Journal of
Information Systems Engineering
and Business Intelligence

Vol.5, No.2, October 2
Available online at: http://e-journal.unair.ac.id/index.php/JISEBI

A Language-Independent Library for Observing Source Code Plagiarism

Ricardo Franclinton[1], Oscar Karnalim[2] *

[1)2)] Maranatha Christian University, Indonesia

M.P.H, Jl. Surya Sumantri No.65, Sukawarna, Kec. Sukajadi, Kota Bandung

[1)]carlclifinton@gmail.com, [2)]oscar.karnalim@it.maranatha.edu

Article history:

Received 20 April 2019
Revised 2 June 2019
Accepted 19 June 2019
Available online 28 October 2019

Keywords:

Language independency, Plagiarism detection, Reusable library, Source code, Tool development

Abstract

Background: Most source code plagiarism detection tools are not modifiable. Consequently, when a modification is required to be applied, a new detection tool should be created along with it. This could be a problem as creating the tool from scratch is time-inefficient while most of the features are similar across source code plagiarism detection tools.
Objective: To alleviate researchers' effort, this paper proposes a library for observing two plagiarism-suspected codes (a feature which is similar across most source code plagiarism detection tools).
Methods: Unique to this library, it is not constrained by the selected programming language for development. It is executed from command line, which is supported by most programming languages.
Results: According to our evaluation, the library is integrable and functional. Moreover, the library can enhance teaching assistants' accuracy and reduce the tasks' completion time.
Conclusion: The library can be beneficial for the development of source code plagiarism detection tools since it is integrable, functional, and helpful for teaching assistants.

I. INTRODUCTION

Source code plagiarism happens when a source code is reused (with or without modification) and that reuse is not acknowledged properly [1]. This could lead to several problems, especially those which are related to authorship and creative works. Hence, several automated plagiarism detection tools dedicated to code domain (e.g., JPlag [2], ES-Plag [3], and IC-Plag [4]) have been proposed. Using those tools, such an illegal behaviour can be detected with limited human effort.

Most of the detection tools are not modifiable, even though some modification attempts can enhance the tools' effectiveness and efficiency. As a result, when a modification needs to be applied, a new detection tool should be proposed along with it. This could be a problem for several researchers since they may not have sufficient time to recreate the tool. The issue becomes worse as some of the components are the same for most detection tools. For instance, a component for observing two plagiarism-suspected codes has similar appearance and features across existing detection tools.

It is true that some tools have their codes stored under an online repository (such as GitHub) and other researchers can partly reuse the codes. However, understanding those codes can be more difficult than creating them from scratch; the programmers could use uncommon terminologies, logic, and programming paradigm as these programmers may come from different backgrounds. Further, the codes may be written in another programming language in which does not suit the researchers' development needs.

To fill the aforementioned gap, this paper proposes a library for observing two plagiarism-suspected codes, which is easily integrable. Further, it is not constrained by the selected programming language for development. The library is represented as a standard application which can be accessed with ease through command line (i.e., a feature that is supported by most programming languages); several parameters are passed by storing them in a particular file prior executing the library. Using this library, prospective researchers are not required to recreate all

* Corresponding author

ISSN 2443-2555 (online) 2598-6333 (print) © 2019 The Authors. Published by Universitas Airlangga.
This is an open access article under the CC BY license (http://creativecommons.org/licenses/by/4.0/)
doi: http://dx.doi.org/10.20473/jisebi.5.2.110-119

components from scratch in developing a new source code plagiarism detection tool. To the best of our knowledge, this is the first attempt to address such a recreation issue by creating a language-independent library.

The library can also act as a standalone tool for detecting source code plagiarism. It is capable to detect the similarity between two source codes (written in either Java, C, or Python) using Running-Karp-Rabin Greedy-String-Tiling [5].

II. LITERATURE REVIEW

Tools for detecting source code plagiarism can be classified in three groups: attribute-based, structure-based, and hybrid detection tool [6]. Attribute-based tool determines the similarity (which is a suspicious hint for plagiarism) based on source code attributes. Structure-based tool determines the similarity based on source code structure. Hybrid tool combines the techniques of attribute- and structure-based tool to determine the similarity.

Attribute-based tool considers two source codes as similar to each other when they share similar attributes (e.g., source code characteristics or fragments). The similarity can be measured either by standard occurrence counting [7], [8], Information Retrieval measurement [9]–[12], or clustering technique [13], [14].

Structure-based tool defines source code similarity based on given codes' shared structure. The structure can be either source code token sequence [2], [15], [16], low-level token sequence [17], [18], program dependency graph [19] or abstract syntax tree [20]. The similarity of the first two structures are commonly measured by string matching algorithms (e.g., Running-Karp-Rabin Greedy-String-Tiling [5]), that have been modified to handle source code tokens instead of characters. Whereas, the similarity of the remaining structures are measured with domain-specific algorithms (e.g., tree kernel algorithm).

Hybrid tool is a combination of attribute- and structure-based tool. It can be derived further to two sub-groups (effectiveness- and efficiency-oriented hybrid tool) based on its primary focus.

