IEEE Access
Multidisciplinary : Rapid Review : Open Access Journal

Received April 5, 2021, accepted April 12, 2021, date of publication April 16, 2021, date of current version April 26, 2021.
Digital Object Identifier 10.1109/ACCESS.2021.3073703

Explanation in Code Similarity Investigation

OSCAR KARNALIM[1,2], (Graduate Student Member, IEEE), AND SIMON[1]
[1]School of Electrical Engineering and Computing, University of Newcastle, Callaghan, NSW 2308, Australia
[2]Faculty of Information Technology, Maranatha Christian University, Bandung 40164, Indonesia

Corresponding author: Oscar Karnalim (oscar.karnalim@uon.edu.au)

This work was supported by the University of Newcastle, Australia.

ABSTRACT When using code similarity detection to uncover code plagiarism and collusion, the marker needs to determine whether any detected similarities might be the result of coincidence. But understanding the similarities can be difficult and might be prone to human error, because few tools facilitate the investigation process, and if they do, the similarities are not explicitly explained in human language. This paper presents STRANGE, an investigation module that exclusively explains code similarities in natural language (English and Indonesian). For the purpose of reusability, STRANGE can be embedded in JPlag and other code similarity detection tools. It can also act as a standalone tool for measuring source code similarity. Our evaluation shows that STRANGE is more helpful than JPlag in the investigation process since it explains the similarities in natural language. Further, its effectiveness is comparable to that of JPlag but higher on trivial disguises of the sort that novice students will tend to apply when disguising copied code.

INDEX TERMS Code similarity detection, collusion, computing education, natural language explanation, plagiarism, programming.

I. INTRODUCTION

Source code plagiarism and collusion are two forms of academic dishonesty in computing education [1]. They both involve reusing source code without adequate acknowledgment to all of the contributors [2], [3], but they differ in the awareness of the contributors: plagiarism is typically copying from people who are unaware that the copying is taking place, whereas with collusion, all of the contributors are generally aware of the copying.

Many strategies have been developed to deal with academic dishonesty [4]. In the context of source code plagiarism and collusion, it is common to educate the students about academic integrity, penalizing them only if misconduct subsequently occurs. At the beginning of the course, the students are taught about academic integrity in programming [5]. Afterwards, for each assessment, the similarity of the students' programs will be measured; all students whose programs are deemed inappropriately similar upon manual investigation will be penalized. Since it can be demanding to check all possible program pairs, the similarity measurement is often performed with the help of an automated similarity detection tool such as JPlag [6] or MOSS [7].

The associate editor coordinating the review of this manuscript and approving it for publication was Alba Amato.

In using a code similarity detection tool for academic purposes, Mišić *et al.* [8] suggest four steps to be followed, adapted from Culwin and Lancaster's work in text similarity detection [9]. *Code collection*, gathering all students' programs for a particular assessment, can be automated with an assessment submission system (e.g., BOSS [10]). *Similarity detection* pairs the students' programs and calculates the degree of similarity of each pair using the detection tool. *Similarity confirmation*, performed manually by the instructors, checks whether the resulting similarities reflect the human perception of similarity. *Plagiarism (or collusion) investigation* is where the instructors examine each program pair that demonstrates high similarity to determine if the similarity constitutes plagiarism or collusion.

Mišić *et al.*'s last step [8] is the one that involves a 'burden of proof' [11], where the instructors should be able to demonstrate that the suspected cases are not a result of coincidence. Knowing this, some detection tools (e.g., JPlag [6], Sherlock [11], and MOSS [7]) enable side-by-side comparison of each suspicious program pair, with similar code fragments marked, to assist with the investigation. This is clearly beneficial for the instructors as the similarities are visually mapped to the students' programs.

Notwithstanding the benefits, the side-by-side comparison module still has four limitations. First, the similarities can be hard to understand: two similar code fragments can look

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/

different at first glance due to surface level variation (changes that do not affect program semantics, such as in comments, white space, and identifier names). Second, the similarities are neither explained nor summarized in natural language, leaving the instructors to 'translate' them during the investigation process, and subsequently if the similarities are to be passed to people who are not necessarily code-literate, such as higher officials or the suspected students. Third, similar comments are not marked even though such comments can be evidence of plagiarism or collusion [12]. Fourth, the comparison module needs to be developed from scratch for each similarity detection tool, which can be demanding and time consuming [13].

To deal with these issues, this paper proposes STRANGE (Similarity TRacker in Academia with Natural lanGuage Explanation). It is a side-by-side comparison module that explains syntax and comment similarities with their surface level variation in natural language. STRANGE can be embedded in other similarity detection tools via command line instructions. It can also update the side-by-side comparison result of JPlag [6]—a common tool for code similarity detection [14]—and it can act as a standalone similarity detection tool. To the best of our knowledge, this is the first tool that combines these features.

The tool has been used for detecting code plagiarism and collusion in six programming classes offered in two academic semesters. The classes covered three courses: introductory programming (Python and Java), basic algorithms and data structures (Python), and advanced algorithms and data structures (Java).

STRANGE is expected to help instructors in raising suspicion of code plagiarism and collusion. Similar code segments can be more identifiable and self-explanatory in STRANGE than in other similarity detection tools, as the similarity is exclusively explained and summarized in natural language. It might also expedite the investigation process. For instructors who are accustomed to ignoring surface level variation when investigating program pairs, the tool can mitigate the occurrence of human errors.

Given that STRANGE can be embedded in other similarity detection tools to explain the code similarities, STRANGE should be at least as effective as the majority of those tools. Consequently, we performed a comparative study between STRANGE and a benchmark tool (JPlag) in terms of effectiveness. We also measured the impact of code similarity explanation by comparing STRANGE's side-by-side comparison with that of JPlag.

Following the introduction of STRANGE, three research questions arise:

- RQ1 How effective is STRANGE in reporting surface level variation?
- RQ2 Is STRANGE at least as effective as a benchmark tool, JPlag [6]?
- RQ3 Is code similarity explanation useful?

II. RELATED WORK

Research on automated code similarity or code reuse detection for plagiarism and collusion has been growing for more than four decades [14]. Ottenstein [15] developed one of the earliest techniques for this task, determining the similarity via four software metrics: the numbers of operators, operands, unique operators, and unique operands. Many similarity detection techniques have been introduced since then, relying on either the submitted source code or the creation process [16].

Based on the similarity measurement, techniques relying on the submitted source code can be further classified into three subcategories: attribute-counting-based, structure-based, and hybrid [16].

Attribute-counting-based techniques measure similarity via the frequencies of occurrence of source code characteristics (e.g., the number of assignment statements [17] or source code tokens [18]); their similarity measurement is often adapted from other domains such as information retrieval [19], [20] and data mining [21], [22].

Structure-based techniques were introduced for greater effectiveness, but with an offset in efficiency [23], as the similarity is based on code structure. JPlag [6] is a popular example of this approach, converting student programs to token strings and then pairwise comparing the strings with running Karp-Rabin greedy string tiling (RKRGST) [24].

Hybrid techniques combine both attribute-counting-based and structure-based techniques for enhanced effectiveness or efficiency. Greater effectiveness is commonly achieved either by displaying the results of both techniques at once [25] or by using the output of one technique as an input for another [26]. For efficiency, an attribute-counting-based technique can be used as a filter for a structure-based technique to reduce the number of program pairs requiring comprehensive comparison based on their structure [23].

The source code creation process has been considered as a way of reducing the number of false positives [27], as high similarity among submitted programs does not necessarily imply plagiarism and collusion [28]. Many additional hints for raising suspicion can be derived from the process, such as save timestamps [29], resubmission activity [30], classroom seating position [31], and social network [32].

Program pairs marked as suspicious by automated detection software should be investigated manually to confirm whether the similarities suggest plagiarism or collusion, or are likely to be coincidental [33]. For that reason, some similarity detection tools offer a side-by-side comparison module, which shows the content of a given program pair with similar fragments marked.

Many similarity detection tools have features similar to those seen in JPlag's side-by-side comparison module. Examples include Plaggie [34] and Deimos [35]. SSID [26] and Parikshak [36] differ only in their selection of similar fragments. SCSDS [37] does not display its output as HTML pages.

Other tools either have simpler modules or incorporate external programs. Tools proposed by Chen *et al.* [38] and Žáková *et al.* [39] mark all similar fragments with the same color. Rather than assigning colors, Sherlock [11] embeds two strings to mark the beginning and the end of each matched fragment. ES-Plag [23] replaces program code with its corresponding token string. MOSS [7], Marble [40], and IC-Plag [31] rely on a *diff* editor (e.g., TortoiseSVN[1]).

