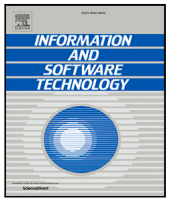
Information and Software Technology 150 (2022) 106994

Contents lists available at ScienceDirect

Information and Software Technology

journal homepage: www.elsevier.com/locate/infsof



Test case recommendation based on balanced distance of test targets

Weisong Sun [a], Quanjun Zhang [a], Chunrong Fang [a,*], Yuchen Chen [a], Xingya Wang [b,c,*], Ziyuan Wang [d,*]

[a] State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, 210093, China
[b] School of Computer Science and Technology, Nanjing Tech University, Nanjing, 211816, China
[c] Command and Control Engineering College, Army Engineering University of PLA, Nanjing, 210008, China
[d] School of Computer Science and Technology, Nanjing University of Posts and Telecommunications, Nanjing, 210023, China

ARTICLE INFO

Keywords:
Test case recommendation
Test target
Balanced distance

ABSTRACT

Context: Unit testing has been widely regarded as an effective technique to ensure software quality. Writing unit test cases is time-consuming and requires developers to have abundant knowledge and experience. Automated test case generation, a promising technology for liberating developers and improving test efficiency, currently performs not satisfactory in real-world projects. As a complement, test case recommendation (TCR) has been receiving the attention of researchers. TCR can improve the efficiency of test case writing by recommending test case code to developers for their reference and reuse. The overarching idea of TCR techniques is that two similar test targets can reuse each other's test cases.

Objective: Existing TCR techniques either fail to recommend relevant test cases for a given test target or are vulnerable to the mismatch of test target signatures. Our objective is to effectively and robustly recommend relevant test cases for test targets given by developers.

Method: In this paper, we propose a novel TCR technique that measures the similarity of test targets based on a balanced distance. The balanced distance integrates the distances on code snippets and comments, making the measurement of test target similarity more accurate and robust. In particular, we take the distance on control flows into account to compensate for the shortcomings in measuring the similarity only based on the literal text of code snippets. As a proof-of-concept application, we implement a test case recommender named BDTCR.

Results: We construct a test case corpus containing more than 13,000 test cases collected from GitHub. Based on this corpus, we conduct comprehensive experiments to evaluate the effectiveness and usefulness of BDTCR. The experimental results show that BDTCR can effectively recommend relevant test cases and outperform the state-of-the-art techniques.

Conclusion: It can be concluded that (1) BDTCR is an effective TCR technique; (2) BDTCR is a robust TCR technique that can effectively resist the interference of the mismatch of test target signatures; (3) BDTCR is practical to help developers write test cases quickly and effectively.

1. Introduction

Unit testing [1] has been widely recognized as an essential and valuable means of improving software quality, as it exposes bugs early in the software development life cycle [2,3]. A lack of adequate testing can have a high economic impact [4]. However, unit test coverage is not high in most open-source projects. The prior studies [5–8] show that most of the developers' efforts are focused on production code development and with a dismissive and negative view on software testing [9]. Besides, writing unit test cases is a time-consuming task and requires developers to have a lot of testing knowledge and experience.

Automated test case generation, a promising technology for liberating developers and improving test efficiency, has been extensively studied [10–15], but its current effectiveness is limited in real-world applications [16,17]. For example, Eduard Enoiu et al. [18] conducted a case study on real-world industrial control software to compare manually and automatically created tests. The study result shows that, although automatically generated tests can efficiently achieve similar code coverage as manually created tests, they do not result in better fault detection than manual testing. Manual tests more effectively detect logical, timer, and negation types of faults than automatically

* Corresponding authors.
E-mail addresses: weisongsun@smail.nju.edu.cn (W. Sun), quanjun.zhang@smail.nju.edu.cn (Q. Zhang), fangchunrong@nju.edu.cn (C. Fang), yuc.chen@outlook.com (Y. Chen), xingyawang@outlook.com (X. Wang), wangziyuan@njupt.edu.cn (Z. Wang).

https://doi.org/10.1016/j.infsof.2022.106994
Received 9 August 2021; Received in revised form 27 June 2022; Accepted 28 June 2022
Available online 2 July 2022

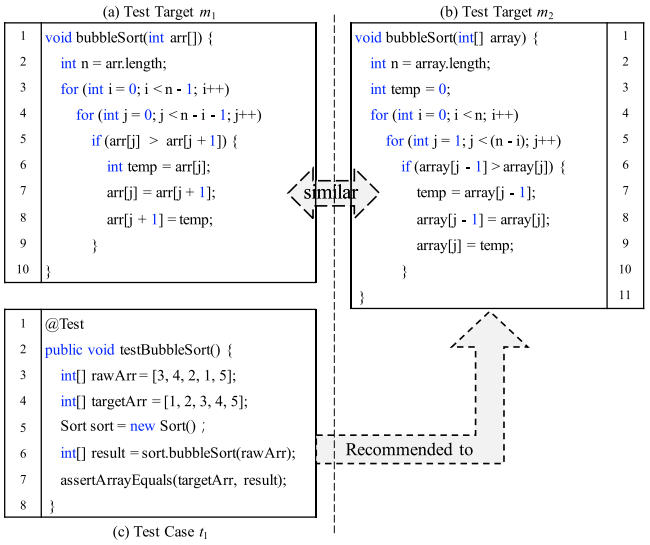


Fig. 1. An example of test case recommendation.

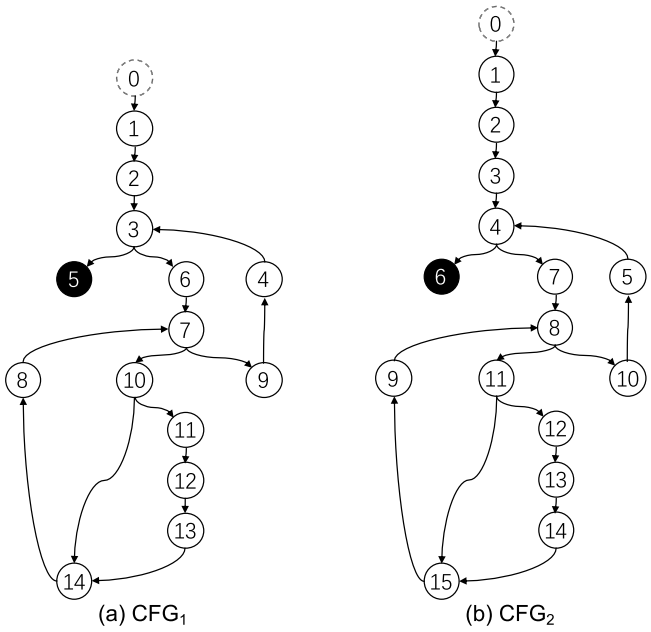


Fig. 2. The CFGs of the methods m_1 and m_2 in Fig. 1.

generated tests. The key findings of the study conducted by Domenico Serra et al. [19] further confirm that the automatic tools can achieve high coverage and mutation score; however, most of the actual defects cannot be identified. In addition, Sina Shamshiri et al. [20] conducted an empirical study to specifically evaluate the influence of automatically generated tests on developers' effort and effectiveness when performing software maintenance tasks. And their results show that, despite being less complex, automatically generated tests are harder to understand because they execute unrealistic scenarios.

Recommendation techniques as an effective means of improving the productivity of software engineering tasks have been widely investigated, such as developer recommendation [21–23], code recommendation [24–26], test case recommendation [6,27,28]. Among them, test case recommendation (TCR), which aims to provide test case code written by previous developers in existing projects to developers for their reference and reuse, has been proved to be an effective technique to assist unit testing. The idea of code reuse is widely adopted by TCR techniques [6,27–30], that is, two similar test targets can reuse each other's test cases. Formally, given a test case t_i whose test target is the method m_i, assuming that we plan to test a new test target m_j, if m_i and m_j are similar measured by a similarity function $sim(\cdot)$, the t_i can be recommended to m_j. For example, as shown in Fig. 1, two test targets m_1 and m_2 are similar in functionality, and thus the test case t_1 that belongs to m_1 can be recommended to the new test target m_2. Obviously, the main problem that TCR solves is to measure the similarity of test targets accurately.

Existing TCR techniques [6,27–30] provide us with some valuable insights and inspirations for measuring the similarity of test targets, but they still have some deficiencies. The test case search algorithms are designed to search for accurate and relevant test cases from the test case corpus. Although these techniques design various $sim(\cdot)$ functions in their search algorithms, they all measure the test target similarity crudely in terms of the code part. The prior works [28–30] design the function $sim(\cdot)$ based on the method signature and class name matching. The method signature (signature for short) is a combination of the method name and parameter types [31]. The accuracy of such $sim(\cdot)$ is easily affected by the modification of the method or class names (e.g., abbreviations, synonyms). The search algorithm used in the work [6] is based on a mature clone detector called NiCad [32]. The core function $sim(\cdot)$ designed by NiCad is based on the longest common sub-sequence matching algorithm. Thus, this $sim(\cdot)$ is easily affected by the modification of the literal text of the code snippet. For example, we feed two test targets m_1 and m_2 shown in Fig. 1 to NiCad, and the detection results show that they are not a clone pair. In this case, the test case t_1 would not be recommended to the test target m_2.

In this paper, we propose a novel TCR technique based on the balanced distance, which combines the distances of code snippets and comments. These distances can reflect the test target similarity in functionality to a certain extent. For the code snippet, considering that two test targets may share a low similarity in the literal text but implement the same or similar functionality, we compensate for this by measuring the similarity in control flows. For example, the test targets m_1 and m_2 shown in Fig. 1 share a low similarity in literal text while almost identical in control flow graphs (as shown in Fig. 2). Besides, the branch coverage is one of the most important evaluation criteria of test adequacy [33–35]. The control flow of a test target can intuitively show which branches of it need to be tested. In control flow-based test adequacy, node coverage is often called statement coverage or basic block coverage, and edge coverage is often called branch coverage [1,33]. If two test targets have similar control flows, developers can learn how to write similar test cases from the recommended test cases to achieve similar branch coverage in the new test target. Therefore, the TCR technique that takes control flows into count may make the recommended test cases suitable for the new test target. Based on the proposed TCR technique, we implement a test case recommender, namely BDTCR.

In addition, test case recommendation cannot be carried out when there is no corpus that provides a large number of high-quality test cases for retrieval. We implement a tool TConstructer to extract test cases automatically. The design details of TConstructer are introduced in Section 3.2. We extract includes not only a test method but also other context code elements it must depend on at runtime, such as external or third-party variables and method calls. We collectively refer to these context code elements as test dependencies. In practice, for TCR across project boundaries, developers may more or less need to modify recommended test cases to adapt to the new test targets. Test cases containing complete test dependencies are highly readable and understandable, reducing the time and cost for developers to understand and modify them. In this paper, our primary motivation toward studying Java projects that use the JUnit framework[1] is applicability.

[1] https://junit.org.

Existing studies indicated that Java and JUnit are one of the most popular programming languages [36] and Java libraries [37], respectively. Hence, constructing a dataset of projects that use Java and JUnit leads to the potential for impact on real-world software development [38]. For practical application, we construct a test case corpus that contains more than 13,000 JUnit test cases. Based on this corpus, we conduct comprehensive experiments to evaluate the effectiveness and usefulness of BDTCR. And the experimental results show that BDTCR outperforms the state-of-the-art technique (i.e., TestTenderer) by 11% and 26.43% in terms of $SuccessRate@5$ and $SuccessRate@10$.

In summary, we make the following contributions.