Effectiveness-oriented hybrid tool focuses the combination to aim higher effectiveness. It is based on a statement which claims attribute- and structure-based tool' techniques complement each other in terms of effectiveness. A tool proposed by [21], for example, displayed the outputs of both techniques at once. Other two examples are tools proposed by [22], [23]. They considered the structure-based technique's output as an input for its attribute-based technique.

Efficiency-oriented hybrid tool focuses on higher efficiency. It is based on a statement which claims attribute-based technique is more time efficient than the structure-based one even though its result can be less effective [3]. Tools proposed by [3], [24], for instances, put attribute-based technique as an initial filter for the inputs of structure-based technique.

Instead of proposing new plagiarism detection tools, some works were aimed to support existing research on source code plagiarism detection. Those supportive works can be classified further to two groups: technical and non-technical supportive works.

On the one hand, technical supportive works foster the research by maturing supplementary components on source code plagiarism detection tools. Works in [25], [26], for example, proposed a mechanism to replace method invocations with their respective contents so that the effectiveness of the detection tool can be improved. A work in [27] proposed a meta-tool which combines several existing detection tools. A work in [28] proposed dynamic thresholding mechanisms to mitigate the number of false results.

On the other hand, non-technical supportive works foster the research by providing clearer philosophical foundation or issues on source code plagiarism detection. Works in [1], [29]–[31] captured the definition of source code plagiarism from both lecturers and students. A work in [32] summarised how lecturers inform programming academic integrity to students. A work in [33] proposed a process model for detecting plagiarism in the source code domain. A work in [34] introduced a learning method to facilitate students in understanding that source code similarity does not always lead to plagiarism.

III. METHOD

Despite they are high in number, most source code plagiarism detection tools do not enable modification. Hence, when a modification is required to be applied, most researchers would create a new detection tool just to implement it. This can be an issue since recreating a plagiarism detection tool from scratch is time-consuming whlist most of the components will be the same as existing detection tools'. It is true that some of the existing tools can be tweaked and modified by obtaining the codes from the authors. Nevertheless, understanding other people's codes can be demanding; those codes can contain unfamiliar terminologies, logic, and programming paradigm due to different author backgrounds. Further, the codes can be written in a different programming language, in which will not suit the researchers' development programming language.

This paper proposes a library for observing two plagiarism-suspected codes (which is called Plago, Plagiarism Observer). The library, upon embedded on a tool, will provide a panel for determining whether two suspected source codes are considered as a plagiarism. It highlights the matched subsequences based on information given by the tool. Further, several common features for observation are also provided: a unique highlight colour per matched subsequence, IDE-like code viewers (which highlights some tokens based on their type), and direct navigation to focus on a selected matching tuple.

Unique to Plago, it is easy to integrate since the prospective researchers are not required to understand much technical details. Further, it is not constrained by development programming languages; it is executed from the command line, which is supported by most programming languages and operating systems.

To integrate Plago in a source code plagiarism detection tool, the library should be programmatically invoked through the command line. It is true that syntaxes for executing the command line can vary across development programming languages. However, according to our observation toward several programming languages (C#, Java, Python, and PHP), these syntaxes are easy to use and intuitive.

Plago's inputs should be provided through a file called comparison log file, and it should be located on the same directory as the library. The log file contains source code filepaths, token sequences and matching tuples. Filepaths represent where the codes are located. Token sequences (which can be generated with the help of ANTLR library [35]) are the compact representation of the codes in which all whitespaces and comments have been removed. Each of these tokens should be featured with information about the location on its respective code (as column and row indexes). Such information will be used to map the matched tokens to code viewers. Matching tuples inform which parts of the codes are similar. Each of the tuples comprises of the matched tokens' start index on the first token sequence, start index on the second token sequence, and size.

Plago's main view can be seen on Fig. 1. It has three panels: information panel and two source code viewers (labeled as A, B, and C respectively). Information panel shows given source code paths and matching tuples. Each matching tuple has three values separated by ":". The first two values are the starting indexes of the matched tokens while the last value is the size of matched tokens. For instance, if a tuple has "3:5:2" as its value, the codes share two tokens, which location starts at the 3rd token on the first token sequence and the 5th token on the second token sequence.

Source code viewers display the inputted codes as in most source code editors; some tokens' foreground is recoloured based on their respective type for user convenience. It also highlights the matched tokens by recolouring the background. The recolouring is performed with the help of AvalonEdit [36].

