Plagiarism detection based on blinded logical test automation results and detection of textual similarity between source codes

Dirson Santos de Campos
Institute of Informatics (INF)
Federal University of Goiás (UFG)
Goiânia-GO, Brazil
dirson@inf.ufg.br

Deller James Ferreira
Institute of Informatics (INF)
Federal University of Goiás (UFG)
Goiânia-GO, Brazil
deller@inf.ufg.br

Abstract— **This Research to Practice Full Paper presents in this paper. Finding logical errors is the most difficult skill for students from all sorts of disciplines that involve computer programming. Unfortunately, because of this difficulty, some students resort to plagiarism. Plagiarism corrupts the evaluation process. Several tools are used in different researches for this purpose. Recent research has defined a taxonomy of the most relevant types of plagiarism that can be found in source codes. The Hybrid Framework was made to map the student plagiarisms using NLP tecnhics and software test automation techniques for automatic exercise correction with tools available for this purpose. This framework has been tested in the laboratory with promising results.**

Keywords— Plagiarism, Code similarity, Tools to Assess Learning, Natural Language Processing, Automatic Exercise Correction, TPACK.

I. INTRODUCTION

Detecting logical errors is a complex activity even for experienced practitioners who need to learn test engineering techniques let alone undergraduate students. Systematic detection of logical errors is critical to student performance in a variety of disciplines involving programming and will have a high impact on students' academic performance.

Not all students can do them and some, unfortunately, resort to plagiarism. Plagiarism is a well-known ethical problem and is analogous to theft, but of intellectual property, that is, of an immaterial good.

Academic plagiarism is a serious flaw and has strong negative impacts on academic and professional life and several types of research are being done in this subject [2, 4, 7, 9, 12, 14, 18, 19].

Automatic correction of programming exercises is the main attraction of using tools in programming disciplines. These types of tools are those that provide immediate or automatic feedback on the result of automated logic tests and blindly, such as, Online Judge tools [1, 3, 5, 8] that are used in laboratories of programming disciplines. From a teaching standpoint, automatic correction is ideal due to the large number of students who produce thousands of different or similar source codes per subject. From the student's perspective, providing immediate feedback, though semantically poor for this audience, is helpful. Feedback is intended to help students improve their work and is arguably an important factor in learning [5].

In this paper, we present empirical research on code similarity detection will be done with students of introductory programming disciplines of University. An innovative hybrid strategy based on the empirical application of taxonomies of the most relevant types of plagiarism that can be found in student source codes. We also added the analysis of automated black-box testing results that are widely used for exercise correction automatic in programming contest and lab.

Plagiarism detection is highly relevant for student assessment, over decades of research several tools and different types of source code similarity detection approaches have been used in academia.

The major contribution of this paper is the hybrid framework, unheard of, at least to our knowledge, and innovative described in this article based on three pillars.

The first pillar that it was used as a pedagogical theoretical basis of our approach research is the framework for teacher knowledge for technology integration called TPACK (Technological Pedagogical Content Knowledge) [25].

The strategy of how to choose the data collections and how to perform the first step empirical study including analysis of automated tests (section V, B - 1) is our contribution to this pillar. The black-box software test automation applies to the analysis of the results of the automatic exercise correction tools (without the teacher's prior knowledge of the internal structure of the submitted student source code) at a given empirically defined checkpoint. For instance, the latest version of the source code submitted by the student in response to an exercise.

The third pillar is NLP (Natural Language Processing). The theoretical and practical foundations of NLP are used for the code similarity detection taxonomy for plagiarism detection synthesized by us in Table 1. This taxonomy also is a relevant contribution to our research because it allows analyzing the results obtained using a state-of-art open tool automatic for determining the similarity between source codes considering plagiarism identifiers that can be strategically associated with the programmatic content of the discipline.

In this empirical research, we chose the Sharif Judge tool [8] for automatic correction of logical errors in exercises of programming and the MOSS tool [10]. The Measure Of Software Similarity (MOSS) tool developed by Stanford University to detect source code plagiarism.