- We propose a novel TCR technique that measures the test target similarity based on the balanced distance. The balanced distance combines the distances of code snippets and commends, which allows our technique to recommend test cases more accurately and robustly.
- We develop an automated tool named TConstructer used to construct a corpus of test cases. We construct a test case corpus that contains 13,000 JUnit test cases through TConstructer.
- We implement a test case recommender BDTCR and integrate it into a mature testing platform Mooctest.[2] The comprehensive evaluation results show that BDTCR can recommend more accurate test cases than the state-of-the-art techniques.
- We release the implementation code of BDTCR and all the data for future researchers. BDTCR (equipped with TConstructer) and the test case corpus can be downloaded on the website.[3]

The remainder of this paper is organized as follows. Section 2 provides basic concepts involved in this study. Section 3 introduces our approach, i.e., the design of BDTCR. Section 4 presents the evaluation in detail. Section 5 describes a discussion about test case recommendation. Section 6 presents threats to validity. Section 7 discusses the related work. We conclude the paper in Section 8.

2. Basic concepts

In this section, we refine related terms in TCR. From one simple view, the source code is composed of production code ($\mathbb{P}$) and test code ($\mathbb{T}$). Test cases are extracted from $\mathbb{T}$ and used to check the test targets in $\mathbb{P}$.

Test Target. In unit testing, a unit under test is the smallest testable part of the software, a single method/function. All methods in $\mathbb{P}$ are potential test targets. Listing 1 shows an example of production code where the method *integerRepresentation()* is a test target. For each test target $m \in \mathbb{P}$, it is an essential task for TCR to understand m's semantics (i.e., functionality). According to software development experience, we can infer the functionality that a test target implements from its code snippets and comments. Code snippets contain lots of textual information (e.g., identifiers) and structural information (e.g., control flows) that explicitly express the functionality. Comments are the natural language descriptions providing additional information that is not readily available in the code itself [39], which are used to explain or describe what the method wants to do. Thus, method comments are valuable information to help developers comprehend programs and reduce additional time spent on reading and navigating source code [40].

```
1  public class ALU {
2  public String integerRepresentation(String
       number, int length) {
3  ...
4  String tmpNum;
5  boolean isMinus;
6  if (number.charAt(0) == '-') {
7  isMinus = true;
8  tmpNum = number.substring(1);
9  } else {
10 isMinus = false;
11 tmpNum = number;
12 }
13 ...
14 return result.toString();
15 }
16 }
```

Listing 1: An Example of Production Code

Test Method. A test method (m^t) is a method in $\mathbb{T}$. It is mainly responsible for completing the execution of the test task. It is usually declared by marking with the annotation "@Test" [38]. Listing 2 shows an example of test code where the method *test()* is a test method.

```
1  import org.junit.Test;
2  import static org.junit.Assert.*;
3  public class ALUTest {
4  ALU alu = new ALU();
5  @Test
6  public void test() {
7  String expected1 = "00001001";
8  String actual1 = alu.integerRepresentation("9
       ", 8);
9  String expected2 = "00010100";
10 String actual2 = alu.floatRepresentation("
       0.01", 2, 5);
11 assertEquals(expected1, actual1);
12 assertEquals(expected2, actual2);
13 }
14 }
```

Listing 2: An Example of Test Code

Test Dependency. Test dependency (D) refers to a set of context statements that a test method depends on at runtime, encompassing global variable declaration statements, callee method code statements, etc. The dependency on variables is considered as the variable dependency. The dependency on method calls is considered as the method dependency. Four kinds of variables are considered in the variable dependency: local, global, external variables, and variables from a third-party library. For example, in Listing 2, the global variable declaration statement *ALU alu = new ALU()* (line 4) is a variable dependency that the test method *test()* must depend on during executing testing. Three kinds of methods are considered in the method dependency: internal, external methods, and methods from third-party libraries. These test dependencies are contexts that a test method depends on during executing testing. For example, in Listing 2, the method code corresponding to the external method call *alu.integerRepresentation("0.9", 8)* is a method dependency. More details about test dependency analysis are introduced in Section 3.2.2.

Test Assertion. Test assertion ($\mathcal{A}$) refers to the assert statement contained in the test method. In unit testing, assert statements provide necessary logic checks to ensure that test targets are functioning properly and produce expected results [38]. For example, in Listing 2, the assert statement *assertEquals(expected1, actual1);* (line 11) is used to check whether the actual logic of the test target *integerRepresentation()* (line 8) works as expected.

Test Case. Above all, in this paper, a test case t is composed of a test method (m^t) and corresponding test dependencies (D). In addition, t should also meet the following conditions:

- t must have explicit test targets. In other words, if the core part m^t of t does not call any test targets, such t is worthless.
- t must have the assert statements. As mentioned earlier, the core part m^t of t should contain assert statements used to check the actual logic of the test target works as expected.

[2] http://www.mooctest.net.
[3] https://github.com/wssun/BDTCR.

Table 1
A summary of notations.

Name	Notation	Meaning
Production Code	$\mathbb{P}$	A part of source code
Test Code	$\mathbb{T}$	A part of source code
Test Target	m	A method under test
Test Method	m^t	A method in $\mathbb{T}$ and used to test m
Test Dependency	$\mathcal{D}$	m^t's test dependencies
Test Assertion	$\mathcal{A}$	Assert statements in m^t
Test Case	t	A test case

```
@Test
public void testIntegerRepresentation() {
String expected1 = "00001001";
String actual1 = alu.integerRepresentation("9
    ", 8);
assertEquals(expected1, actual1);
}

@Test
public void testFloatRepresentation() {
String expected2 = "00010100";
String actual2 = alu.floatRepresentation("
    0.01", 2, 5);
assertEquals(expected2, actual2);
}
```

Listing 3: An Example of Slicing Results

In particular, a test case should only check one test target. It is a discouraged practice that combining multiple unrelated tests into a single test method [27,41–43]. It is inappropriate and not user-friendly to directly recommend the non-normalized test methods to developers. For example, in Listing 2, the test method *test()* checks two test targets, i.e., *integerRepresentation()* and *floatRepresentation()*. In this case, if a novice wants to learn how to test *integerRepresentation()*, it is inappropriate to directly recommend a test case composed of *test()* and its $\mathcal{D}$ to him because *test()* also checks other test targets. Therefore, to make test case more suitable for TCR, with the guide of our previous study [27], we use program slicing technique [44] to slice the original test case into multiple test cases, and each of them has a single assert statement (i.e., $|\mathcal{A}| = 1$). For example, Listing 3 shows two test methods that are sliced from *test()* using program slicing technique. And more details are introduced in Section 3.2.4.

In Table 1, we provide a summary of notations used in defining the terms as well as those introduced later in the paper.

3. Approach

3.1. Overview

As shown in Fig. 3, BDTCR mainly consists of two parts: (a) test case construction – an offline task, and (b) test case search – an online task. Test case construction (TConstructer) is responsible for constructing a corpus (TCC) that encompasses substantial test cases to support subsequent test case search. TConstructer is an offline module. It is inclusive of four core sequential components: test method extraction (TM Extraction), test dependency analysis (TD Analysis), test target recognition (TT Recognition), and test assert normalization (TA Normalization). Given a pair of production code and test code, TConstructer first uses the TM Extraction component to extract all test methods, detailed in Section 3.2.1. These test methods may use/invoke some global variables/methods, which are essential for executing test methods and are therefore considered as test dependencies. TConstructer uses the TD Analysis component to extract test dependencies for test methods, detailed in Section 3.2.2. Considering that some test methods are invalid, that is, they do not test any test targets, TConstructer uses the TT Recognition component to select valid test methods, detailed in Section 3.2.3. As mentioned in Section 2, some test methods are non-normalized, so TConstructer further uses the TA Normalization component to normalize all valid test methods, detailed in Section 3.2.4. Each normalized test method and its test dependencies constitute a test case. Finally, all test cases are stored into the corpus TCC to support subsequent test case retrieval. Test case search (TSearcher), corresponding to the part (b) of Fig. 3 is responsible for retrieving test cases related to the given query (i.e., new test target) in TCC. TSearcher is an online module. It is composed of multiple components that are designed to complete five processes: ① query extraction, ② test target loading, ③ basic distances (i.e., DL, DG, DC) calculation, ④ balanced distance (i.e., BD) calculation, and ⑤ search result renewal. Given a query m, TSearcher first passes it into the query extraction process to extract the code, keywords of the comment, and CFG, detailed in Section 3.3.1. Then, TSearcher retrieves a test target (m') with a test case from TCC, detailed in Section 3.3.2. In the calculation process of the basic distances, TSearcher calculates the distances in three dimensions between m and m' (i.e., DL, DG, and DC), detailed in Section 3.3.3. Based on the distances DL, DG, and DC, TSearcher further calculates the balanced distance (BD), detailed in Section 3.3.4. All pairs of m' and BD are stored into a similar list of test targets M^S. In the process of search result renewal, TSearcher first sorts all m' by BD, then determines whether to trigger the next retrieval, detailed in Section 3.3.5. TSearcher repeats the above five processes until the number of search results reaches the expected number or all the test targets in TCC have been traversed. We discuss the above components and processes in detail in the following sections.

3.2. Test case construction

In this section, we mainly introduce the implementation of the TConstructer module in BDTCR. As described above, TConstructer includes four core sequential components used to complete four sequential tasks, i.e., test method extraction, test dependency analysis, test target recognition, and test assert normalization. We discuss how TConstructer completes these tasks in detail in the following subsections.

3.2.1. Test method extraction (TM extraction)

As mentioned in Section 2, the test method, as an essential part of a test case, is responsible for completing the execution of the test task. Therefore, it is fundamental work for test case construction to extract test methods from $\mathbb{T}$.

TConstructer designs the TM Extraction component to extract test methods from complex $\mathbb{T}$. In this component, TConstructer uses the characteristics of unit testing frameworks and test assertion libraries as the indicators to extract test methods. Specifically, according to the characteristics of JUnit testing frameworks, that is, in JUnit 4.x[4] or higher (JUnit 3.x is outdated, so it not be discussed.), test methods are annotated with '@Test'. TConstructer first extracts all such methods with '@Test' from $\mathbb{T}$. These methods are considered as candidate test methods. TConstructer further checks whether the candidate test methods contain assert statements. Assert statements provide necessary logic checks to ensure that test targets are functioning properly and produce expected results [38]. In addition to the assertions equipped in JUnit, AssertJ [45] and Truth [46] are also commonly used assertion libraries. TConstructer detects whether test methods contain assert statements according to external method calls and import information (e.g., lines 11 and 2 in Listing 2). Test methods that do not contain any assert statements are discarded.

[4] 3.x, 4.x, and 5.x refer to the version of the JUnit testing framework.

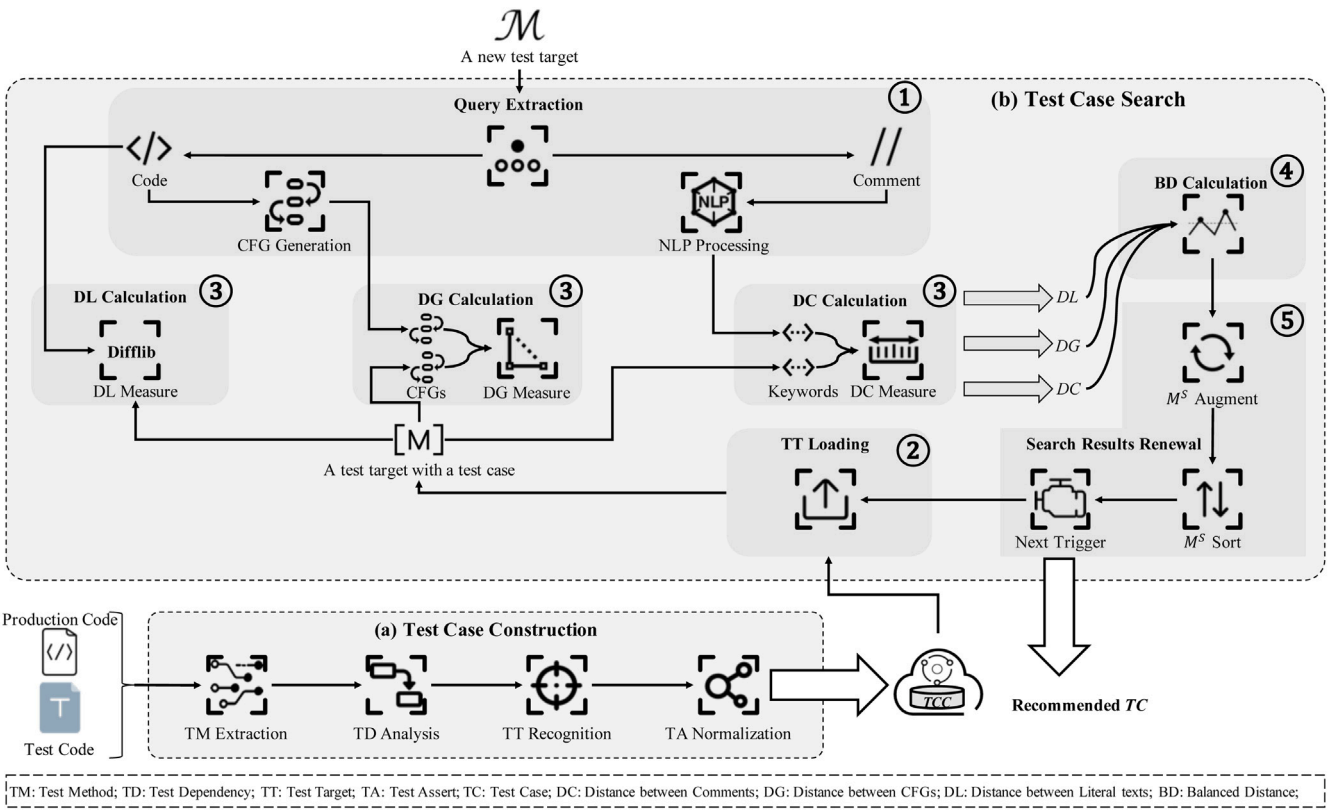


Fig. 3. The framework of BDTCR.