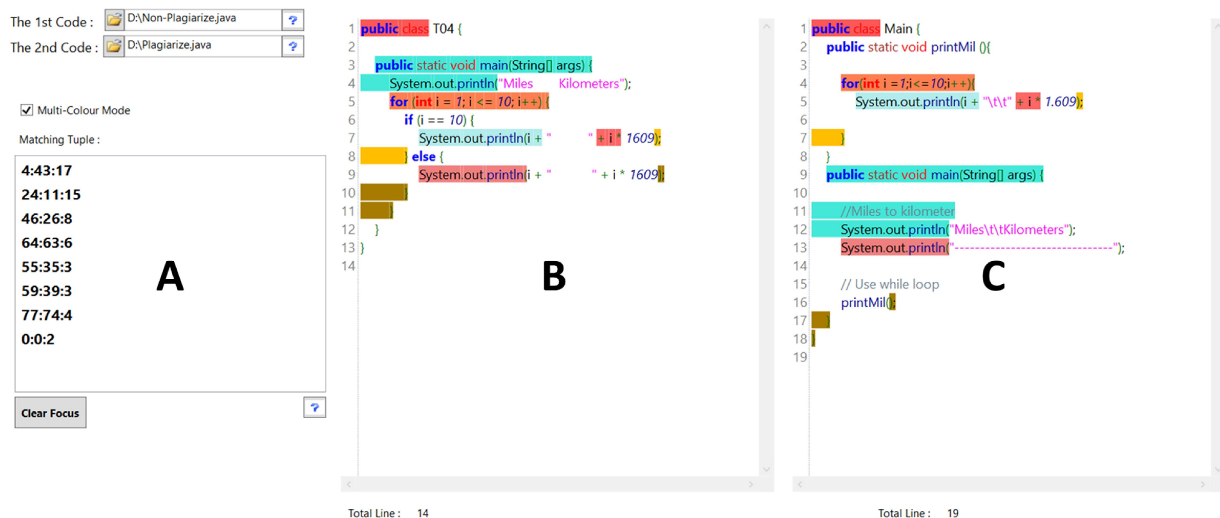

Fig. 1 The main view of Plago

When the user wants to see where a matching tuple's tokens are placed on the codes, they can click that tuple on the matching tuple panel. It will change the focus of the source code viewers to those tokens, highlighting them with darker colour.

If the user wants to see matched tokens' counterpart on another source code viewer, they can click those tokens on their containing viewer. It will change the focus of another source code viewer to the counterpart. Further, it will highlight the tokens on both source code viewers with darker colour.

Plago has two modes for highlighting matched tokens. These modes can be selected with a checkbox placed on the information panel. The first one is single-colour mode. As its name states, all matched tokens are highlighted with the same colour (which is blue in our context). Another one is multi-colour mode. For each matching tuple, its matched tokens are highlighted with a unique colour. The colour is changed based on our 50 pre-defined colours. If the number of matching tuples is higher than 50, some tuples will be highlighted with previously-used colours. It is true that we can use colour gradation to automatically assign colours. However, the resulted gradation can lead to non-contrasting colours when there are so many tuples, adding some difficulties to the user in differentiating the tuples.

In addition to be used as a library, Plago can also act as a standalone plagiarism detection tool. This mode can be accessed by double-clicking the library without providing the comparison log file. It will display a panel (see Fig. 2) which is similar to Fig. 1 except that the information panel is replaced with input panel. As the inputs, the user should provide two compared-to-be source codes and select the target programming language. To date, the standalone mode only covers three programming languages (Java, C++, or Python).

Fig. 2 The standalone view of Plago

Upon pressing the "execute" button, Plago will measure the similarity between the codes and show the result for further observation. The measurement itself is conducted in threefold. At first, each code is tokenised to a token sequence with ANTLR [35] (where comment and whitespace tokens are excluded). Secondly, the matching tuples are determined by comparing the token sequences using Running-Karp-Rabin Greedy-String-Tiling [5] with 2 as its minimum matching length. Finally, the similarity degree is calculated as in (1) where *C1* & *C2* are the token sequences and *match(C1,C2)* is the number of matched tokens.

$$norm(c1, c2) = \frac{2\ match(c1,c2)}{|c1|+|c2|} \quad (1)$$

Plago was developed with C# as its programming language and Visual Studio as its development IDE. It relies on two other external libraries which are AvalonEdit [36] (for source code panel and colour highlighting) and ANTLR [35] (for tokenising the source code files in the standalone tool mode). The ANTLR library can also be used to generate the inputs of Plago's library mode. However, it is not a must considering the comparison log file can be created with other libraries, or even no libraries at all.

The development of Plago can be divided to several stages. At first, the main view of Plago was built based on the analysis of several observation panels from existing source code plagiarism detection tools. Secondly, AvalonEdit [36] was embedded on Plago's source code panels for default code view. Thirdly, the highlighting was applied by tweaking the AvalonEdit; per matching tuple, the code fragment is highlighted by recolouring the background from a particular line and column to another particular line and column. Fourthly, the format of comparison log file was

defined by summarising all inputs required for the library. Finally, the standalone mode was developed by altering the inputs from the comparison log file to user inputs.

IV. RESULT

Our proposed library was evaluated from four aspects: integrability, functionalities, effectiveness, and efficiency. The integrability was measured by integrating our library on three prototypes of source code plagiarism detection tool, written in different programming languages. The functionalities were measured using a blackbox testing. The effectiveness was measured by comparing teaching assistants' accuracy in terms of finding similar subsequences with or without our library. The efficiency was measured in the same way as the effectiveness except that completion time was considered instead of the accuracy.

