test case selection. In: *International Workshop on Automation of Software Test (AST)* 2012:145–151. 10.1109/IWAST.2012.6228983.
165. Menendez HD, Clark D. Hashing Fuzzing: Introducing Input Diversity to Improve Crash Detection. *Transactions on Software Engineering.* 2022;48(09):3540-3553. 10.1109/TSE.2021.3100858.
166. Panichella A, Oliveto R, Di Penta M, De Lucia A. Improving multi-objective test case selection by injecting diversity in genetic algorithms. *Transactions on Software Engineering.* 2015;41(4):358–383. 10.1109/TSE.2014.2364175.
167. Reddy S, Lemieux C, Padhye R, Sen K. Quickly generating diverse valid test inputs with reinforcement learning. In: *International Conference on Software Engineering (ICSE)* 2020:1410–1421. 10.1145/3377811.3380399.
168. Scalabrino S, Grano G, Nucci DD, Oliveto R, Lucia AD. Search-based testing of procedural programs: Iterative single-target or multi-target approach?. In: *International symposium on search based software engineering* 2016:64–79. 10.1007/978-3-319-47106-8_5.
169. Wang W, Chen Z, Zheng Z, Wang H. An adaptive fuzzing method based on transformer and protocol similarity mutation. *Computers & Security.* 2023;129(C):103197. 10.1016/j.cose.2023.103197.
170. Yatoh K, Sakamoto K, Ishikawa F, Honiden S. Feedback-controlled random test generation. In: *Proceedings of the International Symposium on software testing and analysis* 2015:316–326. 10.1145/2771783.2771805.

171. Yuan Y, Pang Q, Wang S. Provably valid and diverse mutations of real-world media data for DNN testing. *IEEE Transactions on Software Engineering.* 2024;50(5). 10.1109/TSE.2024.3370807.
172. Azizi M. A tag-based recommender system for regression test case prioritization. In: *International Workshop on Software Test Architecture (InSTA)* 2021:146–157. 10.1109/ICSTW52544.2021.00035.
173. Bertolino A, Daoudagh S, El Kateb D, et al. Similarity testing for access control. *Information and Software Technology.* 2015;58:355–372. 10.1016/j.infsof.2014.07.003.
174. Bürdek J, Lochau M, Bauregger S, et al. Facilitating reuse in multi-goal test-suite generation for software product lines. In: *International Conference on Fundamental Approaches to Software Engineering* 2015:84–99. 10.1007/978-3-662-46675-9_6.
175. Gong L, Lo D, Jiang L, Zhang H. Diversity maximization speedup for fault localization. In: *Proceedings of the International Conference on Automated Software Engineering* 2012:30–39. 10.1145/2351676.2351682.
176. Hao D, Zhang L, Pan Y, Mei H, Sun J. On similarity-awareness in testing-based fault localization. *Automated Software Engineering.* 2008;15(2):207–249. 10.1007/s10515-008-0025-9.
177. Xia X, Gong L, Le TDB, Lo D, Jiang L, Zhang H. Diversity maximization speedup for localizing faults in single-fault and multi-fault programs. *Automated Software Engineering.* 2016;23(1):43–75. 10.1007/s10515-014-0165-z.
178. Jiang Z, Li H, Wang R. Efficient generation of valid test inputs for deep neural networks via gradient search. *Journal of Software: Evolution and Process.* 2023;n/a(n/a):e2550. 10.1002/smr.2550.
179. Joffe L, Clark D. Constructing Search Spaces for Search-Based Software Testing Using Neural Networks. In: *International Symposium on Search Based Software Engineering* 2019:27–41. 10.1007/978-3-030-27455-9_3.
180. Wang K, Wang Y, Wang J, Wang Q. Fuzzing with Sequence Diversity Inference for Sequential Decision-making Model Testing. In: *IEEE International Symposium on Software Reliability Engineering (ISSRE)* 2023:706–717. 10.1109/ISSRE59848.2023.00041.
181. Xiang Y, Huang H, Li S, Li M, Luo C, Yang X. Automated test suite generation for software product lines based on quality-diversity optimization. *ACM Transactions on Software Engineering and Methodology.* 2023;33(2):1–52. 10.1145/3628158.