3.2.2. Test dependency analysis (TD analysis)

As described in Section 2, test dependency (D) is a set of context statements that a test method (m^t) depends on during executing testing. Test dependency analysis aims at extracting m^t's D. In addition, D is a prerequisite for recognizing the test targets in m^t (i.e., TT Recognition), and more details are introduced in Section 3.2.3.

TConstructer designs the TD Analysis component to extract test methods' contexts from $\mathbb{T}$ and $\mathbb{P}$. In this component, test dependency analysis is further subdivided into variable dependency analysis (*VDA*) and method dependency analysis (*MDA*). Considering that method calls may use the variables declared previously as arguments, the prerequisite for analyzing method calls is to know the data types of these arguments. Therefore, *VDA* must be performed before *MDA*.

Variable dependency analysis (VDA). Given a test method (m^t), *VDA* aims to extract all statements related to the variables that appear in m^t. In practice, TConstructer first uses JavaParser[5] to extract the dependencies among statements and generate a program dependency graph (PDG). Based on PDG, TConstructer employs Bidirectional (Backward and Forward) Static Slicing (BSS and FSS, for short) [42,44,47] to extract statements that are related to the slicing criterion. Static slicing selects all statements that can affect the slicing criterion directly or indirectly without executing the program [44]. All statements in m^t are considered as slicing criteria C. A slicing criterion $c \in C$ is composed of two-tuple $c = \langle s, v \rangle$ where s is a statement and v are variables in s. BSS extracts the statements that c directly or indirectly depends on because of the usage of v. FSS extracts the statements that directly or indirectly depend on c because they used the v declared in s. The slicing results of c is a union set of results from BSS and FSS. All slicing results of C constitute the result of *VDA*. For local variables, they are stored persistently along with test methods where they are declared. For global variables, their declaration statements are extracted and persistently stored. Besides, the code blocks of the static initializers in which the static variables are initialized are also stored. For external variables, declaration statements are extracted by analyzing import information. For variables from the third-party libraries, we store the import statements and the version information of libraries.

Method dependency analysis (MDA). Given a m^t, *MDA* aims to extract all methods related to m^t. In practice, TConstructer first statically scans $\mathbb{P}$ and $\mathbb{T}$ and extracts all methods' signatures. A signature can uniquely represent a method in the project [31]. Signatures are utilized to help TConstructer quickly determine methods when analyzing method calls. For internal and external method calls, TConstructer first analyzes variable dependencies and determines the data types of variables involved in the method calls (e.g., the variables in arguments). Then, TConstructer determines the specific method corresponding to the method call by comparing the signatures obtained in advance. In particular, for external method calls, TConstructer labels where they originate from, production code, or test code. Such a label is instrumental in test target recognition. All signatures of methods that are related to m^t constitute the result of *MDA*. All method code snippets of the internal and external method calls are persistently stored. For method calls from the third-party libraries, the import statements and the version information of libraries are stored. Listing 4 shows an example of the results of test dependency Analysis, which are test dependencies of the test method *test*(). Among which, the top 2 rows (lines 1–2) are *MDA*'s results, and the last row (line 3) is *VDA*'s result.

```
1 import org.junit.Test;
2 import static org.junit.Assert.*;
3 ALU alu = new ALU();
```

Listing 4: An Example of Test Dependency Analysis

3.2.3. Test target recognition (TT recognition)

As described in Section 2, a test case must have explicit test targets. Test target recognition aims to recognize test targets within test methods.

[5] https://javapaser.org.

TConstructer designs the TT Recognition component to recognize test targets tested by test methods. In this component, TConstructer recognizes the test target in a m^t under two progressive constraints: (1) test target is an external method call; and (2) test target is a method originating from $\mathbb{P}$. During TD Analysis, TConstructer has labeled which methods are external method calls and originate from $\mathbb{P}$. Thus, test target recognition is a simple task for TConstructer. In practice, TConstructer first extracts all method calls within the test method. The list of invoked methods is then queried against the previously extracted list of methods defined in $\mathbb{P}$, considering the complete method signature. With the help of the TD Analysis component, like [38,48], TConstructer then considers that the last method call before the assert is the test target of the assert statement. In some instances, the assert statement contains the method call within its parameters. In these instances, we consider the method call within the assertion parameters as the test target.

For example, in Listing 2, there are three method calls in the test method *test()*, i.e., *integerRepresentation()*, *floatRepresentation()* and *assertEquals()*. TConstructer knows *assertEquals()* is a method call from the third-party libraries though the TD Analysis component, it therefore only considers *integerRepresentation()*, *floatRepresentation()* as two test targets. They are the test targets of lines 11 and 12 assert statements, respectively.

3.2.4. Test assert normalization (TA normalization)

As described in Section 2, it is inappropriate and not user-friendly to directly recommend non-normalized test methods to developers. For example, the test method *test()* shown in Listing 2 is coupled and hard to read. Test assert normalization is to normalize each test method and make the number of assert statement (test target) of it 1, i.e., $|\mathcal{A}| = 1$. For example, the two test methods shown in Listing 3 are intuitively easier to understand than the original test method *test()*.

TConstructer designs the TA Normalization component to normalize test targets within test methods. In this component, TConstructer considers each assert statement as a slicing criterion and employs Bidirectional Static Slicing to extract statements related to the slicing criterion. For example, TConstructer considers lines 11 and 12 in Listing 2 as two slicing criteria, and the part of the slicing results are shown in Listing 3. In addition, by their very nature, test methods are a class of methods with a special purpose (i.e., testing) and should therefore follow the lowerCamelCase naming convention [49]. It is standard practice to name test methods the same as the methods being tested with the addition of "test" at the start. For example, the test method used to test the test target *integerRepresentation* should be named *testIntegerRepresentation*.

3.2.5. Test case example

```
1 import org.junit.Test;
2 import static org.junit.Assert.*;
3 ALU alu = new ALU();
4 @Test
5 public void testIntegerRepresentation() {
6 String expected1 = "00001001";
7 String actual1 = alu.integerRepresentation("9
      ", 8);
8 assertEquals(expected1, actual1);
9 }
```

Listing 5: An Example of a Test Case

Listing 5 shows a test case extracted and adjusted by TConstructer from the test code in Listing 2. The test case is composed of a test method *testIntegerRepresentation* (lines 4–9) and its test dependencies (lines 1–3). And its test target is the method *integerRepresentation()*.

All test cases extracted by TConstructer are persisted and stored in the corpus TCC to support the subsequent test case search.

3.3. Test case search

In this section, we mainly introduce the implementation of the TSearcher module in BDTCR. From one simple view, TCR is a test case search process, where the search query is a test target and search results are relevant test cases found by TCR techniques. As shown in Fig. 3, in BDTCR, TSearcher takes a new test target m as input and outputs search results (i.e., recommended test cases TC). TSearcher encompasses multiple components which cooperate to complete five processes: ① extracting m's code and comment (Query Extraction), ② loading a test target m' from TCC (TT Loading), ③ calculating the basic distances between m and m' on code snippets and comments (DL, DG, and DC Calculation), ④ calculating the balanced distance based on DL, DG, and DC (BD Calculation), ⑤ adding m' to a similar list of test targets (M^S), sorting items in M^S based on BD, and determining whether to terminate the search (Search Results Renewal). If the search is finished, TSearcher outputs the test cases TC belonging to M^S, otherwise, it returns to the process ②. We discuss these processes in detail in the following subsections.

3.3.1. Process ①: Query extraction

Once receiving a new test target given by the developer, TSearcher will perform some preprocessing and extract important features from code snippets and comments as queries used in the subsequent processes.

Specifically, in this process, TSearcher takes a test target m given by the developer as input, then separates the code snippet and comments. For the code part, TSearcher first removes all comments in the method body and gets a clean version of the code snippet. Then, TSearcher utilizes an open-source tool PROGEX[6] to generate control flow graphs (CFGs) of the clear version code snippet. Further, TSearcher removes extra white spaces in the code snippet, leaving only a single white space between every two words. In other words, TSearcher converts the formatted code snippet with indentation into a string of words. We call such a string of words the literal text of the code snippet.

For the comment part, three kinds of comments are taken into account, Single-Line comments, Multi-Line (also called Block) comments, and Javadoc comments [39,49]. For the Javadoc comments, TSearcher only extracts the first sentence because such a sentence is a description of the method documented [50]. The research [51] provides us with inspiration for processing the comment text. TSearcher uses the Stanford CoreNLP tools [52] to extract keywords from the comment. In addition to nouns and verbs recommended in existing research [51,53,54], TSearcher also extracts adjectives that also contain important semantic information. For example, the adjectives 'maximum' and 'minimum' in the comment */* calculate the maximum value */* and */* calculate the minimum value */* have opposite semantics.

3.3.2. Process ②: Test target loading (TT loading)

Test target loading is responsible for loading test targets that are compared to the new test target m given by the developer.

In practice, after TCC is constructed, TConstructer further generates the literal text, CFGs, and keywords for each test target by using the same method as that used in the process ①. Therefore, in the TT Loading process, TSearcher directly loads the literal text, CFGs, and keywords of each test target and its test cases from TCC in sequence. This information would further be compared to the queries extracted from m through the process ①. TSearcher executes the TT Loading process in the following two cases:

Case 1: When TSearcher receives a retrieval request (that is, an input 'm') from the developer, it executes the process once.

Case 2: TSearcher executes the process again when the search termination condition is not met, such as when the number of search results does not reach the expected number. This case is discussed in detail in Section 3.3.5.

[6] https://github.com/ghaffarian/progex.

3.3.3. Process ③: Basic distances (DL, DG and DC) calculation

Basic distances (i.e., DL, DG, and DC) determine the similarity of the two test targets and whether they can reuse each other's test cases. Therefore, the calculation of the basic distances is a fundamental work in TSearcher. Below we introduce in detail how TSearcher calculates these distances.

Distance between Comments (DC). TSearcher draws on the prior work [51] to process comments. As shown in Eq. (2), we slightly adjust the Jaccard Index to calculate the similarity between two keyword sets, in which, K_i and K_j denote the keyword sets of natural language descriptions (i.e., comments); $synonyms(K_i, K_j)$ is an interface used to calculate the number of synonym pairs found in K_i and K_j; and $AdaptedJI(K_i, K_j)$ denotes the similarity between two keyword sets K_i and K_j. $DC(m_i, m_j)$ in Eq. (1) denotes the distance between comments of two methods m_i and m_j.