We are aware that in terms of effectiveness and efficiency, the library can be compared with existing source code plagiarism detection tools (such as JPlag [2]). However, we do believe the comparison is subject to bias, favouring our library; existing tools are more complex as they cover the detection process as a whole (while our library only covers a part of it but assures the reusability as a library on other detection tools). Having no knowledge about both our library and the existing tools, it is possible that the teaching assistants work more effectively and efficiently with our library due to its simplicity (even though that is not our aim developing the tool). Further, those two metrics were evaluated just to prove that the output of this library is understandable by humans and it can help humans to observe the suspected source code pairs faster. Instead of outperforming existing source code plagiarism detection tools, this library aims to avoid recreating all components for a source code plagiarism detection tool from scratch.

A. Evaluating the Integrability and The Functionalities

Plago's integrability was evaluated by integrating it on three source code plagiarism detection prototype tools. These prototypes were written in three different programming languages: Java, C++, and Python. All of them work in three simple steps. At first, they accept two source codes and convert them to token sequences with the help of ANTLR [35]. Secondly, they measure the code similarity degree using Running-Karp-Rabin Greedy-String-Tiling [5] with two as its minimum matching length. Finally, they create the comparison log file and pass it to Plago.

In terms of other functionalities, Plago was evaluated with blackbox testing. Twenty three scenarios were created by the first author of this paper and therefore tested on Plago. These scenarios (which details can be seen on Table 1) cover the functionalities in both library and standalone mode. The results of all scenarios were as expected.

B. Evaluating the Effectiveness

The effectiveness was evaluated by comparing teaching assistants' performance in terms of finding matched subsequences (i.e., accuracy) with or without Plago. The former scenario is referred as Plago scenario while the latter is referred as conventional scenario. Plago is considered as effective if its existence enhances the accuracy of teaching assistants.

Ten teaching assistants, who had experienced in manually detecting source code plagiarism, were involved in this evaluation. They are labeled as R01 to R10 respectively. Each assistant was asked to find matched subsequences from plagiarism-suspected source code pairs with both scenarios; where the conventional scenario was conducted first. Since locating the subsequences' lines and columns is considerably demanding, the assistants were only required to mark the starting line of the subsequences. The accuracy is defined by dividing the number of correctly-predicted subsequences with the total number of matched subsequences (calculated using Running-Karp-Rabin Greedy-String-Tiling [5] with two as its minimum matching length).

Table 2 depicts assessment tasks that were given to the assistants. These tasks rely on Java plagiarism-suspected source code pairs defined on Table 3. The first four have the same source code pair for both scenarios while the others have different source code pairs (but with the same difficulty). To balance the number of cases on those two categories, T01 & T02 were only performed by the first half assistants; T03 & T04 were performed by the last half; and T05 & T06 were performed by all assistants.

Fig. 3 shows the accuracy of Plago and conventional scenario when the tasks shared the same code pairs (T01-T04). Horizontal axis refers to cases; these cases are labeled as "X-Y" where X refers to a task ID and Y refers to a teaching assistant ID. Vertical axis refers to the accuracy in percentage. Fig. 4 also show the accuracy of Plago and conventional scenario except that the tasks shared different code pairs with the same difficulty (T05 and T06).

TABLE 1
SCENARIOS FOR BLACKBOX TESTING

ID	Scenario	Expected Result
S01	Start using Plago as a library without providing the first source code file	An error message stating “the first source code file should be provided” occur
S02	Start using Plago as a library without providing the second source code file	An error message stating “the second source code file should be provided” occur
S03	Start using Plago as a library without providing comparison log file	An error message stating “the comparison log file should be provided” occur
S04	Start using Plago as a library with non-parseable comparison log file	An error message stating “the comparison log file should be parseable” occur
S05	Start using Plago as a library without checking the multi-colour mode checkbox	Similar subsequences on the source code panels will be highlighted with the same colour, which is blue.
S06	Start using Plago as a library and clicking a row on the matching tuple panel	The source code panels will show the selected row’s target subsequence
S07	Start using Plago as a library and clicking the clear focus button	Nothing happens
S08	Start using Plago as a library and upon observing, clicking the clear focus button	The selected row on the matching tuple panel will be unselected
S09	Start using Plago as a library and clicking a subsequence from the first source code panel	The second panel will show the selected subsequence’s position and the matching tuple panel will highlight the subsequence’s tuple
S10	Start using Plago as a library and clicking a subsequence from the second source code panel	The first panel will show the selected subsequence’s position and the matching tuple panel will highlight the subsequence’s tuple
S11	Start using Plago as a library and scrolling the first source code panel	The first source code will be focused on lower part of the code
S12	Start using Plago as a library and scrolling the second source code panel	The second source code will be focused on lower part of the code
S13	Start using Plago as a standalone tool without providing the first source code file	An error message stating “the first source code file should be provided” occur
S14	Start using Plago as a standalone tool without providing the second source code file	An error message stating “the second source code file should be provided” occur
S15	Start using Plago as a standalone tool without selecting the programming language	An error message stating “the programming language should be selected” occur
S16	Start using Plago as a standalone tool without checking the multi-colour mode checkbox	Similar subsequences on the source code panels will be highlighted with the same colour, which is blue.
S17	Start using Plago as a standalone tool and clicking a row on the matching tuple panel	The source code panels will show the selected row’s target subsequence
S18	Start using Plago as a standalone tool and clicking the clear focus button	Nothing happens
S19	Start using Plago as a standalone tool and upon observing, clicking the clear focus button	The selected row on the matching tuple panel will be unselected
S20	Start using Plago as a standalone tool and clicking a subsequence from the first source code panel	The second panel will show the selected subsequence’s position and the matching tuple panel will highlight the subsequence’s tuple
S21	Start using Plago as a standalone tool and clicking a subsequence from the second source code panel	The first panel will show the selected subsequence’s position and the matching tuple panel will highlight the subsequence’s tuple
S22	Start using Plago as a tool and scrolling the first source code panel	The first source code will be focused on lower part of the code
S23	Start using Plago as a tool and scrolling the second source code panel	The second source code will be focused on lower part of the code

