MuTCR: Test Case Recommendation via Multi-Level Signature Matching

Weisong Sun[1], Weidong Qian[2], Bin Luo[1], Zhenyu Chen[1]
[1]State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, China
[2]China Ship Scientific Research Center, Wuxi, China
weisongsun@smail.nju.edu.cn, qianwd@cssrc.com.cn, luobin@nju.edu.cn, zychen@nju.edu.cn

***Abstract*—Off-the-shelf test cases provide developers with testing knowledge for their reference or reuse, which can help them reduce the effort of creating new test cases. Test case recommendation, a major way of achieving test case reuse, has been receiving the attention of researchers. The basic idea behind test case recommendation is that two similar test targets (methods under test) can reuse each other's test cases. However, existing test case recommendation techniques either cannot be used in the cross-project scenario, or have low performance in terms of effectiveness and efficiency. In this paper, we propose a novel test case recommendation technique based on multi-level signature matching. The proposed multi-level signature matching consists of three matching strategies with different strict levels, including level-0 exact matching, level-1 fuzzy matching, and level-2 fuzzy matching. For the query test target given by the developer, level-0 exact matching helps to retrieve exact recommendations (test cases), while level-1 and level-2 fuzzy matching contribute to discovering richer relevant recommendations. We further develop a prototype called MuTCR for test case recommendation. We conduct comprehensive experiments to evaluate the effectiveness and efficiency of MuTCR. The experimental results demonstrate that compared with the state-of-the-art, MuTCR can recommend accurate test cases for more test targets. MuTCR is faster than the best baseline by three times based on the time cost. The user study is also performed to prove that the test cases recommended by MuTCR are useful in practice.**

***Index Terms*—unit test case, test case recommendation, test reuse**

I. Introduction

Software reuse has some strong economic value, such as increasing programmer productivity and software quality [1], [2]. Modern code search engines support indexing vast varieties of reusable software code artifacts, including a large number of unit test cases along with production code [3]. Unit test cases writing and execution are done by the developer to make sure that individual units of the production code are working as expected. They can expose bugs early in the software development life cycle [4], [5]. Thus, unit test cases have been widely recognized as important and valuable assets and built into and shipped with the production code [6], [7].

However, the unit test coverage is not high in most open-source projects. After investigating the existing studies [8]–[12], we found that the main reason is that in pursuit of agile development, most of the developers' efforts are focused on production code development and with a dismissive and negative view on test cases [12], [13]. Besides, writing unit test cases not only is a time-consuming task but also requires developers to have a lot of testing knowledge. Therefore, some researchers [9], [14]–[16] envision that, if these valuable test cases could be excavated and recommended to developers for their reference and reuse when writing test cases, it will effectively improve their productivity.

A fundamental problem in test case recommendation is to locate potential test cases in the search corpus for a given test target. The basic idea that two similar test targets (i.e., methods under test) can reuse each other's test cases, has been widely adopted by test case recommendation techniques [9], [12], [14], [16], [17]. Formally, given a test case tc_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 very similar measured by a similarity function $sim(\cdot)$, the tc_i would be recommended to m_j. For example, as shown in Fig. 1, two test targets m_1 and m_2 are identical in functionality, and thus we can recommend the test case tc_1 that belongs to m_1 to the new test target m_2. Obviously, only by accurately measuring test target similarity can we make ensure the recommended test cases are accurate.

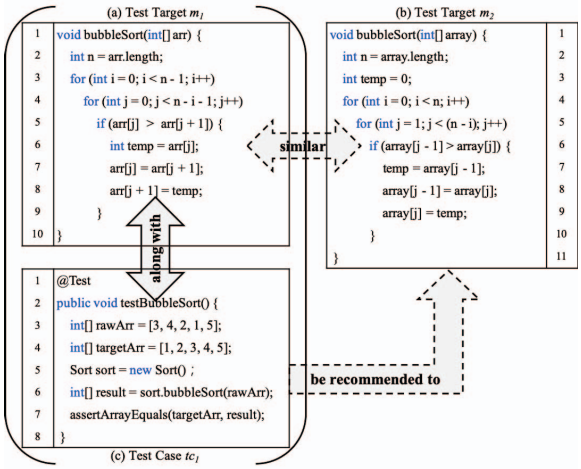


Fig. 1: An example of test case recommendation

Existing test case recommendation techniques can be roughly divided into two categories: *in-project* test case recommendation [12], [17] and *cross-project* test case recommendation [9], [14], [16]. The *in-project* techniques recommend test cases of the project currently under test to developers,

2833-9061/23/$31.00 ©2023 IEEE
DOI 10.1109/AST58925.2023.00022

while the *cross-project* techniques can recommend test cases from other projects to developers. All of them provide us with some valuable insights and inspiration, but they also have some deficiencies. We pay more attention to the *cross-project* techniques because the *in-project* techniques can not work when the project currently under test does not have test cases. The technique in the work [14] designs $sim(\cdot)$ based on method signature information. The method signature information is represented as a three-tuple $\langle PN, CN, S\rangle$ where PN (Package Name) and CN (Class Name) are the names of the package and class where the method appears; S (Signature) is the method signature that consists of the method name (MN) and parameter type list (PT). The $sim(\cdot)$ designed based on method signature information determines whether two test targets are similar by comparing the matching degree of the signature information. Its main advantage is fast, but its disadvantage is low query coverage [1]. For example, assuming that the method name of m_2 (in Fig. 1) is changed to 'sortArrayWithBubbleAlgorithm', TestTenderer [14] would not recommend the test case tc_1 to m_2 because m_1 and m_2 do not match on method names. Mostafa et al. [9] proposed another technique for *cross-project* test case recommendation. They designed $sim(\cdot)$ based on code similarity, which determines whether two test targets are similar by measuring the code similarity between them. Its main advantage is high accuracy, but its disadvantages are low query coverage and high time cost. The code is longer and more complex than the signature information. For example, the work [9] employs a mature clone detector NiCad [18] to measure the code similarity between two test targets. NiCad first transforms the code into a blind mode, and then uses the longest common subsequence matching algorithm to measure the code similarity in blind mode. Both steps are time-consuming. Relatest proposed by Robert et al. [16] is also based on code similarity. Relatest uses the Jaccard index over 3-grams implemented in the Java String Similarity library [19] to measure the code similarity. More details are shown in Section III-C.

In this paper, we propose a multi-level signature matching strategy for test case recommendation. The matching strategy consists of six matching rules that are divided into three groups: level-0 exact matching and two different levels (Level-1 and Level-2) of fuzzy matching. Level-0 exact matching requires that the class name and method name of the corpus test target must be the same as the query test target, while level-2 fuzzy matching only requires the same class name or method name. Based on the proposed signature matching strategy, we further develop a prototype for test case recommendation named MUTCR. MUTCR retrieves matching test targets in the search corpus from strict to loose as instructed by the proposed signature matching strategy. Compared with the matching strategy proposed by TestTenderer [14], our signature matching strategy is sophisticated and productive.

[1]Given a method under test (we also refer to it as a query test target), if the test case recommendation technique can recommend at least one test case for it, we claim that the technique can cover the query test target. More details are shown in Section III-B

We conduct comprehensive experiments to evaluate the effectiveness and efficiency of MUTCR. Compared with the state-of-the-art, MUTCR can rapidly recommend accurate test cases for more test targets. In addition, we perform a user study to evaluate the usefulness of the test cases recommended by MUTCR. The statistical results of the user study prove that MUTCR can recommend useful test cases for more test targets than the state-of-the-art baselines.

In summary, we make the following contributions:

- We propose a novel and lightweight $sim(\cdot)$ that measures the test target similarity accurately and rapidly through multi-level signature matching.
- We develop a prototype for test case recommendation named MUTCR and evaluate it comprehensively. The evaluation results show that MUTCR can recommend more accurate test cases than three baselines. The user study also proves that the test cases recommended by MUTCR are useful in practice.