182. Albunian NM. Diversity in Search-Based Unit Test Suite Generation. In: *Search Based Software Engineering* 2017:183–189. 10.1007/978-3-319-66299-2_17.
183. Nikolik B. Test diversity. *Information and Software Technology.* 2006;48(11):1083–1094. 10.1016/j.infsof.2006.02.001.
184. Singh S, Sharma C, Singh U. A simple technique to find diverse test cases. *International Journal of Computer Applications.* 2013;975(1):88–87.
185. Pizzoleto AV, Ferrari FC, Dallilo LD, Offutt J. SiMut: exploring program similarity to support the cost reduction of mutation testing. In: *International Workshop on Mutation Analysis* 2020:264–273. 10.1109/ICSTW50294.2020.00052.
186. Wu K, Fang C, Chen Z, Zhao Z. Test case prioritization incorporating ordered sequence of program elements. In: *International Workshop on Automation of Software Test (AST)* 2012:124–130. 10.1109/IWAST.2012.6228980.
187. Xie X, Yin P, Chen S. Boosting the Revealing of Detected Violations in Deep Learning Testing: A Diversity-Guided Method. In: *International Conference on Automated Software Engineering* 2022:1–13. 10.1145/3551349.3556919.
188. Kong W. Transcend Adversarial Examples: Diversified Adversarial Attacks to Test Deep Learning Model. In: *IEEE International Conference on Computer Design (ICCD)* 2023:13–20. 10.1109/ICCD58817.2023.00013.
189. Nguyen HL, Grunske L. BEDIVFUZZ: Integrating Behavioral Diversity into Generator-based Fuzzing. In: *International Conference on Software Engineering (ICSE)* 2022:249-261. 10.1145/3510003.3510182.
190. Zhao C, Mu Y, Chen X, Zhao J, Ju X, Wang G. Can test input selection methods for deep neural network guarantee test diversity? A large-scale empirical study. *Information and Software Technology.* 2022;150:106982. 10.1016/j.infsof.2022.106982.
191. Feng J, Yin BB, Cai KY, Yu ZX. 3-way gui test cases generation based on event-wise partitioning. In: *International Conference on Quality Software* 2012:89–97. 10.1109/QSIC.2012.42.
192. He Z, Bai C. GUI test case prioritization by state-coverage criterion. In: *International Workshop on Automation of Software Test* 2015:18–22. 10.1109/AST.2015.11.
193. Leveau J, Blanc X, Réveillére L, Falleri J, Rouvoy R. Fostering the Diversity of Exploratory Testing in Web Applications. In: *International Conference on Software Testing, Validation and Verification (ICST)* 2020:164-174. 10.1109/ICST46399.2020.00026.
194. Cai L, Tong W, Liu Z, Zhang J. Test case reuse based on ontology. In: *Pacific Rim International Symposium on Dependable Computing* 2009:103–108. 10.1109/PRDC.2009.25.
195. Lin J, Wang F, Chu P. Using semantic similarity in crawling-based web application testing. In: *International Conference on Software Testing, Verification and Validation (ICST)* 2017:138–148. 10.1109/ICST.2017.20.
196. Ojdanic M, Garg A, Khanfir A, Degiovanni R, Papadakis M, Le Traon Y. Syntactic versus semantic similarity of artificial and real faults in mutation testing studies. *IEEE Transactions on Software Engineering.* 2023;49(7):3922–3938. 10.1109/TSE.2023.3277564.
197. Cai G, Su Q, Hu Z. Binary searching iterative algorithm for generating test cases to cover paths. *Applied Soft Computing.* 2021;113:107910. 10.1016/j.asoc.2021.107910.
198. Panchapakesan A, Abielmona R, Petriu E. Dynamic white-box software testing using a recursive hybrid evolutionary strategy/genetic algorithm. In: *Congress on Evolutionary Computation* 2013:2525–2532. 10.1109/CEC.2013.6557873.
199. Brooks PA, Memon AM. Introducing a test suite similarity metric for event sequence-based test cases. In: *International Conference on Software Maintenance* 2009:243–252. 10.1109/ICSM.2009.5306305.
200. Flemström D, Potena P, Sundmark D, Afzal W, Bohlin M. Similarity-based prioritization of test case automation. *Software quality journal.* 2018;26(4):1421–1449. 10.1007/s11219-017-9401-7.