TABLE 2
THE ASSESSMENT TASKS FOR EVALUATING EFFECTIVENESS

ID	Plago Scenario's Code Pair	Conventional Scenario's Code Pair
T01	C01	C01
T02	C02	C02
T03	C03	C03
T04	C04	C04
T05	C05	C06
T06	C07	C08

TABLE 3
THE SOURCE CODE PAIRS FOR EVALUATING EFFECTIVENESS

ID	Topic
C01	Output
C02	Input
C03	Function
C04	Looping
C05	Output (Version 2)
C06	Looping (Version 2)
C07	Looping (Version 3)
C08	Array

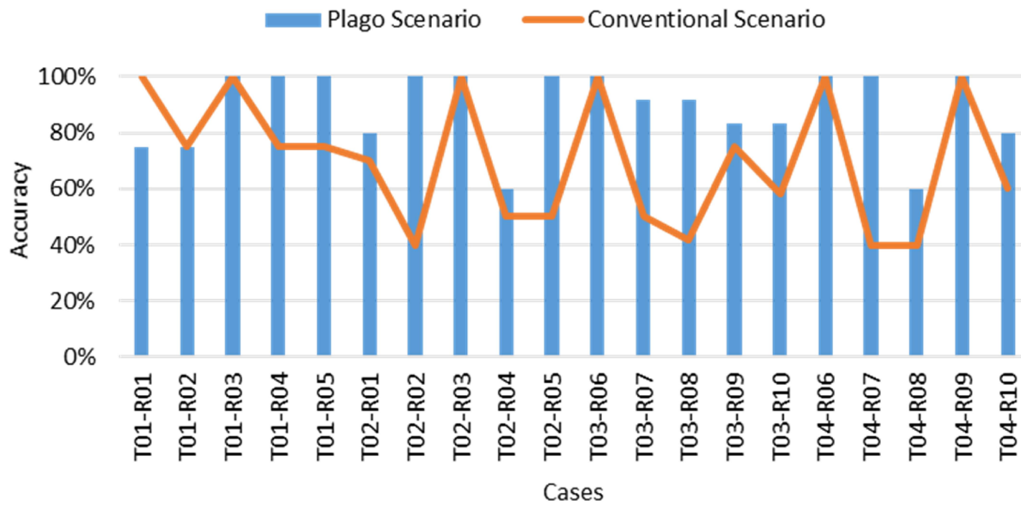

Fig. 3 The accuracy of Plago and conventional scenario on tasks with the same code pair (T01-T04)

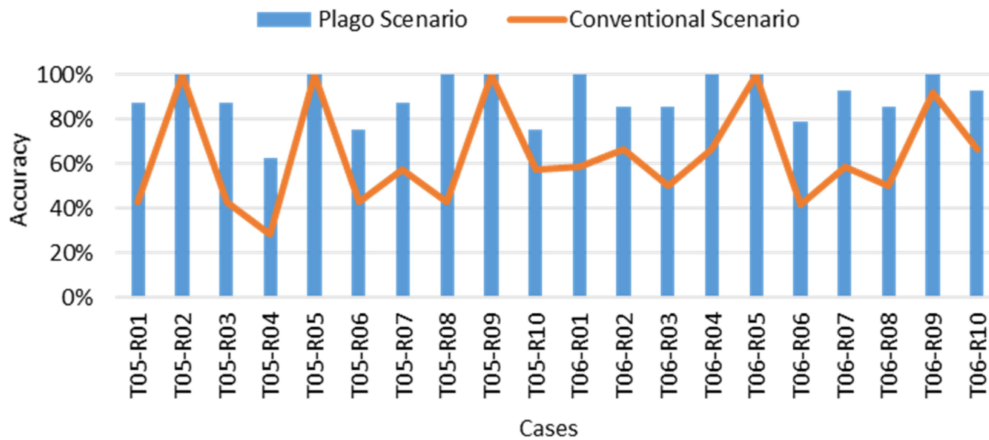

Fig. 4 The accuracy of Plago and conventional scenario on tasks with different code pairs with the same difficulty (T05 & T06)

C. Evaluating the Efficiency

The efficiency was evaluated in a similar manner as the effectiveness except that the accuracy was replaced with the completion time. Plago is considered as efficient if its existence reduces the completion time of given tasks.

