2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering Education and Training (ICSE-SEET)

MAF: Method-Anchored Test Fragmentation for Test Code Plagiarism Detection

Weisong Sun, Xingya Wang*, Haoran Wu, Ding Duan, Zesong Sun, Zhenyu Chen*
State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, China
*corresponding authors: {xingyawang, zychen}@nju.edu.cn

***Abstract*—Software engineering education becomes popular due to the rapid development of the software industry. In order to reduce learning costs and improve learning efficiency, some online practice platforms have emerged. This paper proposes a novel test code plagiarism detection technology, namely MAF, by introducing bidirectional static slicing to anchor methods under test and extract fragments of test codes. Combined with similarity measures, MAF can achieve effective plagiarism detection by avoiding massive unrelated noisy test codes. The experiment is conducted on the dataset of Mooctest, which so far has supported hundreds of test activities around the world in the past 3 years. The experimental results show that MAF can effectively improve the performance (precision, recall and F_1-measure) of similarity measures for test code plagiarism detection. We believe that MAF can further expand and promote software testing education, and it can also be extended to use in test recommendation, test reuse and other engineering applications.**

***Keywords*-Similarity measure; Plagiarism detection; Unit testing; Online training**

I. Introduction

With the advance of information technology, software engineering has become one of the hottest education directions [1], [2]. To improve learning efficiency and reduce learning costs, some software engineering practice platforms, such as LeetCode, Pex4Fun [3], Mooctest, have emerged. Mooctest is one of the most popular testing practice platforms, which supports CST 2016-2018, ISTC 2017-2018, testing contests at ICST 2019 and ISSTA 2019 [1], and hundreds of testing activities around the world [4].

The online programming platforms allow large-scale students, but it needs to maintain the quality of education [5]. For example, there were more than 7000 students attending CST 2017 in China. Plagiarism detection is essential for online programming in examinations and contests, but unfortunately, it lacks in software testing so far. It is an impossible task by manually inspecting for plagiarism detection of a large number of students. Therefore, it needs an automated tool to detect plagiarism of test codes in practice efficiently.

Some similarity measure technologies have been proposed [6] [7] to detect plagiarism of program codes. These technologies measure the similarity of program codes by analyzing the syntax, semantics, or structure of the program. However, there are some differences between program source codes and test codes. The structure of test codes is simpler than source codes, especially for those written by junior testers in examinations and contests. Aside from the inherent structures (such as class and method structures, etc.), test codes are like text in natural language. Besides, test cases in test codes are relatively independent, but most of the methods in source codes are not independent, i.e., methods calling other ones.

Based on these observations, we propose a Method-Anchored test Fragmentation (MAF) technology, combined with similarity measures, to achieve plagiarism detection of test codes effectively. MAF introduces bidirectional static slicing [8], [9] to extract valid test fragments, each of which is a minimum granularity unit test used to test a specific method under test. The critical point is minimizing test granularity, which can capture features of test codes more effectively, so that similarity measures are more accurate. Furthermore, we implemented a tool based on MAF consisting of three modules: test fragment extraction, similarity measure, and combination analysis for plagiarism. It first extracts valid test fragments from test codes and filters out some unrelated test codes. Then, two types of similarity measures, i.e., code-oriented and text-oriented methods [7], are introduced to calculate the similarities of test fragments of two test codes. Then, MAF combines with threshold analysis to solve a specific application scenario, i.e., plagiarism detection here.

In this paper, we utilize the test codes produced in software testing contest to evaluate MAF. The evaluation results show that MAF can effectively improve the accuracy of the test code similarity measure, thereby making the plagiarism judgment more accurate. So, MAF is complementary to existing similarity measures, which helps to measure test code similarity more accurately. In addition, after analyzing the experimental results, one surprising finding is that text-oriented similarity measures are more suitable for test code similarity analysis.

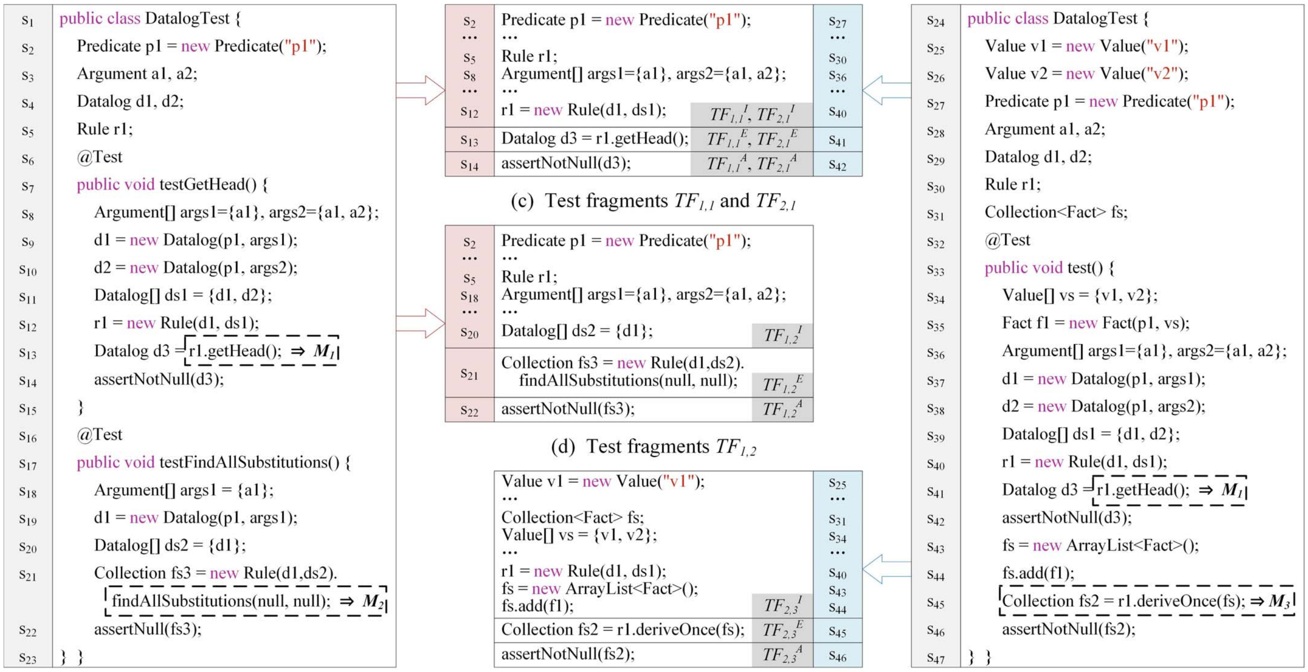
MAF can extract minimum granularity test fragments from non-standard test codes, which have many valuable application scenarios. For instance, it can be used in test recommendation [10] and test reuse [11]. Test recommendation and reuse require an outstanding corpus of test codes. MAF extracts meaningful and minimum granularity test fragments, which can be used to construct excellent tests, and build an outstanding corpus further. MAF can also be used to guide test exercises. Test fragments extracted by MAF from test codes are always well-defined to be a good example for beginners. Hints with test fragments will be studied in the future.

In summary, we make the following contributions.

- To the best of our knowledge, it is the first attempt at plagiarism detection for large-scale test codes. A novel

[1]mooctest.org, swtesting.techconf.org, icst2019.xjtu.edu.cn, conf.researchr.org/home/issta-2019

978-1-7281-1000-4/19/$31.00 ©2019 IEEE
DOI 10.1109/ICSE-SEET.2019.00020



(a) Test code TC_1 (e) Test fragments $TF_{2,3}$ (b) Test code TC_2

Fig. 1: Example of Test Fragmentation

technology, method-anchored test fragmentation (MAF), is introduced to improve similarity measures.
- An experiment based on the dataset of Mooctest is studied to validate the effectiveness of MAF. It can further promote software testing education and training around the world.

