Deposited via The University of Sheffield.

White Rose Research Online URL for this paper:
https://eprints.whiterose.ac.uk/id/eprint/230253/

Version: Published Version

Proceedings Paper:

Latest updates: https://dl.acm.org/doi/10.1145/3756681.3756969

RESEARCH-ARTICLE

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

ISLAM ELGENDY, The University of Sheffield, Sheffield, South Yorkshire, U.K.

R. HIERONS, The University of Sheffield, Sheffield, South Yorkshire, U.K.

PHIL MCMINN, The University of Sheffield, Sheffield, South Yorkshire, U.K.

Open Access Support provided by:

The University of Sheffield

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

Islam Elgendy
School of Computer Science
The University of Sheffield
Sheffield, United Kingdom
i.elgendy@sheffield.ac.uk

Robert Hierons
School of Computer Science
The University of Sheffield
Sheffield, United Kingdom
r.hierons@sheffield.ac.uk

Phil McMinn
School of Computer Science
The University of Sheffield
Sheffield, United Kingdom
p.mcminn@sheffield.ac.uk

Abstract

Regression testing assures software correctness after changes but is resource-intensive. Test Case Prioritisation (TCP) mitigates this by ordering tests to maximise early fault detection. Diversity-based TCP prioritises dissimilar tests, assuming they exercise different system parts and uncover more faults. Traditional static diversity-based TCP approaches (i.e., methods that utilise the dissimilarity of tests), like the state-of-the-art FAST approach, rely on textual diversity from test source code, which is effective but inefficient due to its relative verbosity and redundancies affecting similarity calculations. This paper is the first to study bytecode as the basis of diversity in TCP, leveraging its compactness for improved efficiency and accuracy. An empirical study on seven Defects4J projects shows that bytecode diversity improves fault detection by 2.3–7.8% over text-based TCP. It is also 2–3 orders of magnitude faster in one TCP approach and 2.5–6 times faster in FAST-based TCP. Filtering specific bytecode instructions improves efficiency up to fourfold while maintaining effectiveness, making bytecode diversity a superior static approach.

CCS Concepts

• **Software and its engineering** → **Software testing and debugging**; **Empirical software validation**.

Keywords

test case prioritisation, diversity-based testing, bytecode diversity, textual diversity, Levenshtein distance, static analysis

1 Introduction

Regression testing is a fundamental component of software testing. Its purpose is to verify the correctness of software after modifications to the software itself or its execution environment. Although regression testing is important, it is widely recognised as a resource-intensive and time-consuming process [35]. Test Case Prioritisation (TCP) is a primary strategy to mitigate this challenge. TCP reorders test cases to maximise early fault detection, particularly when it has to be prematurely halted due to budgetary constraints [35]. In the domain of software testing research, TCP has garnered significant attention [14, 18, 23, 35], with diversity-based testing techniques playing a pivotal role [8, 21, 24]. Elgendy et al. [8] provide a formal definition of diversity-based testing techniques, which quantify dissimilarity among test cases through a distance metric derived from various artefacts related to the test, such as inputs or the text of the test cases. The idea of using test diversity is that dissimilar test cases are more likely to exercise different parts of the system than more similar test cases, increasing fault detection.

TCP approaches that utilise the source code of the test cases can be classified mainly as static techniques (i.e., techniques that analyse test cases' text), and dynamic techniques (i.e., techniques that require test suite execution). An example of a static technique is using the source code of the test cases [21], while an example of a dynamic technique is using coverage information [29]. Static approaches may be less effective, but they do not require any complex program instrumentation and program execution.

Many diversity-based TCP techniques leverage different testing artefacts, with static approaches that rely on test case text being commonly used and showing promising results [1, 21, 24, 33, 34]. These static techniques simplify the process by avoiding program instrumentation or model construction. However, relying on test case text has inherent limitations. The string-based representation of test cases is verbose and often large, increasing the time required for similarity calculations, particularly with complex distance metrics like Levenshtein distance [6]. Furthermore, we hypothesise that test cases contain extraneous information for detecting faults such as variable names, comments, and assertion messages, which can distort similarity calculations and affect prioritisation.

Miranda et al. [24] introduced the FAST family of TCP approaches to address runtime concerns with large test suites by leveraging big data techniques such as Shingling, Minhashing, and Locality-Sensitive Hashing to compute test similarity for TCP. However, these techniques still measure similarity based on test case text. While preprocessing to remove comments and whitespace can mitigate some issues, it does not eliminate all irrelevant information. A more concise and efficient representation of test cases could address these limitations by reducing the size of the data used for similarity calculations and eliminating irrelevant information.

In this paper, we propose to use the bytecode of the test cases as the basis for diversity calculations. Bytecode, the intermediate code executed by a virtual machine, offers a more abstract representation than machine code but is more concrete than high-level programming language code. The bytecode is smaller than the

source code, as it is a condensed form of the original source code that strips away information such as comments, whitespace, and variable names. It retains only execution-critical details, making it more efficient to store and process than source code. While this may reduce diversity, our hypothesis it does not hinder effectiveness. Moreover, not all bytecode instructions differentiate tests. Some instructions, such as loading and storing variables in/from the stack, serve a mechanical role rather than providing insight into the logic of the test. Therefore, we further propose to filter out these types of bytecode instructions and extract only instructions that relate to the functionality being tested, such as setting up specific values or calling methods that are central to the test's purpose.