201. Shin D, Yoo S, Bae D. Diversity-aware mutation adequacy criterion for improving fault detection capability. In: *International Workshop on Mutation Analysis* 2016:122–131. 10.1109/ICSTW.2016.37.
202. Shin D, Yoo S, Bae D. A theoretical and empirical study of diversity-aware mutation adequacy criterion. *Transactions on Software Engineering.* 2018;44(10):914–931. 10.1109/TSE.2017.2732347.
203. Kichigin DY. A method of test-suite reduction for regression integration testing. *Programming and Computer Software.* 2009;35(5):282–290. 10.1134/S0361768809050041.
204. Semeráth O, Farkas R, Bergmann G, Varró D. Diversity of graph models and graph generators in mutation testing. *International Journal on Software Tools for Technology Transfer.* 2020;22(1):57–78. 10.1007/s10009-019-00530-6.

205. Sondhi D, Jobanputra M, Rani D, Purandare S, Sharma S, Purandare R. Mining Similar Methods for Test Adaptation. *Transactions on Software Engineering.* 2022;48(07):2262-2276. 10.1109/TSE.2021.3057163.
206. Michael C, Jonathan M, Michael D, Christian N. srcML. https://www.srcml.org/#home; Retrived July. 3, 2024.
207. Clark J. Trang. https://relaxng.org/jclark/trang.html; Retrived July. 3, 2024.
208. Tashiro Y, Song Y, Ermon S. Diversity can be transferred: Output diversification for white-and black-box attacks. *Advances in neural information processing systems.* 2020;33:4536–4548.
209. Hartigan JA, Wong MA. Algorithm AS 136: A k-means clustering algorithm. *Journal of the royal statistical society. booktitle c (applied statistics).* 1979;28(1):100–108. 10.2307/2346830.
210. Keller RM. Formal verification of parallel programs. *Communications of the ACM.* 1976;19(7):371–384. 10.1145/360248.360251.
211. Klees G, Ruef A, Cooper B, Wei S, Hicks M. Evaluating fuzz testing. In: *Proceedings of the ACM SIGSAC conference on computer and communications security* 2018:2123–2138. 10.1145/3243734.3243804.
212. Tsankov P, Dashti MT, Basin D. SECFUZZ: Fuzz-testing security protocols. In: *International Workshop on Automation of Software Test (AST)* 2012:1–7. 10.1109/IWAST.2012.6228985.
213. Padhye R, Lemieux C, Sen K, Papadakis M, Le Traon Y. Semantic fuzzing with zest. In: *Proceedings of the SIGSOFT International Symposium on Software Testing and Analysis* 2019:329–340. 10.1145/3293882.3330576.
214. Pacheco C, Lahiri SK, Ernst MD, Ball T. Feedback-directed random test generation. In: *International Conference on Software Engineering (ICSE)* 2007:75–84. 10.1109/ICSE.2007.37.
215. Pacheco C, Ernst MD. Randoop: feedback-directed random testing for Java. In: *Companion to the SIGPLAN conference on Object-oriented programming systems and applications companion* 2007:815–816. 10.1145/1297846.1297902.
216. Just R, Jalali D, Ernst MD. 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* 2014:437–440. 10.1145/2610384.2628055.
217. Yoo S, Harman M. Regression testing minimization, selection and prioritization: a survey. *Software testing, verification and reliability.* 2010;22(2):67–120. 10.1002/stvr.430.
218. Gligoric M, Eloussi L, Marinov D. Practical regression test selection with dynamic file dependencies. In: *Proceedings of the International Symposium on Software Testing and Analysis* 2015:211–222. 10.1145/2771783.2771784.
219. Leveau J, Blanc X, Réveillère L, Falleri J, Rouvoy R. Fostering the diversity of exploratory testing in web applications. *Software Testing, Verification and Reliability.* 2022;32(5):e1827. 10.1002/stvr.1827.
220. Wong WE, Gao R, Li Y, Abreu R, Wotawa F. A survey on software fault localization. *IEEE Transactions on Software Engineering.* 2016;42(8):707–740. 10.1109/TSE.2016.2521368.
