Scientific Journal of Informatics
Vol. 11, No. 1, Feb 2024

p-ISSN 2407-7658 http://journal.unnes.ac.id/nju/index.php/sji e-ISSN 2460-0040

Damerau-Levenshtein Distance Algorithm Based on Abstract Syntax Tree to Detect Code Plagiarism

Ahlijati Nuraminah[1*], Abdullah Ammar[2]
[1, 2]Computer Science Department, Sekolah Tinggi Ilmu Manajemen dan Ilmu Komputer ESQ, Indonesia

Abstract.
Purpose: This research aimed to detect source code plagiarism based on Abstract Syntax Tree using Damerau-Levenshtein Distance algorithm, which is expected to streamline the inaccuracies and time-consumption associated with the manual process.
Methods: Damerau-Levenshtein Distance algorithm was used to determine the similarity between source code files and calculate F-Measure. The dataset, which consisted of 178 source code files from 20 coursework assignments, was obtained from GitHub by Lawton Nichols in 2019. Damerau-Levenshtein Distance algorithm was used to compute the minimum cost required to transform one line of code into another. Furthermore, ANTLR detected AST, which was processed through preprocessing, including node pruning, function and variable sorting, and log output removal.
Result: The result showed that the two methods took 5.704 seconds and 0.996 seconds to complete. The lowest and highest values obtained using F-Measure were 0.16 and 0.8, respectively. Therefore, the system performed detection processes quickly and effectively detected common forms of code plagiarism with difficulty in the more complex forms.
Novelty: In conclusion, this research used AST and Damerau-Levenshtein Distance algorithm to calculate the 5 levels of similarity in Java programming language source code. For further development, preprocessing steps were needed to prune unnecessary nodes and detect equivalent but differently syntaxed code.

Keywords: Code plagiarism, Code similarity, Abstract syntax tree, Damerau-levensthein distance algorithm.
Received October 2023 / **Revised** November 2023 / **Accepted** December 2023

INTRODUCTION

Plagiarism is a copyright infringement [1] in various forms, such as art, graphic design, and writing. It is prevalent in programming coursework, particularly code plagiarism, defined as when individuals either copy codes entirely or partially, essentially stealing someone else source code [2]. According to preliminary research [3], 65% of students engage in code plagiarism at least once during their academic studies, which [4], poses challenges for lecturers. This is because students focus on obtaining grades [3] rather than learning and gaining programming experience.

The act of acquiring programming skills is typically a result of practical experience gained through constant practice and the successful completion of assignments given by lecturers [5]. Each assignment is manually checked to ensure that no students engaged in code plagiarism, although it is time-consuming and challenging [6]. According to interviews with the lecturer, it takes approximately one to two hours to assess relatively 20 assignments. This labor-intensive process consumes valuable time and diverts energy that could be more efficiently directed toward other productive activities. To address these challenges and enhance efficiency, there is need for an automated tool to detect code plagiarism.

Numerous automated tools, including SIM [7], JPlag [8], and the Measure of Software Similarity (MOSS) [9] are currently used to detect code plagiarism. SIM and MOSS were considered promising in this situation [10], while another research stated that JPlag and MOSS were found to be effective in detecting plagiarism [11]. Despite the efficacy, there is a need for improvements, particularly in enhancing user usability and interaction with the output of these tools [10]. A significant disadvantage identified in existing code

[*]Corresponding author.
Email addresses: ahlijati.nuraminah@esqbs.ac.id (Nuraminah)
DOI: 10.15294/sji.v11i1.48064

plagiarism detection systems is the poor presentation and visualization of output, prompting the need for enhancement in this aspect [11].

In code plagiarism detection, various methods and algorithms are used, including Token [12], Graph [13], Attribute [14], and Structure-based Detection [15]. Furthermore, structure or Parse-based Detection, used to generate Abstract Syntax Trees (AST), is suitable for identifying code plagiarism because it accurately represents the structure [16]. AST is also considered effective in identifying attempts to avoid detection systems, such as variable renaming, adding comments, and function rearrangement. The preprocessing of AST, which includes sorting, filtering, and weighting operations, is effective for creating a more efficient system for comparing code files [17]. Various algorithms, such as Damerau-Levenshtein Distance, are used in the comparison process. Initially designed for detecting spelling errors [18], [19], this algorithm is effective in code detection due to the high accuracy based on the grammar rules used [19].