We implemented two state-of-the-art static TCP approaches, Ledru-TCP [21] and FAST-TCP [24], applying bytecode similarity for the first time instead of textual similarity. We evaluated its effectiveness and efficiency against traditional static TCP approaches using textual diversity. The Levenshtein distance measured textual similarity, as it suits string data representations [8] and varying test sizes, and the same metric was used for bytecode similarity to ensure consistency. Results show that bytecode-based TCP outperforms text-based TCP, achieving 2.3%–7.8% higher APFD and up to 3.9 times better real-fault detection. The filtering bytecode approach improved APFD by 4.5% to 5.1% and doubled real fault detection compared to textual information.

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

Finally, we evaluated how closely static bytecode diversity TCP matches dynamic TCP performance. We implemented two dynamic TCP techniques using greedy total and greedy additional algorithms [29]. As expected, dynamic approaches, with more information, outperformed bytecode diversity in fault detection, achieving 2.8%–3.5% higher APFDs. However, the best bytecode TCP approach was 6.5–8.3 times faster and avoids the complexity of program instrumentation required for coverage-based techniques. We provide a replication package [7] with detailed descriptions of the system. The paper offers the following contributions:

(1) The first known use of bytecode of test cases as a basis for measuring their diversity in a TCP technique.
(2) A novel bytecode filtering method for further enhancing TCP efficiency.
(3) An evaluation of bytecode diversity against textual diversity-based approaches, providing new insights into their effectiveness in mutant and real-fault detection, as well as runtime performance.
(4) An empirical comparison of static and dynamic TCP approaches with bytecode diversity-based TCP, using data from seven projects and 97 versions from Defects4J, offering insights into the benefits of bytecode diversity.

2 Background

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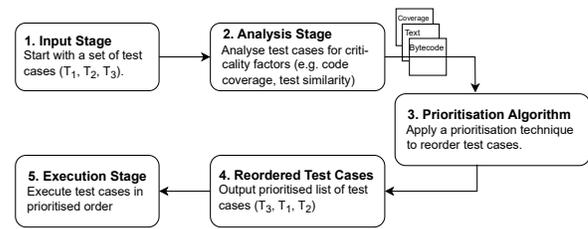

Figure 1: TCP process diagram

2.1 Test Case Prioritisation

Test case prioritisation (TCP) re-orders test cases so that the tester gets the maximum benefit if testing is prematurely stopped at some arbitrary point due to some budget constraints [35]. Figure 1 presents the general TCP process diagram showing the main stages of TCP. The primary goal of TCP is to identify and execute the most critical test cases earlier in the testing process. This leads to earlier fault detection and better resource utilisation. Rothermel et al. [29] formally defined TCP as follows: Given a test suite T, the set of permutations P of T, and an award function f from P to real numbers, find ($T' \in P$) such that $\forall T''$, where $T'' \in P$ and $T'' \neq T'$, $f(T') \geq f(T'')$. The award function f can be based on several criteria, such as code coverage [28, 29], fault detection history [12, 20], high-risk features [5], software requirements [2, 31], or diversity of test cases [21, 24].

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

$$APFD = 1 - \frac{TF_1 + TF_2 + \cdots + TF_m}{nm} + \frac{1}{2n} \qquad (1)$$

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

2.2 Diversity-Based TCP

In this work, we focus on diversity-based TCP, where the function f assesses test cases based on their diversity, either by rewarding those that are most dissimilar to others in an incrementally constructed subset P or by evaluating the overall diversity of the entire test suite. Elgendy et al. [8] provided a recent survey on diversity-based testing techniques in software testing in general and covered diversity-based techniques in TCP. First, diversity-based TCP collects the relevant information of the test case. These can be the source code of the test cases, the inputs to the test, the outputs of the test cases, etc. Then, the pairwise dissimilarity values between test cases are measured according to some distance metric, such as Euclidean distance, Jaccard distance, Levenshtein distance, etc. A "dissimilarity value" is a measure that quantifies how different two test cases are, typically based on shared characteristics or features. In our work, a lower dissimilarity value indicates greater similarity, while a higher value reflects greater dissimilarity. Finally, test cases are ordered based on the dissimilarity values. The ordering occurs in iterations, where in each iteration a test case is removed from the test suite and added to the new permutation set. The first test case

to be added to the permuted set can be selected randomly, or might be the most dissimilar test case from all others. Then, subsequent tests are selected one at a time, where the most dissimilar test from the ones already in the permuted set is selected, and the process continues until the test suite is completed.

Ledru et al. [21] proposed such a diversity-based TCP approach. They used five different distance metrics to calculate the pairwise dissimilarity values between test cases. The dissimilarity values are stored in a "similarity matrix". Miranda et al. [24] developed the FAST family of TCP techniques, which are quick and scalable for large test suites. They applied some big data techniques, such as Shingling, Minhashing, and Locality-Sensitive Hashing to calculate the dissimilarity between test cases. In both works, the source code of the test cases is used as a basis to calculate dissimilarity. We will expand more on these works in Section 3.3.