II. DESIGN

Fig. 2 illustrates the overview of MUTCR. Given a query test target, MUTCR decomposes the test case recommendation process into three phases: signature information extraction, similar test target search, and test case loading. Algorithm 1 details the collaboration of the three phases. The input to MUTCR consists of a query test target (q), the expected number of recommendations (k), and the set of test targets (C). The output is a list of recommended test cases. MUTCR initializes the necessary variables in lines 1-4. C^q, C^{CMN}, C^{CN}, and C^{CN} are lists. C^q is used to store the test targets that are similar to q retrieved by MUTCR. C^{CMN} is used to store test targets with the same class name (CN) and method name (MN) as q. C^{CN} and C^{MN} are used to store test targets with the same class name and method name as q, respectively. Given a query test target, MUTCR extracts signature information from it and its code context, including class name, method name, and parameter types (PT) (lines 7-9, corresponding to the step ① in Fig. 2). The code context represents the class in which the query test target appears. Details are discussed in Section II-A. In the second phase, MUTCR introduces a multi-level signature matching strategy to retrieve the top k similar corpus test targets in C (lines 11-26). The multi-level signature matching strategy includes level-0 exact matching (corresponding to the step ② in Fig. 2 and detailed in Section II-B), level-1 fuzzy matching (step ③ and detailed in Section II-C), and level-2 fuzzy matching (step ④ and detailed in Section II-D). It should be noted that the multi-level matching strategy is conditionally executed sequentially rather than in parallel (lines 15 and 22). After retrieving similar test targets, MUTCR further loads their test cases and recommends them to the developers (lines 29-30).

A. Signature Information Extraction

In this phase, MUTCR aims to extract signature-related information from the given query test target. The signature-related information includes class name, method name, and

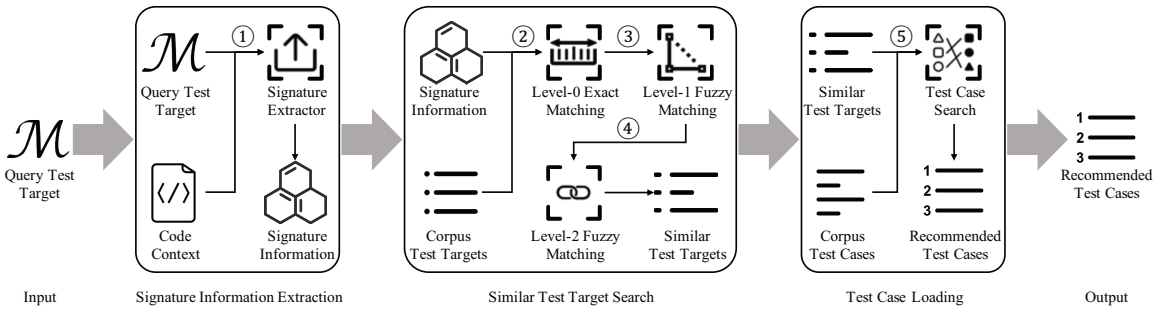


Fig. 2: Framework of MUTCR

Algorithm 1 Multi-Level Signature Matching

INPUT: q query test target
k expected number of recommendations
C corpus test target set
OUTPUT: T recommended test cases

1: $C^q \leftarrow \emptyset$ ▷ list of corpus test targets similar to q
2: $C^{CMN} \leftarrow \emptyset$ ▷ test targets with same CN and MN
3: $C^{CN} \leftarrow \emptyset$ ▷ test targets with same CN
4: $C^{MN} \leftarrow \emptyset$ ▷ test targets with same MN
5:
6: ▷ Signature Information Extraction
7: $CN^q \leftarrow$ extract class name of q from its context
8: $MN^q \leftarrow$ extract method name of q
9: $PT^q \leftarrow$ extract parameter types of q
10:
11: ▷ Level-0 Exact Matching
12: $C^q, C^{CMN}, C^{CN}, C^{MN} \leftarrow$ L0EXACTMATCHING$(CN^q, MN^q, PT^q, k.C)$
13:
14: ▷ Level-1 Fuzzy Matching
15: **if** $|C^q| < k$ **then**
16: $n \leftarrow k - |C^q|$
17: $C^{q1} \leftarrow$ L1FUZZYMATCHING$(C^{CMN}, C^{CN}, C^{MN}, PT^q, n)$
18: $C^q \leftarrow$ append C^{q1} to C^q
19: **end if**
20:
21: ▷ Level-2 Fuzzy Matching
22: **if** $|C^q| < k$ **then**
23: $n \leftarrow k - |C^q|$
24: $C^{q2} \leftarrow$ L2FUZZYMATCHING(C^{CN}, C^{MN}, n)
25: $C^q \leftarrow$ append C^{q2} to C^q
26: **end if**
27:
28: ▷ Test Case Loading
29: $T \leftarrow$ load test cases according to C^q
30: **return** T

parameter type. Like [14], by default, the code context for the query test target is provided along with the query test target by the developer. MUTCR extracts the class name from the code context.

B. Level-0 Exact Matching

In the step of level-0 exact matching, MUTCR (1) searches for the corpus test targets that exactly match the query test target; (2) collects other corpus test targets that are related to the query test target. Algorithm 2 shows the implementation details of the level-0 exact matching step. The input of Algorithm 2 includes the class name (CN^q), method name

Algorithm 2 Level-0 Exact Matching

INPUT: CN^q class name of q
MN^q method name of q
PT^q parameter types of q
k expected number of recommendations
C corpus test target set
OUTPUT: C^q list of corpus test targets similar to q
C^{CMN} test targets with the same CN and MN
C^{CN} test targets with the same CN
C^{MN} test targets with the same MN

1: **function** L0EXACTMATCHING(CN^q, MN^q, PT^q, k, C)
2: $C^q \leftarrow \emptyset$ ▷ list of corpus test targets similar to q
3: $C^{CMN} \leftarrow \emptyset$ ▷ test targets with same CN and MN
4: $C^{CN} \leftarrow \emptyset$ ▷ test targets with same CN
5: $C^{MN} \leftarrow \emptyset$ ▷ test targets with same MN
6: $count \leftarrow 0$
7: **for** each corpus test target $c \in C$ **do**
8: $CN^c \leftarrow$ extract class name of c
9: $MN^c \leftarrow$ extract method name of c
10: $PT^c \leftarrow$ extract parameter types of c
11: **if** $CN^c == CN^q$ **then**
12: **if** $MN^c == MN^q$ **then**
13: **if** $PT^c == PT^q$ **then**
14: ▷ Rule 1: $CN + MN + PT$
15: $C^q \leftarrow$ append c to C^q
16: $count \leftarrow count + 1$
17: **if** $count == k$ **then**
18: **break**
19: **end if**
20: **else**
21: $C^{CMN} \leftarrow$ append c to C^{CMN}
22: **end if**
23: **else**
24: $C^{CN} \leftarrow$ append c to C^{CN}
25: **end if**
26: **else**
27: $C^{MN} \leftarrow$ append c to C^{MN}
28: **end if**
29: **end for**
30: **return** $C^q, C^{CMN}, C^{CN}, C^{MN}$
31: **end function**

(MN^q), and parameter types (PT^q) of the query test target q, the expected number of recommendations k, and corpus test target set C. The output of Algorithm 2 includes four lists, i.e., C^q, C^{CMN} , C^{CN}, and C^{MN}. The meanings of these four lists are the same as in Algorithm 1. MUTCR compares the query test target with the corpus test target one by one (lines 7-29). Specifically, for each corpus test target $c \in C$, MUTCR first extracts its class name CN^c, method name MN^c, and

parameter types PT^c. If c and q have the same class name, method name, and parameter types, we say that c and q match exactly (lines 11-13). In this case, MUTCR appends c to C^q. When the number of retrieved corpus test targets reaches the expected number of recommendations k, MUTCR will stop searching, thereby improving the search efficiency of similar test targets (lines 17-19). If c and q have the same class name and method name, MUTCR will append c to C^{CMN} (line 21). If c and q only have the same class name, MUTCR will append c to C^{CN} (line 24). If c and q only have the same method name, MUTCR will append c to C^{MN} (line 27). C^{CMN}, C^{CN}, and C^{MN} will be used by MUTCR to further pick out recommendations in steps ③ and ④.