Previous research has adopted diverse strategies for plagiarism detection, such as the Winnowing algorithm in MOSS software [20], logistic regression for comparing code differences [21], string tiling algorithm [22] , and syntax-based improvements using Smith-Waterman [17].Despite these approaches, there has been limited exploration of Damerau-Levenshtein Distance in Java programming language. This research aims to fill this gap by investigating the applicability of Damerau-Levenshtein Distance algorithm, specifically for detecting source code plagiarism in Java. It also aims to incorporate a 5-level similarity measure into the evaluation process.

In this research, Java source code served as the main data format. The code, initially represented as a string, is passed to the lexer and parser generated by ANTLR. The lexer translates the string data into an array of tokens converted into Concrete Syntax Tree (CST), which is then transformed into AST through a walker. To enhance analysis, a preprocessing step was adopted to arrange variables, functions, and class declarations, based on the subtree depth in ascending order. The comparison phase was carried out using Damerau-Levenshtein Distance algorithm. This algorithm calculates the minimum cost in numerical units needed to transform one or an array of strings into another, thereby providing a resultant similarity value. The next step used a Confusion Matrix table to assess the successful and failed rate [23], [24] of detecting plagiarism pairs. From the confusion matrix, metrics such as Accuracy [25], Precision, Recall, and F-Measure values were obtained.

METHODS

The sequential steps adopted in this research are shown in Figure 1.

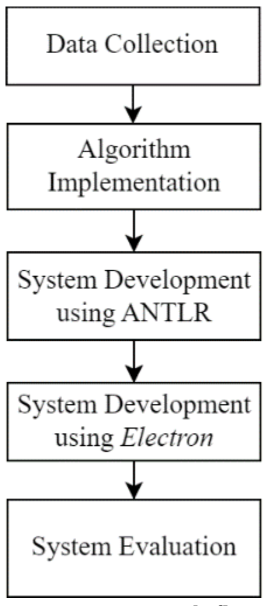
Data Collection
Algorithm Implementation
System Development using ANTLR
System Development using *Electron*
System Evaluation

Figure 1. Research flow

Data Collection

This research used primary data obtained from the Academic Department of STIMIK ESQ. The dataset comprised 24 source code files, divided equally between the two assessment periods of 2019/2020 and 2020/2021 academic years, resulting in 66 pairs accessible on https://github.com/ahlijatinuraminah/plagio. Based on manual analysis, the source code files in this dataset had an average cheating level of 4, with each containing between 100 and 300 lines of code. Manual observation revealed several cheating attributes in the data, including adding comments, modifying identifiers such as variable names and functions, reordering functions, changing data types to its equivalent, and function extraction. A detailed breakdown of this information is shown in Table 1.

Table 1. Frequency of original and plagiarized data for the data structures and algorithms course.

No	Year	Data	Frequency
1	2019/2020	Original	2020_1.java
2	2019/2020	Original	2020_2.java
3	2019/2020	Original	2020_4.java
4	2019/2020	Original	2020_9.java
5	2019/2020	Original	2020_11.java
6	2019/2020	Plagiarized Predicted	2020_0.java
7	2019/2020	Plagiarized Predicted	2020_5.java
8	2019/2020	Plagiarized Predicted	2020_6.java
9	2019/2020	Plagiarized Predicted	2020_7.java
10	2019/2020	Plagiarized Predicted	2020_10.java
11	2019/2020	Plagiarized Predicted	2020_12.java
12	2019/2020	Plagiarized Predicted	2020_13.java
13	2020/2021	Original	2021_2.java
14	2020/2021	Original	2021_3.java
15	2020/2021	Original	2021_5.java
16	2020/2021	Original	2021_7.java
17	2020/2021	Original	2021_8.java
18	2020/2021	Plagiarized Predicted	2021_0.java
19	2020/2021	Plagiarized Predicted	2021_1.java
20	2020/2021	Plagiarized Predicted	2021_4.java
21	2020/2021	Plagiarized Predicted	2021_6.java
22	2020/2021	Plagiarized Predicted	2021_9.java
23	2020/2021	Plagiarized Predicted	2021_10.java
24	2020/2021	Plagiarized Predicted	2021_11.java