We utilize the WordNet [55] to process the problem of languages with relatively more prevalent polysemy (i.e., many possible meanings for a word or phrase). Specifically, we search for the synonyms of keywords from WordNet and take them into account when calculating the distance of comments to avoid the negative impacts of synonyms on analysis.

$$DC(m_i, m_j) = \begin{cases} -1, & \text{if } K_i \text{ or } K_j \text{ is } \emptyset \\ 1 - AdaptedJI(K_i, K_j), & otherwise \end{cases} \tag{1}$$

$$AdaptedJI(K_i, K_j) = \frac{|K_i \cap K_j| + synonym(K_i, K_j)}{|K_i \cup K_j| - synonym(K_i, K_j)} \tag{2}$$

Distance between Literal text (DL). Chaiyong et al. [56] conducted extensive research on the code similarity analyzer. In their research, the similarity analysis of the method-level code is also regarded as the Boiler-plate code detection problem. Boiler-plate code occurs when developers reuse a method code to achieve a particular task. Compared with specialized code similarity tools, Difflib [57] got the highest AUC (Area Under ROC Curve) in the scenario of Boiler-plate code detection, although it is a general textual similarity measure tool. Specifically, it provides an interface *ratio()* that returns a measure of the sequences' similarity as a float in the range [0, 1] where 1.0 if the sequences are identical, and 0.0 if they have nothing in common. The calculation formula behind the *ratio()* interface is shown in Eq. (3).

$$sim(s_1, s_2) = 2 * \frac{matches(s_1, s_2)}{|s_1| + |s_2|} \tag{3}$$

$matches(s_1, s_2)$ is the number of matches detected by Difflib. $|s_1|$ and $|s_2|$ represent the number of elements in s_1 and s_2, respectively. Note that Eq. (3) is asymmetric, i.e., $sim(s_1, s_2) \neq sim(s_2, s_1)$ because of $matches(s_1, s_2) \neq matches(s_2, s_1)$. Thus, we employ Eq. (4) to calculate the distance on the literal text.

$$DL(m_1, m_2) = 1 - max\{sim(s_1, s_2), sim(s_2, s_1)\} \tag{4}$$

Distance between control flow Graphs (DG). Referring to the research [58], we adopt the algorithm based on graph edit distance proposed by Hu et al. [59] to measure the similarity of CFGs. The algorithm can approximate the minimum number of edit operations needed to transform one graph into another. In this paper, we assume uniform costs of edit operations, i.e. $cost \equiv 1$. Besides, to ensure that the operations have unit costs, it only allows adding and deleting zero-degree nodes. In other words, if we want to delete a node, we must first delete its incident edges, and if we want to add a node with some incoming or outgoing edges, we must create the node at first [58]. The basic idea of the algorithm proposed by Hu et al. [59] is to (1) build a cost matrix that represents the costs of mapping the different nodes in the two CFGs. After that, (2) the Hungarian algorithm [60] is used to find matching between the nodes such that the total cost is minimized.

Step (1) *Building Cost Matrix:* Let N_1 and N_2 denote the sets of nodes for CFG_1 and CFG_2 respectively, and $|N_1|$ and $|N_2|$ are the number of nodes. Besides, $|N_2|$ also represents the number of dummy nodes that are added to N_1, and $|N_1|$ also represents the number of dummy nodes that are added to N_2. The cost matrix is thus a $(|N_1|+|N_2|)\times(|N_1|+|N_2|)$ square matrix (e.g., Fig. 4). It represents the cost of matching each of the nodes in CFG_1 to any node in CFG_2. Denote the elements in the cost matrix by e_{ij} where $i, j \in \{|N_1| + |N_2|\}$.

The cost matrix can be divided into four sub-matrices. The first sub-matrix is a $|N_1| \times |N_2|$ matrix at the top left corner. It denotes the cost of matching a real node in CFG_1 to a real node in CFG_2. The values of the elements in it are calculated by Eq. (5), in which the relabeling cost is the cost of editing the instructions in a node to make them the same as those in the node that is matched to it; $nei^-(n_i)$ and $nei^+(n_i)$ are the number of the in-neighbors and out-neighbors of node n_i, respectively. The second sub-matrix is a $|N_2|\times|N_1|$ matrix at the bottom right corner. It is a zero matrix as it represents matching a dummy node to a dummy node which costs nothing. The third sub-matrix is a $|N_2| \times |N_2|$ matrix at the top right corner. It represents the matching of a real node in CFG_1 to a dummy node which essentially means a node deletion. The values of the elements in it are calculated by Eq. (6), where $deg^-(n_i)$ and $deg^+(n_i)$ are the number of the in-degrees and out-degrees of node n_i, respectively. The elements not at the diagonal are set to ∞. The fourth sub-matrix at the bottom left corner is defined similarly to the third sub-matrix. As in the work [58], we also focus on the topology distance and thus ignore the relabeling cost. For example, Fig. 4 shows the cost matrix that we created for CFG_1 and CFG_2 is composed of four sub-matrices enclosed by dotted rectangles.

$$\begin{aligned} e_{ij} = \text{relabeling cost} &+ |nei^-(n_i) - nei^-(n_j)| \\ &+ |nei^+(n_i) - nei^+(n_j)| \end{aligned} \tag{5}$$

$$e_{ij} = 1 + deg^-(n_i) + deg^+(n_i) \tag{6}$$

Step (2) *Finding Best Match of the Nodes:* After the cost matrix is built, the target is to find matching between the nodes in CFG_1 and CFG_2 with the minimum cost. The cost of matching is given by the sum of the costs of the pairs in the matching according to the cost matrix. This is an instance of the *assignment problem* [58]. Hungarian algorithm [60] finds an optimal solution to the assignment problem in $O(n^3)$ time. As shown in Fig. 4, the elements circled by black circles represent matching between nodes in the two control flow graphs, which are found by the Hungarian algorithm. Adding up the costs of these elements, we get the total cost of these edit operations, which is **3**.

The distance between two CFGs is calculated by Eq. (7), among which N_i and E_i are the numbers of nodes and edges in CFG, and *the minimum cost* is got in Step (2).

$$DG(CFG_i, CFG_j) = \frac{the\ minimum\ cost}{|N_i| + |E_i| + |N_j| + |E_j|} \tag{7}$$

Based on the above theory, we use $DG(m_i, m_j)$ represent the distance between control flow graphs of two test targets m_i and m_j, i.e., $DG(m_i, m_j) = DG(CFG_i, CFG_j)$.

3.3.4. Process ④: Balanced Distance (BD) calculation

The balanced distance is used to balance the three basic distances from the process ③ and get a unique representation of similarity between two test targets.

Specifically, in this process, TSearcher first combines the two basic distances on the code snippets to produce a hybrid distance. TSearcher uses Eq. (8) to combine the distances between two test targets on the literal text and control flows to represent the distance on the code snippets, i.e., DB. Eq. (8) is a step-wise formula, where the first condition holds for when the distance on the literal text is 0 (i.e., $DL(m_i, m_j) = 0$) which means two test targets are identical. In this case, $DB(m_i, m_j) = 0$. In the next step, we weight the distances on the literal text and CFGs by adjusting two factors α and β, and $\alpha + \beta = 1$. The resulting hybrid distance DB represents the pairwise distance of the corresponding code snippets of two test targets m_i and m_j. Then, TSearcher uses Eq. (9) to combine the code snippet distance (DB) and the comment distance

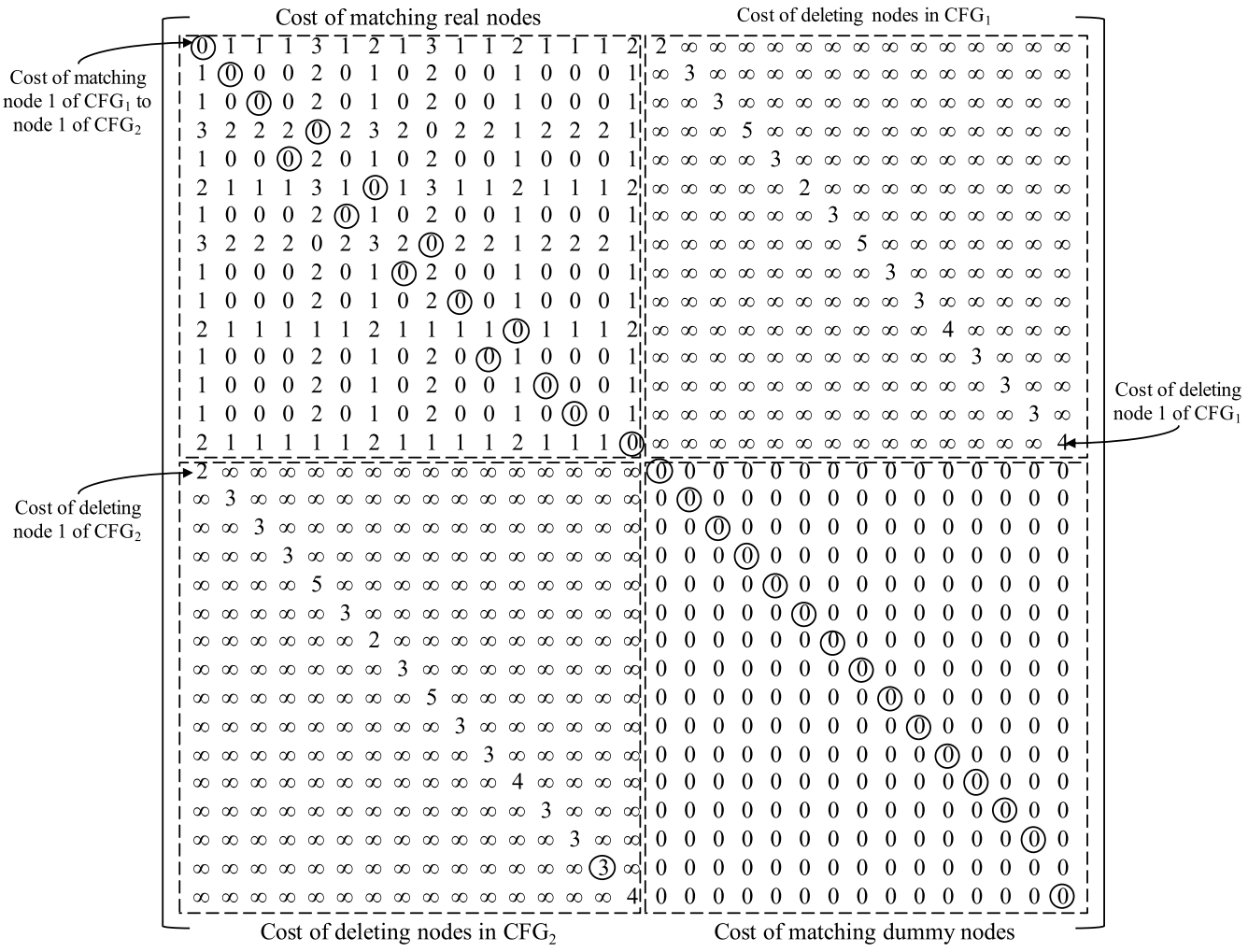


Fig. 4. An example of cost matrix.

(DC). We use $BD(m_i, m_j)$ to unique represent the similarity between two test targets m_i and m_j.

$$DB(m_i, m_j) = \begin{cases} 0, & \text{if } DL(m_i, m_j) = 0 \\ \alpha \times DL(m_i, m_j) + \beta \times DG(m_i, m_j), & otherwise \end{cases} \tag{8}$$

$$BD(m_i, m_j) = \begin{cases} 0, & \text{if } DB(m_i, m_j) = 0 \\ DB(m_i, m_j), & \text{if } DC(m_i, m_j) = -1 \\ \gamma \times DB(m_i, m_j), & \text{if } DC(m_i, m_j) = 0 \\ (1+\omega^2) \times \frac{DB(m_i,m_j) \times DC(m_i,m_j)}{\omega^2 DB(m_i,m_j) + DC(m_i,m_j)}, & otherwise \end{cases} \tag{9}$$

3.3.5. Process ⑤: Search Results Renewal

In this process, TSearcher first updates the search results. The search results are composed of a list of similar test targets and their corresponding test cases. Then, TSearcher needs to give a conclusion on whether to trigger the next search.