It can sometimes be challenging to understand the similarities shown in a side-by-side comparison module; two similar fragments can look different due to their surface level variation or, indeed, to disguises applied by students. The situation can be exacerbated if the instructors do not know what preprocessing has been applied, and they might conclude that the tool is behaving erroneously.

The difficulty in understanding the similarities can also complicate the reporting process, if the similarities need to be passed to higher officials for further investigation or to the suspected students as evidence of their dishonesty, and it becomes necessary to explain the similarities and summarize them in human language.

Another limitation is that the existing modules exclude comment similarity even though similar comments can sometimes be strong evidence of plagiarism or collusion [12]. Comments are typically excluded because they can be easily modified, obfuscating the apparent similarity. However, that issue can be easily addressed by measuring the comment similarity separately from the code similarity and considering only the comments that are similar.

Last but not least, according to a quick observation of the similarity detection tools listed in a recent literature review [16], a side-by-side comparison module is implemented by only about one fifth of them despite the benefits. This is possibly due to the difficulty of creating such a module from scratch. In response to that, Plago [13], a reusable module that can be executed via command line instructions was proposed. However, the module is not easy to use: along with the programs to be compared, the user is required to input the similar fragments and the corresponding token strings.

Natural language text is arguably useful in explaining complex concepts. An obvious example of this in programming is source code comments, which are generally easier to understand than the code itself. In light of this benefit, several studies automatically generate comments in natural language to easily explain some facts. Though many of those studies are related to program comprehension [41], there are a few studies covering different domains such as machine learning reports [42].

III. THE TOOL

This paper proposes STRANGE, a side-by-side comparison module that summarizes the similarities and explains them in natural language. It is also able to visualize surface level variation and capture similarities in comments as well as those in syntax. The module is expected to help instructors in understanding and reporting the similarities.

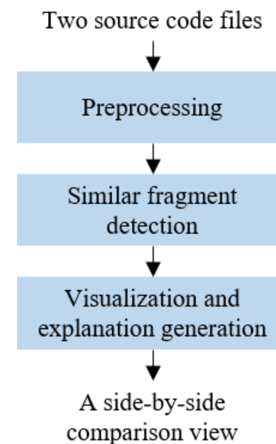

FIGURE 1. **STRANGE's method to generate a side-by-side comparison view from a program pair.**

STRANGE is designed to recognize code similarities either copied verbatim or with variation in comments, white space, identifier names, code fragment location, constants, and data types. Most of the recognized variation was inspired by the low-level disguises in the taxonomy of Faidhi and Robinson [43], which are expected to occur frequently among student programs.

The module accepts a pair of programs as input and returns the side-by-side comparison view via three stages (see Fig. 1). Preprocessing converts the programs to their intermediate representations (syntax and comment strings) prior to comparison. Similar fragment detection recognizes the similarities using RKRGST [24] for syntax and sequence alignment [44] for comments. Visualization and explanation generation summarizes the similarities in natural language and visualizes the applied preprocessing in an interactive HTML page.

STRANGE can act as a standalone similarity detection tool that accepts a set of student programs and returns the pairwise comparison results, but it can also take the output of JPlag [6] and display the same content in its own side-by-side comparison view. It also incorporates three additional features: sub-assessment grouping, code file merging, and template code removal.

All modes can be embedded in the development of other similarity detection tools so long as these tools are able to access the command prompt and run the module as a Java program. For the development of Java similarity detection tools, STRANGE can be attached as an additional library. Further, since the code is also provided on GitHub,[2] developers can adjust it according to their own needs.

STRANGE has a number of noteworthy features. First, it covers Python and Java, the two most common programming languages for introductory programming in the UK and Australasia [45]. Second, the generated explanation can be expressed in either English or Indonesian. The former is an international language used in many countries, while the

[1] https://tortoisesvn.net/

latter is the national language used by the people involved in our evaluation. Third, the applied preprocessing covers white space removal, comment removal, and token renaming. The first is automatically applied when the programs are converted to syntax and comment strings while the other two are selected due to their frequent use in code similarity detection [46]. They cover disguises that can be performed with a little programming knowledge, such as adding blank lines, removing comments, and renaming variables. Fourth, it uses a string matching algorithm, RKRGST [24], to check for similar code fragments, and sequence alignment [44] to check for similar comments. These are used in preference to other algorithms as they are widely-used [16] and the results are relatively easy to interpret. Fifth, while STRANGE might not recognize all similarities detected by a more advanced tool in which it may be embedded, its output can still be used for initial observation, as it recognizes common variations in code [43], especially those that are trivial but distracting. Sixth, compared to JPlag, STRANGE can be more effective on assessments that expect semantically similar solutions as it uses simpler preprocessing. Seventh, STRANGE can be downloaded via this link.[2] Alternatively, a request can be emailed to the corresponding author.

A. PREPROCESSING

In the preprocessing stage, from each input program we extract the syntax string and comment strings with the help of ANTLR [47]. The syntax string is the primary basis for raising suspicion, while comment similarities, if detected, are used only to reinforce the findings.

A syntax string is a sequence of syntax tokens in the order of their occurrence, with identifiers, numbers, strings, numeric data types, and string data types generalized to their corresponding token types. Numeric and string data types also cover non-primitive ones such as Java's *Float*, but only for Java, as Python has no explicit data type declaration. The token set used is based on ANTLR grammars.[3]

The use of syntax strings for raising suspicion negates any variation in comments and white space as only syntax tokens are considered. It is also resistant to changes in identifier names, constants, and data types, as the affected tokens are generalized.

Some sequences of non-keyword tokens can form effective keywords; for example, Java's 'System.out.print', which is a combination of identifiers and periods. To improve accuracy these tokens should not be generalized. To accommodate this, a list of user-defined keywords can be provided to STRANGE. This applies also to 'keywords' that are actually identifiers; for example, those that are introduced by the use of external libraries such as JES for Python.[4]

Comment strings comprise all comments found in the source code, with adjacent ones concatenated to deal with comment merge and split disguises. Five preprocessing steps are applied to each comment to deal with other variation. First, the comment is split into words based on non-alphanumeric characters to negate any changes in white space and symbols. Second, stop words (common words such as 'the' and 'it') are removed from the list, with the help of Apache Lucene [48], as they contribute little to the meaning. Third, the words are converted to lower case to negate capitalization changes. Fourth, each word is reduced to its stem using Porter stemmer in either English [49] or Indonesian [50], according to the selected explanation language, to deal with variations in word forms. Finally, the words are concatenated into a string to deal with word splitting or concatenation (e.g., 'preprocessing' and 'preprocessing').

B. SIMILAR FRAGMENT DETECTION

This phase recognizes similar fragments in the input programs based on their syntax and comment strings.

Similar syntax fragments are recognized with the help of RKRGST [24]. The algorithm is preferred to other similarity measurements since it prioritizes long matches, which are unlikely to be a result of coincidence. Further, it can occasionally deal with code fragment relocation in reasonable amount of time. RKRGST includes a parameter called minimum matching length: a lower value means that the algorithm will be more sensitive to code fragment relocation but in exchange, it may report more short fragments (which is not a problem if the similarity report is still to be investigated manually). The parameter is reconfigurable but set by default to two, which gives it the greatest sensitivity in detecting relocated code fragments.

Because some syntax tokens are generalized in preprocessing, the actual syntax strings of the similar fragments can be paired inaccurately. For example, assume that there are two similar Java variable declaration statements 'x = 5;' and 'y = 7;', and that they are written in a different order in each program. As they share the same generalized form, 'identifier = number;', 'x = 5;' can be paired with 'y = 7;' while 'y = 7;' is paired to 'x = 5;'. STRANGE handles this issue by comparing the actual strings of these fragments in addition to the generalized ones, and swapping them if necessary.