The secondary data used in this research was sourced from the GitHub page [17] uploaded in 2019. The dataset consisted of source code files generated by a program, divided into two categories, namely original and plagiarized data. The original and plagiarized data had an average of 141 and 2063 lines of code, respectively. In addition, both categories had an average cheating level of 5. The generated data comprised several cheating attributes commonly associated with acts of plagiarism, such as the random addition of spaces, modification of every identifier, including variable names, arguments, and functions, changing data types to its equivalent, replacement of operations with equivalent ones, substitution of control structures, namely with or for loops and vice versa, and rearrangement of the order of statements and functions. The frequency of original and plagiarized data for dataset are shown in Table 2.

Table 2. Frequency of original and plagiarized data for dataset

No	Data	Frequency
1	Original	10
2	Plagiarized	10

Implementation of Algorithm

Damerau-Levenshtein Distance algorithm calculated the minimum cost required to transform one string S1 into another S2. The operations allowed include *insertion*, depicting adding a character at a specific position, *deletion*, meaning removing a character from any position, *substitution* implying replacing a character at any position, and *transposition*, representing swapping two adjacent characters. For example, string S1 (aku) needs a minimum of three costs or steps to obtain S2 (kami). These operations include *transposition*, swapping characters at indices 0 and 1, the next is substitution, replacing the character at index 2, and finally, *insertion*, adding a character at index 3. In terms of computational complexity, the algorithm has a time complexity of O(m.n) in the worst-case scenario, along with a corresponding space complexity of O(m.n).

The various processing stages are outlined as follows [26]:

a. Input Code Files
   Existing code files are classified into two categories, namely plagiarized and non-plagiarized. Following this categorization, the files are inputted and read with the built-in functions of the Node.js module, specifically fs/promises.

   The process starts with a manual data analysis to identify plagiarism cases in the examined code files. Furthermore, the data is systematically labeled based on the detected forms of cheating. These standard forms of code plagiarism are further categorized into several levels, outlined in ascending order of severity [27]:
   - Level 1: Copy and paste entirely.
   - Level 2: Altering comments.
   - Level 3: Modifying identifiers (class, function, and variable names).
   - Level 4: Rearranging code sequences.
   - Level 5: Code refactoring (function separation, code replacement).

b. Formation of CST
   After extracting the string, data from the read code files is passed to the lexer and parser generated by ANTLR. The lexer processes the string data into an array of tokens, read by the parser, and converted into CST. CST is then passed to a walker to be transformed into AST.

c. Formation, Preprocessing, and Linearization of AST
   After obtaining CST, a walker is used to transform it into AST. The walker systematically traverses each node in CST, eliminating redundant nodes. Immediately AST is formed, preprocessing is performed to organize variable, function, and class declarations in ascending order based on the subtree depth. Subsequently, the tree is linearized into a string array for comparison using Damerau-Levenshtein Distance algorithm.

d. Comparison with Damerau-Levenshtein Distance Algorithm
   The comparison process was done using Damerau-Levenshtein Distance algorithm, implemented in JavaScript. After the process had been completed, the minimum cost in numerical units was used to transform one string or its arrays into another.

e. Calculating Similarity
   After obtaining the minimum cost or weight, the similarity value was calculated using the following formula [19]:

$$Similarity = \left(1 - \frac{Distance[m,n]}{Max(m,n)}\right) \qquad (1)$$

Distance[m,n] represents the minimum weight obtained from the calculation to transform string or array m into string/array n using Damerau-Levenshtein Distance algorithm. Max(m,n) represents the length of the longest string or array between m and n, in addition, the similarity values are sorted in ascending order.

System Development using ANTLR

The system was developed by generating AST from the source code files using ANTLR4 library (Another Tool for Language Recognition). This library is effective for automatically creating lexers and parsers using specific language rules called grammar [28]. The grammar specified by ANTLR for Java programming language was used for this research. During lexical analysis, the lexer uses this grammar to transform character sequences into token. The parser then uses the resulting token sequence to generate CST. The detailed process of transforming CST to AST formation is shown in Figure 2.