The rest of the paper is organized as follows. Section II describes preliminary. Section III presents an overview of MAF technology. We introduce the experiment in detail in Section IV. Threats to validity and related work are given in Section V and Section VI respectively. The conclusion is in Section VII.

II. Preliminary

Unit testing is a popular technique and always performed by the software developer [12], [13]. In this paper, we focus on unit testing under the JUnit[2] framework [14]. JUnit provides a standard way to encode the four fundamental parts of a test: setting initial state, invoking functionality under test, checking results of testing, and performing any necessary cleanup. In order to formalize our technique, referring to [15], we give the following definition of a test code set.

A test code TC, seen as a statement set, is a union of five subsets: TC^I,TC^E,TC^A, TC^C and TC^U, i.e., $TC = TC^I \cup TC^E \cup TC^A \cup TC^C \cup TC^U$, where

- TC^I is the set of all initialization statements.
- TC^E is the set of all execution statements.
- TC^A is the set of all assertion statements.
- TC^C is the set of all cleanup statements.
- TC^U is the set of all unrelated statements.

In order to explain our motivation and technology, we give an example shown in Fig.1. This example contains two test codes, denoted by TC_1 and TC_2, in which three methods, getHead (M_1), findAllSubstitutions (M_2) and deriveOnce (M_3), are tested. As stated, for each test code, statements that invoke any of these methods are categorized as TC^E, i.e., $TC_1^E = \{s_{13}, s_{21}\}$, $TC_2^E = \{s_{41}, s_{45}\}$. Spontaneously, statements for initializing TC^E are categorized as TC^I, i.e., $TC_1^I = \{s_2 \rightarrow s_5, s_8 \rightarrow s_{12}, s_{18} \rightarrow s_{20}\}$, $TC_2^I = \{s_{25} \rightarrow s_{31}, s_{34} \rightarrow s_{40}, s_{43}, s_{44}\}$, and statements for test verification are categorized as TC^A, i.e., $TC_1^A = \{s_{14}, s_{22}\}$, $TC_2^A = \{s_{42}, s_{46}\}$. Note that both TC_1 and TC_2 have no cleanup statements. Thus, the other statements, which have no relation to initialization, execution, assertion and cleanup, are categorized as TC^U, i.e., $TC_1^U = \{s_1, s_6, s_7, s_{15} \rightarrow s_{17}, s_{23}\}$, $TC_2^U = \{s_{24}, s_{32}, s_{33}, s_{47}\}$.

Given a software under test SUT, a test code TC of unit testing is designed to test all methods of SUT, denoted by $SUT = \{M_1, M_2, \cdots, M_n\}$, in which M_j is a method under test in SUT. A method-anchored test fragment of the test is defined as follows.

DEFINTTION 1 (Test Fragment): Given a software under test $SUT = \{M_1, M_2, \cdots, M_n\}$ and test code TC_i, a test fragment anchors method M_j, denoted by $TF_{i,j} \subseteq TC$, is a subset of all statements to test the method M_j.

Given a test code TC_i, it is clear that $\cup_j TF_{i,j} = TC \backslash TC^U$.

[2] junit.org

For each test fragment $TF_{i,j}$, we have $TF_{i,j} = TF^I_{i,j} \cup TF^E_{i,j} \cup TF^A_{i,j} \cup TF^C_{i,j}$, where $TF^I_{i,j} = TF_{i,j} \cap TC^I$, $TF^E_{i,j} = TF_{i,j} \cap TC^E$, $TF^A_i = TF_{i,j} \cap TC^A$, and $TF^C_{i,j} = TF_{i,j} \cap TC^C$. In this paper, we assume that $TF^E_{i,j} \neq \emptyset$ for a valid test fragment.

We give four examples of test fragments, namely $TF_{1,1}$, $TF_{1,2}$, $TF_{2,1}$ and $TF_{2,3}$, shown on the center side of Fig.1. $TF_{1,1}$ and $TF_{1,2}$ are extracted from TC_1 and used to test M_1 and M_2 respectively. $TF_{2,1}$ and $TF_{2,3}$ are extracted from TC_2 and used to test M_1 and M_3 respectively. Note that $TF_{1,1}$ is exactly the same as $TF_{2,1}$. Therefore, we can conclude that TC_1 and TC_2 are most likely to be plagiaristic. Each example $TF_{i,j}$ is composed of $TF^I_{i,j}$, $TF^E_{i,j}$ and $TF^A_{i,j}$, and they all exclude TF^U_i.

Directly measuring the similarity as well as detecting the plagiarism between TC_1 and TC_2 is difficult because their test targets are different and they follow different programming style rules (e.g., naming conventions and unit test granularity). Moreover, a plagiarist may add a series of unrelated statements to confuse the detector. Fragmentation, which removes the set of unrelated statements and avoids the interferences of the codes that used for testing different targets, rearranges TC_1 and TC_2 to a set of method-anchored test fragments respectively. Then, the set of the finer granularity similarities, which computed by measuring the pair-wise fragments (i.e., $\langle TF_{1,1}, TF_{2,1}\rangle$, $\langle TF_{1,2}, \emptyset\rangle$, $\langle \emptyset, TF_{2,3}\rangle$), becomes a great indicator for test code similarity measure and the subsequently plagiarism detection.

III. Technology

This section provides the framework of MAF and explains test fragment extraction, similarity measure, combination analysis for plagiarism detection in detail.

A. Framework

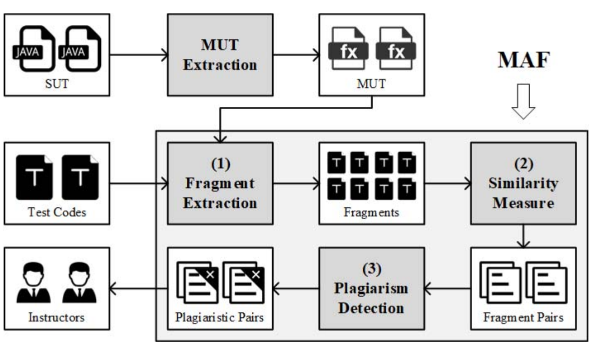


Fig. 2: Framework of MAF

Fig.2 presents the framework of MAF. As mentioned above, MAF highlights the most similar method-anchored test fragments between two submissions. MAF introduces a static slicer to extract the test fragments, and then measures the similarity/distance of each pair of test fragments, finally combines threshold analysis to detect plagiarism pairs. Specifically, MAF works with three parts. (1) Test fragment extraction: test codes are refracted and extracted into a set of test fragments, in which each test fragment corresponds to one unique method under test. (2) Similarity measure: each pair of test fragments that corresponds to the identical method under test will be evaluated by similarity measures. (3) Combination analysis for plagiarism detection: the similarities of test codes obtained by similarity measure are combined with thresholds to conduct plagiarism detection. Finally, some pairs of most suspicious test codes are selected as plagiarism candidates.

Algorithm 1 Framework of MAF

Input: SUT, TC, t and $funSim(TF_{i,k}, TF_{j,k})$;
Output: PP.

```
// Stage I: Test Fragment Extraction
recognize all methods under test M_1, M_2, ..., M_n from SUT, and add them into a list of Method Under Test MUT;
initialize an empty set TF, which is utilized to record the Test Fragments of all students;
for all TC_i in TC do
    initialize an empty set TF_i, which is utilized to record the Test Fragments of ith student;
    for all M_j in MUT do
        analyze TC_i to extract the test fragment TF_{i,j} that corresponds to M_j, and add TF_{i,j} into the ith student's test fragment list TF_i;
    end for
end for
// Stage II: Similarity measure
initialize a Three-Dimensional Similarity Array TDSA, where each element is set to 0;
for i = 1 to num_student-1 do
    for j = i + 1 to num_student do
        for all M_k in MUT do
            get the test fragments, TF_{i,k} and TF_{j,k}, that correspond to M_k from TF_i and TF_j respectively;
            if both TF_{i,k} and TF_{j,k} are not NULL then
                TDSA[i][j][k] = funSim(TF_{i,k}, TF_{j,k});
            end if
        end for
    end for
end for
// Stage III: Plagiarism Pair Detection
initialize an empty set of Plagiarism Pair PP;
for i = 1 to num_student-1 do
    for j = i + 1 to num_student do
        if funMax(TDSA[i][j][]) ≥ t then
            add the pair ⟨i, j⟩ into PP;
        end if
    end for
end for
output PP;
```

Algorithm 1 outlines the details of MAF. It treats Software Under Test (SUT), Test Codes (TC) of each student, a threshold of similarity t, and a similarity function

$funSim(TF_{i,k}, TF_{j,k})$, as the inputs and finally outputs the candidate Plagiarism Pairs (PP).