2.3 Bytecode

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

Java Bytecode is the Java Virtual Machine (JVM)-executable intermediate representation compiled from Java source code. Java Bytecode consists of a set of instructions and operands encoded in a binary format, providing a structured and efficient means of representing the program's logic and structure. ASM [9], a widely used library, enables dynamic bytecode manipulation, supporting class transformation, instrumentation, and analysis.

3 Approach

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

3.1 Motivating Example

Consider three test cases written by developers for the Apache Commons Math project shown in Figure 2 parts a)-c). We made a variation of test case 3, to showcase the problem, shown in Figure 2d. The test cases are designed to test the BigFraction class. Figure 3 shows the bytecodes of the test cases shown in Figure 2 in the human-readable and hexadecimal forms.

Let us take the example of the method testFormatNegative shown in Figure 2b and its corresponding bytecode shown in the second column of Figure 3a to illustrate how high-level Java instructions translate into low-level bytecode instructions. In the source code, a BigFraction object is created and initialised with the values -1 and 2. The bytecode reflects this sequence starting with the new instruction at index 0, which allocates memory for the new BigFraction object. The dup instruction at index 3 duplicates the top value of the stack to prepare for the constructor invocation, and the iconst_m1 and iconst_2 instructions at 4 and 5 push the values -1 and 2 onto the stack, respectively. The invokespecial instruction at 6 calls the constructor to initialise the object. Bytecode method invocation instructions are represented by invokevirtual,

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

(a) Test case 1

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

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

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

(b) Test case 2

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

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

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

(c) Test case 3 (original)

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

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

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

(d) Test case 3 (variation)

Figure 2: An example of four test cases from the Math project, with test case 3 having two variations

invokestatic, and invokespecial. Other bytecode instructions that involve loading and storing references to/from local variables are aload and astore, respectively. Field access instructions, like properFormat or improperFormat.format, are represented by getfield, putfield, and getstatic.

The textual test cases from Figure 2 are 705, 258, 252, and 320 bytes, respectively. Textual representations of test cases include additional elements such as variable names, comments, and formatting, which do not contribute to test behaviour but can inflate dissimilarity calculations. For instance, testFormatZero has two variations (Figures 2c and 2d), differing only in variable names

and comments—purely informational elements to the tester that do not affect execution. However, textual diversity methods treat these variations as more dissimilar than they actually are. In contrast, bytecode diversity captures only the core executable logic, leading to a more compact and precise representation. The two variations of `testFormatZero` produce identical bytecode despite textual differences, as shown in the last column in Figure 3.

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

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

3.2 Filtering Bytecode

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

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

Filtering out routine bytecode instructions from diversity calculations will have a significant advantage: it reduces the overall size of the resulting bytecode. Many bytecode instructions, such as `aload` and `astore`, occur frequently and contribute to the bulk of the bytecode without adding meaningful diversity in terms of test logic. By excluding these routine instructions, the size of the bytecode representation for each test case becomes smaller and more concise. Figure 3b shows the hexadecimal forms of the bytecode and the filtered versions.

This filtering approach allows for a more meaningful analysis of the test suite's variety, highlighting the differences in test goals and

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

(a) The human-readable form

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

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

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

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

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

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

(b) The Hexadecimal form

Figure 3: The human-readable and the Hexadecimal formats of the bytecodes for the tests in Figure 2

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

3.3 TCP Approaches

We implemented Ledru-TCP [21] and FAST-TCP [24], two state-of-the-art TCP approaches. Miranda et al. [24] proposed a family of TCP approaches, and we used one of these, FAST-pw. We chose to use FAST-pw because they reported it to be the most precise in the ranking, as it guarantees the selection of the most dissimilar test case from the ones already prioritised. We will simply refer to it as FAST from now on. Ledru-TCP and FAST-TCP are general TCP algorithms that can operate on either textual or bytecode information. When using textual information, they are referred to as

Algorithm 1: Ledru-TCP algorithm, summarised from [21]

Input: List of tests T; Similarity matrix Sim_T
Output: Prioritised test suite P

1   $P \leftarrow \emptyset$
    // First selection
2   $minValues \leftarrow getMinValues(Sim_T, T)$
3   $maxKey = getMaxOfMins(minValues)$
4   $P \leftarrow Append(P, T[maxKey])$
5   $T \leftarrow Remove(T, T[maxKey])$
    // Subsequent selections
6   **while** $|T| \neq 0$ **do**
7     $minValues \leftarrow getMinValues(Sim_T, P)$
8     $maxKey = getMaxOfMins(minValues)$
9     $P \leftarrow Append(P, T[maxKey])$
10     $T \leftarrow Remove(T, T[maxKey])$
11   **return** P

Algorithm 2: FAST-TCP algorithm [24]

Input: Coded test suite T
Output: Prioritised test suite P

1   $P \leftarrow \emptyset$
2   $I \leftarrow GetTextCaseIDs(T)$
3   $M \leftarrow MHSignatures(T)$
4   $B \leftarrow LSHBuckets(M)$
5   $M(v) \leftarrow MHSignatures(\emptyset)$
6   **while** $|P| \neq |I|$ **do**
7     $C_s \leftarrow LSHCandidates(B, M(v))$
8   **if** $C_s = \emptyset$ **then**
9       $M(v) \leftarrow MHSignatures(\emptyset)$
10       $C_s \leftarrow LSHCandidates(B, M(v))$
11     $C_d \leftarrow (I - P - C_s)$
12     $s \leftarrow \text{argmax}_{c \in C}\{JaccardDistance(M(v), M(c))\}$
13     $M(v) \leftarrow UpdateMHSignatures(M(v), M, s)$
14     $M \leftarrow Remove(M, s)$
15     $P \leftarrow Append(P, s)$
16   **return** P

Ledru-Text and FAST-Text, respectively, and when using bytecode, they are referred to as Ledru-Bytecode and FAST-Bytecode.