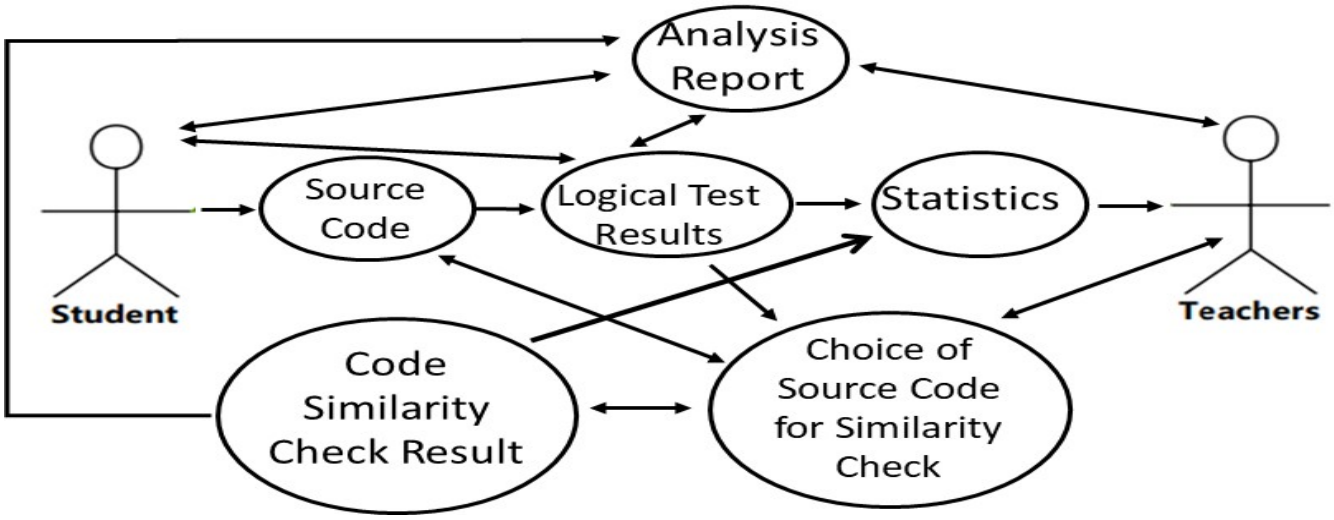

Figure 1 – Hybrid Framework for Plagiarism Detection Simplified

The remainder of this paper discusses in section II the Hybrid Framework for Plagiarism Detection and section III the plagiarism type taxonomy that can be found in source code. Section IV deals with Related Work. Section V deals with the Empirical Study for plagiarism detection based on the results of blinded logic test automation, plagiarism type taxonomy, and textual similarity detection of source codes. Section VI shows the Results and Conclusions. Section VII shows Future Works.

A. Motivation and Research Questions

The analysis of plagiarism in programming exercise submissions in a discipline can have gigantic dimensions. We used for detecting the similarity of the source codes the excellent MOSS tool from Stanford University [10]. It is only available online and it will automatically compare two to two source codes at a time. However, it is possible to call MOSS directly from Sharif Judge.

For instance, if we have 4 source codes from different students, we will have to compare the fourth code with the other 3 (4 with 3, 4 with 2, 4 with 1) because it is not necessary to compare it with itself. Then we owe 3 with others (3 with 2, 3 with 1), it is not necessary to compare it with itself or with the 4 source code that was previously compared. Then we will compare the second code with the others, in this case, the missing comparison (2 with 1). The first code has already been compared with all the other 3 in the previous steps. Thus, there are 6 comparisons.

We used a mathematical induction proof to demonstrate how we generalize this formula. We recall that the proof by induction consists of two cases. The first, the base case (or basis), without assuming any knowledge of other cases. The second case, the induction step, proves that if the statement holds for any given case n = k, then it must also hold for the next case n = k + 1. Considering n = 4, and we apply the formula developed by us (n * (n-1) / 2) we have (4 * (4 -1) / 2 = 6, if the division is not exact, rounding is performed.

The research question is if an analysis of the results of automated logical tests together with the results of the similarity analysis between source codes it contributions or not to detecting plagiarism with more accurately that it using only state-of-art NLP techniques.

This research question was answered from section V and VI, including its implications for practice in the experiment developed in the classroom.

II. HYBRID FRAMEWORK FOR PLAGIARISM DETECTION

There are several research approaches in Computer Science and Engineering Education that emphasize learning lab activities and programming teaching strategies for the discovery of logical errors in students' source code [1, 3, 21].

The experiment in this paper presupposes blind submission, that is, students submit their solutions by file uploading (source code) using a tool, in our case, Sharif Judge [8]. After their submission, the files are tested using the automated black-box techniques implemented in the tool. For pedagogical reasons, it was incorporated into our framework (Figure 1) the phases or steps that include a methodology for tests for detecting logical errors that are done blindly.

Test-Driven Learning (TDL) [22] is based on white-box testing techniques, it requires knowledge of the source code. However, TDL can be adapted to work with black-box testing techniques successfully in the classroom in introductory programming courses [23].

TDL methodology [22] is a pedagogical approach based on software testing fundamentals. It was proposed at SIGCSE'06 as a mechanism for teaching and motivating the use of automated testing as both a design and a verification activity on CS1 and CS2 courses [24]. TDL is based on Test-driven development (TDD) [29] without imposing significant additional time for the test activity. In its turn, TDD is used with agile methods both in academia and in the software industry. TDD methodology, as well as TDL, are based on white-box testing techniques.

In our empirical research approach also it needed a technology integration with didactic foundations in a pedagogical framework. It was used the framework called TPACK (Technological Pedagogical Content Knowledge) [25] because it is pedagogically consistent and comprehensive. It has been analyzed in section IV – A.

Hybrid Framework for Plagiarism Detection was developed to allow the establish connections, and make predictions about possible detections of plagiarism phenomena in the classroom of introductory programming courses using specific free tools for this purpose.

TABLE 1 - PLAGIARISM TAXONOMY THAT CAN BE FOUND IN SOURCE CODES.

Number	Short Description of Pattern Types of Plagiarism	Identifier
0	Copy without syntactic changes.	NO_CHANGE
1	Code rewriting by adding or omitting comments. Changing identifier names (for example, variables), whitespace, indentation, and line breaks.	COMENT NOM_ID FORMAT
2	Change of position of variables in code. Changing the order of operators or operands in logical expressions. Reordering code blocks.	POS_VAR POS_OPE POS_BLO
3	Data type change.	TYPE
4	Introduction of purposeful logical errors.	BUG
5	Changing parameters of methods. Changes may include passing parameters by value or by copy.	PARAMET
6	Replace function calls with their content once or multiple times or vice versa.	FUNCTION MULT_F
7	Substitution of logically equivalent expressions. Replacement of repeating structures. Change of type of selection structures. Code refactoring.	EXP_EQU FOR_WHI IF_CASE INLINE
8	Add redundant commands and / or variables without use.	USELESS
9	Structural Code Redesign.	ESTRUT
10	Combination of copied code with original code.	COP_ORIG

Hybrid Framework for Plagiarism Detection has been simplified in Figure 1. It is technologically hybrid because it uses two different areas of knowledge from computer science and engineering techniques contemplated by SWEBOK (Guide to the Software Engineering Body of Knowledge) [26]. The first area is the automation of logical tests represented in the hybrid framework by the ellipses: Source Code, Logical Test Results, Statistics, and Analysis Report. The second area uses the fundamentals of NLP are represented by the ellipses: Choice of Source for Similarity Check, Code Similarity Check Result, and Statistic.