Specifically, TSearcher first updates a similar list of test targets M^S based on the results of the process ④. As shown in Fig. 3, TSearcher first augments M^S by add m' into it. Then, TSearcher sorts each element in M^S according to its balanced distance with m. The smaller the balanced distance is, the higher the ranking of the test target is. On the one hand, if the number of similar test targets in M^S is less than the expected number, TSearcher triggers the next search. On the other hand, if there are test targets in TCC that have not been compared, the next search will also be triggered. Both of the above conclusions on whether to trigger the next search correspond to Case 2 in Section 3.3.2.

3.3.6. Algorithm of TSearcher

TSearcher uses Algorithm 1 to complete the test case search. The algorithm's inputs are a new test target m_i and the expected number of search results k given by the developer. The algorithm's outputs are top k results related to m_i. Specifically, before the search, TSearcher initializes an empty set TC that is used to store recommended test cases. During the search, TSearcher first executes the processes ① (line 3) and ② (line 4) to obtain the code snippets and comments of two test targets (m_i and m_j). Then, TSearcher executes the process ③ and calculates the distance on the literal text (i.e., dl) of the code snippets (i.e., $m_i.code$ and $m_j.code$) of two test targets (line 5). It should be noted that if the conditional $dl == 0$ is true (line 6), the two code snippets are identical. In this case, there is no need to calculate dg, db, dc, and bd (i.e., lines 10–13), and TSearcher then jumps directly to the process ⑤ to continue execution. If dl is not zero, TSearcher executes the remainder of the process ③ (lines 10–12), and then executes the process ④ (line 13). After getting the balance distance (db) of the two test targets (m_i and m_j), TSearcher executes the process ⑤ (lines 14–22). TSearcher first augments the list of similar test targets M^S with a similar test target $\langle m_j, bd \rangle$ (line 14) and further sorts elements in M^S by db (line 15). If the number of the similar test targets in M^S is less than

Algorithm 1 Test Case Searcher

Input: m_i, // a new test target
k, //the expected number of search results
Output: a set of recommended test cases, TC;
1: $TC \leftarrow \emptyset$;
2: **for each** t_j in TCC **do**
3: $m_i.code, m_i.comment \leftarrow QExtraction(m_i)$;
4: $m_j.code, m_j.comment \leftarrow TTLoading(t_j)$;
5: $dl \leftarrow DL(m_i.code, m_j.code)$;
6: **if** $dl == 0$ **then**
7: $bd \leftarrow 0$;
8: **goto** line 14;
9: **end if**
10: $dg \leftarrow DG(m_i.code, m_j.code)$;
11: $db \leftarrow DB(dl, dg)$;
12: $dc \leftarrow DC(m_i.comment, m_j.comment)$;
13: $bd \leftarrow BD(dc, db)$;
14: $M^S \leftarrow M^S \cup \{\langle m_j, bd\rangle\}$;
15: $M^S \leftarrow$ sort M^S by the balanced distance;
16: **if** $|M^S| \leq k$ **then**
17: $TC \leftarrow TC \cup \{t_j\}$;
18: **continue**;
19: **else**
20: $M^S \leftarrow$ select the top k from M^S;
21: $TC \leftarrow$ update TC according to M^S;
22: **end if**
23: **end for**
24: output TC;

k (line 16), TSearcher stores test case t_j into TC (line 17) and triggers the next search (line 18). Otherwise, TSearcher selects the top k similar test targets from M^S according to the balanced distance as a new M^S (line 20), and further updates TC according to the new M^S (line 21). After the search, TSearcher outputs TC and recommends them to the developer, and the algorithm finishes.

4. Evaluation

In this section, we conduct experiments to evaluate BDTCR and compare it with existing TCR techniques. Since existing techniques did not introduce the test case construction process, in this section, we only compare the test case search process.

4.1. Research questions

To evaluate BDTCR comprehensively, we summarize critical problems into the following research questions and design experiments to answer them respectively:

- **RQ1:** How does the effectiveness of BDTCR compare with state-of-the-art techniques?
- **RQ2:** How does the granularity of query subjects impact the robustness of BDTCR?
- **RQ3:** How do distances in different aspects affect the performance of BDTCR?
- **RQ4:** How do developers perform aided by BDTCR?

4.2. Datasets

The evaluation of the search effect of test cases requires two datasets, one test case dataset used as a corpus to support retrieval, and one test target dataset used as query subjects used in retrieval.

Table 2
The statistics of test cases in TCC.

	JUnit 4.x	JUnit 5.x	Total
# Projects	2,133	268	2,401
# Test Cases	8,083	4,984	13,067

4.2.1. Test case corpus

There is no ready-made test case corpus that can be used to support retrieval. Existing studies [7,61] investigated the usage of the unit testing in open source projects. Their investigated results showed that although the unit testing coverage in most projects has not reached 100%, there are a large number of test cases in them. These test cases are invaluable assets and play a major role in determining the success or failure of a software system, which makes it possible to reuse or recommend test cases. To build a test case corpus, we choose the Java projects that have at least 20 stars from 2017 to 2018 in GitHub. After filtering the projects without test cases, we collect 2,401 useful projects. As shown in Table 2, TConstructer extracts more than 13,000 test cases which constitute the test case corpus (TCC).

4.2.2. Query subjects

For ease of evaluation, we simply refer to the test case recommendation as a test case search process. In this case, we regard the new test target as the search query used in the search process, and the search result is the recommended test case. Thus, we carefully select several representative queries to build a benchmark that satisfies the following criteria: (1) The test target must have a method comment. The method comment is beneficial for developers to understand what the method wants to do. (2) The test target is not a duplicate of the previous test targets. (3) The test target should be highly understandable, as we need to judge whether the recommended test cases are really relevant. Thus for the candidate targets (satisfying the first two criteria), two volunteers with 5-year Java programming experience conduct three rounds of manual inspection to identify whether the candidate targets are understandable or not, according to the previous work [62]. Accordingly, we randomly select 50 test targets from open-source Java projects to build a benchmark of queries Q.

In practice, different developers have different programming styles. No matter how rich TCC is, it cannot guarantee that it contains a test target that is the same as the new method under test. Thus, we aim to evaluate the robustness of BDTCR with a new practical application scenario, where for each query, there are no test targets in TCC that are the same as it. Specifically, we generate 50 new queries (Q') by making minor modifications to the benchmark queries mentioned above, including replacing a verb in the method name with synonyms and abbreviation expansion.

In summary, we prepare two sets of query subjects, Q, and Q', which contain a total of 100 test targets. We also employ postgraduates to conduct a two-phase checking. Firstly, each query is independently checked by two postgraduates to ensure the test targets are high understandable. Then, all queries are labeled with different results that are provided to the first author for final determination.

4.3. Comparison techniques

To evaluate BDTCR more accurately, we compare the following TCR techniques.

Tech 1: Baseline. The recommendation approach is based on the method signature exact matching. Specifically, if the method signatures of the two test targets are identical, they can reuse each other's test cases. This approach is the easiest to implement, so we treat it as a baseline technique.

Tech 2: TestTenderer. The traditional method signature can uniquely represent a method in the same class but cannot uniquely represent a method in different classes. The work *Test Tenderer* [29]

Table 3
An example of the (extending) method signatures.

	Method signature	Extending method signature
Abstraction	MN + PT	CN + MN + PT + RT
Example	integerRepresentation(String,int)	ALU.integerRepresentation(String,int):String

proposed a TCR technique that combines an extending method signature matching with a relaxation strategy. As shown in Table 3, an extending method signature is composed of the method signature (MN + PT), the class name (CN), and the return type (RT) of the test target. The relaxation strategy is applied in the recursive call in the matching algorithm, consists of four levels: (1) search for exact match of the query; (2) add wildcards to the method names; (3) remove the methods and search only for the class name; and (4) add wildcards to the class name.

Tech 3: NiCad-based. The recommendation approach is based on clone detection. Specifically, if two test targets are a clone pair, they can reuse each other's test cases. This approach has been adopted in the work [6] where a mature clone detector NiCad was used.

Tech 4: BDTCR. The recommendation approach we proposed, which measures the test target similarity based on the balanced distance, aims to improve the effectiveness and robustness of the recommendation.

4.4. Experiment setup

The NiCad configuration file allows users to configure different detection parameters, such as detection threshold (degree of dissimilarity for code fragments to be considered clones), and minimum and maximum size of what constitutes a cloned fragment. In this paper, we use the same configuration employed by the work [6] that configured NiCad with function-level granularity, using a blind setting and 0.1 dissimilarity threshold, setting the minimum and the maximum number of cloned lines to 5 and 2500, respectively.

We utilize the tool Difflib to measure the similarity of the literal text of two code snippets. We follow the configuration (i.e., $autojunk = true$ and $whitespace = true$) suggested by the researchers who presented the optimal configuration of Difflib for method-level code similarity measurement in [56]. When comparing lines as sequences of characters, $autojunk = true$ means enabling the automatic junk heuristic that treats certain sequence items as junks, and $whitespace = true$ means ignoring white space.

For BDTCR, we set the factors α and β that are used to adjust the weight of distance on the literal text and control flows to 0.75 and 0.25, respectively. The main reason for this setting is that, considering the complexity of unit test cases, to improve the understandability of the recommended test cases, we think that similarity in the literal text is more important than similarity in the control flow. The factor γ is set to 0.5, which enhances the similarity of two test targets when their comments are identical (i.e., $DC(m_i, m_j) = 0$). The factor ω is used to balance two different distances to 1, which means we equally weigh the two kinds of distances.

For each query $q \in Q \cup Q'$, two developers manually inspect the top 10 recommended test cases returned by different TCR techniques and label their relevance to q. Then they discuss the inconsistent labels and relabel them. The procedure repeats until a consensus is reached.

4.5. Evaluation metrics

We use three common metrics to measure the effectiveness of TCR: $SuccessRate@k$, $Precision@k$, and Mean Reciprocal Rank (MRR). They are widely used metrics in code recommendation literature [26,63–65].

The $SuccessRate@k$ (also known as success percentage at k [64]) measures the percentage of queries for which more than one correct result could exist in the top k ranked results [64,66,67]. It is calculated by Eq. (10) where Q is a set of queries (i.e., test targets), $\delta(\cdot)$ is a function which returns 1 if the input is true and 0 otherwise. $SuccessRate@k$ is important because a better TCR technique should allow developers to discover the needed test cases by inspecting fewer returned results. The higher the metric value, the better the test case search performance [26].

$$SuccessRate@k = \frac{1}{|Q|}\sum_{q=1}^{|Q|}\delta(FRank_q \le k) \tag{10}$$

where $FRank$ is the rank of the first hit result in the search result list [63,64].

The $Precision@k$ [65,68] measures the percentage of relevant results in the top k returned results for each query. It is calculated by Eq. (11), where $|RTC|$ is the number of related test cases (RTC) in the top k test cases. $Precision@k$ is important because developers often inspect multiple results of different usages to learn from [63]. A better test case search engine should allow developers to inspect less noisy results. The higher the metric value, the better the test case search performance. We evaluate $SuccessRate@k$ and $Precision@k$ when k's value is 1, 5, and 10. These values reflect the typical sizes of results that users would inspect [26,64].

$$Precision@k = \frac{|RTC|}{k} \tag{11}$$

$$MRR = \frac{1}{|Q|}\sum_{q=1}^{|Q|}\frac{1}{FRank_q} \tag{12}$$

The MRR [26,65] is the average of the reciprocal ranks of results of a set of queries Q. It is calculated by Eq. (12). The reciprocal rank of a query is the inverse of the rank of the first hit result [69]. The higher the MRR value, the better the test case search performance.