C. Level-1 Fuzzy Matching

Algorithm 3 Level-1 Fuzzy Matching

INPUT:	C^{CMN}	test targets with same class and method names
	C^{CN}	test targets with same class names
	C^{MN}	test targets with same method names
	PT^q	parameter types of the query q
	k	expected number of recommendations
OUTPUT:	C^q	list of corpus test targets similar to q

```
1: function L1FuzzyMatching(C^CMN, C^CN, C^MN, PT^q, k)
2:     count ← 0
3:     ▷ Rule 2: CN + MN + *
4:     for each corpus test target c ∈ C^CMN do
5:         C^q ← append c to C^q
6:         count ← count + 1
7:         if count == k then
8:             return C^q
9:         end if
10:    end for
11:
12:    ▷ Rule 3: CN + * + PT
13:    for each corpus test target c ∈ C^CN do
14:        PT^c ← extract parameter types of c
15:        if PT^c == PT^q then
16:            C^q ← append c to C^q
17:            count ← count + 1
18:            C^CN ← remove c from C^CN
19:        end if
20:        if count == k then
21:            return C^q
22:        end if
23:    end for
24:
25:    ▷ Rule 4: * + MN + PT
26:    for each corpus test target c ∈ C^MN do
27:        PT^c ← extract parameter types of c
28:        if PT^c == PT^q then
29:            C^q ← append c to C^q
30:            count ← count + 1
31:            C^MN ← remove c from C^MN
32:        end if
33:        if count == k then
34:            break
35:        end if
36:    end for
37:    return C^q
38: end function
```

Algorithm 3 shows the implementation details of the level-1 fuzzy matching in MUTCR. The input of Algorithm 3 consists of C^{CMN}, C^{CN}, C^{MN}, PT^q, and the expected number of recommendations k. The output is a list of corpus test targets retrieved by the level-1 fuzzy matching (C^q). In the level-1 fuzzy matching, MUTCR introduces three matching rules, including Rule 2 ($CN+MN+*$), Rule 3 ($CN+*+PT$), and Rule 4 ($*+MN+PT$). '*' represents a wildcard. Specifically, Rule 2 requires that the corpus test target and the query test target have the same class and method names (lines 4-10). Rule 3 requires that the corpus test target and the query test target have the same class name and parameter types (lines 13-23). Rule 4 requires that the corpus test target and the query test target have the same method name and parameter types (lines 26-36). It should also be noted that these three matching rules are also conditionally executed sequentially rather than in parallel (lines 7-9, 20-22, 33-35). Since step ④ will still continue to search for similar test targets with more relaxed conditions, the items that have been selected in step ③ should be removed to prevent repeated selection. MUTCR removes the items that have been selected from C^{CN} and C^{MN} (lines 18 and 31). When the number of retrieved corpus test targets reaches the expected number of recommendations k or all three matching rules have been tried, MUTCR will return the list of similar corpus test targets C^q (lines 8, 21, 37).

D. Level-2 Fuzzy Matching

Algorithm 4 Level-2 Fuzzy Matching

INPUT:	C^{CN}	test targets with same class name
	C^{MN}	test targets with same method name
	k	expected number of recommendations
OUTPUT:	C^q	list of corpus test targets similar to q

```
1: function L2FuzzyMatching(C^CN, C^MN, k)
2:     count ← 0
3: ▷ Rule 5: CN + * + *
4:     for each corpus test target c ∈ C^CN do
5:         C^q ← append c to C^q
6:         count ← count + 1
7:         if count == k then
8:             return C^q
9:         end if
10:    end for
11:
12: ▷ Rule 6: * + MN + *
13:    for each corpus test target c ∈ C^MN do
14:        C^q ← append c to C^q
15:        count ← count + 1
16:        if count == k then
17:            break
18:        end if
19:    end for
20:    return C^q
21: end function
```

Compared with the level-1 fuzzy matching, the level-2 fuzzy matching is more permissive. The level-2 fuzzy matching only requires that the corpus test target and the query test target have the same class name or method name. Algorithm 4 shows the implementation details of the level-2 fuzzy matching in MUTCR. The input of Algorithm 4 is composed of C^{CN}, C^{MN}, and the expected number of recommendations k. The output is a list of similar corpus test targets retrieved by the level-2 fuzzy matching (C^q). In the level-2 fuzzy matching, MUTCR introduces two matching rules, including Rule 5 ($CN+*+*$) and Rule 6 ($*+MN+*$). '*' also represents a wildcard. Matching Rule 5 and Rule 6 are also conditionally executed sequentially. When the number of retrieved corpus test targets reaches the expected number of recommendations

k or two matching rules have been executed, MUTCR will return the list of similar corpus test targets C^q (lines 8, 20).

E. Test Case Loading

In this phase, MUTCR loads test cases in the search corpus according to the similar test targets returned in the second phase. In practice, to build the test case search corpus, we develop an automated tool named TConstructer. To ensure the quality of the search corpus, TConstructer consists of 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. The test method extraction component is used to extract test methods from complex test code. The test dependency analysis component is used to extract test methods' contexts from the test code and production code. The test target recognition component is responsible for recognizing test targets tested by test methods. The test assert normalization component is responsible for normalizing test targets within test methods. For more details on TConstructer please read our previous work [20]. In our search corpus, test targets and test cases exist in pairs. Therefore, it is very direct and simple to load the corresponding test cases from the search corpus through the index of the similar test targets.

III. Evaluation

Our experimental study is designed to answer the following research questions.

RQ1: How effective is MUTCR compared with state-of-the-art test case recommendation techniques?

RQ2: How efficient is MUTCR compared with state-of-the-art test case recommendation techniques?

RQ3: How useful is MUTCR compared with the state-of-the-art TCR techniques in practice?

The research questions **RQ1** and **RQ2** investigate whether MUTCR recommends test cases accurately and efficiently. **RQ3** investigates the usefulness of recommended test cases.

A. Experiment Design

Considering the possible subjective bias in manual evaluation, in this paper, we try to automate the evaluation of test case recommendation techniques. We propose an automated evaluation framework for test case recommendation based on intuition, that is the test cases corresponding to two similar test targets should be similar. Fig. 3 shows the automated evaluation framework.

Specifically, with the guide of the work [21], we simply view the test case recommendation as a test case search process. In this case, the query test targets are regarded as the queries used in the search process. Then, different test case recommendation (TCR) techniques extract different information (e.g., method signature and code text) from queries as search conditions, and search results are the recommended test cases. The evaluation framework employs a similarity metric [22] between search results (i.e., the recommended test cases) and the ground-truth test cases to assess whether a

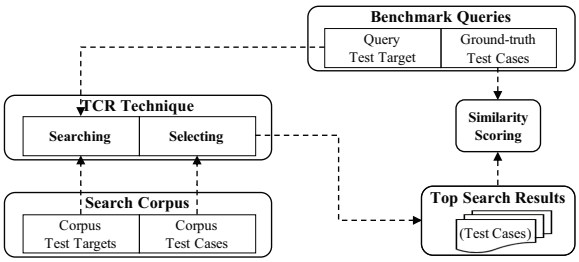


Fig. 3: Automated evaluation framework.

query was correctly answered. In other words, we compute the similarity value for each result with respect to the appropriate ground-truth test case. The higher the similarity value, the closer the recommended test case is to the ground-truth test case, which also indirectly indicates the more accurate the recommended test case of the corresponding TCR technique. According to the experimental design, the evaluation needs two datasets, i.e., the test case search corpus and the benchmark query dataset.