Stage 1: lines 1-9. MAF recognizes the methods under test $\{M_1, M_2, ..., M_n\}$ from SUT and stores them in a list of methods MUT. For each test code TC_i, MAF refracts it into a set of test fragments TF_i. In each iteration (lines 4-9), we expect to find all the relevant test statements for every M_j from TC_i. These extracted test statements with respect to M_j constitute the so-called test fragment $TF_{i,j}$.

Stage 2: lines 10-21. MAF resorts to a Three-Dimensional Similarity Array $TDSA$ to record the test fragment similarity values of the pair-wise students on each M_k. Due to the limited time and test skills, a student may only test some parts of MUT. That is, some test fragments may be NULL. Given two non-null test fragments $TF_{i,k}$ and $TF_{j,k}$, MAF calculates the similarity based on $funSim(TF_{i,k}, TF_{j,k})$, and puts the value into $TDSA\,[i]\,[j]\,[k]$. Otherwise, MAF assigns the default similarity value, 0, to $TF_{i,k}$ and $TF_{j,k}$.

Stage 3: lines 22-31. MAF employs the threshold analysis to detect the plagiarism pairs. Intuitively, the higher the similarity is, the more likely the pair is considered as a plagiarism pair. The identification of plagiarism has no cause-and-effect relationship with the size of plagiarism contents as well as the number of the plagiarism positions in general. Thus, the pair with the maximum similarity value can be utilized for plagiarism judgment. $TDSA\,[i]\,[j]\,[]$ records the similarity values between i and j. If the maximum similarity, i.e. the return value of $funMax(TDSA\,[i]\,[j]\,[])$ between i and j is greater than t, then the pair $\langle i, j\rangle$ is supposed to be plagiarism. The plagiarism detection process is finished after all of the pairs of test codes are analyzed.

B. Test Fragment Extraction

The submitted test codes in Mooctest are required to follow a series of programming style rules, such as naming conventions, unit test granularity, and so on. For xUnit, e.g. JUnit [16] framework, a well-designed xUnit test should satisfy but not limit to the two rules: (1) Naming convention: for a class 'C', the name of its corresponding test class should be either "CTest" or "TestC" in UpperCamelCase, and for a method 'm', the name of its corresponding test method should be either "testM" or "mTest" in lowerCamelCase [17]. (2) Unit test granularity: each test case should only test one method under test and should not combine multiple unrelated tests into a single test case [18]. Moreover, a unit test consists of four fundamental parts, i.e., TC^I, TC^E, TC^A, and TC^C [15].

It is not difficult to extract test statements anchoring methods under test in well-designed test codes. However, test codes written by junior testers (such as students) are always far from well-designed, especially in a high-stress examination or contest. Moreover, many test codes may be incomplete. Some methods in MUT are not tested intentionally or unintentionally; some tests miss assertions, and so on. It also remains some challenges of test fragment extraction for failed or crashed tests.

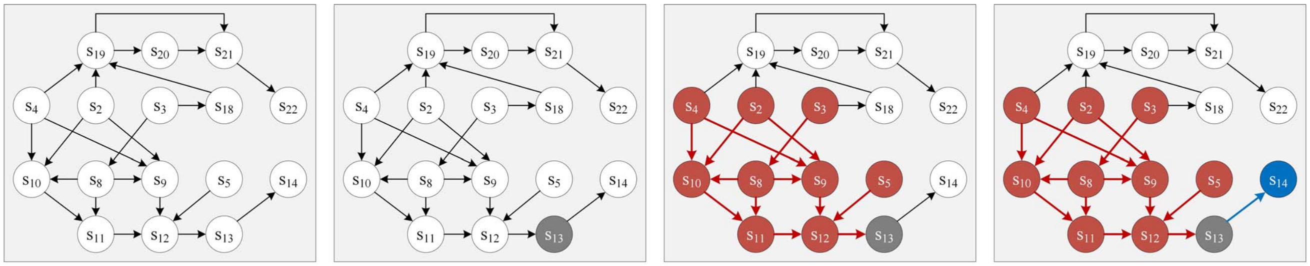
Static slicing, firstly proposed by Weiser [8], is used to select all the statements that can affect the value of a variable in a statement directly or indirectly, so-called backward static slicing. "Static" means that the slicing result does not rely on the program execution as well as the input [19]. Subsequently, Horwitz et al. proposed the forward static slicing to recognize the statements that are directly or indirectly affected by the value of a variable in a statement [20]. Both BSS and FSS rely on program dependence (control dependence and data dependence) analysis to extract some code statements from the original program [19].

Algorithm 2 Test Fragment Extraction Based on Bidirectional Static Slicing

Input: TC_i, M_j;
Output: $TF_{i,j}$.

initialize the test fragment $TF_{i,j}$ as an empty set;
for each execution statement es_k in TC_i **do**
  if the callee method in es_k is not M_j **then**
    continue;
  end if
  initialize $TF^E_{i,j}$ as $\{es_k\}$ and $TF^I_{i,j}, TF^A_{i,j}, TF^C_{i,j}$ as empty sets;
  get the used variables USE and the defined variables DEF in es_k;
  slice TC_i with backward static slicing criteria $\langle es_k, USE\rangle$ and forward static slicing criteria $\langle es_k, DEF\rangle$, where BSSR and FSSR are respectively referred as the slicing results.
  for each statement s in $BSSR$ **do**
    if s is an initialization statement **then**
      $TF^I_{i,j} = TF^I_{i,j} \cup \{s\}$;
    end if
  end for
  for each statement s in $FSSR$ **do**
    if s is an assertion statement **then**
      $TF^A_{i,j} = TF^A_j \cup \{s\}$;
    else if s is a cleanup statement **then**
      $TF^C_{i,j} = TF^C_j \cup \{s\}$;
    end if
  end for
  $TF_{i,j} = TF_{i,j} \cup TF^I_{i,j} \cup TF^E_{i,j} \cup TF^A_{i,j} \cup TF^C_{i,j}$;
end for
output $TF_{i,j}$;

Inspired by the success stories of slicing, we introduce Bidirectional (backward and forward) Static Slicing for Test Fragment Extraction, namely BSS-TFE in brief. Algorithm 2 outlines the details of BSS-TFE. It treats a test code TC_i and the method under test M_j as the inputs and finally outputs the test fragment $TF_{i,j}$ in TC_i, which was coded for testing M_j. In BSS-TFE, each execution statement es_k that invokes M_j will be selected as the key point for slicing. In a test, before the M_j is invoked, it needs to set the initial state (e.g., Object Instantiated) and prepare the essential arguments. Thus, $TF_{i,j}$



(a) *PDG* of TS_1 (b) es_k=s_{13} when M_j=M_1 (c) $BSSR$={$s_2 \rightarrow s_5$, $s_8 \rightarrow s_{12}$} (d) $FSSR$={s_{14}}

Fig. 3: Example of Bidirectional Static Slicing

should contain statements that are utilized for the initial state setting and arguments preparation. A backward static slicing with the criteria $\langle es_k, USE\rangle$ can satisfy the demands (line 8). Moreover, after the M_j is invoked, a test needs to check the results of the test and perform any necessary cleanup. Thus, $TF_{i,j}$ should contain statements that are utilized for result checking and resource cleanup. A forward static slicing with the criteria $\langle es_k, DEF\rangle$ can satisfy the demands (line 9). Thus, the result (i.e., $BSSR$ and $FSSR$) of bidirectional static slicing on es_k contains the statements that are used for testing M_j. Once all execution statements have been analyzed, BSS-TFE outputs $TF_{i,j}$ and the algorithm finishes.