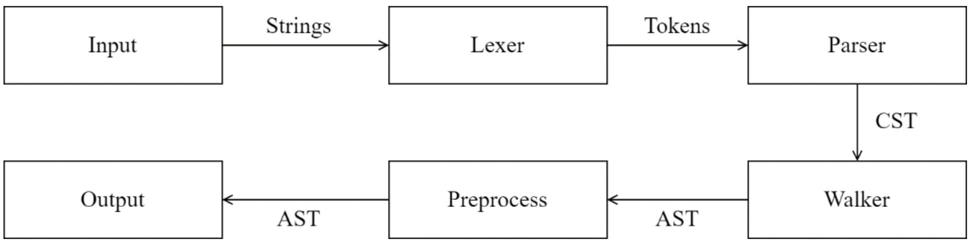

Figure 2. The process of AST formation using ANTLR library

CST is a tree structure that stores the entire representation of the source code file. It is then transformed into AST to simplify and retain only essential information. This simplification process includes removing unnecessary nodes, reorganizing and sorting subtree structures for a function [29]. The next step is the linearization process, where the tree structure is transformed into an array of ruleIndex data. The generated array is crucial for comparing the two source code files using Damerau-Levenshtein Distance algorithm. Immediately, the minimum cost was obtained, calculations were performed to determine the similarity value using a predefined formula. Subsequently, these values are sorted in ascending order, facilitating the identification of the source code file associated with plagiarism.

System Development using Electron

To facilitate the use of code plagiarism detection system, Graphical User Interface (GUI) is needed. In this research, the developed GUI is a desktop application created using the Electron framework. The electron integrates Chromium and Node.js, enabling the development of native applications using web technologies such as JavaScript, HTML, and CSS [30]. Chromium renders the application interface on the screen or windows [31], while Node.js provides essential system-level functionalities such as file access and database interactions [32]. In the Electron framework, two concurrent processes are operated, namely, the main one and the renderer. The main one is responsible for creating or destroying the renderer process windows using the BrowserWindow, where the user interface functions. The renderer process lacks direct access to Node.js modules, therefore, a bridge is needed to connect to them in the main one. This connection is facilitated through Inter-Process Communication (IPC) [33], which defines a listener, often referred to as a channel, operating based on the events exchanged between processes. An illustration of the communication process in the Electron application is shown in Figure 3.

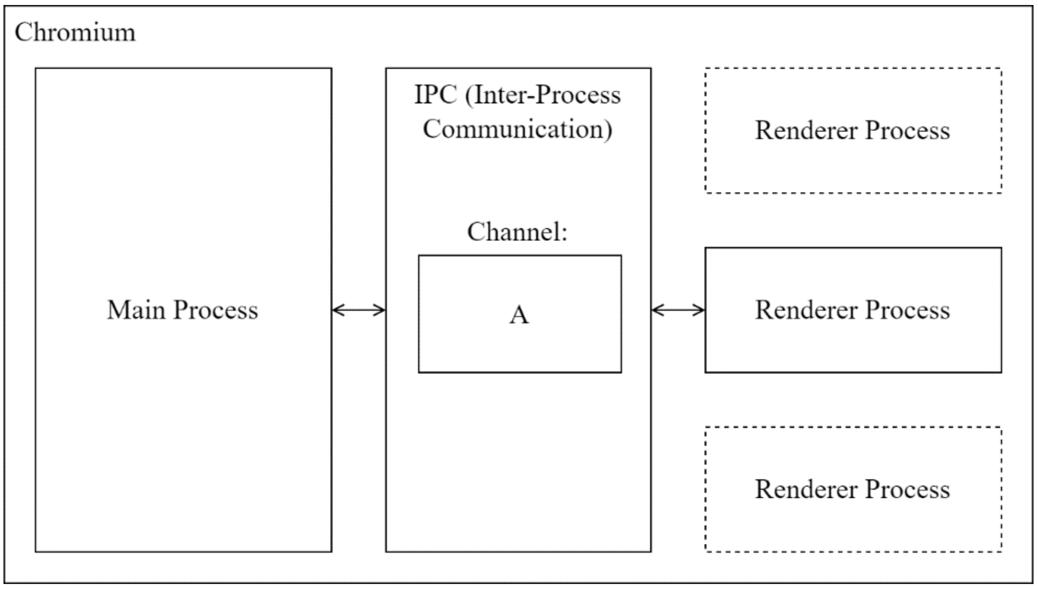

Figure 3. The chromium process in the electron application

In code plagiarism detection system, the main process orchestrates the detection procedure, which is connected to the renderer to receive input file paths for the analysis. The renderer transmits the input file paths to the primary process through a designated channel. Subsequently, the main process implores a function to conduct code plagiarism detection, producing a Map data structure containing keys with file names and corresponding similarity values. This result is then sent back to the renderer to be transformed into a table format for easier user readability. A comprehensive overview of the detection process is shown in Figure 4.