1) Search Corpus: There is no ready-made test case corpus that could be used to evaluate test case recommendations. Existing studies [10], [23] investigated the usage of the unit testing in open-source projects and found 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 key role in determining the success of a software system, which makes it possible to implement test case recommendations. To build a test case corpus supporting recommendations, we choose the Java projects that have at least 20 stars from January 2017 to December 2018 in GitHub. After filtering the projects without test cases, we collect 3,929 useful projects from 13,029. In total, we extract more than 136,000 test cases that constitute a search corpus.

2) Benchmark Query: We are the first to adopt an automated evaluation framework to evaluate test case recommendation techniques, and there is no readily available benchmark query dataset. The evaluation framework requires that every query must have a ground-truth result. In other words, each query test target has at least one ground-truth test case. To build the benchmark query dataset, we crawled 730 Java projects from GitHub that are created in January 2019. In the same way as the search corpus, we extract 12,473 test cases that covered 9,139 test targets. We further randomly select 1,000 from these test targets used as benchmark queries.

In order to detect whether the benchmark query dataset is representative, we compared the difference in the distribution of the number of code lines of test targets in the benchmark query dataset and search corpus. Specifically, we counted the number of code lines of all corpus test targets in the search corpus, and then divided them into 5 intervals. For example, [1,5) means that the number of code lines of the test target is greater than or equal to 1 and less than 5. We further

performed the same statistics and division on the number of code lines of the query test target in the benchmark query dataset. Fig. 4 shows the comparison of distributions of code lines of two datasets, where the blue curve connected by triangles represents the distribution of code lines of corpus test targets, and the orange curve connected by circles represents the query test targets in the benchmark queries. From the figure, we can observe that although these 1,000 queries were randomly selected, the distribution of the number of code lines of the two datasets is almost identical. This also indicates that the benchmark query dataset is representative.

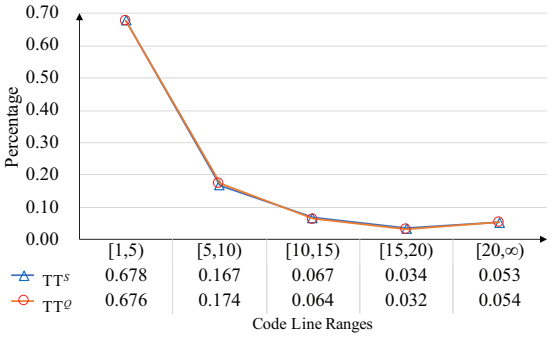


Fig. 4: The distribution of the code lines of two datasets.

B. Evaluation Metrics

Learning from the evaluation experience in the information retrieval and code search literature [24]–[28], we propose the following two evaluation metrics to measure the effectiveness of test case recommendation.

Query Coverage (QC). Given a query (test target), if the test case recommendation technique retrieves at least one test case for it, we deem that the query is covered. The query coverage (QC, for short) measures the percentage of queries in Q that are covered by the test case recommendation technique. It is computed as:

$$QC = \frac{1}{|Q|}\sum_{q=1}^{|Q|}\mathbb{1}(n > 0) \tag{1}$$

where Q is a set of queries (i.e., test targets); n is the number of query results (i.e., test cases) corresponding to the query q, and $\mathbb{1}(\cdot)$ is a function which returns 1 if the input is true and 0 otherwise. QC is important because a better test case recommendation technique should allow developers to search for test cases for more test targets. The higher the metric value, the better the test case recommendation performance.

Coverage Accuracy of the Top k Hits ($CA@k$). The closer the query result is to the ground-truth result, the more accurate the query result is. Specifically, we use the similarity metric between the recommended test case and the ground-truth test case to indicate accuracy. The larger the similarity, the more accurate the recommended test case. For each query, we measure the average accuracy ($Accuracy@k$, for short) of the top k query results when k is 1, 5, and 10. These values reflect the typical sizes of results that users would inspect. $Accuracy@1$ is important as users scan the search results from top to bottom [24]. A larger $Accuracy@1$ implies a lower inspection effort for finding the expected result. $Accuracy@5$ and $Accuracy@10$ are also important because developers often inspect multiple results of different usages to learn from [26]. $Accuracy@k$ is calculated as follows:

$$Accuracy@k = \frac{1}{k}\sum_{rank=1}^{n} ratio(tc^{rank}, tc^{ground\_truth}) \tag{2}$$

where k and n are the expected and actual number of query results respectively. $ratio(\cdot)$ is a function provided by the tool MAF [2] developed in our previous work [22] used to calculate the similarity between two test cases, where tc^{rank} and tc^{ground_truth} represent the recommended and ground-truth test cases, respectively.

In addition, we find that a test target may be tested by multiple test cases. Therefore, when the retrieved similar corpus test target has multiple recommended test cases, we compute the similarity between each recommended test case (i.e., tc^{rank}) and the ground-truth test case (i.e., tc^{ground_truth}), and then select the maximum value as the accuracy score.

The coverage accuracy of the top k hits ($CA@k$) measures the average accuracy of the results of the queries that have been covered by the test case recommendation technique. $CA@k$ is calculated as follows:

$$CA@k = \frac{1}{|Q'|}\sum_{q=1}^{|Q'|} Accuracy@k \tag{3}$$

where $Q' \subseteq Q$ is a set of covered queries. $Accuracy@k$ is used to measure the accuracy of a technique against a single query, and in reality, there may inevitably be coincidences - a technique happens to perform well on a given query but in fact, it performs poorly on other queries. $CA@k$ is a measurement of the accuracy of a set of queries, which can effectively avoid the coincidence problem, so it can more accurately reflect the effectiveness of a technique. A better test case recommendation technique should allow developers to search for test cases for any query test target as much as possible. The higher the metric values, the better the test case recommendation performance.

C. Comparison Techniques (Baselines)

In this paper, we compare MUTCR with the following three baselines.

TestTenderer. TestTenderer proposed by Werner et al. [14] combines the method signature matching with a relaxation strategy. The relaxation strategy is applied in the recursive call in the matching algorithm, and consists of four levels: 1) Search for the exact match of the query; 2) Add wildcards to the method names; 3) Remove the methods and search only for the class name; 4) Add wildcards to the class name. We did not find the test case search engine claimed to be public

[2]https://github.com/wssun/MAF

in their paper, so we implemented the search algorithm they proposed according to their paper description [14].

NiCadBased. The test case recommendation technique is based on clone detection, that is, if the two test targets are a clone pair, they can reuse each other's test cases. This approach has been adopted in the work [9] where a mature clone detector NiCad was used. This work also did not provide a tool that can be directly used for comparison. We thus also implemented its search algorithm according to the description of the paper [9]. For ease of description, we refer to their technique as NiCadBased for short. In the experiment, NiCad is configured with *function-level* granularity, *blind* setting, and 0.1 dissimilarity threshold.

Relatest. Relatest [16] follows the work by Ragkhitwetsagul et al. [29], [30] and designs $sim(\cdot)$ based on the Jaccard index [31]. Specifically, Relatest uses the Jaccard index over 3-grams implemented in the Java String Similarity library [19]. The pair-wise Jaccard index method retrieves a similar corpus test target in the search corpus and adds it to the candidate set if the similarity to the query test target is greater than or equal to a certain threshold τ. The Jaccard index is also a textual similarity measure. Therefore, like the threshold used by FuzzyWuzzy [32], τ is uniformly set to 0.6. The candidate set is then ranked by these similarity scores.

IV. Result Analysis

In this section, we present experimental results and the answer to each of the research questions posed.

RQ1: How effective is MuTCR compared with state-of-the-art test case recommendation techniques?