Furthermore, we resort to Fig.3 to illustrate the process of bidirectional static slicing, in which we want to extract the fragment w.r.t. M_1 from TC_1 in Fig.1. Static slicing works on Program Dependence Graph ($PDG = \langle N, E\rangle$). Generally, the nodes N correspond to the executable statements and the edges E correspond to the dependencies (i.e., data dependence and control dependence) among the nodes. An edge $s_i \rightarrow s_j$ implies that s_j is dependent on s_i. Fig.3 (a) presents the PDG of TC_1. It contains 16 nodes and 22 edges. Obviously, only node s_{13} invokes M_1 in TC_1. Thus, as Fig.3 (b) shows, s_{13} (the grey node) is selected as the key point. Then, its used variables $USE = \{r1\}$ and defined variables $DEF = \{d3\}$ are provided to the subsequently backward static slicing (BSS) and forward static slicing (FSS) respectively. BSS extracts the statements that s_{13} is directly or indirectly depend on, and FSS extracts the statements that are directly or indirectly depend on s_{13}. Once either BSS or FSS finishes, Bidirectional static slicing stops and outputs BSS Result $BSSR = \{s_2 \rightarrow s_5, s_8 \rightarrow s_{12}\}$ (i.e., the red nodes in Fig.3 (c)) and FSS Result $FSSR = \{s_{14}\}$ (i.e., the blue node in Fig.3 (d)).

C. Similarity Measure

We evaluate MAF in association with three typic similarity measure tools (also seen as plagiarism detectors): Difflib [21], FuzzyWuzzy [22] and Plaggie [23]).

$$sim_D(file_1, file_2) = 1 - \frac{min(|file_2|^{line}, |D(file_1, file_2)|^{line})}{|file_2|^{line}} \quad (1)$$

$$sim(file_1, file_2) = max\{sim_D(file_1, file_2), sim_D(file_2, file_1)\} \quad (2)$$

Difflib is a text-oriented similarity measure tool. It relies on the class “difflib.Differ” to compare sequences of lines of text and produce human-readable differences or deltas [21]. Difflib has been used in code plagiarism detection [7]. In that paper, given two files $file_1$ and $file_2$, their similarity is calculated by equation (1), where $|file_1|^{line}$ and $|file_2|^{line}$ correspond to the number of lines in $file_1$ and $file_2$ respectively. $D(file_1, file_2)$ represents the output of Difflib. Note that $sim_D(file_1, file_2)$ is sensitive to parameter order, and thus we have $\text{sim}_D(file_1, file_2) \neq \text{sim}_D(file_2, file_1)$ in most cases. As equation (2) shows, in this paper we use the maximum value as the similarity of two files.

$$sim(file_1, file_2) = 1 - \frac{2.0 * Match^{char}}{|file_1|^{char} + |file_2|^{char}} \quad (3)$$

FuzzyWuzzy is also a text-oriented similarity measure tool. It can seem as a wrapper for Difflib since it relies on Difflib for edit similarity calculation. Differently, it adopts fuzzy string matching to evaluate the similarity between two strings. Document [24] presents a detailed comparison between Difflib and FuzzyWuzzy. In FuzzyWuzzy, the similarity between files $file_1$ and $file_2$ is calculated by equation (3), where $|file_1|^{char}$ and $|file_2|^{char}$ correspond to the number of characters in $file_1$ and $file_2$ respectively, and $Match^{char}$ corresponds to the size of all character matches.

$$sim(file_1, file_2) = 1 - \frac{2.0 * Match^{token}}{|file_1|^{token} + |file_2|^{token}} \quad (4)$$

Different from the two tools mentioned above, Plaggie is a code-oriented tool, which aims to detect plagiarism in Java programming exercises. It is similar to another code-oriented tool JPlag [25] in functionally, where both of them tokenize the code and use greedy string tiling [23] to measure the similarity between two strings. Differently, JPlag does not support local service, which reduces its scalability. In Plaggie, the similarity between files $file_1$ and $file_2$ is calculated by equation (4), where $|file_1|^{token}$ and $|file_2|^{token}$ correspond to the number of tokens in $file_1$ and $file_2$ respectively, and $Match^{token}$ corresponds to the size of all token matches. Plaggie also supports configuring the minimum length of matched token sequences for improving adaptability. For test codes, we use the default value (i.e., 11) that is recommended in [23]. Compared to the test code, the test fragment is quite small; thus using the default value may not appropriate.

D. Combination Analysis

MAF extracts meaningful test fragments from non-standard test codes; then uses similarity measures to measure the similarity of test fragments instead of test codes own's. Based on the similarity got above, we use threshold analysis to detect plagiarism pairs. Intuitively, the higher the similarity is, the more likely the pair is considered a plagiarism pair. The identification of plagiarism has no cause-and-effect relationship with the size of plagiarism contents as well as the number of the plagiarism positions in general. Thus, the pair with the maximum similarity value can be utilized for plagiarism judgment. All pairs with maximum similarity above the threshold will be considered as plagiarism pairs.

IV. Experiment

To evaluate the effectiveness of MAF, we implemented the tool and applied it to a dataset from Mooctest. We investigate the following two research questions.

- RQ1: Is MAF effective for test code plagiarism detection?
- RQ2: Which similarity measure works better in MAF?

A. Experiment Subjects

In this experiment, we utilize the test code dataset produced in the National Student Contest of Software Testing 2017 (CST 2017) in China [26] from Mooctest. Specifically, the software under test Datalog and its corresponding test codes are used. The characteristics of Datalog are: lines of codes 288, number of branches 56, Average Method Complexity [27] 2.00, and Average Block Depth [27] 1.92. In CST 2017, 635 students submitted their test codes against the Datalog, and 619 students' test codes adopted by us since a manual review found that 16 of them submitted useless test codes.

To build a dataset for validation, we need to inspect test codes and label plagiarism manually. For reducing the numbers of both false positives and false negatives, we employed postgraduates to conduct a two-phase checking. Firstly, TC was averagely divided into two sets TC_1 and TC_2. Each set was independently checked by two postgraduates. After TC_1 and TC_2 had been checked, pairs that had been labelled with different results were provided to the other two postgraduates for final determination. Finally, we found 4312 plagiarism pairs and 186959 non-plagiarism pairs in TC, where each pair had been checked at least twice.