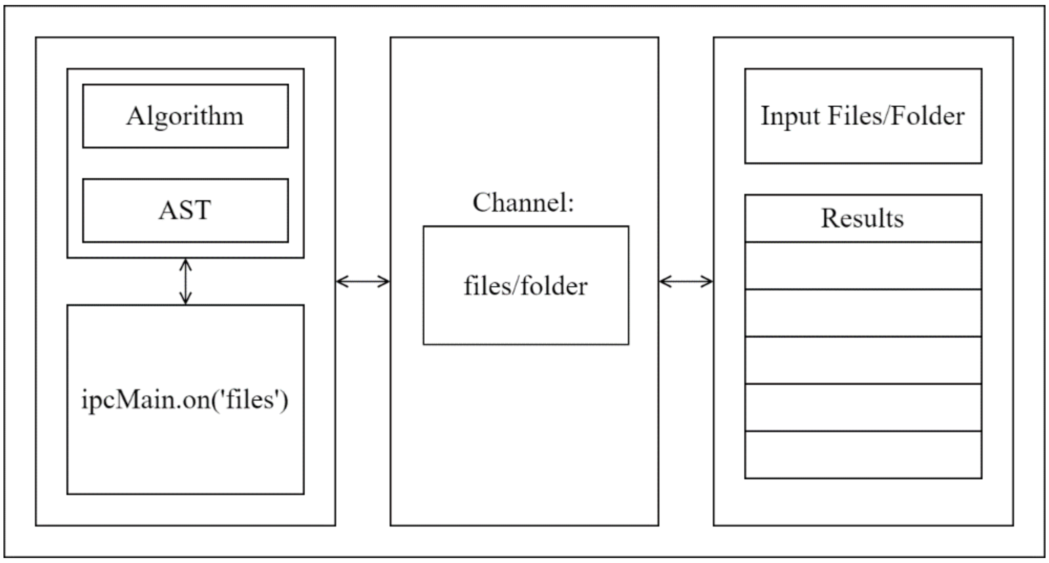

Figure 4. The flow of code plagiarism detection process

System Evaluation

The system is assessed to determine its efficacy in predicting plagiarism, and the evaluated results were presented in a Confusion Matrix table to measure the system performance. In this matrix, True Positive signifies correctly identified plagiarized data. False positive indicates circumstances where plagiarized data is incorrectly classified as original. True Negative represents accurately recognized original and correct data. False negatives denote cases where the original data was mistakenly considered as plagiarized.

From the existing Confusion Matrix table, Precision, Recall, and F-Measure values can be calculated to evaluate the performance of the system in detecting plagiarism. The precision value determines how many predictions are truly positive out of all the total positive ones. The recall value determines how many true positive predictions there are compared to the total data. Meanwhile, F-Measure calculates the average comparison between the precision and recall values.

RESULTS AND DISCUSSIONS

The accuracy of the system was tested by inputting the previously collected data. The first test used generated data, consisting of 20 source code files, with a threshold of 0.5. Subsequently, the second test used source code files from the Data Structures and Algorithms course with a threshold of 0.85. Time measurement testing was conducted to compare the efficiency of manually detecting code plagiarism by experts with that of the system. Seven source code files, equivalent to 21 pairs of files, were used for comparison. Manual detection time was calculated using a stopwatch, while the system was measured using the built-in Node.js performance.now() function.

The first accuracy test, conducted on 20 generated source code files, lasted 5.704 seconds. The results showed a significantly low system accuracy in detecting code plagiarism, reflected in an F-Measure value of 0.16. A comprehensive breakdown of the outcomes from this initial accuracy test is shown in Table 3.

Table 3. The result of the first accuracy test for generated data.