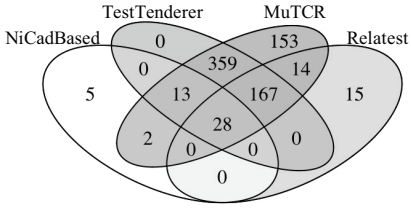


Fig. 5: The Venn diagram of queries covered by MuTCR and three baselines

We analyze the effectiveness of MuTCR from two automated evaluation metrics, i.e., query coverage and coverage accuracy. Fig. 5 shows the Venn diagram of queries covered by MuTCR and three baselines. Observe that MuTCR covers the most queries in Q ($736 = 2{+}13{+}28{+}359{+}167{+}14{+}153$, $QC \approx 0.74$), followed by TestTenderer ($567 = 13 + 28 + 359 + 167$, $QC \approx 0.57$), Relatest ($224 = 28 + 167 + 14 + 15$, $QC \approx 0.22$), and NiCadBased ($48 = 3 + 4 + 13 + 28$, $QC \approx 0.05$). Moreover, MuTCR covers most of the queries covered by NiCadBased (43/48=90%), Relatest (209/224=93%), and TestTenderer (567/567=100%).

In terms of coverage accuracy (i.e., $CA@k$), as shown in Table I, overall, MuTCR outperforms NiCadBased, Relatest, and TestTenderer when k is 1, 5 and 10. It should be noted that the values in Table I are the average scores of all covered queries.

TABLE I: The comparison between MuTCR and baselines on the intersection of covered queries in $CA@k$ and search efficiency.

Techniques	$CA@1$	$CA@5$	$CA@10$	time cost (ms)
NiCadBased	0.87	0.23	0.14	5,841
MuTCR	**0.89**	**0.60**	**0.46**	**7**
Relatest	0.78	0.31	0.20	5,343
MuTCR	**0.78**	**0.56**	**0.46**	**9**
TestTenderer	0.73	0.52	0.40	21
MuTCR	**0.74**	**0.53**	**0.43**	**7**

The average scores cannot reflect whether the improvement is significant. Fig. 6 shows the comparison results between MuTCR and three baselines in terms of $Accuracy(q)@k$ scores in detail. To test statistical significance, we apply the Wilcoxon matched-pairs signed-rank test to the comparison of $Accuracy(q)@k$ scores between MuTCR and three baselines. The Wilcoxon matched-pairs signed-rank test is a nonparametric method to compare 'before-after', or matched subjects [33]. In our case, 'before' refers to three baselines, whereas 'after' refers to MuTCR. The test results are presented in Fig. 6 in the form of '*'. For example, in Fig. 6(b), the '****' on the above NiCadBased and MuTCR indicates that there is a significant difference between NiCadBased and MuTCR in terms of $CA@5$ (i.e., p-value < 0.05).

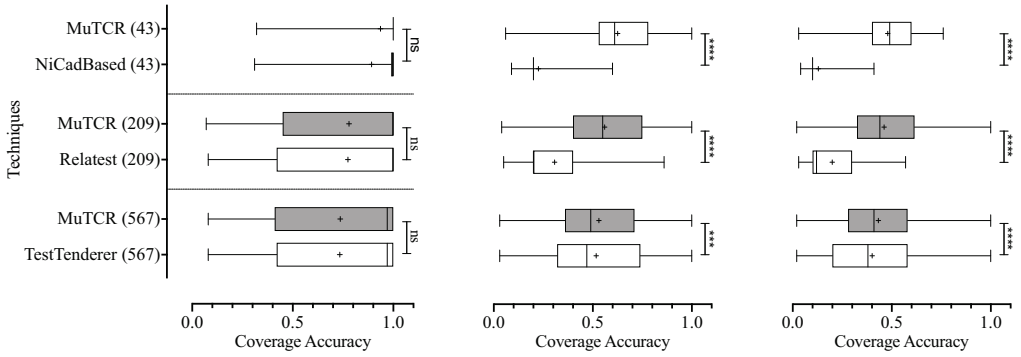
From Fig. 6, we can observe that, compared with NiCadBased, Relatest and TestTenderer, when k is 1, MuTCR does not have significant improvement, but it is significantly better than them when k is 5 and 10. In Relatest, the maximum recommendation list size is set to five (i.e., $k = 5$) as it has been shown that the average person is only able to reason about five to nine different items at a given time [34]. Therefore, we can conclude that MuTCR is significantly better than baselines under the common k setting ($k = 5$).

TABLE II: The comparison between MuTCR and baselines on all 1,000 queries in terms of $CA@k$ and search efficiency

Techniques	QC	$CA@1$	$CA@5$	$CA@10$	time cost
NiCadBased	0.05	0.05	0.02	0.01	5,343
Relatest	0.22	0.17	0.07	0.04	6,499
TestTenderer	0.57	0.42	0.29	0.23	23
MuTCR	**0.74**	**0.48**	**0.35**	**0.29**	**9**

In the above, we primarily compare MuTCR and baselines on the intersection of queries covered by them. We further compare them on all 1,000 benchmark queries, that is under the setting $|Q'| \equiv 1000$. Table II shows the results of MuTCR and three baselines on all 1,000 queries. From this table, we can observe that, MuTCR performs the best in QC and $CA@k$ and outperforms all three baselines. MuTCR improves the best baseline TestTenderer by 30% in QC, 14% in $CA@1$, 21% in $CA@5$, and 26% in $CA@10$.

Answer to RQ1.
Based on the above analysis, we can make a conclusion that, in terms of search effectiveness, MuTCR *is significantly better than the state-of-the-art techniques in terms of QC and $CA@k$. In other words,* MuTCR *can effectively recommend test cases to more test targets.*



(a) $k = 1$ (b) $k = 5$ (c) $k = 10$

Fig. 6: The distribution of the $Accuracy(q)@k$ scores attained by MUTCR and three baselines. In the figure, '+' denotes the mean, which is the value filled in Table I. '*' ($0.01 < p < 0.05$), '**' ($0.001 < p < 0.01$), '***' ($0.0001 < p < 0.001$), and '****' ($p < 0.0001$) represent the differences between the two groups are Significant, Very significant, Extremely significant, and Extremely significant, respectively. 'ns' ($p \geq 0.05$) means Not significant.

RQ2: How efficient is MUTCR compared with state-of-the-art test case recommendation techniques?

We further present the comparison results between MUTCR and three baselines in terms of search efficiency. For a given query, in addition to the search process, it also includes other follow-up processes such as loading test dependencies, transmitting search results to the client, and so on. To eliminate the other bias, we only compare the time it takes for different techniques to search the top k results for a given query. The last column 'time cost (ms)' in Table I and Table II shows the average time it takes for different test case recommendation techniques to search the top 10 results for any query in Q. From the two tables, we can observe that among three baselines, TestTenderer takes about 20-23 milliseconds (ms) to search the top 10 results for any query in Q, while NiCadBased and Relatest take 5–6 seconds. MUTCR takes the shortest time (about 7–9 ms). Compared to the best baseline TestTenderer, MUTCR improves test case search efficiency by 3 times.

Answer to RQ2.
In terms of search efficiency, MUTCR *is superior to NiCadBased, Relatest, and TestTenderer.* MUTCR *can search 10 results for a given query in a time scale of milliseconds on average. Thus,* MUTCR *can efficiently recommend test cases to test targets.*

RQ3: How useful is MUTCR compared with the state-of-the-art TCR techniques in practice?

In this section, we further investigate how useful is MUTCR compared with the state-of-the-art test case recommendation techniques in practice through a user study.