B. Variables and Metrics

The primary goal of this study is to evaluate the effectiveness of MAF we proposed. To accomplish this, we utilize two independent and three dependent variables. The first independent variable is using MAF or not, and the second is which similarity measure would be chosen to use. The dependent variables are three measure metrics of performance: precision (P), recall (R), and F$_1$-measure (F_1).

$$P = \frac{num_{tp}}{num_{tp} + num_{fp}} \tag{5}$$

$$R = \frac{num_{tp}}{num_{tp} + num_{fn}} \tag{6}$$

$$F_1 = \frac{2 * P * R}{P + R} \tag{7}$$

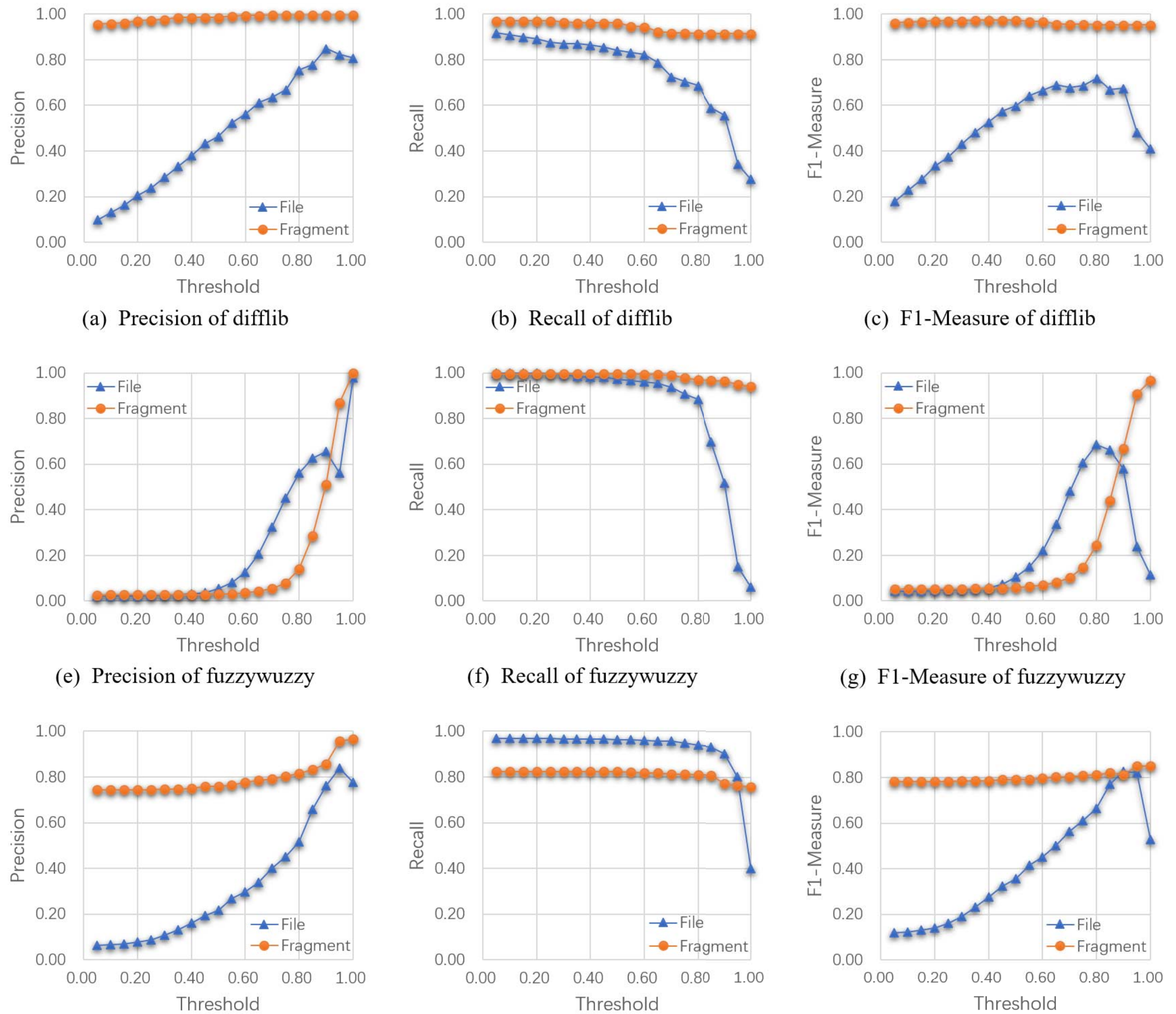
We use precision and recall to evaluate the effectiveness of MAF for test plagiarism detection. Precision corresponds to the plagiarism pairs among the pairs detected by the threshold, which indicates how useful the detected results are. Recall corresponds to the plagiarism pairs among all plagiarism pairs labeled manually, which indicates how complete the detected results are. Equations 5 and 6 present the approaches to calculate precision and recall respectively, in which num_{tp} and num_{fp} correspond to the number of plagiarism pairs and non-plagiarism pairs among the detected results respectively, num_{tn} and num_{fn} correspond to the number of non-plagiarism pairs and plagiarism pairs among the rest results respectively. Since both precision and recall are important in test plagiarism detection, we use the F$_1$-measure to evaluate MAF in plagiarism detection. Equation 7 presents the way for F$_1$-measure calculation.

C. Experiment Setup

To answer our two research questions, we use the experiment to examine the effectiveness of the fragmentation module of MAF and make a comparison between the text-oriented and the code-oriented similarity measure tools.

The central part of MAF includes extracting the test fragments, computing the similarity for pairs of test codes, and detecting the plagiarism pairs based on a given threshold. In practice, we first establish the $TPDS$ (Test Code DataSet) with 619 students' test codes as described in IV-A. Secondly, for each student's test codes in $TPDS$, MAF extracts the test fragments that anchor some methods under test. Thirdly, we measure the similarity of test codes in the granularity of both test file (sim_{file}) and test fragment (sim_{frag}), which we called non-fragmentation and fragmentation respectively. MAF provides the framework for measuring sim_{frag}. Since the test code of one student may contain multiple files, for sim_{file} of two students, we use the maximum similarity among the pairs of files to represent. The files in a pair come from these two students respectively. Finally, we compose all sim_{file} and sim_{frag} and generate a test plagiarism detection report based on the threshold analysis. For each of the detected results, we compute the precision, recall and F$_1$-measure based on the determined results in $TPDS$. Then, we compare fragmentation with non-fragmentation, as well as the outperforming among a set of similarity measure tools. The designed experiments are described as follows:

The first experiment is to evaluate the fragmentation by comparing our approach with non-fragmentation under the same similarity measure tool. In this comparison, we use three



(a) Precision of difflib (b) Recall of difflib (c) F1-Measure of difflib

(e) Precision of fuzzywuzzy (f) Recall of fuzzywuzzy (g) F1-Measure of fuzzywuzzy

(h) Precision of plaggie (i) Recall of plaggie (j) F1-Measure of plaggie

Fig. 4: Results of SimTFile and SimTFrag

typic tools (i.e., Difflib FuzzyWuzzy and Plaggie) to compute the similarities of both test file and test fragment respectively.

The second experiment is to evaluate between code-oriented similarity measure tool (i.e., Plaggie) and the text-oriented similarity measure tools (i.e., Difflib and FuzzyWuzzy) in test code plagiarism detection.

1) Similarity between Test Files, SimTFile: Most of the tools presented in this paper are based on comparisons between source files and do not support comparing submissions as a whole. Besides, the number of test files submitted by each student may be more than one, and the name of the test files may also be non-standard. Therefore, we compare the performance of plagiarism detection tools on all file pairs written by two students. We compare two students' test codes similarity with Difflib, FuzzyWuzzy, and Plaggie on source test files. To be exact, we compare all the test files in the submission directory of the two students. For example, if student A writes m test files and student B writes n test files, we will get $n * m$ similarity comparison results. Difflib and Plaggie can receive the file address as input and generate a test report. However, FuzzyWuzzy receives two sequences as input, and the output is the similarity value of the two sequences. Therefore, we extract the contents of the test files into string sequences as FuzzyWuzzy's inputs.