For Ledru-Bytecode, we used the ASM Java library to read the bytecode of the test cases and applied the Levenshtein distance [22] to calculate the pairwise dissimilarity values and save them in a similarity matrix. Algorithm 1 shows the steps of the Ledru-TCP [21]. The information we have is the pairwise similarity matrix Sim_T, and the list of tests T. The similarity matrix is a two-dimensional array, where each cell contains the textual/bytecode dissimilarity value between two test cases. We keep another list of the permuted tests P, which is initially empty (Line 1). First, using the similarity matrix, the minimum value of each row is saved into an array `minValues` (Line 2). For the first selection, we get the test that has the maximum value of the `minValues` (Line 3). This is the test that is the farthest away from all the rest. If multiple candidates have the same maximum value, we pick the first one. We add this test into P and remove it from T (Lines 4 and 5).

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

In FAST-Bytecode, we used the ASM Java library to read the bytecodes, which are then converted into a hexadecimal format to serve as the encoded representation of the test suite. Algorithm 2 outlines the steps involved in FAST-TCP [24]. Initially, the permuted test suite P is empty (Line 1), and the indices of the test cases are stored in I (Line 2). Using the encoded representation (text or bytecode) of the test suite T, FAST generates the MinHash signatures M (Line 3). The algorithm then computes the Locality-Sensitive Hashing (LSH) buckets B (Line 4), while the cumulative signatures of the ordered test cases, denoted as $M(v)$, are initially set to empty (Line 5).

The candidate sets are generated within a while loop (Lines 6-15), where the prioritisation process occurs. $M(v)$ is divided into b bands, each of which is hashed. If a collision is detected with the corresponding bucket in B, the test cases within that bucket are

added to the candidate set C_s (Line 7). The actual candidate set C_d, used by FAST, is calculated as the complement of C_s, excluding the test cases that have already been prioritised in P (Line 11). FAST then selects the candidate test case that is farthest from $M(v)$ based on the Jaccard distance (Line 12). $M(v)$ is updated to reflect the cumulative signatures of P (Line 13). Lastly, the selected test case is removed from M and added to P (Lines 14 and 15).

4 Evaluation

We evaluated the proposed use of bytecode as a diversity artefact in TCP with the textual artefact in two static TCP approaches. We described the diversity-based TCP approaches in Section 3.3.

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

4.1 Research Questions

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

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

Table 1: The test subjects used in our work

ID	Project name	#Versions	#Unique Classes	#Real Faults	#Mutants	# Test Cases			
						Randoop	EvoSuite	Developer	Total
Cli	jfreechart	5	4	7	851	7425	279	242	7946
Compress	commons-compress	7	5	7	1051	10769	133	38	10940
Csv	commons-csv	10	5	10	767	15050	260	550	15860
Jsoup	jsoup	16	10	16	767	20575	278	632	21485
Lang	commons-lang	15	11	15	4136	25893	1103	688	27684
Math	commons-math	42	36	50	10986	27235	962	1612	29809
Time	joda-time	2	2	2	268	499	172	69	740
Total		97	67	107	18826	107446	3187	3831	114464

This RQ seeks to evaluate the effectiveness and applicability of bytecode diversity as an alternative to textual diversity.

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

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

4.2 Evaluation Metrics

To assess the effectiveness and efficiency of testing processes in our study, we used the following evaluation metrics. We used mutation testing evaluated by APFD defined in Equation 1 and described in Section 2.1. Also, we utilised the real faults recorded in Defects4J [17] and used the number of the first test to reveal the real fault as the evaluation metric. A test that can reveal a real fault is one that fails in the buggy version of the Defects4J but passes in the fixed version.

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

4.3 Subjects

We ran our experiments using Java projects from the Defects4J framework [17]. While we initially considered all available projects, we selected seven Maven-based projects where JaCoCo [16] successfully generated coverage reports and PIT mutation analysis [3] ran without issues. Other projects were excluded due to failures in

meeting these requirements. We attempted to debug these issues for a week but ultimately did not pursue them further.

For our test suites, we used the developer-written tests augmented with automatically generated tests from Randoop [27] and EvoSuite [10]. We undertook this to create large, diverse test pools that possess high fault detection rates (i.e., mutation score) and can identify the real faults. Table 1 shows the projects and the size of the test suites for each project used in our study, where a total of 97 versions and 67 unique classes are used in our experiments. Also, we reported the number of mutants generated using the PIT mutation tool, the number of Randoop tests, the number of EvoSuite tests, and the number of developer-written tests for each project. PIT is a widely adopted tool in both academic and industrial settings, and we employed Randoop and EvoSuite, which are state-of-the-art tools for automated test generation in Java.