It is possible to verify in Figure 1 all the phases or stages that were contemplated by the information flow, represented by arrows in the hybrid framework, from the receipt of students' source codes to the analysis of results by the teacher.

The two main differences from the hybrid framework concerning the state-of-the-art, at the time of writing this paper, are:

- analyze the results of automated software tests done blindly in the students' source codes before any analysis on code similarity.

- interpret the results of the similarity detection between source codes considering not only the results obtained by the MOSS tool [10] but also considering the plagiarism taxonomy (Table 1).

The plagiarism taxonomy uses identifiers that help to standardize and classify different types of plagiarism attempts into degrees of difficulty that it facilitates the analysis of plagiarism by the teacher about a given programmatic content of the discipline.

In the software test literature, this type of code is called a mutant code, usually, the mutant code is created by the tester [28], but in our experiment, it is created by the student himself, not by the evaluator represented by the discipline staff.

Mutant code is understood as the different versions of source code submitted by the same student for the same programming exercise.

Source code submitted blindly is understood, in this paper, as a source code that the student submits in an Online Judge tool with automatic feedback. In this case, the teacher knows absolutely nothing about the internal structure and data of the student solution.

The fundamentals of NLP are present both in syntactic analysis techniques, such as the based on source code characteristics developed by other researchers [2, 7, 12, 17, 20] that we analyzed to build Table 1, as well as in techniques for detecting similarities between source codes implemented in the MOSS tool how the Winnowing Algorithm [16] that it was explained in section IV - C of this paper.

III. PLAGIARISM TAXONOMY THAT CAN BE FOUND IN SOURCE CODES

There are several types of research about techniques NLP for detecting plagiarism with different approaches in the literature. Among them, we find both for the description of the type of plagiarism and for the standardization of the identifiers that classify them. So a plagiarism taxonomy that can be applied in source codes can be very useful for researchers and educators.

The definition of taxonomy can help the standardization and evaluation of the results obtained provided by several types of tools used in the detection of plagiarism in different experiments pedagogical in programming learning of introductory disciplines. Thus, a description of patterns types of plagiarism it is considered when a similar type of plagiarism occurs. Using patterns, one can measure the degree of difficulty in detecting plagiarism more consistently among different research.

It is not trivial to detect specific plagiarism similarity identifiers in source codes. Thus, there are several types of research on these subjects. In this paper, we will analyze

some of them that we consider useful for our research approach.

Faidhi and Robinson [2] that these measures: no. of characters per line, no. of comments lines, no. of indented lines, no. of blank lines, average procedure, function length, no. of reserved words, average identifier length, average spaces percentage per line, no. of labels and gotos, and variety of identifiers. They are the most likely to be altered by a novice programmer committing plagiarism [2].

Joy and Luck [7] discuss techniques for plagiarism: lexical and structural changes in source code.

The main lexical changes are comments can be reworded, added and omitted and formatting can be changed and Identifier names can be modified.

The main structural changes:

- loops can be replaced,
- nested if statements can be replaced by case statements, and vice-versa,
- statement order can be changed,
- provided this does not affect the meaning of the program,
- procedure calls may be replaced by function calls, and vice-versa,
- calls to a procedure may be replaced by a copy of the body of the procedure, ordering of operands may be changed (e.g. x < y may become y >= x).

Mozgovoy [12] list the possible techniques that a plagiarizer can use. They are:

- changing comments (rewording, adding, changing comment syntax and omitting),
- changing white space and layout, renaming identifiers, reordering code blocks,
- reordering statements within code blocks, changing the order of operands/operators in expressions,
- changing data types, adding redundant statements or variables,
- replacing control structures with equivalent structures (while-loop by do-while loop,
- nested if statements by a switch-case block and so on),
- replacing the functional call by the body of the function.

Chowdhury and Bhattacharyya [17] analyze two plagiarism type: textual and source code plagiarism. The second one allows manipulation from vicinity plagiarism, reordering structure, no change plagiarism, and language switching plagiarism.

Meuschke et al. [20] proposed approaches that achieved promising results, most of the research still addresses text-based plagiarism detection only.

Considering the researches analyzed [2, 7, 12, 17, 20] and also our observations on empirical experiment described in section V. We provide, in Table 1, a plagiarism taxonomy that can be found in source codes. It represents the number (zero to ten) for identifying the type of plagiarism, a brief description of its semantic meaning, and the acronym of the similarity Identifier using in this paper.

IV. RELATED WORKS

There are three different types of researches that influenced our research approach are the theoretical abstract framework for the pedagogical use of technology, automatic correction of programming exercises and, text similarity for plagiarism detection, in particular source codes, that it has been implemented in tools for this purpose.

A. Research on Abstract Framework for the Pedagogical use of Technology

TPACK [25] is an abstract framework that makes connections among technological knowledge (TK), pedagogical knowledge (PK), and content knowledge (CK). At the intersection of this knowledges are TPK (Technological Pedagogical Knowledge), TCK (Technological Content Knowledge), and Pedagogical Content Knowledge (PCK). The core of TPACK was formed by the interconnection of these parts.

Technology knowledge, in the various dimensions of research using the TPACK abstract framework, is always in states that correspond to the specific flux in empirical research. Considering aspects as tool-and-its-use, whereby data is gathered and analyzed, and inferences according to the researchers' needs.