Similar comment fragments are detected by iterating through the comment strings in the first program and pairing each of them with a similar string from the counterpart if any is found. The paired string should have at least 50% similarity, measured with Equation 1, where A and B are the comment strings, $sim(A, B)$ is the number of similar characters between both strings recognized via sequence alignment [44], and $size(A)$ and $size(B)$ are the total numbers of characters in A and B respectively. For this process, sequence alignment [44] is preferred to RKRGST [24] as the former is more sensitive to order, making it easier to understand the similarity of the shared comments.

$$csim(A, B) = \frac{2 \times sim(A, B)}{size(A) + size(B)} \tag{1}$$

[2] https://github.com/oscarkarnalim/strange

[3] https://github.com/antlr/grammars-v4

[4] http://coweb.cc.gatech.edu/mediaComp-teach/26

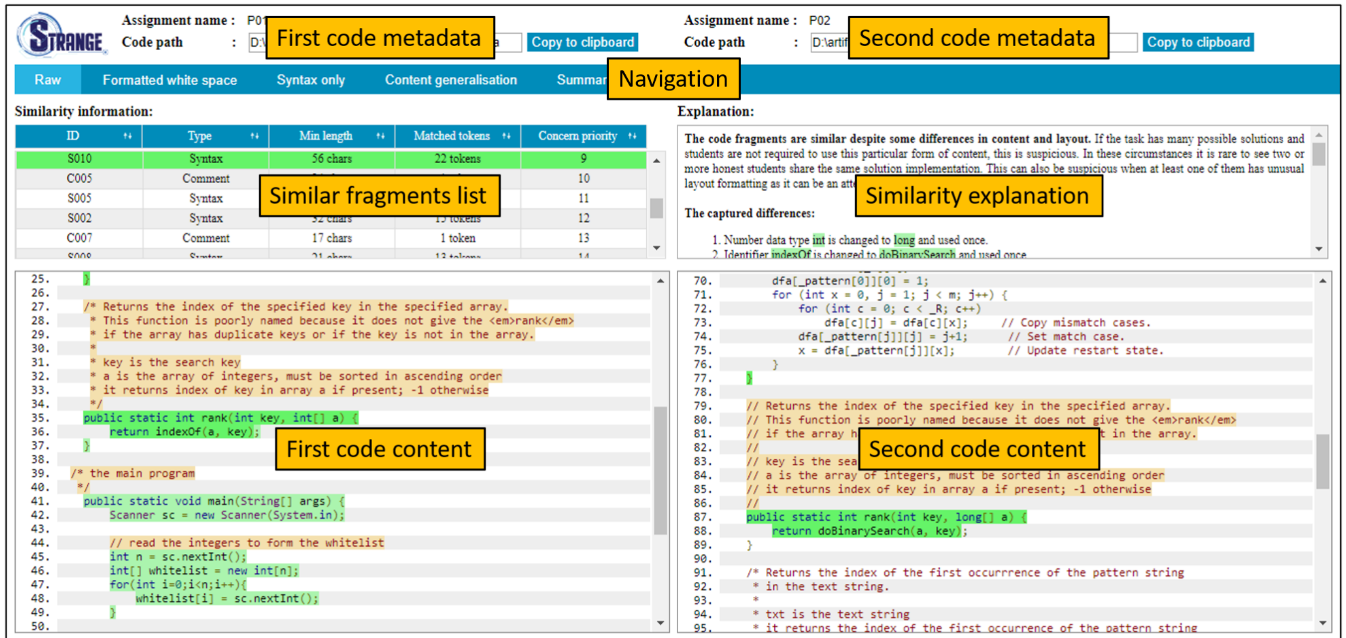

FIGURE 2. STRANGE's side-by-side comparison view comprising seven panels. Similar fragments are highlighted based on their corresponding type, either syntax or comment. The figure aims to provide a brief overview of the layout without expecting the text to be readable. The orange labels are added for clarity.

The 50% threshold is arbitrarily defined under a logical assumption that two entities can only be considered similar if they share at least half of their content. No empirical evaluation was performed for this since the threshold contributes only a little to STRANGE; it is used only to determine whether two comments are similar, not whether two programs share similar comments. In addition, it has no impact in selecting suspected programs since STRANGE's suspicion is based on syntax similarity. Although comment similarity is more striking, it acts only as an additional hint for instructors in case further evidence is needed. Moreover, it is not possible to measure the effectiveness unless STRANGE's suspicion mechanism is altered to rely on comment similarity–a change that is strongly discouraged since it implicitly suggests that comment similarity alone can be sufficient for raising suspicion, thus placing undue importance on similarity of comments.

We are aware that comment addition, comment removal, and paraphrasing can also be used to disguise the comments of copied program. However, reporting the first two can lead to confusion due to superfluous information, as comments can be unique to each student so long as no explicit instructions are given; and detection of paraphrasing is not worth the effort, as it takes a considerable amount of time, and again, comment similarity is not the primary basis for raising suspicion.

C. VISUALIZATION AND EXPLANATION GENERATION

In this phase, the similar fragments are suitably displayed in an HTML page. First, a template HTML page is selected according to the selected natural language. Then the similar fragments are converted to HTML code and embedded in the HTML page via a find-and-replace mechanism: each targeted field is denoted by a unique string and that string will be replaced by the corresponding HTML code. An example of this visualization can be seen in Fig. 2. It involves panels for navigation, similar fragment list, similarity explanation, left code metadata, left code content, right code metadata, and right code content.

The navigation panel enables the user to change the current layout (the various layouts will be explained later) or to access the summary, which comprises four metrics, each dealing with syntax tokens, comment tokens, and the combination of both. Average similarity [6] considers all token differences, and is calculated as in Equation 2, where *sim(A, B)* is the number of matches, *size(A)* is the number of tokens in the left program, and *size(B)* is the number of tokens in the right program. This metric is best used when all differences are equally important. Maximum similarity [6] is less strict than average similarity; identicality can be achieved when one program is a subset of the other. With the same terminology as the average, it is defined as in Equation 3, and is best used when unnecessary tokens can be added without changing the meaning of the program. Left-to-right and right-to-left similarities, calculated as in Equations 4 and 5 respectively, are useful to check the extent to which one program is a subset of the other.

$$avgsim(A, B) = \frac{2 \times sim(A, B)}{size(A) + size(B)} \quad (2)$$

The code fragments are similar despite some differences in content. If the task has many possible solutions and students are not required to use this particular form of content, this is suspicious. In these circumstances it is rare to see two or more honest students share the same solution implementation. This can also be suspicious when the layout is unusually formatted, as unusual formatting is rarely shared by coincidence.

The captured differences:

1. Number data type int is changed to long and used 2 times.
2. Identifier indexOf is changed to doBinarySearch and used once.
3. Identifier lo is changed to hi and used once.

Aligned contents (with differences highlighted):

Aligned position	Left form	Left position	Right form	Right position
1	int (number data type)	line 14 column 18	int (number data type)	line 39 column 18
2	indexOf (identifier)	line 14 column 22	doBinarySearch (identifier)	line 39 column 22
3	(	line 14 column 29	(	line 39 column 36
4	int (number data type)	line 14 column 30	long (number data type)	line 39 column 37
5	[	line 14 column 33	[	line 39 column 41
6	]	line 14 column 34	]	line 39 column 42

FIGURE 3. An example of STRANGE's explanation panel for a particular fragment. It describes the similarities, the possible causes, and the surface level variation (in both list and table views, displayed only partially for conciseness).

$$maxsim(A, B) = \frac{sim(A, B)}{min(size(A), size(B))} \quad (3)$$

$$lrsim(A, B) = \frac{sim(A, B)}{size(B)} \quad (4)$$

$$rlsim(A, B) = \frac{sim(A, B)}{size(A)} \quad (5)$$

The similar fragment list panel records any similar fragments (both syntax and comment) in a table. Each row is associated with one similar fragment and contains five pieces of information: fragment ID, fragment type (either syntax or comment), minimum character length, number of similar tokens, and concern priority. The last three are specifically tailored to indicate a fragment's strength in raising suspicion. Concern priority is proportional to the lengths of the matching strings, and is arbitrarily doubled for comments as similarity in comments is more striking than syntax similarity. Clicking on any row will focus the targeted fragment in both code content panels and display the summary in the similarity explanation panel.

The similarity explanation panel gives a natural-language description of the similarities of a matched pair of fragments and their surface level variation, starting with a summary sentence and then giving possible reasons for raising suspicion. Any surface level variations will then be listed, sorted by their frequency of occurrence, or by their position in the left program if the frequencies are the same. A table aligning the syntax tokens or the comment words, with their differences highlighted, is also provided for further investigation. An example can be seen in Fig. 3.