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

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

4.4 Methodology

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

4.4.1 Data Collection. Our tool parses the test files' source code and extracts each test case. The lines of each test case are concatenated into a single string, where the tool calculates the pairwise dissimilarity values between the strings of test cases using Levenshtein distance. We used the Levenshtein distance because it was reported by Elgendy et al. [8] to be more appropriate with string data, which is the nature of our test cases. In another study, Elgendy et al. [6] reported that Levenshtein distance performs well as a string distance metric. Then, a textual similarity CSV file is

generated that represents the textual similarity matrix. We compiled the source code using Apache Maven 3.6.3, which in turn used the `javac` compiler from OpenJDK 1.8.0_442. The tool also uses the ASM Java library to read the bytecode files (.class) in their binary format and uses the same Levenshtein distance to calculate the pairwise bytecode similarity between test cases. A separate bytecode similarity CSV file is generated. The textual and bytecode similarity files are to be used in Ledru-TCP.

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

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

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

4.4.2 Prioritising Test Cases. We are now ready to perform our TCP strategies, which we described in Section 3.3 and the coverage-based approaches described at the start of Section 4. We apply Ledru-TCP, described in Algorithm 1, using three different similarity matrices: textual similarity, all bytecode similarity, and filtered bytecode similarity. Also, we apply FAST-TCP, described in Algorithm 2, with three different input files, using the text of the test cases, the hexadecimal form of all bytecode, and the hexadecimal form of filtered bytecode.

4.4.3 TCP Evaluation. As described in Section 4.2, we evaluate the TCP strategies through mutation score, real-fault detection, and runtime. The tool uses the newly ordered test suite and the mutation information to calculate the APFD using Equation 1 for each TCP strategy.

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

To validate our results, we employed the non-parametric "Mann-Whitney U" statistical test, as the data are independent and we cannot assume a normal distribution. Additionally, we used the Vargha-Delaney (A_{12}) effect size test [32], a non-parametric measure that compares two groups by estimating the probability that a randomly selected value from one group is larger than a randomly selected value from the other group.

4.5 Threats to Validity

In this section, we address validity threats that can affect our results.

Construct validity: The selected string distance to measure test case similarity could be wrong. We mitigated this by using a widely known distance metric (Levenshtein), which is suitable given the nature of the data as reported by other works [6, 8, 21]. Another threat would be that the implementation of Ledru-TCP and FAST-TCP can be invalid. We mitigated this by running the implemented Ledru-TCP against the case studies provided by Ledru et al. [21], getting the same results, and by using the replication package for FAST-TCP provided by Miranda et al. [25]. Miranda et al. [24] used the text of the entire test classes as records for the input, while we used the individual test cases as records. We had an email correspondence with one of the authors to verify that we used the latest version of FAST, and that using individual test cases is valid.

Internal validity: The results can be affected by the nature of the used tests in our test suite, as the test suites we used are comprised of developer-written tests and automatically generated tests from Randoop and EvoSuite. To mitigate this risk, we ran our experiments on different test suites, where we used developer-written tests only, Randoop tests only, and EvoSuite tests only. We found negligible differences and our findings are not affected. These results are reported in our replication package [7].

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

Reliability: To address reproducibility concerns, we: (1) detailed all TCP approaches and subject characteristics, (2) thoroughly documented our methodology (including data collection and bytecode filtering criteria), and (3) provided a complete replication package on GitHub [7] with subjects, resources, implementation details, and an executable README.

5 Results

5.1 RQ1: Bytecode vs. Textual Diversity

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

Table 2: APFDs for all static TCP approaches across all projects. SD is the standard deviation and A_{12} is the effect size. The p-values and the effect sizes are between textual diversity and bytecode diversity for each TCP technique. A small effect size is marked with (*), while a medium effect size is marked with (**).

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

Table 3: Real-fault detection for all static TCP approaches across all projects.

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

and marked different effect sizes with stars, where 1, 2, and 3 stars correspond to a small, medium or large effect size, respectively. The p-values and effect sizes shown in the table are between the textual diversity and the corresponding bytecode diversity.

The median APFD of Ledru-Bytecode is 89.9, which is slightly higher than Ledru-Text. The median real fault detection is 124, which means that on median, after 124 test cases run, the real fault can be detected. This is slightly worse than the 110 tests of Ledru-Text. However, the differences in the APFD and real fault detection values are not statistically significant and have small effect sizes over Ledru-Text.

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

In terms of runtime, Figure 4 shows the preparation times for all static TCP approaches. Table 4 shows the runtime for all TCP approaches in each project in our study. For Ledru-TCP and FAST-TCP, the diversity artefact has the biggest impact on the preparation time, while the prioritisation time is usually the same per TCP approach. Thus, we reported the preparation times of each TCP approach and the average prioritisation time for Ledru-TCP and FAST-TCP. Using bytecode rather than the text of test cases hugely improves the efficiency of the Ledru-TCP and FAST-TCP.