221. Do H, Elbaum S, Rothermel G. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Software Engineering.* 2005;10:405–435. 10.1007/s10664-005-3861-2.
222. Beller M, Gousios G, Zaidman A. TravisTorrent: Synthesizing Travis CI and GitHub for Full-Stack Research on Continuous Integration. In: *International Conference on Mining Software Repositories (MSR)* 2017:447–450. 10.1109/MSR.2017.24.
223. Locascio N, Narasimhan K, DeLeon E, Kushman N, Barzilay R. Neural generation of regular expressions from natural language with minimal domain knowledge. *arXiv preprint arXiv:1608.03000.* 2016. 10.48550/arXiv.1608.03000.
224. Widyasari R, Sim SQ, Lok C, et al. Bugsinpy: a database of existing bugs in python programs to enable controlled testing and debugging studies. In: *Proceedings of the ACM joint meeting on European software engineering conference and symposium on the foundations of software engineering* 2020:1556–1560. 10.1145/3368089.3417943.
225. Inc M. Simulink Design Verifier. https://uk.mathworks.com/help/sldv/examples/; Retrived July. 29, 2024.
226. Inc M. Simulink. https://uk.mathworks.com/help/simulink/examples/; Retrived July. 29, 2024.
227. JS-frameworks . Front-end JavaScript frameworks. https://github.com/collections/front-end-javascript-frameworks; Retrived July. 29, 2024.
228. Alshahwan N, Harman M. Automated web application testing using search based software engineering. In: *International Conference on Automated Software Engineering (ASE)* 2011:3–12. 10.1109/ASE.2011.6100082.
229. Rogstad E, Briand L, Dalberg R, Rynning M, Arisholm E. Industrial experiences with automated regression testing of a legacy database application. In: *International Conference on Software Maintenance (ICSM)* 2011:362-371. 10.1109/ICSM.2011.6080803.
230. Braiek HB, Khomh F. Deepevolution: A search-based testing approach for deep neural networks. In: *International Conference on Software Maintenance and Evolution (ICSME)* 2019:454–458. 10.1109/ICSME.2019.00078.
231. Krizhevsky A, Sutskever I, Hinton GE. Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems.* 2012;25.
232. LeCun Y, Bottou L, Bengio Y, Haffner P. Gradient-based learning applied to document recognition. *Proceedings of the IEEE.* 1998;86(11):2278–2324. doi: 10.1109/5.726791
233. Krizhevsky A, Hinton G. Learning multiple layers of features from tiny images. *Toronto, ON, Canada.* 2009.
234. Netzer Y, Wang T, Coates A, et al. Reading digits in natural images with unsupervised feature learning. In: *NIPS workshop on deep learning and unsupervised feature learning*. 2011. 2011:4.
235. Baidu . Apollo: Open Source Autonomous Driving. https://github.com/ApolloAuto/apollo; Retrived July. 29, 2024.
236. Rong G, Shin BH, Tabatabaee H, et al. LGSVL Simulator: A High Fidelity Simulator for Autonomous Driving. In: *International conference on intelligent transportation systems (ITSC)* 2020:1–6. 10.1109/ITSC45102.2020.9294422.
237. Nikolik B. Test suite oscillations. *Information processing letters.* 2006;98(2):47–55. 10.1016/j.ipl.2005.11.016.
238. Nikolik B. The π measure. *IET software.* 2008;2(5):404–416. 10.1049/iet-sen:20070094.
239. Nikolik B. Software quality assurance economics. *Information and Software Technology.* 2012;54(11):1229–1238. 10.1016/j.infsof.2012.06.003.
240. Rogstad E, Briand LC. Clustering deviations for black box regression testing of database applications. *IEEE Transactions on Reliability.* 2015;65(1):4–18. 10.1109/TR.2015.2437840.
241. Rogstad E, Briand L. Cost-effective strategies for the regression testing of database applications: Case study and lessons learned. *Journal of Systems and Software.* 2016;113:257–274. 10.1016/j.jss.2015.12.003.
242. Rosero RH, Gómez OS, Rodríguez G. An approach for regression testing of database applications in incremental development settings. In: *International Conference on Software Process Improvement (CIMPS)* 2017:1–4. 10.1109/CIMPS.2017.8169952.