Table 4
Overall accuracy of four techniques in Q.

Techniques	$R@1$	$R@5$	$R@10$	$P@1$	$P@5$	$P@10$	MRR
Baseline	0.84	0.84	0.84	0.84	0.25	0.13	0.90
TestTenderer	1.00	1.00	1.00	1.00	0.46	0.26	1.00
NiCad-based	1.00	1.00	1.00	1.00	0.22	0.11	1.00
BDTCR	1.00	1.00	1.00	1.00	0.51	0.35	1.00

4.6. Results

4.6.1. Effectiveness of BDTCR

Table 4 shows the overall performance of the four techniques in Q, measured in terms of $SuccessRate@k$, $Precision@k$ and MRR. The columns $R@1$, $R@5$ and $R@10$ show the results of $SuccessRate@k$ when k is 1, 5 and 10, respectively. The columns $P@1$, $P@5$ and $P@10$ show the results of the average $Precision@k$ over all queries when k is 1, 5 and 10, respectively. The column MRR shows the MRR values of the four techniques.

Based on the results in Table 4, we can have the following observations. First of all, compared to Baseline, the other three techniques can find accurate results (test cases) for each query (new test target) and rank the results friendly in the top 1. Besides, the results show that BDTCR returns more relevant test cases than Baseline, TestTenderer, and NiCad-based. For example, the $R@5$ value is 1.00, which means that for 100% of the queries, the relevant test cases can be found within the top 5 return results. The $P@5$ value is 0.51, which means that 51% of the top 5 results are deemed accurate. For the $SuccessRate@5$

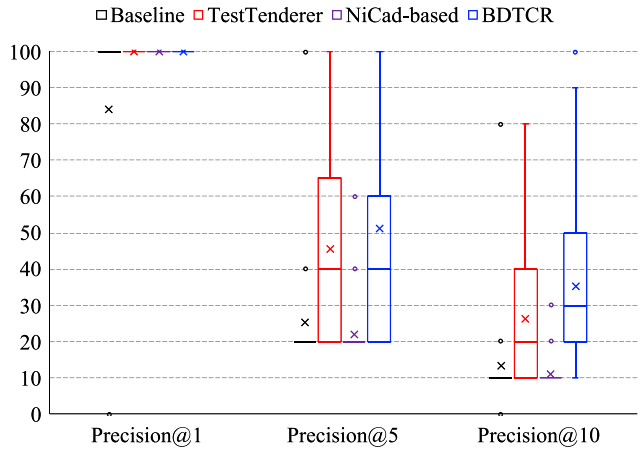


Fig. 5. The statistical comparison of *Precision@k* for four techniques.

Table 5
Overall accuracy of four techniques in Q'.

Techniques	*R@1*	*R@5*	*R@10*	*P@1*	*P@5*	*P@10*	*MRR*
Baseline	0.00	0.00	0.00	0.00	0.00	0.00	0.09
TestTenderer	0.98	1.00	1.00	0.98	0.42	0.25	0.99
NiCad-based	1.00	1.00	1.00	1.00	0.22	0.11	1.00
BDTCR	1.00	1.00	1.00	1.00	0.51	0.35	1.00

and *SuccessRate@10*, the improvements to TestTenderer are 11% and 35%, respectively. Fig. 5 shows the statistics of *Precision@k* for the three approaches when k is 1, 5 and 10, respectively. The symbol '+' indicates the average *Precision@k* value achieved by each technique. We observe that BDTCR achieves better overall precision values than the other three techniques.

4.6.2. Robustness of BDTCR

Table 5 shows the overall performance of the four techniques in Q'. Each query in Q' is derived by making minor modifications to the literal text of each query in Q while maintaining the same semantic (functionality). From the results in Table 5, we can observe that, in the new actual scenario, the performance of the four techniques has declined to varying degrees, among which Baseline has the most severe decline, almost completely failed; NiCad-based and BDTCR have the least decline.

4.6.3. Influence of subdivided distances on BDTCR

The balanced distance used by BDTCR consists of three subdivided distances, including the distance between literal text (DL), the distance between control flow graphs (DG), and the distance between comments (DC). Therefore, we carry out ablation experiments to understand the effects of these subdivided distances on the performance of BDTCR. The experimental results are shown in Table 6. In the table, BDTCR-DC and BDTCR-DC-DG are two variants of BDTCR, which represent remove the distance DC and both DC and DG from BDTCR, respectively.

From the table, we can observe that when the distances on comments and CFGs are removed, (1) the performance of BDTCR in terms of *P@5* and *P@10* degrades significantly; (2) the performance of BDTCR in terms of *R@k* and *MRR* remains stable. In the case of removing both DC and DG (i.e., BDTCR-DC-DG), we are still able to get stable *R@k* and *MRR*, which is attributed to the fact that we give more weight to the distance on the literal text. Based on the above results and observations, we can draw the conclusion that all the three subdivided distances (i.e., DL, DG, and DC), which promote each other, improve the performance of BDTCR jointly.

Table 6
Influence of different distances on BDTCR.

Techniques	*R@1*	*R@5*	*R@10*	*P@1*	*P@5*	*P@10*	*MRR*
BDTCR-DC	1.00	1.00	1.00	1.00	0.47	0.29	1.00
BDTCR-DC-DG	1.00	1.00	1.00	1.00	0.37	0.21	1.00
BDTCR	1.00	1.00	1.00	1.00	0.51	0.35	1.00

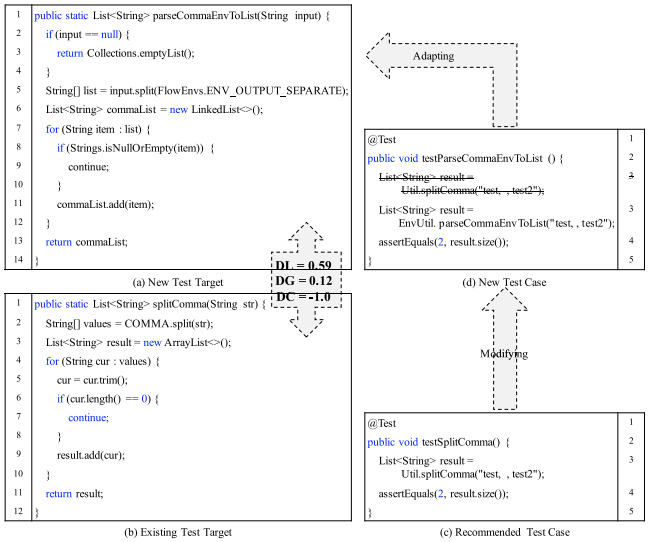


Fig. 6. A concrete example of test case recommendation.

4.6.4. Usefulness of BDTCR

Besides, to evaluate the usefulness of BDTCR in practice, we invite 40 students to perform unit test generation tasks and collect feedback. All the students are majoring in software engineering with rich experience to solve test problems by using BDTCR.

The statistics of feedback results demonstrate that 86.21% of them think, on the basis of meeting the test coverage goals, the number of test cases recommended by BDTCR is appropriate. 96.55% of them say that BDTCR is very helpful in improving the efficiency of test writing, which indicates that BDTCR obtains high user satisfaction.

The experiment also finds that most developers can complete the problem in 10–30 min and reach coverage requirements after using BDTCR, while it takes 1–2 h to achieve similar results without using BDTCR according to the former exam results of software testing courses. To sum up, BDTCR can recommend useful test cases, which brings great help to test writing.

5. Discussion

5.1. A case study of BDTCR

We now provide a concrete example of test case recommendation to demonstrate the advantages of BDTCR. Fig. 6(a) shows a new test target *parseCommaEnvoToList()* that is treated as a query and fed to BDTCR, and the searching result returned by BDTCR is a recommended test case as shown in Fig. 6(c). The test target in the test case *testSplitComma()* is the method *splitComma()* that shares a low similarity in the literal text with the new test target ($DL = 0.59$) although they implement the similar functionality. Both of Baseline and TestTenderer failed to search for the test case because the (extending) method signatures of two test targets are not matched. NiCad-based also failed to search for the test case because it thinks two test targets are not a clone pair. BDTCR successfully found this test case by measuring the similarity of two test targets more accurately based on the balanced distance. For example, two test targets in Fig. 6 show a high similarity in control

flows ($DG = 0.12$). Further, we can reuse the recommended test case *testSplitComma()* by simply modifying it and making it adapted to the new test target. For example, as shown in Fig. 6(d), the adjusted test case *testParseCommaEnvToList()* can be used to test the new test target *parseCommaEnvoToList()*.

5.2. The quality of test cases from BDTCR

During the evaluation, each recommended test case is scored −1, 0, and 1. Among them, 0 indicates that the test target of the recommended result is textually identical to the test target of the query. In other words, these test cases can be used directly without modification. Note that we default each recommended test case to be of high quality relative to its test target. Therefore, the quality of such recommended test cases is the highest. Recommended test cases with a score of −1 are considered irrelevant to the query, that is, of the worst quality. Recommended test cases with a score of 1 are considered relevant to the query but need to be modified appropriately to apply to the query. Modification usually consists of three parts: initialization, invocation, and assertion. To reduce the efforts in modifying them to adapt to the new test targets, the test cases we recommend to developers contain contexts to improve their understandability and reusability.

5.3. The advance of BDTCR

We have identified three advantages of BDTCR that may explain its effectiveness and robustness in test case recommendation:

Accurate measurement of test target similarity. The literal text, control flows, and comments contain valuable information that can reflect the functionality of the test target. By combining them to measure the similarity of the test target, BDTCR can effectively resist the negative effects caused by a deficiency in a single view (e.g., signature matching), thereby making the measurement more accurate.

Test-oriented criteria selection. We select the distance on control flows as one of the criteria instead of other structures (e.g., data flows), which helps developers to understand the branch distribution in the test target and improve the efficiency of branch coverage [35]. If the control flows of the two test targets are similar, it means that their branch distributions are similar.

High understandability test case recommendation. The test cases that we recommend to developers contain contexts that aim at improving their understandability and reusability. It helps to reduce the efforts in modifying them to adapt to the new test targets.

6. Threats to validity

In our experiments, the relevancy of returned results was manually graded and could suffer from subjectivity bias. To mitigate this threat, we employed postgraduates to conduct a two-phase checking, (i) the manual checking was performed independently by two developers; (ii) the developers performed an open discussion to resolve conflict grades for the 100 queries. We will further mitigate this threat by inviting more developers for the grading in the future. Besides, we refer to the work [26] and consider only the top 10 recommended test cases. Queries that fail are identically assigned with an $FRank$ of 11 and could be biased from the real relevancy of test cases. We believe that the setting is reasonable. In real-world code search, developers usually inspect the top k results and ignore the remaining. That means it does not make much difference if a test case appears at rank 11 or 20 if k is 10.

The test case recommendation techniques we compared do not provide publicly available tools or source code. Thus we implemented the corresponding tool prototypes according to the recommendation strategies described in their papers. There may be deficiencies in the implementation. To mitigate this threat, we show the specific configuration of each argument in Section 4.4 consistent with their papers. We make all the implementation code public available for further inspection.

7. Related work

7.1. Code similarity measurement

Test target similarity analysis is essentially the task of code similarity measurement. Code similarity measurement is a traditional and mature research field [56,70] and has many application scenarios, such as code search or recommendation [71,72], code clone detection [73–75], and code plagiarism detection [42,70,76]. A large number of code similarity measurement methods have been proposed one after another [56,70,73]. These methods mainly measure code similarity from two levels of text and structure. The textual-level methods include string-based [77,78] and token-based [79–81] methods. The structural-level methods include tree-based [82–84] and graph-based [85–87] methods. All of these methods have advantages and disadvantages. Overall, their performance (i.e., accuracy) has improved over time, but their complexity (including time cost) has also increased. In this paper, we mainly focus on applying code similarity measurement in test case recommendation scenarios. NiCad-based [6] utilizes a clone detector to measure test target similarity. TestTenderer [29] uses signature matching to represent test target similarity. Both NiCad-based and TestTenderer are textual-level methods. Different from them, BDTCR takes into account both textual and structural similarities to more accurately measure the test target similarity. At the textual level, considering the understandability of the recommended test cases, BDTCR focuses on the raw sequences of the code rather than the abstract tokens. At the structural level, BDTCR focuses on control flow because it is closely related to unit test coverage. In addition to the code part, BDTCR also considers comments, which provide a highly readable explanation of the functionality implemented by the test target [88].