To conduct a user study, we recruit 10 graduate students to check the usefulness of test cases recommended by MUTCR, NiCadBased, Relatest, and TestTenderer. The 10 participants consist of 8 masters (including 6 first-year and 2 second-year masters) and 2 second-year PhD candidates. Their programming experience ranged from 3 years up to 6 years. Their testing experience ranged from 2 years to up to 4 years. We randomly select 50 queries from the 1,000 benchmark queries used in the automatic evaluation. We ask them to independently score the usefulness of test cases recommended by four techniques for these queries based on their development and testing experience. We mingle the ground-truth test cases with the test cases recommended by the four test case recommendation techniques (i.e., NiCadBased, TestTenderer, Relatest, and MUTCR), so that we can check whether the participants blindly trust these recommended test cases. Therefore, we can get 10 objective scores for each recommended test case. All scores are integers, ranging from 0 to 5, according to previous studies [35], [36]. Table III shows the score interpretation of the user study, where a score of 1 indicates the test case is useless, and a score of 5 indicates the test case is good enough and can directly be used to test the query test target without modification. Like the automatic evaluation, each technique recommends the top 10 test cases for each query.

TABLE III: The score interpretation

Score	Interpretation
1	The test case is useless.
2	The test case can provide test ideas.
3	The test case can provide test ideas and a small amount of reusable source code.
4	The test case can provide test ideas and a large amount of reusable source code.
5	The test case can be used without modification.

The statistical results of the user study are shown in Table IV. All values in the table are average scores calculated on 50 queries. The first column (i.e., $Score_{all}$) shows the average scores of all recommended test cases from TCR tools. The $Score@1$, $Score@5$, and $Score@10$ columns present the average scores of the top 1, 5, and 10 test cases, respectively. $Score@k$ is computed as:

$$Score@k = \frac{\sum_{i=1}^{k} score(tc_i)}{k} \quad (4)$$

where k is set to 1, 5, and 10; $score(tc_i)$ represents the average score of i-th test case scored by 10 students.

Table IV shows the usefulness scores of the recommended test cases on the queries covered by different techniques respectively. In the $Score_{all}$ column, we can observe that the ground-truth test case (i.e., Ground-truth) gets the highest score, followed by our MUTCR, and the worst is NiCadBased. From the $Score@1$ columns, we can observe that our MUTCR gets a score of 3.68, and outperforms TestTenderer (3.01), Relatest (1.15), and NiCadBased (0.29). According to the score interpretation in Table III, we can conclude that most of the test cases recommended by MUTCR can provide valuable references for developers to test the query test target. In addition, from the $Score@5$ and $Score@10$ columns, we can observe that all four techniques get low scores. The dominant reason is that the number of the test cases recommended by the four techniques is less than the expected k. In terms of $Score@5$ and $Score@10$, our MUTCR still outperforms the three baselines.

In Table IV, we only present the usefulness scores averaging over users and queries. To show the detailed results, we further present and compare the distribution of usefulness scores attained by our MUTCR and three baselines (NiCadBased, Relatest, and TestTenderer). Fig. 7 shows the distribution of the usefulness scores. From the figure, we can observe that the distribution of the usefulness scores of the recommendations found by MUTCR is significantly better than the three baselines under all k settings (i.e., $k = 1$, 5, and 10).

TABLE V: The relationships between usefulness score (US) and similarity score (SS).

US	SS
$[0.0, 1.0)$	0.00
$[1.0, 2.0)$	0.29
$[2.0, 3.0)$	0.38
$[3.0, 3.5)$	0.39
$[3.5, 4.0)$	0.45
$[4.0, 4.5)$	0.66
$[4.5, 5.0]$	0.95

TABLE VI: The number of useful recommendations

Techniques	k = 1	k = 5	k = 10
NiCadBased	41	44	45
Relatest	180	208	210
TestTenderer	446	498	513
MUTCR	**504**	**613**	**640**

We also analyze the correlation between the usefulness score used in the user study and the similarity score used in the automatic evaluation. As shown in Table V, we can observe that, in general, the usefulness score is positively correlated with the similarity score. When the similarity score between the recommended test case and the ground-truth test case is greater than 0.4, its usefulness score has exceeded 3.5. If we simply consider recommendation results with a similarity greater than 0.4 as useful recommendations, otherwise as useless recommendations, the performance of these four techniques is shown in Table VI. From this table, we can observe that, compared to NiCadBased, Relatest, and TestTenderer, MUTCR is able to recommend useful test cases for more queries (640 when $k = 10$).

Answer to RQ3.
The statistics of user feedback results demonstrate that (1) Compared to NiCadBased, Relatest, and TestTenderer, the test cases recommended by MUTCR *are more useful and closer to ground-truth test cases. (2)* MUTCR *can recommend useful test cases for more test targets than three baselines.*

V. THREATS TO VALIDITY

• *Internal Validity.*

The threat to internal validity lies in the implementation of the three baselines, i.e., TestTenderer, NiCadBased, and Relatest. To mitigate this threat, when implementing their algorithm, we set all involved configurations strictly as described in their paper.

In addition, the selection of participants in the human study might be a threat to internal validity. Due to a monetary limitation, following existing work [37]–[39], we recruit students instead of professional developers from industry, which may introduce a bias in our conclusions. To mitigate this threat, we select participants with varied experience (i.e., formal work experience, internship work experience, and limited work experience). Each recommended test case is evaluated by 10 human students, and we use the average score of the 10 students as the final score. Meanwhile, the empirical study performed by Salman et al. demonstrates that both students and professional developers have similar performance for a new software engineering task [40]. As such, we believe the selection strategy may not be a key point to our user study.

• *External Validity.*

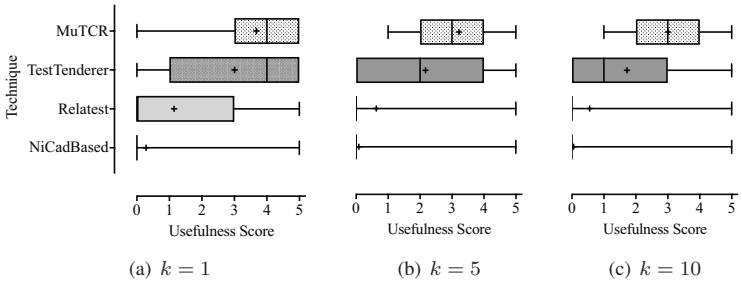
The threats to external validity mainly lie in the search corpus and benchmark queries. The richness of the test targets contained in the search corpus may affect the performance of different techniques. To mitigate the threat resulting from the search corpus, we used a large number of Java projects from GitHub to build the search corpus. In the future, we will continue to enrich the search corpus through more projects on Github or other open-source platforms (e.g., SourceForge [3]). The selection of benchmark queries may also affect the performance of different techniques. To mitigate the threat resulting from the benchmark queries, we randomly selected 1,000 queries from 9,139 test targets extracted from more than 700 Java projects. Although these queries were randomly selected, the distribution of the code lines of the methods under test in these queries is almost the same as that in the large-scale search corpus. This means that the benchmark queries are representative. In the future, we will integrate MUTCR into a mature IDE (e.g., Eclipse and IntelliJ IDEA) to receive any possible methods under test from developers as queries.

In this paper, we do not evaluate the usefulness of the recommended test cases by test case recommendation techniques in reducing the effort in creating new test cases, which might be another threat to external validity. The baseline Relatest [16] shows that the recommended test cases can

[3]https://sourceforge.net/

TABLE IV: The results of the user study on queries covered by different techniques

Techniques	$Score_{all}$	$Score@1$	$Score@5$	$Score@10$
Ground-truth	–	**4.59**	–	–
NiCadBased	0.29	0.29	0.07	0.03
Relatest	1.09	1.15	0.50	0.37
TestTenderer	2.38	3.01	1.72	1.13
MuTCR	**3.03**	**3.68**	**2.55**	**1.98**



(a) $k = 1$ (b) $k = 5$ (c) $k = 10$

Fig. 7: The distribution of the usefulness scores attained by MuTCR and three baselines.

significantly benefit the task of test creation. We use a similar automatic evaluation method as in the baseline Relatest [16], that is to measure the textual similarity between ground-truth test cases and recommended test cases. In the automatic evaluation, our MuTCR is significantly better than Relatest in terms of QC and $CA@k$. Therefore, we have reasons to believe that the test cases recommended by MuTCR can help developers reduce the efforts of creating new test tests for query test targets. We leave the evaluation of the usefulness of recommendations in reducing the effort in creating tests in future work.