Fig. 5 depicts the completion time for the first four tasks (T01-T04 which relied on the same code pairs). Horizontal axis refers to cases; these cases are labeled as "X-Y" where X refers to a task ID and Y refers to a

teaching assistant ID. Vertical axis refers to the completion time in minutes. Fig. 6 shows the same on the remaining two tasks (T05 and T06 which shared different code pairs with the same difficulty).

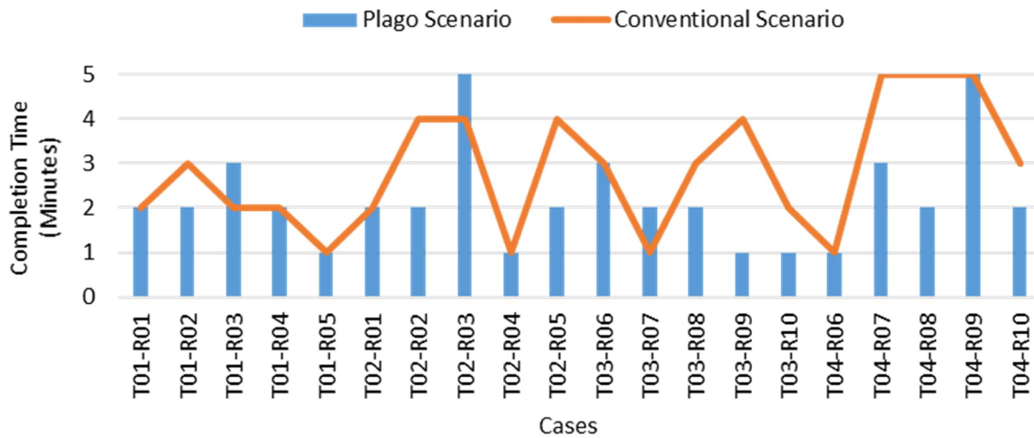

Fig. 5 The completion time of Plago and conventional scenario on tasks with the same pair (T01-T04).

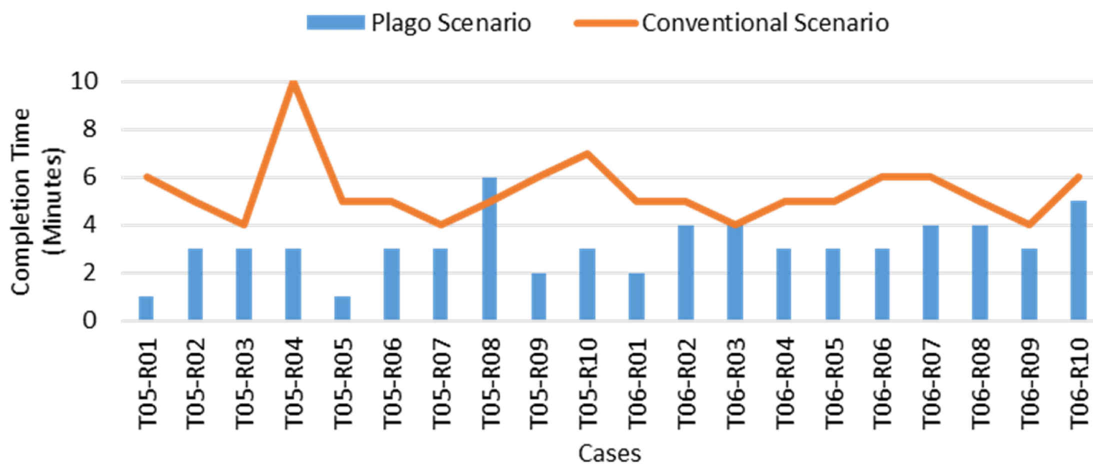

Fig. 6 The completion time of Plago and conventional scenario on tasks with different code pairs with the same difficulty (T05 & T06)

V. DISCUSSION

A. Discussion on the Integrability and The Functionalities

In general, Plago can be easily integrated with Java, C++, and Python prototypes. The integration only took ten minutes each for Java and C++ prototype. Python prototype took longer (about a hour) due to Python's consideration of whitespace tokens. Some adjustments were required to make sure that the information from matched tuples were displayed correctly on source code viewers. In our case, we removed all whitespace tokens before measuring the similarity degree.

According to the blackbox testing which details are listed on Table 1, all scenarios worked as expected. Every feature is functioned well. Hence, it can be stated that Plago is also functional.

B. Discussion on the Effectiveness

Fig. 3 shows the accuracy of Plago and conventional scenario when the tasks shared the same code pairs (T01-T04). Plago generates higher accuracy than the conventional one on more than half cases. The largest improvement occured on T02-R02 and T04-R07 with 60% increase. In average, it generated 19% higher accuracy. When tested

using two-tailed t-test with 95% confidence rate, the result was statistically significant; its p-value is 0.001, which is lower than the maximum threshold for significance (0.005).

When the tasks shared different code pairs with the same difficulty (T05 and T06), all cases led to a non-negative accuracy improvement when Plago was used. (see Fig. 4). The largest improvement was 57.14% while the lowest improvement was 0% (since both scenarios led to 100% accuracy at that time). Plago led to 26.61% improvement in average. The improvement was statistically significant when measured using two-tailed t-test with 95% confidence rate; its p-value (1.2E-06) was lower than the maximum threshold for significance.