2) Similarity between Test Fragments, SimTFrag: We extract test fragments for each student. In order to enable Plaggie to compare the similarity of test fragments, we have specially processed the test fragment (e.g., wrapping the test fragment

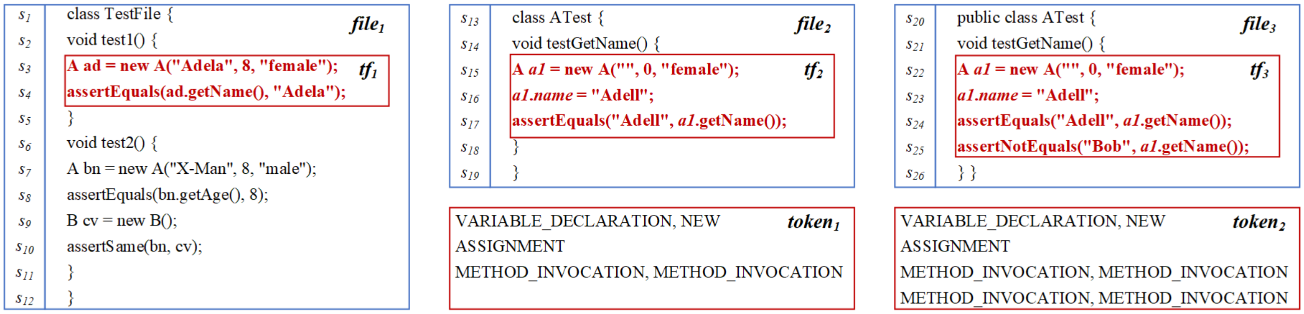


Fig. 5: Example of Token and Test Codes

with "class{{ tf }}", 'tf' is a test fragment.). This is because Plaggie has a requirement for the format of the file content (in accordance with the basic specification of Java code). The difference between SimTFrag and SimTFile is that SimTFrag compares two test fragments that test the same method, while SimTFile compares all test files in the two students' directory.

D. Result Analysis

We conduct the first experiment described in Section IV-C and present the results in Fig.4. In this experiment, we verify the effectiveness of fragmentation by using three similarity measure tools (i.e., Difflib, FuzzyWuzzy and Plaggie) and compare them under precision, recall and F_1-measure. As shown in Fig.4, the sub-figures (a)-(c) respectively illustrate the precision, recall and F_1-measure of conducting non-fragmentation (File) and fragmentation (Fragment) when Difflib is used. Similar to Difflib, the latter six subfigures, (d)-(i), represent the precision, recall and F_1-measure of conducting non-fragmentation (File) and fragmentation (Fragment) when FuzzyWuzzy and Plaggie are used respectively. In each of the sub-figure, the horizontal axis represents the threshold we configured (i.e., 0.05, 0.10, ..., 1.00). Along the vertical axis, we present the value of precision (or recall, F_1-measure). We resort to the blue curve with triangle points to represent the results of non-fragmentation and resort to the red curve with circular points to represent the results of fragmentation. Detailed results are given as follows.

1) Effectiveness of Fragmentation: For ease of understanding, we use the test code example in Fig.5 to explain the experimental results. Now, assume that $file_1$, $file_2$, and $file_3$ in Fig.5 are the test files written by students A, B, and C respectively, where B and C is a plagiarism pair.

In precision, as shown in Fig.4-(a), (d) and (g), the precision of Difflib and Plaggie in the SimTFrag scenario is always higher than that in the SimTFile scenario. For the tool FuzzyWuzzy, the precision in SimTFile is higher than that in SimTFrag when the similarity threshold $t \in (0.4, 0.9)$, but after t is greater than 0.95, it is reversed. The reason is that the similarity value calculated in SimTFrag is higher than that in SimTFile on the whole. For example, we use FuzzyWuzzy to calculate the similarity between A and B. In SimTFile scenario, $sim(A, B) = sim(file_1, file_2) = 56$, while the $sim(A, B) = sim(tf_1, tf_2) = 67$ in SimTFrag. Similarly, A and C, B and C are also the same. So, the number of the plagiarism pairs chosen by the same threshold is accordingly larger, especially the false positive pairs. Therefore, we can conclude that fragmentation can make sense for improving the performance of all experimental tools in precision.

In recall, as shown in Fig.4-(b), (e) and (h), the recalls of Difflib and FuzzyWuzzy in the SimTFrag scenario is always higher than that in the SimTFile scenario. For the tool Plaggie, the recall in SimTFile is higher than that in SimTFrag when $t \in (0, 0.95)$, but after t is greater than 0.95, The recall of Plaggie in the SimTFile scenario drops rapidly while declining smoothly in the SimTFrag scenario. In addition, Fig.4-(e) shows, in general, when t increases from 0 to 1, the recall of FuzzyWuzzy changes little, almost always at 0.8 in the SimTFrag scenario. To our surprise, even if t is small (e.g. $t = 0.05$), the recall of Plaggie in the SimTFrag scenario is still very low ($recall = 0.8$ while which is close to 1.0 in FuzzyWuzzy and Difflib), the main reason is that , for Plaggie, it first converts test fragments tf_2 and tf_3 to tokens $token_1$ (three tokens) and $token_2$ (four tokens) respectively before calculating the similarity of tf_2 and tf_3 . In this case, if $minMatchLength = 4$, $sim(B, C) = 0$. We can find once $minMatchLength > min\{|tf_1.tokens|, |tf_2.tokens|\}$, where $|tf_1.tokens|$ and $|tf_2.tokens|$ are the numbers of token of tf_1 and tf_2, the similarity is 0. Because the number of tokens is not fixed, no matter what the t value is, there are always true plagiarism pairs would be omitted, so the recall rate will not reach particularly high. In total, we can find fragmentation can make sense for improving the performance of all experimental tools in recall.

In F_1-measure, as we all know, F_1 is used to balance precision and recall, and it is calculated through them. Therefore, the changing curve of F_1 value is related to the greater influencing factor. From Fig.4-(c), (f) and (i), we can see fragmentation can make sense for improving the performance of all experimental tools in F_1-measure.

Based on the above analysis of precision, recall, and F_1-measure, we can conclude that fragmentation is a significant stage to improve test plagiarism detection.

2) Code or Text-Oriented Similarity measures: Next, we conduct the second experiment to evaluate the code-oriented

TABLE I: Results of Threshold Analysis

Threshold	Tool	P	R	F_1	FN
0.1	T_D	0.961	0.972	0.966	
	T_F	0.027	0.997	0.052	
	T_P	0.744	0.825	0.782	
0.2	T_D	0.973	0.972	0.972	
	T_F	0.027	0.997	0.052	
	T_P	0.744	0.825	0.782	
0.3	T_D	0.978	0.965	0.971	
	T_F	0.027	0.997	0.052	
	T_P	0.746	0.825	0.784	
0.4	T_D	**0.986**	**0.964**	**0.975**	**156**
	T_F	0.028	0.997	0.054	
	T_P	0.751	0.824	0.786	
0.5	T_D	0.987	0.962	0.974	
	T_F	0.03	0.995	0.058	
	T_P	0.76	0.824	0.791	
0.6	T_D	0.994	0.944	0.969	
	T_F	0.036	0.994	0.069	
	T_P	0.775	0.819	0.797	
0.7	T_D	0.998	0.919	0.957	
	T_F	0.054	0.991	0.102	
	T_P	0.792	0.813	0.803	
0.8	T_D	0.998	0.915	0.955	
	T_F	0.14	0.971	0.245	
	T_P	0.815	0.81	0.812	
0.9	T_D	0.998	0.915	0.955	
	T_F	0.511	0.965	0.668	
	T_P	0.855	0.772	0.812	
1.0	T_D	0.998	0.915	0.955	
	T_F	**0.998**	**0.939**	**0.968**	**262**
	T_P	**0.967**	**0.758**	**0.85**	**1044**

similarity measure (i.e., Plaggie) and the text-oriented similarity measure (i.e., Difflib and FuzzyWuzzy) in the test code plagiarism detection. We present the results of precision, recall and F_1-measure when threshold value ranging from 0.00 to 1.00 in Table I, when employing Difflib (T_D), FuzzyWuzzy (T_F) and Plaggie (T_P).

Since Plaggie's performance in different scenarios is affected by the parameter 'minMatchLength', we have done a supplementary experiment for Plaggie to explore a relatively suitable 'minMatchLength' value so that Plaggie can reach a relatively good performance state in the SimTFrag scenario. The experimental results show that when $minMatchLength = 17$, Plaggie can achieve relatively optimal performance in the SimTFrag scenario, and the detail results are shown in Fig.6.