Threshold	Time	TP	FP	TN	FN	Precision	Recall	F-Measure
0.5	5.704s	5	5	137	43	0.5	0.1	0.16

The second test was conducted twice using information from the Data Structures and Algorithms course, comprising two sets of 12 source code files. The first and second sets were drawn from assessment 2 of the 2019/2020 and the 2020/2021 academic year, respectively. The processing time for the first and second tests lasted for 0.996 seconds and 0.631 seconds, respectively. The detailed results of the first accuracy test are shown in Table 4.

Table 4. The result of the second accuracy test.

Year	Threshold	Time	TP	FP	TN	FN	Precision	Recall	F-Measure
2019/2020	0.85	0.996s	2	2	60	0	0.5	1	0.67
2020/2021	0.85	0.631s	4	2	58	0	0.67	1	0.8

The results of the two tests showed a significant difference in the ability of the system to detect code plagiarism. In the first accuracy test, the system struggled to identify cheating effectively, primarily because the source code files in this test presented more diverse and complex forms of cheating compared to those in the second test. The complex nature of the cheating methods could have helped the efficiency of the system. However, in the second test, the system was able to detect code plagiarism effectively because the cheating in the source code files was less complicated.

The time processing tests showed a significant difference between manual and automatic code plagiarism detection. On average, manual processing lasted 53.18 seconds, while the system processed each pair of source code files within 0.36 seconds. The total time for manual processing was 1116.90 seconds, significantly surpassing the system processing time of 7.58 seconds.

Table 5. Difference in time for manual and automatic detection.

Source Code A	Source Code B	Manual Detection (s)	Automatic Detection (s)
2021_1.java	2021_2.java	84.66	0.39
2021_1.java	2021_3.java	13.38	0.63
2021_1.java	2021_7.java	83.22	0.11
2021_1.java	2021_9.java	181.92	0.07
2021_1.java	2021_10.java	153.60	0.55
2021_1.java	2021_11.java	61.92	0.50
2021_2.java	2021_3.java	68.88	0.10
2021_2.java	2021_7.java	66.96	0.20
2021_2.java	2021_9.java	25.98	0.69
2021_2.java	2021_10.java	16.02	0.42
2021_2.java	2021_11.java	17.94	0.09
2021_3.java	2021_7.java	24.12	0.34
2021_3.java	2021_9.java	15.60	0.19
2021_3.java	2021_10.java	17.88	0.61
2021_3.java	2021_11.java	10.98	0.68
2021_7.java	2021_9.java	77.04	0.11
2021_7.java	2021_10.java	27.00	0.30
2021_7.java	2021_11.java	24.90	0.07
2021_9.java	2021_10.java	24.00	0.83
2021_9.java	2021_11.java	30.42	0.35
2021_10.java	2021_11.java	90.48	0.36

Code plagiarism detection system testing results showed variations across different source code file sets. In respect to the three tests, the processing times ranged from a minimum of 0.631 seconds to a maximum of 5.704 seconds. Furthermore, the highest and lowest effectiveness level in code plagiarism detection, as determined by F-Measure value, are 0.8 and 0.16. This showed that the system could only detect four out of five existing cheating levels. The detailed breakdown of code plagiarism detection results is shown in Table 6.

Table 6. Code plagiarism detection results from generated source code files and the data structures and algorithms course from 2019-2021

Test Number	Total Pairs of Source Code Files Processed	Avg Lines of Code	Cheating Level	Time	F-Measure
1	190	2063	5	5.704s	0.16
2	66	300	4	0.996s	0.67

This present investigation comprehensively measured five distinct levels of similarity as opposed to previous research that only assessed accuracy at one level of similarity [17]. Smith-Waterman algorithm [12] can detect more cases of plagiarism compared to software such as MOSS and JPLAG. However, the time required to detect code plagiarism is less efficient than MOSS and JPLAG applications. The implementation of Damerau-Levenshtein Distance algorithm applied in this research significantly reduced the time needed to detect code plagiarism.

Table 7. Result comparation with previous research

Methods	Avg Time (s)	Precision	Recall	F-Measure
Smith-Waterman [12]	38.39	0.673	0.660	0.667
Damerau-Levenshtein Distance	0.36	0.670	1	0.800

Table 7 shows a comparison between the results of this research and the findings of previous ones [12] In the prior investigation, three source code files were tested, while the existing one expanded the evaluation to seven. The average processing time is 38.39 seconds and 0.36 seconds, respectively.