For syntax fragments, four explanation templates are provided: verbatim copy, similar but different white space, similar but different content, similar but different white space and content. An appropriate template is selected by comparing the syntax strings. If all of the original tokens are the same and located in the same relative position (both row and column), the verbatim copy template will be used. If all of the original tokens are the same but they are located in different relative positions, it is similar but different white space. If the original tokens are different but become similar once generalized, and they share the same relative position, it is similar but different content. If the original tokens are different but become similar once generalized, and they have different relative positions, it is similar but different white space and content.

If the fragment has different content prior to generalization, the surface level variation will be listed as points: each point has one sentence highlighting one variation and its pattern is "*CT OCL* is changed to *OCR* and used *F*". *CT* is the content type (e.g., identifier); *OCL* is the original token from the left code; *OCR* is the original token from the right code; and *F* depicts the occurrence frequency of that variation in the fragment.

A table aligning the syntax tokens contains three pieces of information for each entry, mapped to five columns: aligned position (i.e., the relative order of the syntax tokens), the original token from the left program and its absolute position, and the original token from the right program and its absolute position. For convenience, the varied tokens are highlighted.

Three templates are provided for comment fragments: verbatim copy, similar but different white space and/or non-alphanumeric characters, and partially similar. The template selection is based on layered comparison of the comment strings. If the comments are identical before preprocessing, it is verbatim copy. If the preprocessed comment words are identical, it is similar but different white

space and/or non-alphanumeric characters. If only some of the preprocessed comment words are identical, it is partially similar. For reporting purposes, the preprocessing of stop word removal and concatenation is not applied: stop words are worth reporting, and concatenation makes the comments hard for humans to read.

For comment fragments that are partially similar, the surface level variations will be listed as bullet points and an aligned table. The mechanism is similar to that for syntax fragments except that the syntax tokens are now replaced with comment words.

The left and right code metadata panels store the assignment name and file path of each program, while the programs are displayed in the code content panels with Google Prettify,[5] which highlights some tokens, adds line numbers, and distinguishes odd and even lines. Similar fragments are also highlighted using either green for syntax or orange for comments. The highlighting mechanism splits the fragment line-wise, encapsulates each line with a link tag, and assigns a unique ID derived from the fragment ID. Each link tag in the left code targets the corresponding code on the right and vice versa.

Some side-by-side comparison modules (e.g., that of JPlag [6]) highlight each fragment with a unique color to distinguish it from others. However, we avoid that as the mechanism can lead to inconspicuous colors, especially if many fragments are involved, and instructors may find it difficult to read the highlighted code fragments. Moreover, STRANGE has its own mechanism for distinguishing matched fragments: a single click on a fragment will highlight the fragment in both code content panels with a darker color, making the fragment more visible against the white and gray background of the containing panel.

Four layouts are provided for similarity investigation. The raw layout displays the programs as they are. The formatted white space layout shows the programs reformatted with either google-java-format[6] for Java or YAPF[7] for Python. This layout can be useful if the instructor prefers to ignore white space variation. The formatted layout is then updated to the syntax-only layout by removing all comments; this is useful if the investigation ignores comments. In the content generalization layout, tokens displayed in the syntax only layout are replaced with their generalized form if applicable. The layout can also be used to understand how the similarity is determined.

The similarity investigation layouts are generated in four steps. First, the whole method is executed to generate the raw layout. Second, the programs are reformatted and then passed to the whole method once again to generate the formatted white space layout. Third, the syntax only layout is generated by removing comments from the formatted white space layout. Finally, some tokens in the syntax-only layout are generalized to generate the content generalization layout.

Changing the layout will affect the information displayed in the similar fragments list and the similarity explanation panel. Comment fragments are displayed only in the raw and the formatted white space layouts. White space modification is discussed only in the raw layout. The syntax generalization layout only discusses the generalized forms of the programs.

D. OTHER MODES

STRANGE can update JPlag's output by taking its directory as the input and displaying the same data in its own side-by-side comparison view. The replacement is based on the program pairs listed in JPlag's 'index.html'. The new views will be assigned the same names as the replaced ones, making them automatically accessible from JPlag's 'index.html'.

STRANGE can also act as a standalone tool to detect similarities shared in a programming assessment, though only syntax and comment similarities are considered. It accepts an assessment directory of student programs, in which each program is written in Java or Python and is in its own subdirectory. As output, STRANGE returns a directory containing the side-by-side comparison views. An entry HTML page will also be generated to navigate through the resulting views.

For large classes, it can be time consuming to generate all possible comparison views when the instructor is likely to examine only a few of them. STRANGE's standalone mode can limit the comparison views to program pairs with undue similarity by setting the minimum similarity threshold for suspicion.

In addition to the aforementioned modes, STRANGE features three supporting modes: sub-assessment grouping, code file merging, and template code removal.

Sub-assessment grouping is applicable when the assessment consists of several sub-assessments that are to be graded or investigated separately. This mode generates a new directory for each sub-assessment, containing only the relevant programs. The programs are grouped according to the sub-assessment's filename pattern expressed as a Java regular expression. The generated directories can be used directly as the input of STRANGE's standalone mode to detect similarities.

Code file merging is applicable when the complete assessment task involves more than one code file. This often happens in courses that partly focus on code modularity and reusability, such as object-oriented programming. The mode generates a new assessment directory where all code files for each student submission are merged into one large code file, separated by comments describing the original file's relative path. Again, the generated directory can be used directly as the input of STRANGE's standalone mode to detect similarities.

Template code removal is applicable when code has been provided or suggested for students to use, and should therefore not be considered in raising suspicion. This mode excludes the given code from each code file found in each

[5] https://github.com/google/code-prettify
[6] https://github.com/google/google-java-format
[7] https://github.com/google/yapf

```
package p3;
/* ================================ */
// line below prior to template removal:
// public class Temp {
/* ================================ */
               Temp{
/* ======================================== */
// line below prior to template removal:
//  public static void main(String[] args) {
/* ======================================== */
                                         {
        System.out.println("Hello World");
    }
}
```

FIGURE 4. An example of template code removal on a Java code file with 'public class' and 'public static void main (String[] args)' as the template code.

student's assessment directory. For readability, the excluded code will be replaced with spaces of the same size, and a comment showing the original form will be added for each affected line. See Fig. 4 for an example. It is worth noting that upon removal, the code files are not guaranteed to be compilable as many template code fragments are integral to the syntax and/or semantics of the program. These modified code files can also be used as the input of other similarity detection tools, so long as the detection techniques can deal with incomplete parsing.

The template code is removed in two consecutive stages, inspired by Durić & Gašević [37]. First, the template code is converted to a syntax string with the help of ANTLR [47], with comments and white space removed. Then each student code file is converted to a syntax string in a similar way, and compared to the template string using RKRGST [24] with two as the minimum matching length (for the reason explained in subsection III-B); any similar tokens from the student code file will be replaced with same-sized spaces and each affected line will be indicated by a comment showing the original content prior to removal.

Being a greedy algorithm, RKRGST does not guarantee a completely accurate result. However, it is still preferred for reasons of time efficiency; more computation and contextual information are required for higher accuracy. Further, RKRGST's result is arguably acceptable as that algorithm is often used for detecting code similarities [16].

IV. EVALUATION

This section evaluates STRANGE on the basis of the three research questions presented in the introduction. RQ1 is addressed by evaluating STRANGE's effectiveness in reporting surface level variation. RQ2 is addressed by comparing its effectiveness with that of JPlag in dealing with similarity disguises and recognizing copied programs. RQ3 is addressed by highlighting the differences between STRANGE's and JPlag's side-by-side comparison views, and testing both views on tutors. Appropriate ethics approval was obtained for the evaluation involving tutors and student submissions.

A. ADDRESSING RQ1: THE EFFECTIVENESS OF STRANGE IN REPORTING SURFACE LEVEL VARIATION

If two code files share a high number of similar code fragments, this can potentially raise suspicion of plagiarism or collusion. Being aware of this, perpetrators sometimes obfuscate some of the similarities by applying disguises. However, as the perpetrators are often unable to write the code unaided, the applied disguises are commonly superficial and only affect the surface representation. This section summarizes the capabilities of STRANGE in detecting and reporting some common disguises.

To create a test suite of programs, we began with four Java programs from Sedgewick and Wayne's algorithms book [51], covering binary search, Knuth-Morris-Pratt string matching, linear regression, and shell sort. Each program was modified by applying ten disguises, two each from five categories: comment modification, white space modification, identifier renaming, code fragment relocation, and constant value and data type change. They were inspired by the low-level disguises in the taxonomy of Faidhi and Robinson [43], which are expected to be common among perpetrators who have little programming skill. The tool's effectiveness on more advanced disguises such as structural modification is reported in subsection IV-C; it is not discussed here since these disguises are not specifically reported by STRANGE. Several dummy methods were also added to make the cases more realistic, as perpetrators often mix the copied code with their own to reduce the chance of being caught.