TPACK framework is abstract and generalist while the hybrid framework proposed by us is used in a specific technology knowledge about programming learning, instructions, assessments of programming exercises blindly submitted by students, and the detection of plagiarism that occurs in these submissions. The design of our hybrid framework is based on TPACK guidelines that can be used to represent / research and to create matter knowledge about a specific subject. In our research, this includes relevant pedagogical considerations for teaching and the learning of programming.

B. Research on Automatic Correction of Programming Exercises

There is research based on automatic corrections of programming exercises using black-box techniques [1, 3, 15] or white-box techniques [22, 24, 29] with or without TDL methodology [22]. TDL was analyzed in section II.

Different from our research approach, other researchers [1, 3, 5, 15, 22, 24, 29] use the results of the logical tests of students in their classroom or programming contest, but do not provide support for plagiarism detection. However, the research that has both the black-box techniques and also provides automated feedback are related works.

They are considered related works because they are similar to phases of the hybrid framework responsible for automated testing blindly.

Keuning et. al. [5] reviewed 102 papers and cited 69 different tools used in programming teaching that provide some type of student interaction. These studies confirm that tools that provide automatic feedback are important for students because they receive feedback on the results of the logical tests immediately.

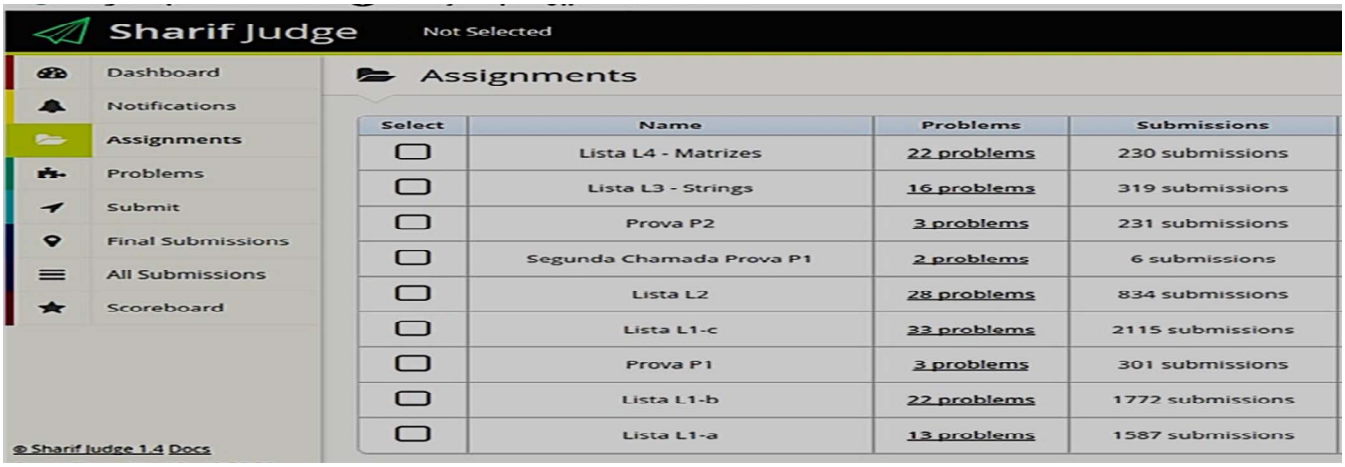

Fig. 2. Sharif Judge Student Submission Results

Good feedback is essential to improve students' learning in Higher Education (HE) [27]. Good feedback decreases students doubts and the more knowledge, less plagiarism.

Research involving online Judge tools [1, 3, 15] in the classroom, in addition to the benefits of automated feedback to students, it allows teachers to check didactic activities involving logical exercises using one or more programming languages.

The analysis report of the results of the logical tests produced by these tools allows for obtaining valuable information. This manipulated information correctly allows the monitoring of student academic performance regarding the learning of programming logic. Logical errors are a good metric for this.

The difference between these related works in this section and the one developed in this paper is that the logical errors are analyzed differently. They represent faults in programming learning. So, in our research approach, the difficulty with logical error and the pressure to deliver the correct answer can lead the student to copy another colleague's solution, and, consequently, to cause the plagiarism.

C. Research on text similarity for plagiarism detection

For pedagogical reasons to make clear the construction process of plagiarism taxonomy (Table 1) some of the most important related work [2, 7, 12, 17, 20] was analyzed in section III.

Schleimer at all. [16] uses source code similarity identification techniques based on or derived from Winnowing Algorithms: Local Algorithms for Document Fingerprinting.

This was the main algorithm that allows MOSS [10] accuracy initially.

The tool has been improved over time by including several features. According to the information contained on the tool's page [10], the algorithm behind MOSS has a significant improvement over other known plagiarism detection algorithms, at least over those known to the authors of the tool itself. MOSS is the second most widely used tool to detect source code similarity in the Systematic Review promoted by Matija [9]. Matija [9] reviewed 150 papers in his Systematic Review (SR) on this subject and none of the approaches described in this SR have the innovative and unpublished hybrid approach (at least to our knowledge) used by us in this empirical study. The first is JPLAG [6].

In this paper, we adopted the MOSS tool [10].

The general idea of the Winnowing Algorithm [16] is:

- Divide input into k-grams (a contiguous substring of length k that forms substrings)
- For each k-gram, associate it with a hash value. Hashing strings of length k is costly for large k.
- Select minimum values (in each window) as document fingerprints
- To compare file similarity, just look at the corresponding fingerprints.

Preprocessing aims to eliminate unimportant data, and depends on the type of content analyzed. This technique is applied to both natural language and source code, but differently. If it is a Source Code we have: Variable name → id, Function name → id, and Comments → .