VI. Related Work

A. Code Similarity Measurement

The test target similarity analysis is essentially a code similarity measurement that is a traditional and mature research field. A large number of code similarity measurement methods have been proposed one after another [29], [41]. These methods mainly measure code similarity from two levels of text and structure. The text level mainly includes string-based [42], [43] and token-based [44]–[46] methods. The structural level mainly includes tree-based [47]–[49] and graph-based [50]–[52] methods. All of these methods have advantages and disadvantages. Overall, their performance (i.e., accuracy) has improved over time, but their complexity (including cost) has also increased. MuTCR measures the test target similarity based on multi-level signature matching. Compared with long code text and complex structure, signature information is relatively short and simple, which can greatly improve the efficiency of similar test target searches and help realize real-time test case recommendations. We plan to explore advanced techniques, such as deep learning-based code semantic representation method [21], [53], [54] in future work.

B. Test Case Recommendation

Along with software testing development, test case search or recommendation [9], [12], [14], [17] gets more and more attention. *Test Recommender* [12] recommends test cases from the project's test cases to newcomers of the project, aiming at facilitating learning and test writing. *Test Recommender* is useful in facilitating newcomers to learn to write tests, but it requires that the project that newcomers join is rich in test cases. Thus, for a new project, it does not work. Mostafa et al. [9] recommend 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 work [12], their technique can achieve the test case recommendation across project boundaries. But they only evaluated the proposed technique on a few projects and did not compare it with others' techniques. Werner et al. [14] 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. Unfortunately, we haven't found either SENTRE or TestTenderer. TeSRS [17] is an online test recommendation system that can effectively assist test novices in learning unit testing. TeSRS gets test snippets from superior crowdsourcing test scripts by program slicing and recommends test cases by method signature matching. However, it is only used on an educational platform and its source code is not public. Its recommendation accuracy may decline because it only uses method signature as the metric. The search algorithm proposed by Relatest requires a high textual similarity between the entire code snippets of the two test targets. However, two test targets that implement the same function usually have similar method names but may have completely different code text because developers may have different coding styles and wording habits. Compared with these techniques, MuTCR introduces a multi-level signature matching strategy to measure test target similarity, which is not only efficient but also can effectively find more relevant recommendations.

VII. Conclusion

In this paper, we propose a novel multi-level signature matching for test case recommendation. We develop a prototype named MuTCR for test case recommendation. MuTCR achieves better performance by reasonably leveraging the advantages of three matching strategies with different strict levels. Our comprehensive experiments have shown that MuTCR is effective and efficient, and outperforms the state-of-the-art baselines. Furthermore, statistical results from the user study show that MuTCR can find useful recommendations for more query test targets and is better than all baselines.

ACKNOWLEDGMENT

This work is supported partially by Science, Technology, and Innovation Commission of Shenzhen Municipality (CJGJZD20200617103001003), Innovative Research Foundation of Ship General Performance(25422207), and the Program B for Outstanding PhD Candidate of Nanjing University (202201B054). Weidong Qian is the corresponding author.

REFERENCES

[1] T. A. Standish, "An essay on software reuse," *IEEE Transactions on Software Engineering*, vol. 10, no. 5, pp. 494–497, 1984.

[2] W. B. Frakes and B. A. Nejmeh, "Software reuse through information retrieval," *SIGIR Forum*, vol. 21, no. 1-2, pp. 30–36, 1987.

[3] W. Janjic, O. Hummel, and C. Atkinson, "Reuse-oriented code recommendation systems," in *Recommendation Systems in Software Engineering*. Springer, 2014, pp. 359–386.

[4] T. Xie, N. Tillmann, and P. Lakshman, "Advances in unit testing: theory and practice," in *Proceedings of the 38th International Conference on Software Engineering - Companion Volume*. Austin, TX, USA: ACM, May 14-22 2016, pp. 904–905.

[5] E. Daka and G. Fraser, "A survey on unit testing practices and problems," in *Proceedings of the 25th International Symposium on Software Reliability Engineering*. Naples, Italy: IEEE Computer Society, November 3-6 2014, pp. 201–211.

[6] S. Planning, *The economic impacts of inadequate infrastructure for software testing*. National Institute of Standards and Technology, 2002.

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

[8] J. Lee, S. Kang, and D. Lee, "Survey on software testing practices," *IET software*, vol. 6, no. 3, pp. 275–282, 2012.

[9] M. Erfani, I. Keivanloo, and J. Rilling, "Opportunities for clone detection in test case recommendation," in *Proceedings of the 37th Annual Computer Software and Applications Conference*. Kyoto, Japan: IEEE Computer Society, July 22-26 2013, pp. 65–70.

[10] P. S. Kochhar, F. Thung, D. Lo, and 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*. Jeju, South Korea: IEEE Computer Society, December 1-4 2014, pp. 215–222.

[11] M. Beller, G. Gousios, A. Panichella, and A. Zaidman, "When, how, and why developers (do not) test in their ides," in *Proceedings of the 2015 Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. Bergamo, Italy: ACM, August 30 - September 4 2015, pp. 179–190.

[12] R. Pham, Y. Stoliar, and K. Schneider, "Automatically recommending test code examples to inexperienced developers," in *Proceedings of the 2015 Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. Bergamo, Italy: ACM, August 30 - September 4 2015, pp. 890–893.

[13] R. Pham, S. Kiesling, O. Liskin, L. Singer, and K. Schneider, "Enablers, inhibitors, and perceptions of testing in novice software teams," in *Proceedings of the 22nd International Symposium on Foundations of Software Engineering*. Hong Kong, China: ACM, November 16 - 22 2014, pp. 30–40.

[14] W. Janjic and C. Atkinson, "Utilizing software reuse experience for automated test recommendation," in *Proceedings of the 8th International Workshop on Automation of Software Test*. San Francisco, CA, USA: IEEE Computer Society, 2013, pp. 100–106.

[15] C. Zhu, W. Sun, Q. Liu, Y. Yuan, C. Fang, and Y. Huang, "Homotr: Online test recommendation system based on homologous code matching," in *Proceedings of the 35th International Conference on Automated Software Engineering*. Melbourne, Australia: IEEE, September 21-25 2020, pp. 1302–1306.

[16] R. White, J. Krinke, E. T. Barr, F. Sarro, and C. Ragkhitwetsagul, "Artefact relation graphs for unit test reuse recommendation," in *Proceedings of the 14th Conference on Software Testing, Verification and Validation*. Porto de Galinhas, Brazil: IEEE, April 12-16 2021, pp. 137–147.

[17] R. Qian, Y. Zhao, D. Men, Y. Feng, Q. Shi, Y. Huang, and Z. Chen, "Test recommendation system based on slicing coverage filtering," in *Proceedings of the 29th International Symposium on Software Testing and Analysis*. Virtual Event, USA: ACM, July 18-22 2020, pp. 573–576.

[18] J. R. Cordy and C. K. Roy, "The NiCad clone detector," in *Proceedings of the 19th International Conference on Program Comprehension*. Kingston, ON, Canada: IEEE Computer Society, June 22-24 2011, pp. 219–220.

[19] Tdebatty, "Java string similarity – Jaccard index," site: https://github.com/tdebatty/java-string-similarity#jaccard-index, 2015, accessed 10 January 2023.

[20] W. Sun, Q. Zhang, C. Fang, Y. Chen, X. Wang, and Z. Wang, "Test case recommendation based on balanced distance of test targets," *Information & Software Technology*, vol. 150, no. 1, p. 106994, 2022.

[21] J. Cambronero, H. Li, S. Kim, K. Sen, and S. Chandra, "When deep learning met code search," in *Proceedings of the 2019 Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. Tallinn, Estonia: ACM, August 26-30 2019, pp. 964–974.