The process was repeated with different disguise instances, resulting in eight modified programs, each of which was paired with the program on which it was based. Python versions of the original programs were then created and modified in a similar way, doubling the program pairs to 16. Disguises in constant values and data types are not applicable in Python due to its loose typing, so these were replaced with additional disguises from the other categories. The natural language used in the programs is limited to English, as changing the human language affects mainly the comments.

STRANGE's capability to detect the disguises was assessed by measuring the syntax similarity of each program pair with maximum normalization, a form of normalization that is resistant to the added dummy methods (see Equation 3). Its capability in reporting the disguises was also summarized, via manual observation of the resulting side-by-side comparison view.

STRANGE was confirmed to be highly effective, leading to 100% similarity for 10 of the 16 pairs, while the remaining six pairs averaged 99%. Comment and white space modification is handled by removing those prior to comparison. Identifier renaming, constant value change, and data type change are handled with token generalization, with the affected tokens being replaced with their corresponding token types. Code fragment relocation can be partially detected with the help of RKRGST. However, as RKRGST does not search for optimal matches and the similarity is defined based on

the generalized form, this only applies when the fragments are unusually long or have uncommon generalized token representations. Code fragment relocation is a reason why STRANGE failed to recognized some similarities in six of the 16 program pairs.

In terms of reporting the disguises, comment modification is reported by pairing the similar comments and explaining their differences at word level. White space modification is reported by stating its existence in the summary sentence. As white space changes are not marked in the similar fragments, the differences are not easy to see in the code content panels, especially if the modification is quite minor; for example, removing all spaces in a Python program statement (e.g., 'x = x + 3' to 'x = x + 3'). Identifier renaming, constant value change, and data type change are reported in the same way as comment modification except that the differences are described at token level. Code fragment relocation is not explicitly reported as such. Rather, each of the relocated fragments is just paired with the corresponding fragment located at a different position in the other program.

The difference in programming language does not affect the similarity degree and report except in the number of involved tokens, as Python tends to use fewer tokens than Java in expressing program statements. For example, Python does not require statements to be terminated by semicolons so long as they are on their own lines.

B. ADDRESSING RQ2: COMPARING STRANGE WITH JPlag IN DETECTING AND REPORTING CODE SIMILARITY DISGUISES

STRANGE can be used to update similarity reports of existing similarity detection tools. At least, it should have comparable effectiveness with a benchmark tool (JPlag [6]) in detecting and reporting code similarity disguises. This subsection compares both tools using two evaluation metrics: average degree of similarity, to assess whether the tool detects the disguises; and proportion of reported disguises, to assess whether the tool reports the disguises.

The evaluation data set was created by 21 tutors of introductory programming, assuming that their experience in marking student assignments equips them to act as plagiarists. All of them scored at least B+ when they completed the course, had a GPA of at least 3 out of 4, and had at least six months experience of tutoring. They were grouped based on their preferred programming language (11 chose Java and 10 chose Python), and were asked to disguise four programs from Sedgewick and Wayne's book [51] (i.e., binary search, descriptive statistics, Boyer-Moore string matching, and selection sort) in which the comments had been simplified and translated to Indonesian, the tutors' native language. For Python tutors, the programs had been manually converted from Java to Python.

Tutors were required to complete the disguising process in 45 minutes, making minimum changes to the program logic and recording all applied disguises. The process was expected to result in 84 disguised code files, 44 in Java and 40 in Python. However, only 83 code files were collected since one Java tutor submitted only three of the four programs. The disguised code files were then paired with their corresponding raw code files and became the data set.

This data set is preferred to the data sets from subsection IV-A as the applied disguises are not limited to those known by the authors.

On the data set, STRANGE results in 98% average similarity degree, which is comparable to that of JPlag. A paired two-tailed t-test with 95% confidence rate found no significant difference between the two tools. It was not practical to check whether both tools found the same similarities in the same program pairs, but the high average degree of similarity in both tools suggests that it is very likely.

The proportion of reported disguises is based on how many of the tutor's recorded disguises were noted and explained by the tool. Since JPlag only highlights the similarities without describing the differences (i.e., applied disguises), its proportion of reported disguises is automatically assigned to 0%. STRANGE was found to report 203 of 373 applied disguises (54%), which means that it is reasonably effective in reporting the disguises. The proportion increases to 70% if disguises focusing on addition or removal are excluded; few similarity detection tools report such differences, focusing instead on similarities.

In this tutor-generated data set, identifier names are the most common elements to be disguised, followed by comments and white space. Modification of those components is easy as it requires limited programming knowledge. This is aligned to STRANGE's preprocessing, which generalizes identifiers and removes comments and white space.

STRANGE failed to report 32 disguises involving code fragment relocation as the relocated fragments are short and RKRGST is not designed to find optimal matched fragments. Other unreported disguises are:

- Major comment change (3 occurrences). Paraphrased or summarized comments are undetected as comment semantics are ignored.
- Importing mechanism change (6 occurrences). These changes were applied mainly by Java tutors, who changed the coverage of the importing mechanism (e.g., replacing 'import java.util.Scanner' with 'import java.*'). STRANGE recognizes these statements as standard syntax, resulting in some mismatches.
- Function structure change (4 occurrences). This was applied by changing the order of function parameters, changing the return value type of a function, or replacing a function call with the code of its body. While these are not addressed by STRANGE, they do require a level of programming knowledge that might be rare among students who choose to copy program code.
- Program statement replacement (40 occurrences). This disguise was commonly applied to arithmetic or logical expressions; some such disguises are obvious as the replacing expressions are inefficient (e.g., replacing 'i = i + 1' with 'i = i − 2 + 1'). Other common disguises

were to introduce or remove a temporary variable, or to change the order of the branches of an *if* statement. Language-specific features such as dynamic parameters in Python's *for* loop syntax and inbuilt mathematical functions were also often used as replacements. Again, as STRANGE does not specifically address these, they are not detected or reported.

C. ADDRESSING RQ2: COMPARING STRANGE WITH JPlag IN RECOGNIZING COPIED PROGRAMS

STRANGE can be used to investigate suspicious program pairs suggested by existing similarity detection tools. Hence, it should be at least as effective in recognizing copied programs as a benchmark tool (JPlag [6]). This subsection conducts the comparison using f-score, the harmonic mean of precision and recall, calculated as $(2 \times precision \times recall)/(precision + recall)$. Precision is the proportion of suspected program pairs that are copied to all suspected pairs. Recall is the proportion of suspected program pairs that are copied to all copied pairs.

To perform this comparison we used the IR-Plag data set [52]. This data set is based on seven initial Java programs that cover introductory programming materials (output, input, branching, looping, function, array, and matrix). These seven programs were disguised by nine introductory programming tutors to 355 variants through the application of specific types of similarity disguise. In addition, another set of tutors wrote 15 independent – not copied – versions of each of the seven programs, so the IR-Plag data set includes both 355 disguised and 105 independent programs to carry out the same tasks. The data sets from previous subsections cannot be used here, as the metrics that we are using require independent programs.

The disguises were drawn from the highest six levels of Faidhi and Robinson [43], in which lower levels describe simpler disguises and higher levels indicate more complex disguises. Level 1, the lowest level after verbatim copy (level 0), involves modification of comments and white space; level 2 involves identifier modification; level 3, component declaration relocation; and level 4, function structure change. The last two levels are program statement replacement or structural change (level 5) and logic change (level 6). Level 0 (verbatim copy) is not included since it is trivial to detect copied programs with no disguises.

For each level, the original programs were paired with their corresponding disguised and independent programs. The effectiveness of STRANGE and JPlag was then measured, with suspected program pairs deemed to be those with similarity degree greater than or equal to both 75% and the average similarity. The 75% threshold ensures that the suspected pairs share high similarity, even when the average similarity is low. Our manual observation of 92 Java and 24 introductory programming assessments shows that program pairs whose similarity is at least 75% are indeed suspicious. The statistical significance was measured with a two-tailed paired t-test with 95% confidence rate.