If it is a text in Natural Language we have: Whitespace → λ, the score → λ, Non-differentiation in upper and lower case letters, ie A - Z → a – z

The representational similarity techniques of the MOSS tool are syntactic, so it works to the textual similarity of the source code. The difference is that we adoption also of plagiarism taxonomy (Table 1) it is very important because one can go beyond simple syntax analysis and incorporate important patterns related to programming activity.

However, it should be noted that there are other techniques reported in the literature indirectly involving source code syntax, such as the metric similarity technique, feature-based similarity [4].

Metric similarity involves information about the internal structures of the source code, ie requires access to this code and includes the number of function calls, number of defined and used local variables, number of non-local variables used or defined, number of parameters, number of sentences, number of selection structures, number of repetition structures.

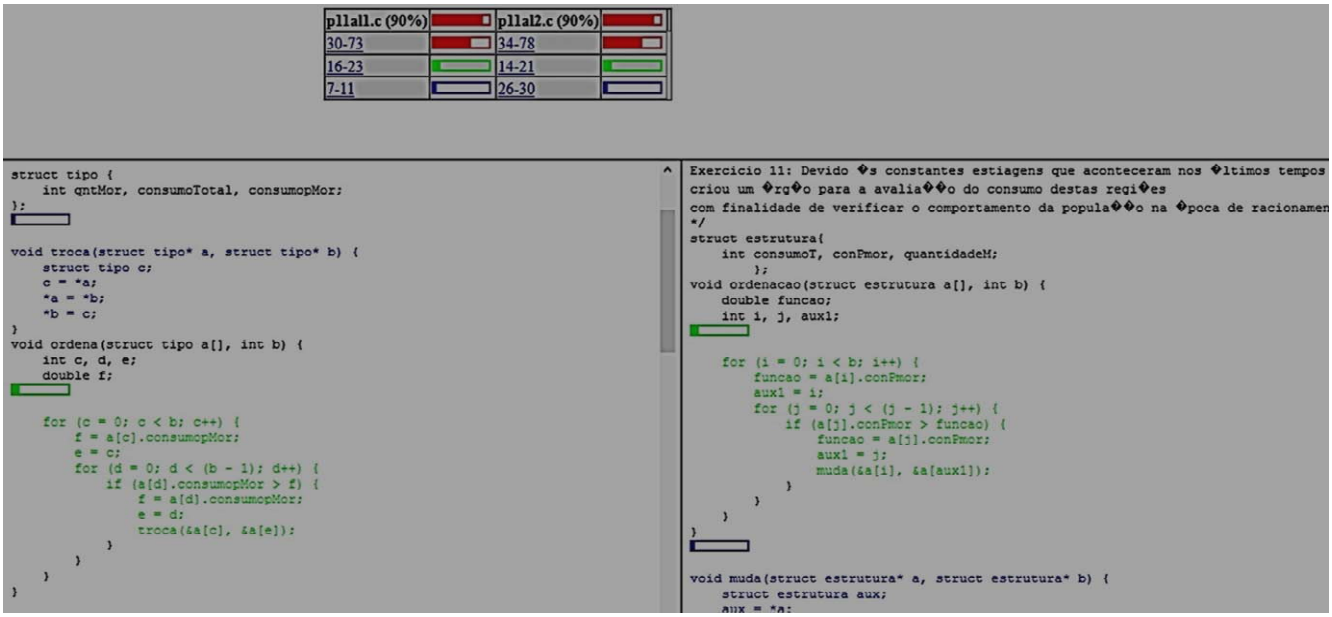

Figure 3: Comparison of 2 source codes from data sample with identifiers of type NO_CHANGE, COMMENT, NOM_ID, FORMAT, POS_VAR, POS_BLO.

V. EMPIRICAL STUDY FOR PLAGIARISM DETECTION BASED ON HYBRID FRAMEWORK

First, it is important to mention that the didactic objective of the empirical study was not to punish the students for possible plagiarism, after all, the plagiarism in source code cannot be categorically affirmed based only on empirical results measuring similarity as stated by the authors of the MOSS tool [10]. The empirical results allow to substantially decrease both the number of students and the number of comparisons among the select source codes suspected of fraud. Pedagogically, it is strongly advised to talk to the students invo lved in case of complaints. In this case, comparisons can be presented as in Figure 3.

We have adopted the strategy pedagogical evaluation with low score in the submissions tasks. The validation of learning is made with formal evaluations in the classroom and the lab, involving logical aspects explored in the submitted exercises. The evaluation of the lab is done in Sharif Judge [8].

We adopted the Hybrid Framework that maps the parts referring to the flows of the TPK, TCK, PCK components of the TPACK framework [25]. We applied technological knowledge to the didactic objective of measuring the detection of students' logical errors, eliminating possible attempts at plagiarism. The technological knowledge refers to NLP methods and techniques and software test automation used by automatic exercises correction implemented in MOSS [10] and Sharif Judge [8] respectively.

MOSS tool was used to detect similarity only in suspicious codes, after analyzing the results obtained by the automated tests, not in the whole data set described the Figure 2. We used MOSS with Perl scripts [13] in line from a command window and MossSwing [11] with the graphical interface.

Sharif Judge [8] is used as a support tool for teaching programming.

The tool automatically tests the functional source code requirements submitted by students based on the execution of an automated black-box test case test suite that provides automatic feedback on the results of the logical tests.

A. Data Collection Description

The data collection that corresponds to the corpus of this paper was made in the introductory programming discipline from 2018 to 2019 by the Institute of Informatics (INF) of the Federal University of Goiás (UFG). It was produced 7,395 source codes in C language (Figure 2).