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

[23] P. S. Kochhar, T. F. Bissyandé, D. Lo, and L. Jiang, "An empirical study of adoption of software testing in open source projects," in *Proceedings of the 13th International Conference on Quality Software*. Najing, China: IEEE, July 29-30 2013, pp. 103–112.

[24] X. Gu, H. Zhang, and S. Kim, "Deep code search," in *Proceedings of the 40th International Conference on Software Engineering*. Gothenburg, Sweden: ACM, May 27 - June 03 2018, pp. 933–944.

[25] M. Raghothaman, Y. Wei, and Y. Hamadi, "SWIM: synthesizing what i mean: code search and idiomatic snippet synthesis," in *Proceedings of the 38th International Conference on Software Engineering*. Austin, TX, USA: ACM, May 14-22 2016, pp. 357–367.

[26] X. Li, Z. Wang, Q. Wang, S. Yan, T. Xie, and H. Mei, "Relationship-aware code search for JavaScript frameworks," in *Proceedings of the 24th International Symposium on Foundations of Software Engineering*. Seattle, WA, USA: ACM, November 13-18 2016, pp. 690–701.

[27] F. Lv, H. Zhang, J. Lou, S. Wang, D. Zhang, and 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*. Lincoln, NE, USA: IEEE Computer Society, November 9-13 2015, pp. 260–270.

[28] X. Ye, R. C. Bunescu, and C. Liu, "Learning to rank relevant files for bug reports using domain knowledge," in *Proceedings of the 22nd International Symposium on Foundations of Software Engineering*. Hong Kong, China: ACM, November 16 - 22 2014, pp. 689–699.

[29] C. Ragkhitwetsagul, J. Krinke, and D. Clark, "A comparison of code similarity analysers," *Empirical Software Engineering*, vol. 23, no. 4, pp. 2464–2519, 2018.

[30] C. Ragkhitwetsagul, J. Krinke, and D. Clark, "Similarity of source code in the presence of pervasive modifications," in *Proceedings of the 16th International Working Conference on Source Code Analysis and Manipulation*. Raleigh, NC, USA: IEEE Computer Society, October 2-3 2016, pp. 117–126.

[31] J. P, "Étude comparative de la distribution florale dans une portion des alpes et des jura," *Bull Soc Vaudoise Sci Nat*, vol. 37, no. 1, pp. 547–579, 1901.

[32] Seatgeek, "FuzzyWuzzy." site: https://github.com/seatgeek/fuzzywuzzy, 2021, accessed 10 January 2023.

[33] H. J. Motulsky, "GraphPad statistics guide." site: http://www.graphpad.com/guides/prism/8/statistics/index.htm, 2016, accessed 10 January 2023.

[34] G. A. Miller, "The magical number seven, plus or minus two: some limits on our capacity for processing information," *Psychological review*, vol. 63, no. 2, pp. 343—352, 1956.

[35] Z. P. Fry, B. Landau, and W. Weimer, "A human study of patch maintainability," in *Proceedings of the 21th International Symposium on Software Testing and Analysis*. Minneapolis, MN, USA: ACM, July 15-20 2012, pp. 177–187.

[36] Y. Tao, J. Kim, S. Kim, and C. Xu, "Automatically generated patches as debugging aids: a human study," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. Hong Kong, China: ACM, November 16 - 22 2014, pp. 64–74.

[37] X. Xie, Z. Liu, S. Song, Z. Chen, J. Xuan, and B. Xu, "Revisit of automatic debugging via human focus-tracking analysis," in *Proceedings*

of the 38th International Conference on Software Engineering. Austin, TX, USA: ACM, May 14-22 2016, pp. 808–819.

[38] Y. Feng, Z. Chen, J. A. Jones, C. Fang, and B. Xu, "Test report prioritization to assist crowdsourced testing," in *Proceedings of the 10th Joint Meeting on Foundations of Software Engineering*. Bergamo, Italy: ACM, August 30 - September 4 2015, pp. 225–236.

[39] C. Parnin and A. Orso, "Are automated debugging techniques actually helping programmers?" in *Proceedings of the 20th International Symposium on Software Testing and Analysis*. Toronto, ON, Canada: ACM, July 17-21 2011, pp. 199–209.

[40] I. Salman, A. T. Misirli, and N. J. Juzgado, "Are students representatives of professionals in software engineering experiments?" in *Proceedings of the 37th International Conference on Software Engineering*. Florence, Italy: IEEE Computer Society, May 16-24 2015, pp. 666–676.

[41] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. Merlo, "Comparison and evaluation of clone detection tools," *IEEE Transactions on Software Engineering*, vol. 33, no. 9, pp. 577–591, 2007.

[42] S. Ducasse, M. Rieger, and S. Demeyer, "A language independent approach for detecting duplicated code," in *Proceedings of the 7th International Conference on Software Maintenance*. Oxford, England, UK: IEEE Computer Society, August 30 - September 3 1999, pp. 109–118.

[43] J. H. Johnson, "Identifying redundancy in source code using fingerprints," in *Proceedings of the 3th Conference of the Centre for Advanced Studies on Collaborative Research*. Toronto, Ontario, Canada: IBM, October 24-28 1993, pp. 171–183.

[44] M. J. Wise, "YAP3: improved detection of similarities in computer program and other texts," in *Proceedings of the 27th SIGCSE Technical Symposium on Computer Science Education*. Philadelphia, Pennsylvania, USA: ACM, February 15-17 1996, pp. 130–134.

[45] Z. Li, S. Lu, S. Myagmar, and Z. Yuanyuan, "CP-Miner: finding copy-paste and related bugs in large-scale software code," *IEEE Transactions on software Engineering*, vol. 32, no. 3, pp. 176–192, 2006.

[46] H. Sajnani, V. Saini, J. Svajlenko, C. K. Roy, and C. V. Lopes, "Sourcerercc: scaling code clone detection to big-code," in *Proceedings of the 38th International Conference on Software Engineering*. Austin, TX, USA: ACM, May 14-22 2016, pp. 1157–1168.

[47] I. D. Baxter, A. Yahin, L. M. de Moura, M. Sant'Anna, and L. Bier, "Clone detection using abstract syntax trees," in *Proceedings of the 6th International Conference on Software Maintenance*. Bethesda, Maryland, USA: IEEE Computer Society, November 16-19 1998, pp. 368–377.

[48] L. Jiang, G. Misherghi, Z. Su, and S. Glondu, "DECKARD: scalable and accurate tree-based detection of code clones," in *Proceedings of the 29th International Conference on Software Engineering*. Minneapolis, MN, USA: IEEE Computer Society, May 20-26 2007, pp. 96–105.

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

[50] J. Krinke, "Identifying similar code with program dependence graphs," in *Proceedings of the 8th Working Conference on Reverse Engineering*. Stuttgart, Germany: IEEE Computer Society, October 2-5 2001, pp. 301–309.

[51] R. Komondoor and S. Horwitz, "Using slicing to identify duplication in source code," in *Proceedings of the 8th International Static Analysis Symposium*. Paris, France: Springer, July 16-18 2001, pp. 40–56.

[52] G. Zhao and J. Huang, "DeepSim: deep learning code functional similarity," in *proceedings of the 2018 Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. Lake Buena Vista, FL, USA: ACM, November 04-09 2018, pp. 141–151.

[53] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, and M. Zhou, "CodeBERT: a pre-trained model for programming and natural languages," in *Proceedings of the 25th Conference on Empirical Methods in Natural Language Processing: Findings*. Online Event: Association for Computational Linguistics, 16-20 November 2020, pp. 1536–1547.

[54] C. Zeng, Y. Yu, S. Li, X. Xia, Z. Wang, M. Geng, B. Xiao, W. Dong, and X. Liao, "deGraphCS: Embedding variable-based flow graph for neural code search," *CoRR*, vol. abs/2103.13020, pp. 1–21, 2021.