C. Discussion on the Efficiency

Fig. 5 depicts the completion time for the first four tasks (T01-T04 which rely on the same code pairs). In most occasions, Plago leads to shorter completion time (with 0.65 minute reduction in average). The completion time reduction was proved to be statistically significant when measured using two-tailed t-test with 95% confidence rate; its p-value was 0.02, that was lower than 0.05 (the maximum threshold for significance). T03-R09 and T04-R08 experienced the largest completion time reduction; Plago took three minutes shorter per case. T01-R03, T02-R03, and T03-R07, on the contrary, were the cases where Plago took more completion time (which is one minute longer per case).

For the remaining two tasks (T05 and T06 which shared different code pairs with the same difficulty), Fig. 6 depicts that Plago leads to shorter completion time on most cases; its average reduction was 2.25 minutes. The reduction was statistically significant; its p-value (when measured using two-tailed t-test with 95% confidence rate) was 3.2E-05 and it was lower than the maximum threshold for significance. Plago's largest reduction (7 minutes) occurs on T05-R04 while the lowest reduction (-1 minute) occurs on T05-R08.

VI. CONCLUSIONS

This paper proposes a language-independent library for observing source code plagiarism. The library is called Plago, Plagiarism Observer. It can be integrated to any plagiarism detection tools as long as those tools support command line arguments. Further, it can also act as a standalone plagiarism detection tool.

The library is expected to deal with recreation issue on source code plagiarism detection tools in which some components should be recreated from scratch even though the components share the same traits among the tools. To the best of our knowledge, this is the first attempt to address such an issue by proposing a language-independent library.

According to our evaluation, Plago is integrable since it has been successfully integrated with three prototype tools for detecting source code plagiarism. Further, it is functional; the features work as expected.

In terms of effectiveness and efficiency, Plago can help teaching assistants to find matched subsequences on plagiarism-suspected source code pairs. It boosted up the assistants' accuracy by 19% for tasks with the same code pair and 26.61% for tasks with different code pairs that have the same difficulty. Further, it cut up the tasks' completion time by 0.65 minute for tasks with the same code pair and 2.25 minutes for tasks with different code pairs that have the same difficulty.

For future work, we intend to make other frequently-used components on source code plagiarism detection tools (e.g., a panel for comparing all source codes to each other) as language-independent libraries. Further, we also intend to integrate more advanced similarity measurements on Plago's standalone mode.

REFERENCES

[1] G. Cosma and M. Joy, "Towards a definition of source-code plagiarism," *IEEE Transactions on Education*, vol. 51, no. 2, pp. 195–200, May 2008.
[2] L. Prechelt, G. Malpohl, and M. Philippsen, "Finding plagiarisms among a set of programs with JPlag," *Journal of Universal Computer Science*, vol. 8, no. 11, pp. 1016–1038, 2002.
[3] L. Sulistiani and O. Karnalim, "ES-Plag: efficient and sensitive source code plagiarism detection tool for academic environment," *Computer Applications in Engineering Education*, vol. 27, no. 1, pp. 166–182, 2019.
[4] A. E. 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.
[5] M. J. Wise, "Yap3: improved detection of similarities in computer program and other texts," in *The 27th SIGCSE Technical Symposium on Computer Science Education*, 1996, vol. 28, no. 1, pp. 130–134.
[6] O. Karnalim, "A low-level structure-based approach for detecting source code plagiarism," *IAENG International Journal of Computer Science*, vol. 44, no. 4, pp. 501–522, 2017.
[7] K. J. Ottenstein, "An algorithmic approach to the detection and prevention of plagiarism," *ACM SIGCSE Bulletin*, vol. 8, no. 4, ACM, pp. 30–41, 01-Dec-1976.
[8] J. A. W. Faidhi and S. K. Robinson, "An empirical approach for detecting program similarity and plagiarism within a university programming environment," *Computers & Education*, vol. 11, no. 1, pp. 11–19, 1987.