Students submitted the same exercises in the same discipline, made available on the same exercise lists, using the same Sharif Judge tool [8]. In addition, they underwent the same evaluations with the same test cases for detecting logical errors. For each exercise to be submitted, input and output should be included in the statement to avoid presentation errors in blind tests.

B. Execution of the Empirical Study

The empirical study was done based on the hybric framework following the control flows between the student and the teacher in figure 1 in two technological dimensions:

- The first is the analysis of the results the exercises automatic correction tool, in the Figure 1, it corresponds the phases are Source Code, Logical Test Results, Statistics and Analysis Report on logical tests.
- The second is the analysis the similarities of suspicious source codes, in the Figure 1, it corresponds the phases are Source Code, Choice of Source Code for Similarity Check , Code Similarity Check Result and Analysis Report on similarity analysis.

1) First Step Empirical Study

In this first part of the empirical study it was used as a direct metric for detecting logical errors in source code the satisfaction of the test case. For example, if 4 test cases were

necessary and sufficient to detect logical errors in a given list exercise. The student submits the source code with his answer. If his code satisfies one of the test cases automatically tested by the tool, it satisfies one test case and fails in 3 cases. In this case, the percentage is 25% (hit 1 of 4 test cases).

For each source code submission, the result obtained by the student is automatic feedback, in this way, the student knows the hit percentage in his submission. If the result is 100% the source code is logically correct according to the logical tests, although that the possibility of plagiarism exists. It will be investigated in second step this empirical study.

The first part of the empirical study was done statistically using the paired sample technique. This parametric test applies when the two sets of observations are from the same individual. The two samples are taken from of the n submissions made by the same student during the time the submission was open, if there is a second sample (n> 1), otherwise the only sample is chosen. The first sample analyzes is the first source code submitted by students' without syntax errors.

The second sample is the student's final version chosen by the student. It is not necessarily the last version submitted in the Sharif Judge tool because the student can choose which version will be submitted for teacher evaluation.

If statistically there was an improvement in the test case hit percentage between the two paired samples. It was considered that there was an improvement in the source code quality of the student as they satisfy more test cases designed to test logical errors. The comparison using paired samples is a differential of our research since it is much more common to analyze only a single version of the students' source code, the last version submitted. In this paper, we call the analysis of paired samples by simply intra-code analysis.

Figure 2 represents the results of the empirical study submissions where students were offered 140 different exercises with subjects typical of introductory programming disciplines in the imperative paradigm. Analyzing Figure 2, we observed that 7,395 source codes were submitted.

This version, the second paired sample, is used for the final exercise evaluation as well as for possible plagiarism detection. At any time the code can be evaluated by teachers. For instance, he can choose to evaluate only the source codes submission by students in the last 24 hours after opening the task. In this case, it is using the All Submission option of Sharif-Judge it is possible to select one of the lists exercices, for example, the L4 in Figure 2, list for matrices and collect the submission data in excel spreadsheet format (all_excel option of tool).

2) Second Step Empirical Study

In this part of the empirical study, the list chosen for sampling refers to the dynamic allocation of memory that is considered difficult for students to understand, that is, the subject is more challenging and prone to more errors.

A sample of the source codes from one of the exercise lists submitted by the students of the discipline was made. The list in question involves Structs exercises and dynamic memory allocation in C (List 1-a with 13 issues and 1587 submissions from Figure 1). The average number of submissions per student was 61.04 (1587 submissions / 26 students). There are 13 exercises so on average per exercise was 4.7 (61.04 submissions / 13 exercises). This list was submitted by 26 students and not all students submitted the 13 exercises contained in it.

The choice of source codes compared under suspicion is not automatically defined, an intra-code and inter-code analysis were performed.

The intra-code analysis is done with two paired source code samples, ie from the same student, the first version submitted by him, and the second sample the version chosen by him. This was done to validate the code similarity of the same author which should hypnotically have a high similarity as made by the same author.

The percentage of results is given by the MOSS tool [10] that compares pairs. The result is unique for each student and ranged from 0.2387 (23.87%) to 0.9999 (99.99%). It was performed 26 intra-code comparisons by exercises.

Assuming that all students submit at least two different versions of maximum codes and that they submit all exercises the maximum number of comparisons is 388 (26 student comparisons x 13 exercises). As not everyone submitted all the exercises and the outlines were removed. Comparisons dropped to 136 comparisons.

Outlines were considered students who did not submit even half of the rounding up exercises (8 exercises).

If the first version was logically incomplete (more than 50% of the functionalities missing), in this case, we moved to the next submitted version, if it exists because although the average submission rate is 4.7 sources per exercise there were smaller submissions.

Students who submitted the first version of the logically correct and / or practically correct source code were also considered candidates for plagiarism.

Students who had symmetry variations above 30% between the code itself were also candidates for plagiarism. Students who submitted all versions of the source code without any logical test case hit were also considered outlines because if there was plagiarism between them there was no academic gain.

Comparison between codes of different students was done only by students with the same percentage accuracy in automated logic tests since plagiarism aims to improve the grade, not to make it worse.

It was further considered whether the similarity involved code snippets with semantics from Table 1.

Note that the percentage among different students was 90%, for instance, it is a serious candidate for plagiarism.

Remember that the plagiarism is a statement that someone copied code deliberately without attribution, and while Moss automatically detects program similarity, it has no way of knowing why codes are similar. It is still up to a human to go and look at the parts of the code that Moss highlights and make a decision about whether there is plagiarism or not.