Using Ledru-TCP, the preparation times using all bytecode information were 124 to 814 times faster than using textual test cases, where the bigger the test suite size and the larger the test case, the more improvements can be observed. Figure 4a shows a substantial

difference between using text and bytecode in Ledru-TCP. Similarly, Figure 4c highlights a significant reduction in preparation time when using bytecode instead of text in FAST-TCP. The smallest project in our study is the `Time` project, where the test suite size is 740 test cases. The preparation time in Ledru-TCP for building the similarity matrix dropped from 633.3 seconds (almost 10 minutes) using text to 5.1 seconds using all bytecode information. In FAST-TCP, the preparation time for building the signatures dropped from 2.6 seconds to 0.6 seconds. The `Math` project has the largest number of different versions and the largest number of test cases (29,610), but the test cases are small. In Ledru-TCP, the preparation times dropped from 347037.9 seconds (almost 96 hours) to 757.7 seconds (almost 12 minutes), while in FAST-TCP, the preparation times dropped from 88.9 seconds to 33.2 seconds. In FAST-TCP, the preparation times using all bytecode information were 2.5 to 6 times faster than using the text of test cases.

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

5.2 RQ2: Filtering Bytecode

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

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

Once more, the biggest improvement is the efficiency of filtering bytecode information. Ledru-Bytecode-Filter is 455 to 2798 times faster than Ledru-Text and 2.5 to 4.1 times faster than Ledru-Bytecode. Figures 4b- 4c visualise these time reductions for both full and filtered bytecode approaches. FAST-Bytecode-Filter is 4
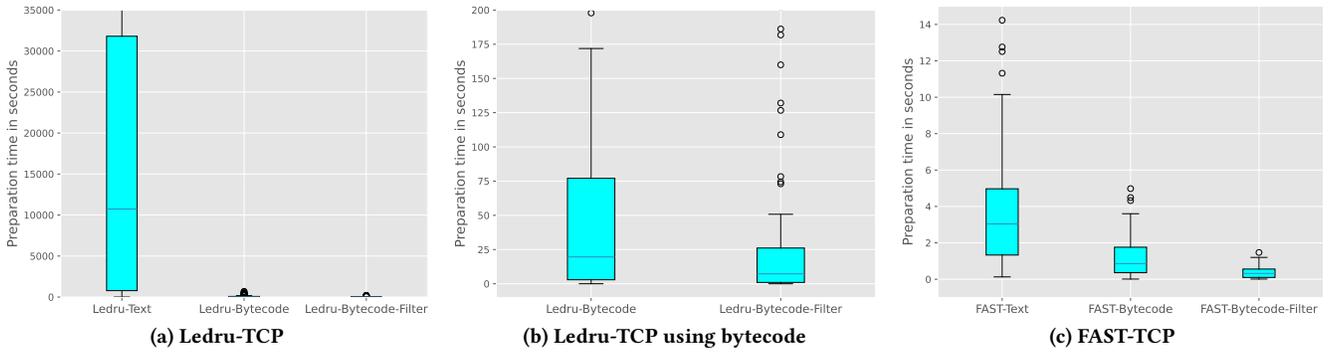
Figure 4: The preparation times in seconds for all static TCP approaches

Table 4: Runtime for all TCP approaches across all projects. Avg P-Time is the average prioritisation time in seconds for the TCP approach. Cov-Tot is coverage-based greedy total, while Cov-Add is coverage-based greedy additional.

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

to 17.9 times faster than FAST-Text and 1.4 to 4 times faster than FAST-Bytecode. The preparation time in the Math project dropped to only 261.4 seconds (4 minutes) in Ledru-TCP, while it dropped to only 9.2 seconds in FAST-TCP.

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

5.3 RQ3: Bytecode TCP vs. Dynamic TCP

We compared the best-performing static TCP approach (FAST-Bytecode) with the dynamic coverage-based TCP approaches. Tables 5 and 6 show the APFDs and real fault detection results. The median APFD of greedy additional is 96.7, which is slightly more than the greedy total (96), but the difference is not statistically significant. FAST-Bytecode has lower APFD by 2.8% to 3.5% than greedy total and greedy additional, respectively. Also, the standard deviation for the two coverage-based approaches is lower than bytecode diversity showing better consistency. Furthermore, the

Table 5: APFDs for FAST-bytecode TCP and coverage-based TCP across all projects.

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

Table 6: Real-fault detection for FAST-bytecode TCP and coverage-based TCP across all projects.

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

results are statistically significant at the 99% significance level, and there is a medium effect size for coverage-based TCP approaches.

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

Finally, the overall time of FAST-Bytecode, which includes the preparation and prioritisation times, is better than coverage-based TCP. As explained earlier in Section 4.2, we included only the prioritisation times for greedy total and greedy additional, shown in Table 4. Nonetheless, even using prioritisation times only for

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

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

---

6 Related Work

There is a plethora of work in TCP in general, and using diversity-based TCP in particular. This shows how TCP is a very active topic in the software testing literature. Yoo et al. [35] presented a comprehensive survey of techniques and strategies used in regression testing, where TCP is one of the techniques described to deal with regression testing. They categorised the TCP techniques as coverage-based (i.e., use structural coverage as metric), interaction testing (i.e., use multiple combinations of different components), distribution-based (i.e., use the distributions of the profiles of test cases), human-based (i.e., use case-based reasoning), history-based (i.e., use past bug fixes records), requirement-based (i.e., use software requirements), model-based (i.e., use system models), and other approaches (e.g. use of mutation scores, or use of recorded sessions for web applications). Lou et al. [23] presented 191 papers on TCP from 1997 to 2016, and investigated six aspects, which are the algorithms, criteria, measurements, constraints, empirical studies, and scenarios. Elgendy et al. [8] provided a comprehensive survey of diversity-based testing in software testing and covered diversity-based techniques in TCP. They identified text, test steps (i.e., individual actions that need to be followed to execute a test case), program executions, and clustering as common diversity artefacts in TCP but highlighted the need to explore new diversity artefacts. Our work directly addresses this gap by introducing byte-code as a novel diversity artefact for TCP, offering a more concise and efficient representation of test cases compared to traditional textual diversity.