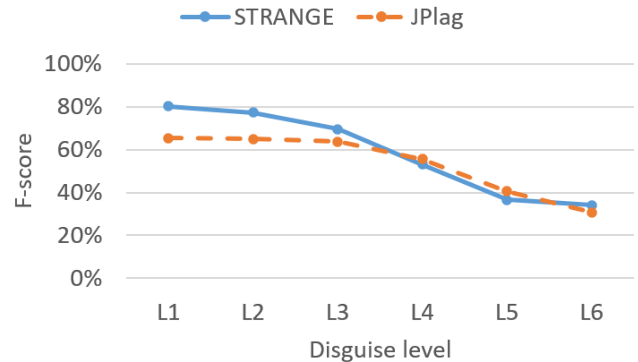

FIGURE 5. STRANGE vs JPlag on IR-Plag data set: f-score.

Fig. 5 shows that STRANGE generates higher f-scores than JPlag on the first two levels; the differences are statistically significant (p-values ≤ 0.01). JPlag's program statement generalization (e.g., considering 'i++' and 'i = i + 1' as the same) complicates the detection process at these lower levels as the degree of coincidental similarity is increased; many copied pairs can have degrees of similarity comparable to the independent pairs. That same generalization is expected to be beneficial with program statement replacement (level 5), but with this data set there is no significant improvement. On the remaining levels, the performance of the tools is comparable and there are no significant differences since the disguises are handled in similar ways.

D. ADDRESSING RQ3: DIFFERENCES BETWEEN THE SIDE-BY-SIDE COMPARISON VIEWS OF STRANGE AND JPlag

This subsection highlights differences between STRANGE and a benchmark tool, JPlag [6], in terms of their side-by-side comparison views. Both views were qualitatively compared based on 164 suspicious program pairs (92 Java and 72 Python).

The Java program pairs were selected from student submissions to 92 distinct introductory programming assessment tasks (with three to 22 programs each), by pairing the student programs for each task, and choosing one for which JPlag's average similarity is closest to but no lower than the suspicion threshold used in Subsection IV-C (i.e., the greater of 75% and the average of all pairwise similarities). Manual observation confirms that for this evaluation, program pairs whose similarity degree is at least as high as the suspicion threshold are indeed suspicious. The Python program pairs were selected in the same manner, except that three pairs were selected for each task since only 24 tasks are available (with ten to 19 programs each). The first two pairs are those for which JPlag's average similarity is closest to but no lower than the suspicion threshold, and the third is the pair with the highest similarity. These pairs are preferable to those in other data sets as the programs are written by students rather than tutors or the authors, and should therefore provide more realistic cases for this qualitative assessment.

Based on the side-by-side comparison views, JPlag seems to deal with more code disguises than STRANGE. For example, it nullifies the impact of changing 'i++' to 'i = i + 1' or changing 'String [] args' to 'String args []' in Java. Nevertheless, if the instructor is not aware of the mechanisms, these can be perceived as errors. STRANGE's detection mechanism is less advanced, but ensures that each marked similarity is explained. To further assist the instructor in understanding the similarities, STRANGE provides natural language explanations and several layouts showing the code in various representations.

Not all of JPlag's nullifying mechanisms are guaranteed to be beneficial. The exclusion of comments, for example, ignores the fact that similar comments can be evidence of copying, as comments are commonly unique to each student if they are neither automatically generated nor explicitly instructed. STRANGE deals with this by separating the comments from the code and measuring their similarity separately. In the data set, STRANGE found 24 similar comments containing unique information such as computer ID, potentially used for raising suspicion. Seventeen of them were found in Java program pairs and seven in Python pairs.

Although JPlag ignores comments, they are still displayed in code content panels and occasionally highlighted if they occur within similar code fragments. This can be confusing for instructors since the comments can be perceived to play some part in determining similarity. STRANGE separates comments from program code and therefore ensures that the former will not be unwittingly highlighted with the latter.

JPlag also seems to consider expressions and constants as the same, possibly under an assumption that the given programs share the same semantics. For assessments with many possible solutions, this can lead to incorrect matches as not all expressions can be replaced with constants and vice-versa. Although STRANGE generalizes constants, it still considers expressions as they are.

JPlag appears to generalize some program statements to single tokens (e.g., Java's 'System.out.println') to reduce false positive matches. Some of those statements can then go undetected, as the number of matched tokens can be lower than RKRGST's minimum matching length. STRANGE does not generalize in this manner, so it is able to capture those statements, but at a slight expense in efficiency.

To differentiate similar code fragments, JPlag displays each of them in a unique color. According to our observation, the generated colors can be lighter ones, making them hard to detect against the white background of the code content panel. Further, if many fragments are involved, some colors look alike. STRANGE uses only two colors for highlighting: green for syntax and orange for comment. To avoid confusion, each code fragment can be highlighted with a click, and will appear in a darker version of the same color, adding more contrast against the white and gray background.

JPlag's similar fragments are highlighted on top of the given code, which is displayed as plain text. This might lead to more investigation effort as reading code as plain text can be challenging. It can be worse if the code fragments are different at surface level (e.g., different comments, white space, or identifiers) as the instructor needs to deduce why the tool considers the code fragments to be similar. STRANGE shows the code as it might be displayed in an IDE; some tokens are highlighted and the lines are numbered. Several layouts are provided to help exclude surface differences and thus to draw attention to the similarities. The formatted white space layout reformats the white space according to particular rules, nullifying modification in that aspect. The syntax only layout excludes comments from consideration. The content generalization layout shows the generalized version of each syntax token, showing why the code fragments are considered similar despite having different surface tokens.

TABLE 1. Average usability measures of JPlag's and STRANGE's views.

Metric	JPlag	STRANGE	Variances	P-value
Time in minutes	42.5	39.3	Equal	0.15
Recorded instances	20.4	34.5	Unequal	≤ 0.001

E. ADDRESSING RQ3: THE USEFULNESS OF CODE SIMILARITY EXPLANATION

Code similarity explanation is expected to be useful if STRANGE's side-by-side comparison view, which alone provides such an explanation, is more helpful than JPlag's during the investigation process carried out by the instructors. To examine this expectation, the 21 tutors mentioned in subsection IV-B participated in this phase of the study. Each tutor was shown eight program pairs written in their preferred programming language and asked to list any similar fragments and the corresponding disguises. Four pairs would be observed with JPlag's view in up to 45 minutes, and the other four with STRANGE's view in up to 45 minutes. We hypothesized that if one view was more helpful, the task using that view would be completed in less time, and/or the tutors would record more similar fragments and disguises. The statistical significance was measured with a two-tailed t-test with 95% confidence rate; not the paired test, as the tutors' responses were collected anonymously.

The program pairs were taken from the data set used in subsection IV-A, covering four topics from Sedgewick and Wayne's book [51]: binary search, Knuth-Morris-Pratt string matching, linear regression, and shell sort. For this particular data set, each topic has two copied programs with comparable disguises, permitting one copy to be examined with JPlag and the other with STRANGE, without favoring either tool.

Table 1 shows that the use of STRANGE's view leads to more recorded fragments and disguises than JPlag's. STRANGE summarizes the similarities and the disguises in natural language, making it easier for tutors to discern and report them. There was no significant difference in completion time, as the tutors found more to report with STRANGE.

After completing both tasks the tutors were asked to complete a survey summarizing their perspective of code similarity explanation. The survey consisted of 14 statements, each

TABLE 2. Survey questions about code similarity explanation and the responses.

ID	Question	Agree	Do not know	Disagree
SQ01	Natural language explanation helps me to understand what kinds of similarity have been found.	21	0	0
SQ02	Natural language explanation helps me to identify white space modification (e.g., adding extra blank lines between program statements).	21	0	0
SQ03	Natural language explanation helps me to identify comment modification (e.g., replacing a comment word from 'changing' to 'change').	19	2	0
SQ04	Natural language explanation helps me to identify identifier renaming (e.g., changing a variable name from 'height' to 'ht').	21	0	0
SQ05	Natural language explanation helps me to identify variable type change (e.g., replacing 'int' with 'float'). This is not applicable to Python tutors.	11	0	0
SQ06	Natural language explanation helps me to identify changes in constant values (e.g., assigning a variable to 1 instead of 0).	18	3	0
SQ07	Natural language explanation helps me to identify rearrangement of program statements (e.g., moving some statements to a different location in the program).	19	2	0
SQ08	Having the white space formatted helps me to not consider white space in my manual observation.	20	0	1
SQ09	Having the comments removed helps me to not consider comments in my manual observation.	17	0	4
SQ10	Having some identifier names generalized to the same token helps me to not consider identifier name variation in my manual observation.	18	1	2
SQ11	Having some variable types generalized to the same token helps me to not consider variable type variation in my manual observation. This is not applicable to Python tutors.	11	0	0
SQ12	Having some constant values generalized to the same token helps me to not consider constant value variation in my manual observation.	18	2	1
SQ13	Having different types of similarity displayed in different colors helps me to understand what kinds of similarity have been found.	20	0	1
SQ14	Ordering similar fragments according to their concern priority enables me to focus first on obviously similar fragments.	18	1	0

of which can be answered with 'agree', 'do not know', or 'disagree'. For each question, respondents could also provide a short explanation supporting the choice. The survey questions and the responses can be seen in Table 2. SQ05 and SQ11 are not applicable to Python tutors as the language has no variable type declaration. SQ14 had no response from two tutors due to human error.