For Difflib, it can achieve relatively the best performance when $t = 0.4$, and $t = 1.0$ for FuzzyWuzzy and Plaggie. In the best-performing state, according to precision, recall and F_1, we found that Difflib and FuzzyWuzzy are better than Plaggie, in other words, they are better suited to the new scenario of test code plagiarism detection than Plaggie. On the one hand, this is caused by the method of computing similarity using Difflib. If there are differences between the two lines in two files, even if the difference is small, Difflib will still mark them as different. For example, assuming that the student A wrote a test file named F_A, and the student B wrote a test file named F_B. The test codes in file F_A were copied from file F_B and made certain modifications to each statement (such as the modification of the identifier). In this case, the output $|D(F_A, F_B)|$ generated by Difflib will be vast, which also leads to the value of $sim_D(F_A, F_B)$ to be small. However, it is a common practice to make certain modifications after plagiarism. Thus, the similarity calculated by Difflib is generally small than FuzzyWuzzy and Plaggie. On the other hand, we find a large number of students who plagiarized others' test code without any modification, so that the similarity calculated by Difflib is 1.0. All these can explain why the performance of Difflib can achieve the best easily with a small threshold, such as $t = 0.4$. Besides, we can also find that the number of false positives ($FN = 1044$) pairs is the highest when Plaggie achieves relatively the best performance.

Based on the above analysis, we can get an exciting conclusion that the effectiveness and performance of the text-oriented plagiarism detection tools are better than those of the code-oriented tools in test plagiarism detection scenario.

V. Threats to Validation

A. Construct validity

Since we are the first to study the test code plagiarism detection, there is no ready-made test code dataset available for our experiment. Relying on Mooctest, we got many test codes submitted by students. We have carefully sorted out these test codes and produced a good test code dataset. We will make the test code dataset publicly available so that it can be used in future studies of tool evaluation and comparison.

B. Internal validity

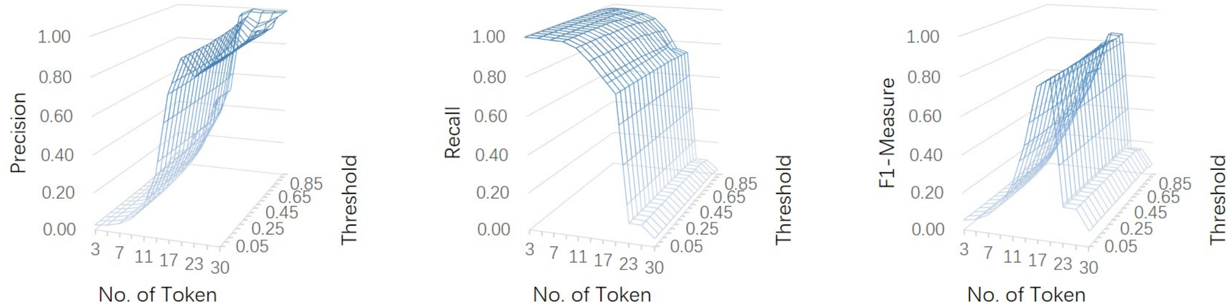
The test fragments extracted by MAF may be incomplete (some statements may be lost). Although some test fragments may be incomplete, they are still sufficient to indicate the testing process of the students, so it is enough to prove whether there exist plagiarism among the students. We will further improve and optimize the extraction of test fragments to make the extracted test fragments more complete in the future.

We use the threshold analysis to find all pairs that are plagiarism. There is a certain degree of difference in performance with different thresholds. In order to reduce the bias of threshold t, we used different t (t increased from 0 to 1 by 0.05 each time) for many experiments.

We used three typic detection tools to evaluate the effectiveness of fragmentation of MAF. The performance of Plaggie is affected by a parameter of 'minMatchLength', which may affect the evaluation of MAF. For this threat, we did an auxiliary experiment to find a suitable 'minMatchLength' value that could make Plaggie achieve a relatively good performance.

C. External validity

We only used one dataset to conduct the experiment, which may affect the generalization of MAF. Although we used only one dataset for evaluation, the dataset contains test codes submitted by 635 students from all over the country, which can produce more than 150 thousand comparison pairs, so it has certain representativeness. In the future, we will do further experiments on more datasets.



(a) Precision of Plaggie (b) Recall of Plaggie (c) F1-Measure of Plaggie

Fig. 6: Results of Supplementary Experiment for Plaggie

VI. Related Work

Plagiarism Detection is essential for education. A large number of similarity measures have been proposed, providing a good infrastructure for software plagiarism detection. Such as, token-based [23], [25], structure-based [28]–[30], syntax-based [31] and semantics-based [32], and so on measures. Meanwhile, many plagiarism detection tools have been developed. Plague [28] was introduced by G.Whale in 1988, and it is a structure-based plagiarism detector, which can be used to detect plagiarism code written in C programming language. YAP3 [33] uses Running Karp-Rabin Greedy String Tiling (RKS-GST) as a comparison algorithm that proposed by Michael Wise [34]. RKS-GST is suitable for plagiarism detection since it prioritizes longer substrings and it is not greatly affected by the order of substrings [35]. MOSS [36], [37] and JPlag [25] provide a web service for detecting plagiarism. However, if source codes are confidential information, it will be not suitable to send codes to MOSS or JPlag. Plaggie [23] is similar in functionality to JPlag, but its source code is open so that someone can do secondary development based on it. However, the similarity measures and tools listed above are mostly used to detect software or source code plagiarism. There is no specific test code plagiarism detection tool. MAF is the first plagiarism detection framework for test code. It takes the difference between the test code and source code into account, so it can be combined with existing similarity measures to detect test code plagiarism well.

Test Similarity is most relevant to test code plagiarism detection so far and focus on test report or test case similarity measure. Usually, the test report consists of natural language text and some screenshots. Existing research on test report similarity measure is mainly used to solve a lot of redundancy problem in crowdsourced testing [38], [39]. Such as Feng et al. [38] measure the similarity of test reports by combining natural language processing technique and image analysis technique. TERFUR [40] is a fuzzy clustering framework to cluster crowdsourced test reports for reducing the costs of manual inspection. These measures may also be suitable for test code plagiarism detection, but so far we have not seen any practice. Besides, all of them do not take test behavior (e.g., the test case is used to test a specific method under test) into account. For test case similarity measure, lots of research focus on test case prioritization [41]–[43]. Fang et al. [42] employ ordered sequences of program entities to measure the similarity of test cases. Noor et al. [43] use the sequence of method calls of test case to measure similarity. The test cases they focused on mostly are standard and minimum granularity. However, when used for non-standard test code, their performance is not satisfactory. By introducing slicing technology to anchor methods under test and extracting fragments from non-standard test codes, MAF can effectively improve the performance of similarity measures for test code plagiarism detection.

VII. Conclusion

In this paper, we propose MAF technology to extract meaningful test fragments, then measuring the similarity between the test fragments instead of test codes own's. We evaluated MAF with three typic tools on a test code dataset from the Mooctest platform. The evaluation results show that MAF can extract a large number of meaningful test fragments from the non-standard test codes submitted by students. And, measuring test code similarity based on test fragments instead of test code directly can effectively improve the performance of similarity measures for test code plagiarism detection. Besides, to our surprise, we found that, in the test code similarity measure, the effectiveness and performance of text-oriented similarity measure tools are better than that are code-oriented. MAF is a very flexible framework, in which the similarity measure module can be easily replaced by most of the existing similarity measures. In the future, we will use more promising similarity measures to verify the effectiveness and performance of our MAF on more datasets.

Acknowledgment

This work was supported by the National Natural Science Foundation of China (Grant No. 61690201, 61772014, 61802171) and Jiangsu Planned Projects for Postdoctoral Research Funds (Grant No. 2018K028C).

REFERENCES

[1] N. R. Mead, "Software engineering education: How far we've come and how far we have to go," *Journal of Systems and Software*, vol. 82, no. 4, pp. 571–575, 2009.