[9] D. Ganguly, G. J. F. Jones, A. Ramírez-de-la-Cruz, G. Ramírez-de-la-Rosa, and E. Villatoro-Tello, "Retrieving and classifying instances of source code plagiarism," *Information Retrieval Journal*, vol. 21, no. 1, pp. 1–23, Sep. 2018.
[10] F. Ullah, J. Wang, M. Farhan, S. Jabbar, Z. Wu, and S. Khalid, "Plagiarism detection in students' programming assignments based on semantics: multimedia e-learning based smart assessment methodology," *Multimedia Tools and Applications*, Mar. 2018.
[11] G. Cosma and M. Joy, "An approach to source-code plagiarism detection and investigation using Latent Semantic Analysis," *IEEE Transactions on Computers*, vol. 61, no. 3, pp. 379–394, Mar. 2012.
[12] O. Karnalim, "Source code plagiarism detection with low-level structural representation and information retrieval," *International Journal of Computers and Applications*, Mar. 2019.
[13] L. Moussiades and A. Vakali, "PDetect: a clustering Approach for detecting plagiarism in source code datasets," *The Computer Journal*, vol. 48, no. 6, pp. 651–661, Nov. 2005.
[14] T. Ohmann and I. Rahal, "Efficient clustering-based source code plagiarism detection using PIY," *Knowledge and Information Systems*, vol. 43, no. 2, pp. 445–472, May 2015.
[15] A. B. Franca, D. L. Maciel, J. M. Soares, and G. C. Barroso, "Sherlock N-Overlap: invasive normalization and overlap coefficient for the similarity analysis between source code," *IEEE Transactions on Computers*, 2018.
[16] C. Kustanto and I. Liem, "Automatic source code plagiarism detection," in *The 10th ACIS International Conference on Software Engineering, Artificial Intelligences, Networking and Parallel/Distributed Computing*, 2009, pp. 481–486.
[17] O. Karnalim, "Python Source Code Plagiarism Attacks on Introductory Programming Course Assignments," *Themes in Science and Technology Education*, vol. 10, no. 1, 2017.
[18] F. S. Rabbani and O. Karnalim, "Detecting source code plagiarism on .NET programming languages using low-level representation and adaptive local alignment," *Journal of Information and Organizational Sciences*, vol. 41, no. 1, pp. 105–123, Jun. 2017.
[19] C. Liu, C. Chen, J. Han, and P. S. Yu, "Gplag: detection of software plagiarism by program dependence graph analysis," in *The 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2006, p. 872.
[20] D. Fu, Y. Xu, H. Yu, and B. Yang, "WASTK: a weighted abstract syntax tree kernel method for source code plagiarism detection," *Scientific Programming*, vol. 2017, pp. 1–8, Feb. 2017.
[21] M. El Bachir Menai and N. S. Al-Hassoun, "Similarity detection in Java programming assignments," in *The 5th International Conference on Computer Science & Education*, 2010, pp. 356–361.
[22] S. Engels, V. Lakshmanan, and M. Craig, "Plagiarism detection using feature-based neural networks," in *The 38th SIGCSE Technical Symposium on Computer Science Education*, 2007, vol. 39, no. 1, p. 34.
[23] J. Y. H. Poon, K. Sugiyama, Y. F. Tan, and M.-Y. Kan, "Instructor-centric source code plagiarism detection and plagiarism corpus," in *The 17th ACM Annual Conference on Innovation and Technology in Computer Science Education*, 2012, p. 122.
[24] S. Burrows, S. M. M. Tahaghoghi, and J. Zobel, "Efficient plagiarism detection for large code repositories," *Software: Practice and Experience*, vol. 37, no. 2, pp. 151–175, Feb. 2007.
[25] O. Karnalim, "An abstract method linearization for detecting source code plagiarism in object-oriented environment," in *The 8th IEEE International Conference on Software Engineering and Service Science*, 2017, pp. 58–61.
[26] O. Karnalim, "IR-based technique for linearizing abstract method invocation in plagiarism-suspected source code pair," *Journal of King Saud University - Computer and Information Sciences*, Feb. 2018.
[27] A. O. Portillo-Dominguez, V. Ayala-Rivera, E. Murphy, and J. Murphy, "A unified approach to automate the usage of plagiarism detection tools in programming courses," in *The 12th International Conference on Computer Science and Education*, 2017, pp. 18–23.
[28] O. Karnalim and L. Sulistiani, "Dynamic thresholding mechanisms for IR-based filtering in efficient source code plagiarism detection," in *The 2018 International Conference on Advanced Computer Science and Information Systems*, 2018, pp. 23–28.
[29] M. Joy, G. Cosma, J. Y.-K. Yau, and J. Sinclair, "Source code plagiarism—a student perspective," *IEEE Transactions on Education*, vol. 54, no. 1, pp. 125–132, Feb. 2011.
[30] D. Chuda, P. Navrat, B. Kovacova, and P. Humay, "The Issue of (software) plagiarism: a student view," *IEEE Transactions on Education*, vol. 55, no. 1, pp. 22–28, Feb. 2012.
[31] D. Zhang, M. Joy, G. Cosma, R. Boyatt, J. Sinclair, and J. Yau, "Source-code plagiarism in universities: a comparative study of student perspectives in China and the UK," *Assessment & Evaluation in Higher Education*, vol. 39, no. 6, pp. 743–758, Aug. 2014.
[32] Simon, J. Sheard, M. Morgan, A. Petersen, A. Settle, and J. Sinclair, "Informing students about academic integrity in programming," in *The 20th Australasian Computing Education Conference*, 2018, pp. 113–122.
[33] D. Kermek and M. Novak, "Process model improvement for source code plagiarism detection in student programming assignments," *Informatics in Education*, vol. 15, no. 1, pp. 103–126, 2016.
[34] F.-P. Yang, H. C. Jiau, and K.-F. Ssu, "Beyond plagiarism: an active learning method to analyze causes behind code-similarity," *Computers & Education*, vol. 70, pp. 161–172, Jan. 2014.
[35] T. Parr, *The definitive ANTLR 4 reference*. Pragmatic Bookshelf, 2013.
[36] D. Grunwald, "AvalonEdit by icsharpcode," 2001. [Online]. Available: http://avalonedit.net/. [Accessed: 05-Jan-2019].