CONCLUSION

Based on the conducted analysis and experiments, it was concluded that the use of an automated system significantly reduced the processing time for seven source code files or 21 pairs, from 1116.90 seconds or 18.61 minutes to 7.58 seconds or 0.12 minutes for code plagiarism detection in the Data Structures and Algorithms course. F-Measure value obtained from the system detection test on generated source code files with a cheating level of five was 0.16, indicating its inability to detect code plagiarism at this level. Meanwhile, F-Measure values obtained from the system detection test on source code files from the Data Structures and Algorithms course from 2019 to 2021 with a cheating level of four were 0.8 and 0.67. This showed that the reasonable effectiveness in detecting code plagiarism at a cheating level is four.

Several recommendations for improvement were identified from the test results. First, there was a suggestion to explore hybrid-based detection methods by combining syntax and structure-based approaches aimed at enhancing the accuracy of the system in identifying code plagiarism. Additionally, the incorporation of other algorithms was recommended to further refine the system performance. Another suggestion was to include a function in the system for detecting lines of code considered templates, providing a more comprehensive approach to identifying plagiarism.

REFERENCES

[1] M. Krokoscz, "Plagiarism in articles published in journals," *Int. J. Educ. Integr.*, vol. 17, no. 1, pp. 1–22, 2021, [Online]. Available: https://doi.org/10.1007/s40979-020-00063-5

[2] G. Cosma and M. Joy, "Towards a definition of source-code plagiarism," *IEEE Trans. Educ.*, vol. 51, no. 2, pp. 195–200, 2008, doi: 10.1109/TE.2007.906776.

[3] J. Pierce and C. Zilles, "Investigating student plagiarism patterns and correlations to grades," *Proc. Conf. Integr. Technol. into Comput. Sci. Educ. ITiCSE*, pp. 471–476, 2017, doi: 10.1145/3017680.3017797.

[4] M. N. Tran, S. Marshall, and L. Hogg, "Development of doctoral student perceptions of plagiarism and academic integrity: the roles of agency and aspirational identity," *Acad. Qual. Integr. New High. Educ. Digit. Environ.*, pp. 143–162, 2023, doi: 10.1016/B978-0-323-95423-5.00006-5.

[5] L. Sun, L. Hu, and D. Zhou, "Programming attitudes predict computational thinking: Analysis of differences in gender and programming experience," *Comput. Educ.*, vol. 181, no. 27, p. 104457, 2022, doi: 10.1016/j.compedu.2022.104457.

[6] A. A. Pandit and G. Toksha, "Review of Plagiarism Detection Technique in Source Code," pp. 393–405, 2020, doi: 10.1007/978-981-15-0633-8_38.

[7] D. Gitchell and N. Tran, "Sim," *ACM SIGCSE Bull.*, vol. 31, no. 1, pp. 266–270, 1999, doi: 10.1145/384266.299783.

[8] A. Ahtiainen, S. Surakka, and M. Rahikainen, "Plaggie," *Proc. 6th Balt. Sea Conf. Comput. Educ. Res. Koli Call. 2006*, pp. 141–142, 2006, doi: 10.1145/1315803.1315831.

[9] V. T. Martins, D. Fonte, P. R. Henriques, and D. Da Cruz, "Plagiarism detection: A tool survey and comparison," *OpenAccess Ser. Informatics*, vol. 38, pp. 143–158, 2014, doi: 10.4230/OASIcs.SLATE.2014.143.

[10] A. Ahadi and L. Mathieson, "A Comparison of Three Popular Source code Similarity Tools for Detecting Student Plagiarism," *Proc. Twenty-First Australas. Comput. Educ. Conf. ACM*

(Association Comput. Mach., pp. 112–117, 2019, doi: 10.1145/3286960.3286974.

[11] M. J. Mišić, J. Protić, and M. V. Tomašević, “Improving source code plagiarism detection: Lessons learned,” *2017 25th Telecommun. Forum, TELFOR 2017 - Proc.*, vol. 2017-January, pp. 1–8, 2018, doi: 10.1109/TELFOR.2017.8249481.

[12] Y. Golubev, V. Poletansky, N. Povarov, and T. Bryksin, “Multi-threshold token-based code clone detection,” *Proc. - 2021 IEEE Int. Conf. Softw. Anal. Evol. Reengineering, SANER 2021*, pp. 496–500, 2021, doi: 10.1109/SANER50967.2021.00053.