[2] N. R. Mead, D. Garlan, and M. Shaw, "Half a century of software engineering education: The CMU exemplar," *IEEE Software*, vol. 35, no. 5, pp. 25–31, 2018.

[3] N. Tillmann, J. De Halleux, T. Xie, S. Gulwani, and J. Bishop, "Teaching and learning programming and software engineering via interactive gaming," in *Proceedings of the 35th ICSE*, pp. 1117–1126, 2013.

[4] W. E. Wong, L. Hu, , H. Wang, and Z. Chen, "Improving software testing education via industry sponsored contest," in *Proceedings of the 8th FIE*, 2018. in press.

[5] S. Li, X. Xiao, B. Bassett, T. Xie, and N. Tillmann, "Measuring code behavioral similarity for programming and software engineering education," in *Proceedings of the 38th ICSE Companion*, pp. 501–510, 2016.

[6] I. Bosnic, B. Mihaljevic, M. Orlic, and M. Zagar, "Source code validation and plagiarism detection - Technology-rich course experiences," in *Proceedings of the 4th CSEDU*, pp. 149–154, 2012.

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

[8] M. Weiser, "Programmers use slices when debugging," *Communications of the ACM*, vol. 25, no. 7, pp. 446–452, 1982.

[9] X. Zhang, N. Gupta, and R. Gupta, "Locating faulty code by multiple points slicing," *Journal of Software: Practice and Experience*, vol. 37, no. 9, pp. 935–961, 2007.

[10] H. Aman, T. Nakano, H. Ogasawara, and M. Kawahara, "A topic model and test history-based test case recommendation method for regression testing," in *Proceedings of the 11th ICST Workshops*, pp. 392–397, 2018.

[11] L. Cai, W. Tong, Z. Liu, and J. Zhang, "Test case reuse based on ontology," in *Proceedings of the 15th PRDC*, pp. 103–108, 2009.

[12] T. Xie, N. Tillmann, and P. Lakshman, "Advances in unit testing: theory and practice," in *Proceedings of the 38th ICSE Companion*, pp. 904–905, 2016.

[13] E. Daka and G. Fraser, "A survey on unit testing practices and problems," in *Proceedings of the 25th ISSRE*, pp. 201–211, 2014.

[14] P. Louridas, "JUnit: unit testing and coiling in tandem," *IEEE Software*, vol. 22, no. 4, pp. 12–15, 2005.

[15] A. Orso and G. Rothermel, "Software testing: a research travelogue (2000–2014)," in *Proceedings of the 21th FOSE*, pp. 117–132, 2014.

[16] "JUnit testing framework." site: https://junit.org. Accessed: 2018.

[17] "Google java style guide." site: https://google.github.io/styleguide/javaguide.html. Accessed: 2018.

[18] "Alibaba java coding guidelines." site: https://alibaba.github.io/Alibaba-Java-Coding-Guidelines/. Accessed: 2018.

[19] B. Xu, J. Qian, X. Zhang, Z. Wu, and L. Chen, "A brief survey of program slicing," *ACM SIGSOFT Software Engineering Notes*, vol. 30, no. 2, pp. 1–36, 2005.

[20] S. Horwitz, T. W. Reps, and D. W. Binkley, "Interprocedural slicing using dependence graphs," in *Proceedings of the 6th PLDI*, pp. 35–46, 1988.

[21] "Difflib-helpers for computing deltas." site: https://docs.python.org/2/library/difflib.html. Accessed: 2018.

[22] "Fuzzywuzzy: Fuzzy string matching in python." site: https://chairnerd.seatgeek.com/fuzzywuzzy-fuzzy-string-matching-in-python/. Accessed: 2018.

[23] A. Ahtiainen, S. Surakka, and M. Rahikainen, "Plaggie: GNU-licensed source code plagiarism detection engine for Java exercises," in *Proceedings of the 6th Koli Calling*, pp. 141–142, 2006.

[24] "Fuzzwuzzy v.s difflib." site: https://python.libhunt.com/compare-fuzzywuzzy-vs-difflib. Accessed: 2018.

[25] L. Prechelt, G. Malpohl, and M. Philippsen, "Finding plagiarisms among a set of programs with JPlag," *Journal of Universal Computer Science*, vol. 8, no. 11, pp. 1016–1038, 2002.

[26] "National student contest of software testing." site: http://www.mooctest.org. Accessed: 2018.

[27] S. Bauersfeld, T. E. J. Vos, K. Lakhotia, S. Poulding, and N. Condori, "Unit testing tool competition," in *Proceedings of the 6th ICST Workshops*, pp. 414–420, 2013.

[28] G. Whale, "Plague: plagiarism detection using program structure," tech. rep., Department of Computer Science Technical Report 8805, University of New South Wales, 1988.

[29] C. Liu, C. Chen, J. Han, and P. S. Yu, "GPLAG: detection of software plagiarism by program dependence graph analysis," in *Proceedings of the 12th KDD*, pp. 872–881, 2006.

[30] D.-K. Chae, J. Ha, S.-W. Kim, B. Kang, and E. G. Im, "Software plagiarism detection: a graph-based approach," in *Proceedings of the 22th CIKM*, pp. 1577–1580, 2013.

[31] M. Chilowicz, É. Duris, and G. Roussel, "Syntax tree fingerprinting for source code similarity detection," in *Proceedings of the 17th ICPC*, pp. 243–247, 2009.

[32] L. Luo, J. Ming, D. Wu, P. Liu, and S. Zhu, "Semantics-based obfuscation-resilient binary code similarity comparison with applications to software plagiarism detection," in *Proceedings of the 22th FSE*, pp. 389–400, 2014.

[33] M. J. Wise, "Yap3: Improved detection of similarities in computer program and other texts," in *Proceedings of the 27th SIGCSE*, pp. 130–134, 1996.

[34] M. J. Wise, "Running karp-rabin matching and greedy string tiling," tech. rep., Basser Department of Computer Science Technical Report 463, University of Sydney, 1993.

[35] M. J. Wise, "String similarity via greedy string tiling and running Karp-Rabin matching," *Online Preprint, Dec*, vol. 119, no. 1, pp. 1–17, 1993.

[36] A. Aiken, "Moss (measure of software similarity) plagiarism detection system." site: https://theory.stanford.edu/~aiken/moss/. Accessed: 2018.

[37] K. W. Bowyer and L. O. Hall, "Experience using MOSS to detect cheating on programming assignments," in *Proceedings of the 29th FIE*, vol. 3, pp. 13B3–18, 1999.

[38] Y. Feng, J. A. Jones, Z. Chen, and C. Fang, "Multi-objective test report prioritization using image understanding," in *Proceedings of the 31th ASE*, pp. 202–213, 2016.

[39] R. Hao, Y. Feng, J. Jones, Y. Li, and Z. Chen, "Ctras: Crowdsourced test report aggregation and summarization," in *Proceedings of the 41th ICSE*, 2019. in press.

[40] H. Jiang, X. Chen, T. He, Z. Chen, and X. Li, "Fuzzy clustering of crowdsourced test reports for apps," *ACM Transactions on Internet Technology*, vol. 18, no. 2, pp. 18:1–18:28, 2018.

[41] Q. Luo, K. Moran, and D. Poshyvanyk, "A large-scale empirical comparison of static and dynamic test case prioritization techniques," in *Proceedings of the 24th FSE*, pp. 559–570, 2016.

[42] C. Fang, Z. Chen, K. Wu, and Z. Zhao, "Similarity-based test case prioritization using ordered sequences of program entities," *Software Quality Journal*, vol. 22, no. 2, pp. 335–361, 2014.

[43] T. B. Noor and H. Hemmati, "A similarity-based approach for test case prioritization using historical failure data," in *Proceedings of the 26th ISSRE*, pp. 58–68, 2015.