243. Sadeghiyan B, Mouzarani M. A New View on Classification of Software Vulnerability Mitigation Methods. *Global Journal of Computer Science and Technology.* 2017;17(C1):41–61.
244. Alqahtani S. A study on the use of vulnerabilities databases in software engineering domain. *Computers & Security.* 2022;116:102661. 10.1016/j.cose.2022.102661.
245. Liu B, Lucia , Nejati S, Briand LC, Bruckmann T. Simulink fault localization: an iterative statistical debugging approach. *Software Testing, Verification and Reliability.* 2016;26(6):431–459. 10.1002/stvr.1605.
246. Matinnejad R, Nejati S, Briand LC, Bruckmann T. SimCoTest: A test suite generation tool for Simulink/Stateflow controllers. In: *Proceedings of International Conference on Software Engineering Companion* 2016:585–588. 10.1145/2889160.2889162.
247. Yang Y. Improve Model Testing by Integrating Bounded Model Checking and Coverage Guided Fuzzing. *arXiv preprint arXiv:2211.04712.* 2022;n/a(n/a):arXiv:2211.04712. 10.48550/arXiv.2211.04712.
248. Rabin RI, Alipour MA. Configuring test generators using bug reports: a case study of gcc compiler and csmith. In: *Proceedings of the Annual ACM Symposium on Applied Computing* 2021:1750–1758. 10.1145/3412841.3442047.
249. Zheng Y, Liu Y, Xie X, et al. Automatic web testing using curiosity-driven reinforcement learning. In: *International Conference on Software Engineering (ICSE)* 2021:423–435. 10.1109/ICSE43902.2021.00048.
250. Tsai CY, Taylor GW. DeepRNG: Towards Deep Reinforcement Learning-Assisted Generative Testing of Software. 2022. 10.48550/arXiv.2201.12602.
251. Fahmy H, Pastore F, Briand L, Stifter T. Simulator-based Explanation and Debugging of Hazard-triggering Events in DNN-based Safety-critical Systems. *ACM Transactions on Software Engineering and Methodology (TOSEM).* 2023;32(4). 10.1145/3569935.
252. Zhi Y, Xie X, Shen C, Sun J, Zhang X, Guan X. Seed Selection for Testing Deep Neural Networks. *ACM Transactions on Software Engineering and Methodology (TOSEM).* 2023;33(1). 10.1145/3607190.
253. Zohdinasab T, Riccio V, Tonella P. An Empirical Study on Low-and High-Level Explanations of Deep Learning Misbehaviours. In: *International Symposium on Empirical Software Engineering and Measurement (ESEM)* 2023:1–11. 10.1109/ESEM56168.2023.10304866.
254. Zohdinasab T, Riccio V, Tonella P. DeepAtash: Focused Test Generation for Deep Learning Systems. In: *Proceedings of the International Symposium on Software Testing and Analysis* 2023:954–966. 10.1145/3597926.3598109.
255. Huang D, Bu Q, Qing Y, Fu Y, Cui H. Feature Map Testing for Deep Neural Networks. *arXiv preprint arXiv:2307.11563.* 2023. 10.48550/arXiv.2307.11563.
256. Biagiola M, Stocco A, Riccio V, Tonella P. Two is better than one: digital siblings to improve autonomous driving testing. *Empirical Software Engineering.* 2024;29(4):1–33. 10.1007/s10664-024-10458-4.
257. Zohdinasab T, Riccio V, Tonella P. Focused Test Generation for Autonomous Driving Systems. *ACM Transactions on Software Engineering and Methodology (TOSEM).* 2024;33(6). 10.1145/3664605.
258. Dola S, McDaniel R, Dwyer MB, Soffa ML. CIT4DNN: Generating Diverse and Rare Inputs for Neural Networks Using Latent Space Combinatorial Testing. In: *Proceedings of the International Conference on Software Engineering (ICSE)* 2024:1–13. 10.1145/3597503.3639106.
259. Selay E, Zhou ZQ, Chen TY, Kuo FC. Adaptive random testing in detecting layout faults of web applications. *International Journal of Software Engineering and Knowledge Engineering.* 2018;28(10):1399–1428. 10.1142/S0218194018500407.