7.2. Test case recommendation

Along with software testing development, test case search or recommendation [6,27,29,30] gets more and more attention. *Test Recommender* [30] and *TeSRS* [27] recommend test cases within the project itself to newcomers of the project, aiming at facilitating learning and test writing. But both of them require the project that newcomers join is rich in test cases. Thus, for a new project, it does not work. NiCad-based [6] recommends test cases mined from software repositories to developers with the help of clone detection techniques, aiming at supporting developers in creating new test cases. Compared with the works *Test Recommender* and *TeSRS*, NiCad-based can achieve cross-project test case recommendation. But NiCad-based only was evaluated on a few projects and did not compare with others' techniques. Werner et al. [29] first built a test case search engine SENTRE which contains a lot of test cases collected from the open web. Based on SENTRE, they developed a tool, namely TestTenderer, used to recommend test cases to developers. TestTenderer searches for test cases in SENTRE using method signatures matching and relaxation algorithm. TestTenderer can also be used for the cross-project test case recommendation. Unfortunately, we did not find either SENTRE or TestTenderer. In this paper, we build a large-scale test case corpus. Based on this corpus, we evaluate NiCad-based, TestTenderer, and our BDTCR in cross-project test case recommendation scenarios.

8. Conclusion

In this paper, we propose a test case recommendation technique named BDTCR. BDTCR measures test target similarity effectively based on the balanced distance that combines distances on the literal text, control flows, and comments. In addition, we implement the tool TConstructer that can be used to automatically extract test cases from complex source code. We build a test case corpus containing more than 13,000 test cases using TConstructer. Based on it, we conduct comprehensive experiments to evaluate BDTCR. The experimental results show

that BDTCR is effective and robust and is superior to the state-of-the-art techniques.

In the future, we plan to use some advanced techniques to measure the test target similarity, such as deep learning-based embedding techniques [24,74,84,87,89]. And further, it would be interesting to investigate how BDTCR performs when deployed in the regular workflow (e.g., IDE plugins) and collect feedback from developers to improve the practicality.

CRediT authorship contribution statement

Weisong Sun: Methodology, Experiment design, Investigation, Writing – original draft. **Quanjun Zhang:** Validation, Writing – review & editing. **Chunrong Fang:** Conceptualization, Resources, Writing – review & editing, Supervision, Project administration, Funding acquisition. **Yuchen Chen:** Software, Validation, Data curation. **Xingya Wang:** Writing – review & editing. **Ziyuan Wang:** Writing – review & editing.

Declaration of competing interest

No author associated with this paper has disclosed any potential or pertinent conflicts which may be perceived to have impending conflict with this work. For full disclosure statements refer to https://doi.org/10.1016/j.infsof.2022.106994.

Data availability

No data was used for the research described in the article.

Acknowledgments

The authors would like to thank the anonymous reviewers for insightful comments. This research is partially supported by the National Natural Science Foundation of China (61932012, 62141215) and Science, Technology and Innovation Commission of Shenzhen Municipality, China (CJGJZD20200617103001003).

References

[1] P. Ammann, J. Offutt, Introduction to Software Testing, Cambridge University Press, 2016.
[2] T. Xie, N. Tillmann, P. Lakshman, Advances in unit testing: theory and practice, in: Proceedings of the 38th International Conference on Software Engineering - Companion Volume, ACM, Austin, TX, USA, 2016, pp. 904–905.
[3] E. Daka, G. Fraser, A survey on unit testing practices and problems, in: Proceedings of the 25th International Symposium on Software Reliability Engineering, IEEE Computer Society, Naples, Italy, 2014, pp. 201–211.
[4] S. Planning, The Economic Impacts of Inadequate Infrastructure for Software Testing, National Institute of Standards and Technology, 2002.
[5] J. Lee, S. Kang, D. Lee, Survey on software testing practices, IET Softw. 6 (3) (2012) 275–282.
[6] M. Erfani, I. Keivanloo, J. Rilling, Opportunities for clone detection in test case recommendation, in: Proceedings of the 37th Annual Computer Software and Applications Conference, IEEE Computer Society, Kyoto, Japan, 2013, pp. 65–70.
[7] P.S. Kochhar, F. Thung, D. Lo, J.L. Lawall, An empirical study on the adequacy of testing in open source projects, in: Proceedings of the 21st Asia-Pacific Software Engineering Conference, IEEE Computer Society, Jeju, South Korea, 2014, pp. 215–222.
[8] M. Beller, G. Gousios, A. Panichella, A. Zaidman, When, how, and why developers (do not) test in their IDEs, in: Proceedings of the 10th Joint Meeting on Foundations of Software Engineering, ACM, Bergamo, Italy, 2015, pp. 179–190.
[9] R. Pham, S. Kiesling, O. Liskin, L. Singer, K. Schneider, Enablers, inhibitors, and perceptions of testing in novice software teams, in: Proceedings of the 22th International Symposium on Foundations of Software Engineering, ACM, Hong Kong, China, 2014, pp. 30–40.
[10] A. Sakti, G. Pesant, Y. Guéhéneuc, Instance generator and problem representation to improve object oriented code coverage, IEEE Trans. Softw. Eng. 41 (3) (2015) 294–313.
[11] G. Fraser, A. Arcuri, EvoSuite: automatic test suite generation for object-oriented software, in: Proceedings of the 19th Symposium on the Foundations of Software Engineering, ACM, Szeged, Hungary, 2011, pp. 416–419.
[12] P. Machado, A. Sampaio, Automatic Test-Case Generation, Springer Berlin Heidelberg, 2010, pp. 59–103.
[13] C. Pacheco, S.K. Lahiri, M.D. Ernst, T. Ball, Feedback-directed random test generation, in: Proceedings of the 29th International Conference on Software Engineering, IEEE Computer Society, Minneapolis, MN, USA, 2007, pp. 75–84.
[14] P. McMinn, Search-based software test data generation: a survey, Softw. Test. Verif. Reliab. 14 (2) (2004) 105–156.
[15] W. Sun, Z. Gao, W. Yang, C. Fang, Z. Chen, Multi-objective test case prioritization for GUI applications, in: Proceedings of the 28th Annual ACM Symposium on Applied Computing, ACM, Coimbra, Portugal, 2013, pp. 1074–1079.
[16] G. Fraser, M. Staats, P. McMinn, A. Arcuri, F. Padberg, Does automated unit test generation really help software testers? A controlled empirical study, ACM Trans. Softw. Eng. Methodol. 24 (4) (2015) 23:1–23:49.
[17] A. Arcuri, An experience report on applying software testing academic results in industry: we need usable automated test generation, Empir. Softw. Eng. 23 (4) (2018) 1959–1981.
[18] E.P. Enoiu, D. Sundmark, A. Causevic, P. Pettersson, A comparative study of manual and automated testing for industrial control software, in: Proceedings of the 10th International Conference on Software Testing, Verification and Validation, IEEE Computer Society, Tokyo, Japan, 2017, pp. 412–417.
[19] D. Serra, G. Grano, F. Palomba, F. Ferrucci, H.C. Gall, A. Bacchelli, On the effectiveness of manual and automatic unit test generation: ten years later, in: Proceedings of the 16th International Conference on Mining Software Repositories, IEEE / ACM, Montreal, Canada, 2019, pp. 121–125.
[20] S. Shamshiri, J.M. Rojas, J.P. Galeotti, N. Walkinshaw, G. Fraser, How do automatically generated unit tests influence software maintenance? in: Proceedings of the 11th International Conference on Software Testing, Verification and Validation, IEEE Computer Society, Västerås, Sweden, 2018, pp. 250–261.
[21] K. Mao, Y. Yang, Q. Wang, Y. Jia, M. Harman, Developer recommendation for crowdsourced software development tasks, in: Proceedings of the 8th Symposium on Service-Oriented System Engineering, IEEE Computer Society, San Francisco Bay, CA, USA, 2015, pp. 347–356.
[22] J. Wang, Y. Yang, S. Wang, Y. Hu, D. Wang, Q. Wang, Context-aware in-process crowdworker recommendation, in: Proceedings of the 42nd International Conference on Software Engineering, ACM, Seoul, South Korea, 2020, pp. 1535–1546.
[23] J. Wang, S. Wang, J. Chen, T. Menzies, Q. Cui, M. Xie, Q. Wang, Characterizing crowds to better optimize worker recommendation in crowdsourced testing, IEEE Trans. Softw. Eng. 47 (6) (2021) 1259–1276.
[24] J. Shuai, L. Xu, C. Liu, M. Yan, X. Xia, Y. Lei, Improving code search with co-attentive representation learning, in: Proceedings of the 28th International Conference on Program Comprehension, ACM, Seoul, Republic of Korea, 2020, pp. 196–207.
[25] W. Sun, C. Fang, Y. Chen, G. Tao, T. Han, Q. Zhang, Code search based on context-aware code translation, in: Proceedings of the 44th International Conference on Software Engineering, IEEE/ACM, Pittsburgh, USA, 2022, pp. 1–13.
[26] X. Gu, H. Zhang, S. Kim, Deep code search, in: Proceedings of the 40th International Conference on Software Engineering, ACM, Gothenburg, Sweden, 2018, pp. 933–944.
[27] R. Qian, Y. Zhao, D. Men, Y. Feng, Q. Shi, Y. Huang, Z. Chen, Test recommendation system based on slicing coverage filtering, in: Proceedings of the 29th International Symposium on Software Testing and Analysis, ACM, Virtual Event, USA, 2020, pp. 573–576.
[28] C. Zhu, W. Sun, Q. Liu, Y. Yuan, C. Fang, Y. Huang, HomoTR: Online test recommendation system based on homologous code matching, in: Proceedings of the 35th International Conference on Automated Software Engineering, IEEE, Melbourne, Australia, 2020, pp. 1302–1306.
[29] W. Janjic, C. Atkinson, Utilizing software reuse experience for automated test recommendation, in: Proceedings of the 8th International Workshop on Automation of Software Test, IEEE Computer Society, San Francisco, CA, USA, 2013, pp. 100–106.
[30] R. Pham, Y. Stoliar, K. Schneider, Automatically recommending test code examples to inexperienced developers, in: Proceedings of the 10th Joint Meeting on Foundations of Software Engineering, ACM, Bergamo, Italy, 2015, pp. 890–893.
[31] Oracle, The java ™ tutorials: Defining methods, 2021, site: https://docs.oracle.com/javase/tutorial/java/javaOO/methods.html. (Accessed 2021).
[32] J.R. Cordy, C.K. Roy, The NiCad clone detector, in: Proceedings of the 19th International Conference on Program Comprehension, IEEE Computer Society, Kingston, ON, Canada, 2011, pp. 219–220.
[33] H. Zhu, P.A.V. Hall, J.H.R. May, Software unit test coverage and adequacy, ACM Comput. Surv. 29 (4) (1997) 366–427.
[34] Q. Yang, J.J. Li, D.M. Weiss, A survey of coverage-based testing tools, Comput. J. 52 (5) (2009) 589–597.
[35] T.T. Chekam, M. Papadakis, Y.L. Traon, M. Harman, An empirical study on mutation, statement and branch coverage fault revelation that avoids the unreliable clean program assumption, in: Proceedings of the 39th International Conference on Software Engineering, IEEE / ACM, Buenos Aires, Argentina, 2017, pp. 597–608.

[36] TIOBE, TIOBE index for january 2021, 2021, site: https://www.tiobe.com/tiobe-index/. (Accessed 2021).