As seen in Table 2, five survey questions (SQ01, SQ02, SQ04, SQ05, and SQ11) were answered unanimously with 'agree', suggesting that the tutors find those features crucial in the investigation process.

The remaining questions still drew mainly positive responses but with a few 'do not know' or 'disagree' responses. To help us understand the causes, we noted the explanations given by the tutors. For SQ06 and SQ12, a few respondents were not aware of a changed constant value as that change altered only one source-code token. For questions SQ08 and SQ09, a few respondents did not find the formatted white space layout or syntax only layout to be helpful, believing that the raw layout is sufficient for the investigation process. Three non-positive responses to SQ10 were from respondents who were unaware of the importance of the token generalization. For SQ13, one respondent felt that it could be confusing having only two colors for highlighting matched fragments. The respondents who answered 'do not know' for SQ03 and SQ07 did not explain their reasons for that choice.

V. CONCLUSION AND FUTURE WORK

This paper presents STRANGE, a side-by-side comparison module for code similarity investigation. It explains and summarizes similarities, in both syntax and comments, using natural language. STRANGE can be used in the development of other similarity detection tools. It can also be used to reinterpret the output from JPlag, and can act as a standalone tool for measuring source code similarity.

STRANGE effectively reports many surface level variations (RQ1), including comment modification, white space modification, identifier renaming, constant value change, and data type change. Code fragment relocation is handled only when the fragments are fairly long and have uncommon generalized forms.

STRANGE is more effective than JPlag [6], a benchmark tool for code similarity detection, in reporting similarity disguises and detecting copied programs with surface level variation (RQ2). The tools are comparable in dealing with advanced disguises.

Code similarity explanation, as featured in STRANGE's side-by-side comparison view, is useful for the investigation process (RQ3). It helps instructors to recognize more similarities and the disguises applied to them. The instructors generally agree that such explanation is helpful in the investigation process.

Our evaluation is subject to four limitations that suggest future work. First, all data sets are at the level of introductory programming since that is probably the best course in which to start educating students about academic integrity. It is important to validate the findings on different data sets such as the SOCO data set [19], the OCD data set [53], and the BigCloneBench data set [54]. Second, not all possible disguises are discussed as we focus on the common ones according to Faidhi and Robinson [43]. Additional findings might be obtained by including other disguises, even those that are cross-language [20]. Third, the evaluation was performed with JPlag version 2.11.8. Some findings might not

be applicable on other versions. Fourth, the JPlag analysis was based only on the user perspective and the corresponding paper [6]. As the tool is open source and has been updated several times since the publication of that paper, an analysis of the tool's code might provide a different perspective.

In addition to addressing the aforementioned limitations, we expect to extend STRANGE to cover more programming languages and more natural languages, and to deal with more advanced disguises. It will also be interesting to test the tool's scalability on larger data sets as that factor might be important for programming courses with hundreds of students.

ACKNOWLEDGMENT

The authors would like to thank Australia Awards Scholarship for financially supporting the first author and William Chivers from University of Newcastle, Australia, for his overall contribution.

REFERENCES

[1] Simon, B. Cook, J. Sheard, A. Carbone, and C. Johnson, "Academic integrity: Differences between computing assessments and essays," in *Proc. 13th Koli Calling Int. Conf. Comput. Educ. Res. (Koli Calling)*, 2013, pp. 23–32.

[2] R. Fraser, "Collaboration, collusion and plagiarism in computer science coursework," *Informat. Educ.*, vol. 13, no. 2, pp. 179–195, Oct. 2014.

[3] G. Cosma and M. Joy, "Towards a definition of source-code plagiarism," *IEEE Trans. Educ.*, vol. 51, no. 2, pp. 195–200, May 2008.

[4] J. Sheard, Simon, M. Butler, K. Falkner, M. Morgan, and A. Weerasinghe, "Strategies for maintaining academic integrity in first-year computing courses," in *Proc. ACM Conf. Innov. Technol. Comput. Sci. Educ.*, Jun. 2017, pp. 244–249.

[5] Simon, J. Sheard, M. Morgan, A. Petersen, A. Settle, and J. Sinclair, "Informing students about academic integrity in programming," in *Proc. 20th Australas. Comput. Educ. Conf. (ACE)*, 2018, pp. 113–122.

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

[7] S. Schleimer, D. S. Wilkerson, and A. Aiken, "Winnowing: Local algorithms for document fingerprinting," in *Proc. ACM SIGMOD Int. Conf. Manage. Data (SIGMOD)*, 2003, pp. 76–85.

[8] M. J. Mišić, J. U. Z. Protić, and M. V. Tomašević, "Improving source code plagiarism detection: Lessons learned," in *Proc. 25th Telecommun. Forum (TELFOR)*, Nov. 2017, pp. 1–8.

[9] F. Culwin and T. Lancaster, "Visualising intra-corpal plagiarism," in *Proc. 5th Int. Conf. Inf. Visualisation*, 2001, pp. 289–296.

[10] M. Joy, N. Griffiths, and R. Boyatt, "The boss online submission and assessment system," *J. Educ. Resour. Comput.*, vol. 5, no. 3, p. 2, Sep. 2005.

[11] M. Joy and M. Luck, "Plagiarism in programming assignments," *IEEE Trans. Educ.*, vol. 42, no. 2, pp. 129–133, May 1999.

[12] U. Inoue and S. Wada, "Detecting plagiarisms in elementary programming courses," in *Proc. 9th Int. Conf. Fuzzy Syst. Knowl. Discovery*, May 2012, pp. 2308–2312.

[13] R. Franclinton and O. Karnalim, "A language-independent library for observing source code plagiarism," *J. Inf. Syst. Eng. Business Intell.*, vol. 5, no. 2, pp. 110–119, 2019.

[14] M. Novak, M. Joy, and D. Kermek, "Source-code similarity detection and detection tools used in academia: A systematic review," *ACM Trans. Comput. Educ.*, vol. 19, no. 3, pp. 27:1–27:37, 2019.

[15] K. J. Ottenstein, "An algorithmic approach to the detection and prevention of plagiarism," *ACM SIGCSE Bull.*, vol. 8, no. 4, pp. 30–41, Dec. 1976.

[16] O. Karnalim, Simon, and W. Chivers, "Similarity detection techniques for academic source code plagiarism and collusion: A review," in *Proc. IEEE Int. Conf. Eng., Technol. Educ. (TALE)*, Dec. 2019, pp. 1–8.

[17] Z. A. Al-Khanjari, J. A. Fiaidhi, R. A. Al-Hinai, and N. S. Kutti, "PlagDetect: A java programming plagiarism detection tool," *ACM Inroads*, vol. 1, no. 4, pp. 66–71, Dec. 2010.

[18] F. B. Allyson, M. L. Danilo, S. M. José, and B. C. Giovanni, "Sherlock N-overlap: Invasive normalization and overlap coefficient for the similarity analysis between source code," *IEEE Trans. Comput.*, vol. 68, no. 5, pp. 740–751, May 2019.

[19] E. Flores, A. Barrón-Cedeño, L. Moreno, and P. Rosso, "Uncovering source code reuse in large-scale academic environments," *Comput. Appl. Eng. Educ.*, vol. 23, no. 3, pp. 383–390, May 2015.

[20] E. Flores, A. Barrón-Cedeño, L. Moreno, and P. Rosso, "Cross-language source code re-use detection using latent semantic analysis," *J. Universal Comput. Sci.*, vol. 21, no. 13, pp. 1708–1725, 2015.