These scores are useful for judging the relative amount of matching between different pairs of programs and for more

easily seeing which pairs of programs stick out with unusual amounts of matching. But the scores are certainly not proof of plagiarism.

Someone must still look at the code. In this article we analyze the source code with a hybrid approach, that is, we analyze, according to Figure 3, semantic aspects described in

Table 1 and not only syntactic aspects of similarity allowing us to analyze detecting plagiarism more accurately, fewer comparisons of code.

Since there are no tools that automatically classify a code snippet into one of the identifiers in table 1. The following strategy has been adopted. Since the teacher has a valid solution to the problem, ie a logically correct answer to the exercise (passes all blindly automated test cases) mutants variations of the correct source code can be created to simulate plagiarism accordingly. with the identifiers in Table 1.

Then we just compared the percentage of symmetry obtained between mutant codes. The codes produced by students who had a similar percentage were examined as shown in Figure 3.

VI. RESULTS AND CONCLUSIONS

The first result important was to discover the formula that allows us to generalize the similarity comparisons in source codes in our experiment. The formula was demonstrated by mathematical induction in section I, letter A.

Strictly speaking to find plagiarism in n submissions of the same student we must compare it with the other students in the class by analyzing $n (n-1) / 2$ pairs of submissions.

It is essential to know the maximum amount of comparison and defining strategies for comparing them. In the case of this paper, the results of automated logical tests were analyzed together with the plagiarism taxonomy that can be found in source codes described in table 1.

Therefore the need for exercise correction automation and also for plagiarism detection strategies are fundamental for the viability of empirical studies in Informatics and Education. This set of changes generated is used to emphasize that this type of approach methodology considers measuring learning pedagogical aspects using software testing techniques to make it possible to develop the experiment pedagogical in the lab.

In this methodology the submission of the students' source code is always done blindly, that is, the logical test is done without any prior analysis of the student's source code by any staff member of the discipline.

Technology knowledge, in the dimensions of our experiment, must include pedagogical research for teaching and learning. In this sense, it was important to use the TPACK abstract framework [25] as a pedagogical basis for our hybrid framework.

Besides, they proposed has two major subdivisions of technological knowledge: NLP and automatic black-box testing. This framework was implemented in steps.

Regarding the first step in analyzing the results of the logical tests of the exercises performed by the students, it is observed clearly that, often, the student, especially of low performance, does not know how to take the next step in the correction of logical errors based on the obtained result. However, there is an empirically proven student gain by doing the qualitative analysis of the evolution of the source codes submitted by the students which occurred in 78% of the cases in the analysis of the paired samples of the students who went until the end of the course.

Students who did not go to the end were not analyzed because they were considered statistically outsides. In our approach to the empirical study this does not mean that the student got the exercise right, only that there was progress in the programming logic detectable by automated logic tests, that is, in learning, for example, in the first sample submitted the student did not hit any case. of the test sample and in the evaluated sample hit 1 or more test cases.

This was done to assess the student's evolution as the learning assessment consisted of a heavyweight in the summative formal assessments.

The number of possible comparisons is $n (n-1) / 2$ pairs of submissions. For the 7395 submissions of Figure 1 ($n = 7395$) the number of possible comparisons is extremely huge.

Therefore, the development of proposed hybrid solutions that greatly reduce comparisons for plagiarism detection based on intra-code comparisons (same student) and between different students is needed. Clear criteria for defining outsides have been presented, as seen empirically, it is not enough to analyze only the percentage of similarity, one should examine this percentage for errors based on typical plagiarism patterns and robust taxonomies tested on them in other empirical studies. using robust tools available from other researchers such as Shariff Judge and other Online judge tools.

We analyzed threats to internal validity, in the case of our experiment using an imperfect instrument used to measure the plagiarism detection capabilities that affect the results. To correct such threats we made a visual inspection of the source code and an analysis of the results obtained in theoretical and practical evaluations with the students that allowed us to empirically validate the results.

An experiment has good external validity if the results are not unique to a particular set of circumstances but are generalizable. The best way to demonstrate generalizability is to repeat the experiment many times in many different situations. As we adopted a series of exercises from each topic we believed that if other exercises were used the results would be similar.

This way, the empirical study may be reproducible by other researchers and with different experimental parameters.

VII. FUTURE WORKS

Plagiarism is a delicate academic problem. Future works we intend to analyze the sentiment analysis of students regarding the difficulty of programming using others advanced text mining techniques.

ACKNOWLEDGMENT

The authors would like to thank the funding for the Postgraduate Program in Computer Science (PPGCC) of Institute of Informatics (INF) of Federal University of Goiás (UFG), Brazil. The authors thank INF-UFG for their support throughout the research.

REFERENCES

[1] R. Year and L. Martínez. A recommendation approach for programming online judges supported by data preprocessing techniques. Appl Intell 47, 277–290 (2017).

[2] J. A. W. Faidhi and S. K. Robinson. An empirical approach for detecting program similarity and plagiarism within a university programming environment. Computers & Education, Elsevier Science Ltd, 1987, 11:11–19.

[3] S. Wasik, M. Antczak, J. Badura, A. Laskowski, and T. Sternal. A Survey on Online Judge Systems and Their Applications. ACM Comput. Surv. 51, 1, Article 3 (January 2018), 34 pages. DOI:https://doi.org/10.1145/3143560.

[4] M. Heon and D. Murvihill. Program Similarity Detection with Checksims. Technical Report of Bachelor of Science. Available at: https://web.wpi.edu/Pubs/E-project/Available/E-project-043015-122310/unrestricted/CheckSims.pdf. Accessed in 01/11/2019.