Several researchers have developed TCP strategies to enhance early fault detection and test suite efficiency. Wu et al. [34] improved similarity-based prioritisation by incorporating program element execution times, while Khojah et al. [19] compared lexical and semantic diversity, concluding that semantic diversity provides better requirement coverage. Altiero et al. [1] prioritised tests based on their coverage of modified code parts, and Ledru et al. [21] introduced a greedy algorithm to maximise test diversity using textual representations. While these studies demonstrate the effectiveness of diversity-based TCP, they rely on textual diversity. In contrast, our work uses textual diversity as a baseline for comparison with bytecode diversity.

For manual black-box system testing, Hemmati et al. [13] adapted test prioritisation techniques, and empirical studies have evaluated the effectiveness of similarity-based TCP. Huang et al. [15] compared local and global TCP techniques, finding global approaches superior in coverage and fault detection. Haghighatkhah et al. [11] identified the most effective similarity-based TCP implementation, testing various distance measures. Noor and Hemmati [26] proposed prioritising test cases by their similarity to previously failing test cases, using metrics like Basic Counting, Hamming distance, and Levenshtein distance, suggesting that tests similar to past failures are more likely to detect new faults. In addressing scalability and performance for large test suites, Miranda et al. [24] introduced the FAST family of techniques, which leverage big data methods like Shingling, Minhashing, and Locality Sensitive Hashing (LSH) for efficient similarity detection in both white-box and black-box testing. We implemented our bytecode diversity TCP and compared it against Ledru et al. [21] and FAST [24].

7 Conclusions and Future Work

This study introduces and evaluates, for the first time, the use of bytecode (a more concise representation of test cases) as the basis for Test Case Prioritisation (TCP). The results show that bytecode diversity improves fault detection rates and significantly reduces processing times compared to traditional textual diversity-based methods. Also, the results show that filtering bytecode information greatly improves efficiency while achieving better fault detection than textual information and comparable effectiveness to using all bytecode information. Although coverage-based techniques yield better fault detection, they require extensive program instrumentation, making them less practical in certain contexts. Bytecode diversity thus provides a suitable alternative, offering a balance between effectiveness and operational simplicity. Our findings suggest that bytecode-based TCP can be particularly advantageous in resource-constrained environments or where coverage data is unavailable.

Future research can explore different ways to filter bytecode information and gain insights into which instructions can be the most beneficial in TCP. One potential enhancement to our approach and an avenue for future work is using normalised bytecode for similarity analysis. Raw bytecode can vary due to compiler optimisations and structural differences that do not affect program behaviour. Techniques such as bytecode canonicalisation, variable renaming, and control flow normalisation (e.g., SootDiff et al. [4], Schott et al. [30]) could help filter out these variations, leading to more precise similarity calculations. We could also conduct experiments on datasets having multi-fault programs and use the cost-cognizant version of the APFD metric to investigate different fault severity. Additionally, a deeper investigation into diversity-based approaches, particularly focusing on the effectiveness of bytecode diversity in addressing stubborn mutants (those hardest to detect/kill) could provide valuable insights. Such work would further validate the robustness of bytecode diversity and its potential to improve the fault detection capabilities of regression testing.

References

[1] F. Altiero, A. Corazza, S. Di Martino, A. Peron, and L. Libero Lucio Starace. 2024. Regression test prioritization leveraging source code similarity with tree kernels.

Journal of Software: Evolution and Process (2024), e2653. doi:10.1002/smr.2653

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

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

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

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

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

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

[8] I. Elgendy, R. Hierons, and P. McMinn. 2025. A Systematic Mapping Study of the Metrics, Uses and Subjects of Diversity-Based Testing Techniques. *Software Testing, Verification and Reliability* 35, 2 (2025), e1914. doi:10.1002/stvr.1914

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

[10] G. Fraser and A. Arcuri. 2011. Evosuite: automatic test suite generation for object-oriented software. In *Proceedings of the SIGSOFT symposium and the European conference on Foundations of software engineering*. 416–419. doi:10.1145/2025113.2025179

[11] A Haghighatkhah, M. Mäntylä, M. Oivo, and P. Kuvaja. 2018. Test case prioritization using test similarities. In *Proceedings of the International Conference on Product-Focused Software Process Improvement (PROFES)*. 243–259. doi:10.1007/978-3-030-03673-7_18

[12] A. Haghighatkhah, M. Mäntylä, M. Oivo, and P. Kuvaja. 2018. Test prioritization in continuous integration environments. *Journal of Systems and Software* 146 (2018), 80–98. doi:10.1016/j.jss.2018.08.061

[13] H. Hemmati, Z. Fang, and M. V Mantyla. 2015. Prioritizing manual test cases in traditional and rapid release environments. In *Proceedings of the International Conference on Software Testing, Verification and Validation (ICST)*. 1–10. doi:10.1109/ICST.2015.7102602

[14] C. Henard, M. Papadakis, M. Harman, Y. Jia, and Y. Le Traon. 2016. Comparing white-box and black-box test prioritization. In *Proceedings of the International Conference on Software Engineering*. 523–534. doi:10.1145/2884781.2884791

[15] R. Huang, Y. Zhou, W. Zong, D. Towey, and J. Chen. 2017. An empirical comparison of similarity measures for abstract test case prioritization. In *Proceedings of the Annual Computer Software and Applications Conference (COMPSAC)*, Vol. 1. 3–12.