[37] Y. Poirierl, What are the most popular libraries java developers use? Based on github's top projects, 2021, site: https://blogs.oracle.com/java/top-java-libraries-on-github. (Accessed 2021).

[38] C. Watson, M. Tufano, K. Moran, G. Bavota, D. Poshyvanyk, On learning meaningful assert statements for unit test cases, in: Proceedings of the 42nd International Conference on Software Engineering, ACM, Seoul, South Korea, 2020, pp. 1398–1409.

[39] M. Demoney, S. Hommel, P. King, P. Naughton, J. Kaverna, K. walrath, Java Code Conventions, Sun Microsystems, California, 1997.

[40] X. Hu, G. Li, X. Xia, D. Lo, Z. Jin, Deep code comment generation, in: Proceedings of the 26th International Conference on Program Comprehension, ACM, Gothenburg, Sweden, 2018, pp. 200–210.

[41] E.M. Burke, B.M. Coyner, Java Extreme Programming Cookbook, O'Reilly Media, Inc., 2003.

[42] W. Sun, X. Wang, H. Wu, D. Duan, Z. Sun, Z. Chen, MAF: method-anchored test fragmentation for test code plagiarism detection, in: Proceedings of the 41st International Conference on Software Engineering, Software Engineering Education and Training, IEEE / ACM, Montreal, QC, Canada, 2019, pp. 110–120.

[43] X. Wang, W. Sun, L. Hu, Y. Zhao, W.E. Wong, Z. Chen, Software-testing contests: observations and lessons learned, IEEE Comput. 52 (10) (2019) 61–69.

[44] M. Weiser, Program slicing, in: Proceedings of the 5th International Conference on Software Engineering, IEEE Computer Society, San Diego, California, USA, 1981, pp. 439–449.

[45] S. Cordio, O. Contributors, AssertJ - fluent assertions java library, 2021, site: https://assertj.github.io/doc. (Accessed 2021).

[46] G. team, Truth - Fluent assertions for Java and Android, 2021, site: https://truth.dev. (Accessed 2021).

[47] S. Horwitz, T.W. Reps, D.W. Binkley, Interprocedural slicing using dependence graphs, ACM Trans. Program. Lang. Syst. 12 (1) (1990) 26–60.

[48] A. Qusef, R. Oliveto, A.D. Lucia, Recovering traceability links between unit tests and classes under test: An improved method, in: Proceedings of the 26th International Conference on Software Maintenance, IEEE Computer Society, Timisoara, Romania, 2010, pp. 1–10.

[49] Google, Google java style guide, 2021, site: https://google.github.io/styleguide/javaguide.html. (Accessed 2021).

[50] Oracle, How to write doc comments for the javadoc tool, 2021, site: https://www.oracle.com/technical-resources/articles/java/javadoc-tool.html. (Accessed 2021).

[51] Y. Feng, J.A. Jones, Z. Chen, C. Fang, Multi-objective test report prioritization using image understanding, in: Proceedings of the 31st International Conference on Automated Software Engineering, ACM, Singapore, 2016, pp. 202–213.

[52] C.D. Manning, M. Surdeanu, J. Bauer, J. Finkel, S.J. Bethard, D. McClosky, The stanford CoreNLP natural language processing toolkit, in: Association for Computational Linguistics System Demonstrations, 2014, pp. 55–60.

[53] A.-M. Popescu, O. Etzioni, Extracting product features and opinions from reviews, in: Natural Language Processing and Text Mining, Springer, 2007, pp. 9–28.

[54] E. Shutova, L. Sun, A. Korhonen, Metaphor identification using verb and noun clustering, in: Proceedings of the 23rd International Conference on Computational Linguistics, Tsinghua University Press, Beijing, China, 2010, pp. 1002–1010.

[55] G.A. Miller, WordNet: An Electronic Lexical Database, MIT Press, 1998.

[56] C. Ragkhitwetsagul, J. Krinke, D. Clark, A comparison of code similarity analysers, Empir. Softw. Eng. 23 (4) (2018) 2464–2519.

[57] P.S. Foundation, difflib – Helpers for computing deltas, 2021, site: docs.python.org/3.8/library/difflib.html. (Accessed 2021).

[58] P.P.F. Chan, C.S. Collberg, A method to evaluate CFG comparison algorithms, in: Proceedings of the 14th International Conference on Quality Software, IEEE, Allen, TX, USA, 2014, pp. 95–104.

[59] X. Hu, T. Chiueh, K.G. Shin, Large-scale malware indexing using function-call graphs, in: Proceedings of the 16th Conference on Computer and Communications Security, ACM, Chicago, Illinois, USA, 2009, pp. 611–620.

[60] H.W. Kuhn, The Hungarian method for the assignment problem, Nav. Res. Logist. Q. 2 (1–2) (1955) 83–97.

[61] P.S. Kochhar, T.F. Bissyandé, D. Lo, L. Jiang, An empirical study of adoption of software testing in open source projects, in: Proceedings of the 13rd International Conference on Quality Software, IEEE, Najing, China, 2013, pp. 103–112.

[62] Z.P. Fry, B. Landau, W. Weimer, A human study of patch maintainability, in: Proceedings of the 21st International Symposium on Software Testing and Analysis, ACM, Minneapolis, MN, USA, 2012, pp. 177–187.

[63] M. Raghothaman, Y. Wei, Y. Hamadi, SWIM: synthesizing what i mean: code search and idiomatic snippet synthesis, in: Proceedings of the 38th International Conference on Software Engineering, ACM, Austin, TX, USA, 2016, pp. 357–367.

[64] X. Li, Z. Wang, Q. Wang, S. Yan, T. Xie, H. Mei, Relationship-aware code search for JavaScript frameworks, in: Proceedings of the 24th International Symposium on Foundations of Software Engineering, ACM, Seattle, WA, USA, 2016, pp. 690–701.

[65] F. Lv, H. Zhang, J. Lou, S. Wang, D. Zhang, J. Zhao, CodeHow: effective code search based on API understanding and extended boolean model (E), in: Proceedings of the 30th International Conference on Automated Software Engineering, IEEE Computer Society, Lincoln, NE, USA, 2015, pp. 260–270.

[66] I. Keivanloo, J. Rilling, Y. Zou, Spotting working code examples, in: Proceedings of the 36th International Conference on Software Engineering, ACM, Hyderabad, India, 2014, pp. 664–675.

[67] X. Ye, R.C. Bunescu, C. Liu, Learning to rank relevant files for bug reports using domain knowledge, in: Proceedings of the 22th International Symposium on Foundations of Software Engineering, ACM, Hong Kong, China, 2014, pp. 689–699.

[68] L. Nie, H. Jiang, Z. Ren, Z. Sun, X. Li, Query expansion based on crowd knowledge for code search, IEEE Trans. Serv. Comput. 9 (5) (2016) 771–783.

[69] M. Grechanik, C. Fu, Q. Xie, C. McMillan, D. Poshyvanyk, C.M. Cumby, A search engine for finding highly relevant applications, in: Proceedings of the 32nd International Conference on Software Engineering, ACM, Cape Town, South Africa, 2010, pp. 475–484.

[70] Source-code similarity detection and detection tools used in academia: a systematic review, ACM Trans. Comput. Educ. 19 (3) (2019) 27:1–27:37.

[71] K. Kim, D. Kim, T.F. Bissyandé, E. Choi, L. Li, J. Klein, Y.L. Traon, FaCoY: a code-to-code search engine, in: Proceedings of the 40th International Conference on Software Engineering, ACM, Gothenburg, Sweden, 2018.

[72] F. Silavong, S. Moran, A. Georgiadis, R. Saphal, R. Otter, DeSkew-LSH based code-to-code recommendation engine, 2021, CoRR abs/2111.04473.

[73] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, E. Merlo, Comparison and evaluation of clone detection tools, IEEE Trans. Softw. Eng. 33 (9) (2007) 577–591.

[74] C. Fang, Z. Liu, Y. Shi, J. Huang, Q. Shi, Functional code clone detection with syntax and semantics fusion learning, in: Proceedings of the 29th International Symposium on Software Testing and Analysis, ACM, Virtual Event, USA, 2020, pp. 516–527.

[75] B. van Bladel, S. Demeyer, Clone detection in test code: an empirical evaluation, in: Proceedings of the 27th International Conference on Software Analysis, Evolution and Reengineering, IEEE, London, ON, Canada, 2020, pp. 492–500.

[76] D.-K. Chae, J. Ha, S.-W. Kim, B. Kang, E.G. Im, Software plagiarism detection: a graph-based approach, in: Proceedings of the 22nd International Conference on Information and Knowledge Management, 2013, pp. 1577–1580.

[77] S. Ducasse, M. Rieger, S. Demeyer, A language independent approach for detecting duplicated code, in: Proceedings of the 7th International Conference on Software Maintenance, IEEE Computer Society, Oxford, England, UK, 1999, pp. 109–118.

[78] J.H. Johnson, Identifying redundancy in source code using fingerprints, in: Proceedings of the 3rd Conference of the Centre for Advanced Studies on Collaborative Research, IBM, Toronto, Ontario, Canada, 1993, pp. 171–183.

[79] M.J. Wise, YAP3: Improved detection of similarities in computer program and other texts, in: Proceedings of the 27th Technical Symposium on Computer Science Education, ACM, Philadelphia, Pennsylvania, USA, 1996, pp. 130–134.

[80] Z. Li, S. Lu, S. Myagmar, Y. Zhou, CP-MIner: Finding copy-paste and related bugs in large-scale software code, IEEE Trans. Softw. Eng. 32 (3) (2006) 176–192.

[81] H. Sajnani, V. Saini, J. Svajlenko, C.K. Roy, C.V. Lopes, SourcererCC: scaling code clone detection to big-code, in: Proceedings of the 38th International Conference on Software Engineering, ACM, Austin, TX, USA, 2016, pp. 1157–1168.

[82] I.D. Baxter, A. Yahin, L.M. de Moura, M. Sant'Anna, L. Bier, Clone detection using abstract syntax trees, in: Proceedings of the 6th International Conference on Software Maintenance, IEEE Computer Society, Bethesda, Maryland, USA, 1998, pp. 368–377.

[83] L. Jiang, G. Misherghi, Z. Su, S. Glondu, DECKARD: scalable and accurate tree-based detection of code clones, in: Proceedings of the 29th International Conference on Software Engineering, IEEE Computer Society, Minneapolis, MN, USA, 2007, pp. 96–105.

[84] J. Zhang, X. Wang, H. Zhang, H. Sun, K. Wang, X. Liu, A novel neural source code representation based on abstract syntax tree, in: Proceedings of the 41st International Conference on Software Engineering, IEEE / ACM, Montreal, QC, Canada, 2019, pp. 783–794.

[85] J. Krinke, Identifying similar code with program dependence graphs, in: Proceedings of the 8th Working Conference on Reverse Engineering, IEEE Computer Society, Stuttgart, Germany, 2001, pp. 301–309.

[86] R. Komondoor, S. Horwitz, Using slicing to identify duplication in source code, in: Proceedings of the 8th International Static Analysis Symposium, Springer, Paris, France, 2001, pp. 40–56.

[87] G. Zhao, J. Huang, DeepSim: deep learning code functional similarity, in: Proceedings of the 17th Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ACM, Lake Buena Vista, FL, USA, 2018, pp. 141–151.

[88] J. Zhai, X. Xu, Y. Shi, G. Tao, M. Pan, S. Ma, L. Xu, W. Zhang, L. Tan, X. Zhang, CPC: automatically classifying and propagating natural language comments via program analysis, in: Proceedings of the 42nd International Conference on Software Engineering, ACM, Seoul, South Korea, 2020, pp. 1359–1371.

[89] L. Xu, H. Yang, C. Liu, J. Shuai, M. Yan, Y. Lei, Z. Xu, Two-stage attention-based model for code search with textual and structural features, in: Proceedings of the 28th International Conference on Software Analysis, Evolution and Reengineering, IEEE, Honolulu, HI, USA, 2021, pp. 342–353.