[5] H. Keuning, J. Jeuring and B. Heeren. Towards a Systematic Review of Automated Feedback Generation for Programming Exercises. In the 21th Annual Conference on Innovation and Technology in Computer Science Education (ITiCSE '16), July 11–13, 2016, Arequipa, Peru, p. 41-46.

[6] JPLAG: Java Detecting Software Plagiarism. (2010) Available at: https://jplag.ipd.kit.edu/. Accessed in 03/03/2020.

[7] M. Joy and M. Luck. Plagiarism in programming assignments. IEEE Transactions on Education, Institute of Electrical and Electronics Engineers, 42:129–133.

[8] M. Naderi. Sharif Judge version 1.4 Documentation. Available at: https://github.com/mjnaderi/Sharif-Judge/tree/docs/ v1.4. Accessed in 05/06/2019.

[9] M. Novak, M. Joy and D. Kermek, Source-code Similarity Detection and Detection Tools Used in Academia (2019). In, ACM Transactions on Computing Education, 10.1145/3313290, 19, 3, (1-37).

[10] MOSS: A System for Detecting Software Similarity. Available at:: https://theory.stanford.edu/~aiken/moss/. Accessed in 02/06/2019.

[11] MossSwing: a Swíing GUI for Moss. Available at: https://github.com/alkanmuhammet/MossSwing. Accessed in 03/06/2019.

[12] M. Mozgovoy. Desktop tools for offline plagiarism detection in computer programs. Informatics in education 5, 1 (January 2006), 97–112.

[13] Perl: Perl for MS Windows. Available at: http://strawberryperl.com/. Accessed in 03/06/2019.

[14] J. Y. Poon, K. Sugiyama,, Y. F. Tan and M.Y. Kan. Instructor-centric source code plagiarism detection and plagiarism corpus. Proceedings of the 17th ACM Annual Conference on Innovation and Technology in CSE, pages 122– 127.

[15] M. T. Pham, and T. B. Nguyen. "The DOMJudge Based Online Judge System with Plagiarism Detection," 2019 IEEE-RIVF International Conference on Computing and Communication Technologies (RIVF), Danang, Vietnam, 2019, pp. 1-6.

[16] S. Schleimer, D. S. Wilkerson and A. Aiken. Winnowing: Local Algorithms for Document Fingerprinting. ACM Special Interest Group on Management of Data (SIGMOD), June 9-12, 2003, San Diego, CA.

[17] H.A. Chowdhury and D. K. Bhattacharyya. Plagiarism: taxonomy, tools and detection techniques. In: Paper of the 19th National Convention on Knowledge, Library and Information Networking (NACLIN 2016) held at Tezpur University, Assam, India (2016).

[18] T. Foltýnek, N. Meuschke, and B. Gipp. Academic Plagiarism Detection: A Systematic Literature Review. ACM Comput. Surv. 52, 6, Article 112 (October 2019), 42 pages. https://doi.org/10.1145/3345317

[19] I. Albluwi. 2019. Plagiarism in Programming Assessments: A Systematic Review. ACM Trans. Comput. Educ. 20, 1, Article 6 (December 2019), 28 pages.

[20] N. Meuschke, V. Stange, M. Schubotz, M. Kramer, and B. Gipp. 2019. Improving academic plagiarism detection for STEM documents by analyzing mathematical content and citations. In Proceeedings of the ACM/IEEE Joint Conference on Digital Libraries (JCDL'19).

[21] R. P. Medeiros, G. L. Ramalho and T. P. Falcão, "A Systematic Literature Review on Teaching and Learning Introductory Programming in Higher Education," in IEEE Transactions on Education, vol. 62, no. 2, pp. 77-90, May 2019.

[22] D. Janzen, and H. Saiedian, "Test-Driven Learning in early programming courses." In Proceedings of the 39th SIGCSE technical symposium on Computer science education, March 12-15, 2008, Portland, OR, USA. DOI = 10.1145/1352135.1352315.

[23] D. S. Campos, A. J. Mendes and M. J. Marcelino, "Case study using Test-driven Learning methodology for building contextualized feedback by logical errors in Introductory Programming Learning.". in IEEE Frontiers in Education Conference (FIE), pp. 606-613, October 22-25, 2014 Madrid, Spain.

[24] C. Desai, D. Janzen and K. Savage, K, "A survey of evidence for test-driven development in academia". Proceeding in ITiCSE-WGR '07 Working group reports on ITiCSE on Innovation and technology in Computer Science Education. Volume 39, Issue 4, December 2007, Pages 204-223. DOI = 10.1145/1345443.1345441.

[25] M. Koehler, M. and P. Mishra. "What is Technological Pedagogical Content Knowledge (TPACK)?" Contemporary Issues in Technology and Teacher Education, 9(1), 60-70. Waynesville, NC USA: Society for Information Technology & Teacher Education, 2009.

[26] P. Bourque and R.E. Fairley, eds., Guide to the Software Engineering Body of Knowledge, Version 3.0, IEEE Computer Society, 2014; www.swebok.org.

[27] B. Deirdre, "Strategies for using feedback students bring to higher Education". In Assessment & Evaluation in Higher Education, Vol. 34 (1), pp. 41-50, 2009. DOI = 10.1080/02602930801895711.

[28] T. Lewowski and L. Madeyski, Mutants as Patches: Towards a formal approach to Mutation Testing, Foundations of Computing and Decision Sciences, 44(4), 379-405, 2019. doi: https://doi.org/10.2478/fcds-2019-0019.

[29] J. Spacco and W. Pugh. Helping students appreciate test-driven development (TDD). In Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications (OOPSLA '06), 2006. Association for Computing Machinery, New York, NY, USA, 907–913. DOI:https://doi.org/10.1145/1176617.1176743