[16] JaCoCo. 2025. JaCoCo Implementation Design. http://www.jacoco.org/jacoco/trunk/doc/implementation.html. [Last accessed: 25-March-2025].

[17] R. Just, D. Jalali, and M. D Ernst. 2014. Defects4J: A database of existing faults to enable controlled testing studies for Java programs. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*. 437–440. doi:10.1145/2610384.2628055

[18] M. Khatibsyarbini, M A. Isa, D. NA Jawawi, and R. Tumeng. 2018. Test case prioritization approaches in regression testing: A systematic literature review.

Information and Software Technology 93 (2018), 74–93. doi:10.1016/j.infsof.2017.08.014

[19] R. Khojah, C. H. Chao, and d F G Oliveira Neto. 2023. Evaluating the trade-offs of text-based diversity in test prioritisation. In *Proceedings of the International Conference on Automation of Software Test (AST)*. 168–178. doi:10.1109/AST58925.2023.00021

[20] J-M Kim and A. Porter. 2002. A history-based test prioritization technique for regression testing in resource constrained environments. In *Proceedings of the International Conference on Software Engineering (ICSE)*. 119–129. doi:10.1145/581339.581357

[21] Y. Ledru, A. Petrenko, S. Boroday, and N. Mandran. 2012. Prioritizing test cases with string distances. *Automated Software Engineering* 19, 1 (2012), 65–95. doi:10.1007/s10515-011-0093-0

[22] V. I. Levenshtein. 1966. Binary codes capable of correcting deletions, insertions, and reversals. 10, 8 (1966), 707–710.

[23] Y. Lou, J. Chen, L. Zhang, and D. Hao. 2019. A survey on regression test-case prioritization. In *Advances in Computers*. Vol. 113. 1–46. doi:10.1016/bs.adcom.2018.10.001

[24] B. Miranda, E. Cruciani, R. Verdecchia, and A. Bertolino. 2018. FAST approaches to scalable similarity-based test case prioritization. In *Proceedings of the International Conference on Software Engineering (ICSE)*. 222–232. doi:10.1145/3180155.3180210

[25] B. Miranda, E. Cruciani, R. Verdecchia, and A. Bertolino. 2025. FAST replication package. https://github.com/icse18-FAST/FAST. [Last accessed: 25-March-2025].

[26] T. B. Noor and H. Hemmati. 2015. A similarity-based approach for test case prioritization using historical failure data. In *Proceedings of the International Symposium on Software Reliability Engineering (ISSRE)*. 58–68. doi:10.1109/ISSRE.2015.7381799

[27] C. Pacheco and M. D Ernst. 2007. Randoop: Feedback-directed random testing for Java. In *Proceedings of the Companion to the SIGPLAN Conference on Object-Oriented Programming Systems and Applications Companion (OOPSLA Companion)*. 815–816. doi:10.1145/1297846.1297902

[28] G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold. 1999. Test case prioritization: An empirical study. In *Proceedings of the International Conference on Software Maintenance (ICSM)*. 179–188. doi:10.1109/ICSM.1999.792604

[29] G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold. 2001. Prioritizing test cases for regression testing. *IEEE Transactions on Software Engineering* 27, 10 (2001), 929–948. doi:10.1109/32.962562

[30] S. Schott, S. E. Ponta, W. Fischer, J. Klauke, and E. Bodden. 2024. Java Bytecode Normalization for Code Similarity Analysis. In *European Conference on Object-Oriented Programming (ECOOP)*, Vol. 313. 37:1–37:29. doi:10.4230/LIPIcs.ECOOP.2024.37

[31] H. Srikanth, L. Williams, and J. Osborne. 2005. System test case prioritization of new and regression test cases. In *International Symposium on Empirical Software Engineering*. 10–pp. doi:10.1109/ISESE.2005.1541815

[32] A. Vargha and H. D Delaney. 2000. A critique and improvement of the CL common language effect size statistics of McGraw and Wong. *Journal of Educational and Behavioral Statistics* 25, 2 (2000), 101–132. doi:10.3102/10769986025002101

[33] R. Wang, S. Jiang, and D. Chen. 2015. Similarity-based regression test case prioritization. In *Proceedings of the International Conference on Software Engineering and Knowledge Engineering (SEKE)*. 358–363. doi:10.18293/seke2015-115

[34] K. Wu, C. Fang, Z. Chen, and Z. Zhao. 2012. Test case prioritization incorporating ordered sequence of program elements. In *Proceedings of the International Workshop on Automation of Software Test (AST)*. 124–130. doi:10.1109/IWAST.2012.6228980

[35] S. Yoo and M. Harman. 2010. Regression testing minimization, selection and prioritization: A survey. *Software testing, verification and reliability* 22, 2 (2010), 67–120. doi:10.1002/stvr.430