[21] T. Ohmann and I. Rahal, "Efficient clustering-based source code plagiarism detection using PIY," *Knowl. Inf. Syst.*, vol. 43, no. 2, pp. 445–472, May 2015.

[22] G. Acampora and G. Cosma, "A fuzzy-based approach to programming language independent source-code plagiarism detection," in *Proc. IEEE Int. Conf. Fuzzy Syst. (FUZZ-IEEE)*, Aug. 2015, pp. 1–8.

[23] L. Sulistiani and O. Karnalim, "ES-Plag: Efficient and sensitive source code plagiarism detection tool for academic environment," *Comput. Appl. Eng. Educ.*, vol. 27, no. 1, pp. 166–182, Jan. 2019.

[24] M. J. Wise, "YAP3: Improved detection of similarities in computer program and other texts," in *Proc. 27th SIGCSE Tech. Symp. Comput. Sci. Educ.*, 1996, pp. 130–134.

[25] M. E. B. Menai and N. S. Al-Hassoun, "Similarity detection in java programming assignments," in *Proc. 5th Int. Conf. Comput. Sci. Educ.*, Aug. 2010, pp. 356–361.

[26] J. Y. H. Poon, K. Sugiyama, Y. F. Tan, and M.-Y. Kan, "Instructor-centric source code plagiarism detection and plagiarism corpus," in *Proc. 17th ACM Annu. Conf. Innov. Technol. Comput. Sci. Educ. (ITiCSE)*, 2012, pp. 122–127.

[27] V. Ljubovic and E. Pajic, "Plagiarism detection in computer programming using feature extraction from ultra-fine-grained repositories," *IEEE Access*, vol. 8, pp. 96505–96514, 2020.

[28] S. Mann and Z. Frew, "Similarity and originality in code: Plagiarism and normal variation in student assignments," in *Proc. 8th Australas. Conf. Comput. Educ.*, 2006, pp. 143–150.

[29] P. Vamplew and J. Dermoudy, "An anti-plagiarism editor for software development courses," in *Proc. 7th Australas. Conf. Comput. Educ.*, 2010, pp. 83–90.

[30] N. Tahaei and D. C. Noelle, "Automated plagiarism detection for computer programming exercises based on patterns of resubmission," in *Proc. ACM Conf. Int. Comput. Educ. Res.*, Aug. 2018, pp. 178–186.

[31] A. Budiman and O. Karnalim, "Automated hints generation for investigating source code plagiarism and identifying the culprits on in-class individual programming assessment," *Computers*, vol. 8, no. 1, p. 11, Feb. 2019.

[32] A. Zrnec and D. Lavbič, "Social network aided plagiarism detection," *Brit. J. Educ. Technol.*, vol. 48, no. 1, pp. 113–128, Jan. 2017.

[33] D. Weber-Wulff, "Plagiarism detection software: Promises, pitfalls, and practices," in *Handbook of Academic Integrity*. Singapore: Springer, 2016, pp. 625–638.

[34] A. Ahtiainen, S. Surakka, and M. Rahikainen, "Plaggie: GNU-licensed source code plagiarism detection engine for java exercises," in *Proc. 6th Baltic Sea Conf. Comput. Educ. Res. Koli Calling (Baltic Sea)*, 2006, pp. 141–142.

[35] C. Kustanto and I. Liem, "Automatic source code plagiarism detection," in *Proc. 10th ACIS Int. Conf. Softw. Eng., Artif. Intell., Netw. Parallel/Distrib. Comput.*, May 2009, pp. 481–486.

[36] S. Sharma, C. S. Sharma, and V. Tyagi, "Plagiarism detection tool, 'Parikshak,'" in *Proc. Int. Conf. Commun., Inf. Comput.*, 2015, pp. 1–7.

[37] Z. Đurić and D. Gašević, "A source code similarity system for plagiarism detection," *Comput. J.*, vol. 56, no. 1, pp. 70–86, 2013.

[38] X. Chen, B. Francia, M. Li, B. McKinnon, and A. Seker, "Shared information and program plagiarism detection," *IEEE Trans. Inf. Theory*, vol. 50, no. 7, pp. 1545–1551, Jul. 2004.

[39] K. Žáková, J. Pištej, and P. Bisták, "Online tool for student's source code plagiarism detection," in *Proc. IEEE 11th Int. Conf. Emerg. eLearning Technol. Appl. (ICETA)*, Oct. 2013, pp. 415–419.

[40] J. Hage, P. Rademaker, and N. van Vugt, "Plagiarism detection for Java: A tool comparison," in *Proc. Comput. Sci. Educ. Res. Conf.*, 2011, pp. 33–46.

[41] X. Song, H. Sun, X. Wang, and J. Yan, "A survey of automatic generation of source code comments: Algorithms and techniques," *IEEE Access*, vol. 7, pp. 111411–111428, 2019.

[42] J. M. Alonso and A. Bugarin, "ExpliClas: Automatic generation of explanations in natural language for Weka classifiers," in *Proc. IEEE Int. Conf. Fuzzy Syst. (FUZZ-IEEE)*, Jun. 2019, pp. 1–6.
[43] J. A. W. Faidhi and S. K. Robinson, "An empirical approach for detecting program similarity and plagiarism within a university programming environment," *Comput. Educ.*, vol. 11, no. 1, pp. 11–19, Jan. 1987.
[44] X. Huang, R. C. Hardison, and W. Miller, "A space-efficient algorithm for local similarities," *Bioinformatics*, vol. 6, no. 4, pp. 373–381, 1990.
[45] Simon, R. Mason, T. Crick, J. H. Davenport, and E. Murphy, "Language choice in introductory programming courses at australasian and UK universities," in *Proc. 49th ACM Tech. Symp. Comput. Sci. Educ.*, Feb. 2018, pp. 852–857.
[46] O. Karnalim, Simon, and W. Chivers, "Preprocessing for source code similarity detection in introductory programming," in *Proc. 20th Koli Calling Int. Conf. Comput. Educ. Res. (Koli Calling)*, Nov. 2020, pp. 1–10.
[47] T. Parr, *The Definitive ANTLR 4 Reference*. Raleigh, NC, USA: Pragmatic, 2013.
[48] M. McCandless, E. Hatcher, and O. Gospodnetic, *Lucene in Action: Covers Apache Lucene 3.0*, 2nd ed. Shelter Island, NY, USA: Manning, 2010.
[49] M. F. Porter, "An algorithm for suffix stripping," in *Program: Electronic Library and Information Systems*. Bingley, U.K.: Emerald Group Publishing, 2006.
[50] F. Z. Tala, "A study of stemming effects on information retrieval in Bahasa Indonesia," M.S. thesis, Inst. Logic, Lang. Comput., Universiteit van Amsterdam, Amsterdam, The Netherlands, 2003.
[51] R. Sedgewick and K. Wayne, *Algorithms*, 4th ed. London, U.K.: Pearson, 2011.
[52] O. Karnalim, S. Budi, H. Toba, and M. Joy, "Source code plagiarism detection in academia with information retrieval: Dataset and the observation," *Informat. Educ.*, vol. 18, no. 2, pp. 321–344, Oct. 2019.
[53] C. Ragkhitwetsagul, J. Krinke, and D. Clark, "A comparison of code similarity analysers," *Empirical Softw. Eng.*, vol. 23, no. 4, pp. 2464–2519, Aug. 2018.
[54] J. Svajlenko, J. F. Islam, I. Keivanloo, C. K. Roy, and M. M. Mia, "Towards a big data curated benchmark of inter-project code clones," in *Proc. IEEE Int. Conf. Softw. Maintenance Evol.*, Sep. 2014, pp. 476–480.

OSCAR KARNALIM (Graduate Student Member, IEEE) is currently pursuing the Ph.D. degree with the University of Newcastle, Australia. He is currently an Assistant Professor with Maranatha Christian University, Indonesia. His primary research interest includes code similarity detection. He has published 24 articles on that topic, many of which are indexed in IEEE Xplore, ACM Digital Library, and ScienceDirect. He also serves as a reviewer for many journals and conferences, including IEEE Access, *CAE* (Wiley), *Heliyon*, IEEE SMC, IEEE EDUCON, and IEEE TALE.

SIMON is currently a Senior Lecturer with the University of Newcastle, Australia. His main research interest includes academic integrity in programming. He has published more than 20 papers in that topic, mostly in computing education conferences, such as ACM SIGCSE, ITiCSE, and ICER. He is a reviewer for many computing education journals and conferences.