[13] N. Kumar, “A graph based automatic plagiarism detection technique to handle artificial word reordering and paraphrasing,” *Lect. Notes Comput. Sci. (including Subser. Lect. Notes Artif. Intell. Lect. Notes Bioinformatics)*, vol. 8404 LNCS, no. PART 2, pp. 481–494, 2014, doi: 10.1007/978-3-642-54903-8_40.

[14] O. Karnalim, “IR-based technique for linearizing abstract method invocation in plagiarism-suspected source code pair,” *J. King Saud Univ. - Comput. Inf. Sci.*, vol. 31, no. 3, pp. 327–334, 2019, doi: 10.1016/j.jksuci.2018.01.012.

[15] O. Karnalim, “A Low-Level Structure-based Approach for Detecting Source Code Plagiarism,” 2019, [Online]. Available: http://orcid.org/0000-0003-4930-6249

[16] M. Duracik, E. Krsak, and P. Hrkut, “Scalable Source Code Plagiarism Detection Using Source Code Vectors Clustering,” *Proc. IEEE Int. Conf. Softw. Eng. Serv. Sci. ICSESS*, vol. 2018-November, pp. 499–502, 2018, doi: 10.1109/ICSESS.2018.8663708.

[17] L. Nichols, K. Dewey, M. Emre, S. Chen, and B. Hardekopf, “Syntax-based Improvements to Plagiarism Detectors and their Evaluations,” *Annu. Conf. Innov. Technol. Comput. Sci. Educ. ITiCSE*, pp. 555–561, 2019, doi: 10.1145/3304221.3319789.

[18] Y. Chaabi and F. Ataa Allah, “Amazigh spell checker using Damerau-Levenshtein algorithm and N-gram,” *J. King Saud Univ. - Comput. Inf. Sci.*, vol. 34, no. 8, pp. 6116–6124, 2022, doi: 10.1016/j.jksuci.2021.07.015.

[19] F. J. Damerau, “A technique for computer detection and correction of spelling errors,” *Commun. ACM*, vol. 7, no. 3, pp. 171–176, 1964, doi: 10.1145/363958.363994.

[20] A. M. Bejarano, L. E. García, and E. E. Zurek, “Detection of source code similitude in academic environments,” *Comput. Appl. Eng. Educ.*, vol. 23, no. 1, pp. 13–22, 2015, doi: 10.1002/cae.21571.

[21] N. Tahaei and D. C. Noelle, “Automated plagiarism detection for computer programming exercises based on patterns of resubmission,” *ICER 2018 - Proc. 2018 ACM Conf. Int. Comput. Educ. Res.*, pp. 178–186, 2018, doi: 10.1145/3230977.3231006.

[22] T. Sağlam, S. Hahner, J. W. Wittler, and T. Kühn, “Token-based plagiarism detection for metamodels,” *Proc. - ACM/IEEE 25th Int. Conf. Model Driven Eng. Lang. Syst. Model. 2022 Companion Proc.*, pp. 138–141, 2022, doi: 10.1145/3550356.3556508.

[23] Y. Yustikasari, H. Mubarok, and R. Rianto, “Comparative Analysis Performance of K-Nearest Neighbor Algorithm and Adaptive Boosting on the Prediction of Non-Cash Food Aid Recipients,” *Sci. J. Informatics*, vol. 9, no. 2, pp. 205–217, 2022, doi: 10.15294/sji.v9i2.32369.

[24] N. Hazimah, S. Harahap, A. Amirullah, M. B. Saputro, and I. A. Tamaroh, “Classification of potential customers using C4.5 and k-means algorithms to determine customer service priorities to maintain loyalty,” *J. Soft Comput. Explor.*, vol. 3, no. 2, pp. 123–130, 2022, doi: 10.52465/joscex.v3i2.89.

[25] W. F. Abror and M. Aziz, “Journal of Information System Bankruptcy Prediction Using Genetic Algorithm-Support Vector Machine (GA-SVM) Feature Selection and Stacking,” vol. 1, no. 2, pp. 103–108, 2023.

[26] M. B. Miles, A. M. Huberman, and J. Saldana, *Qualitative Data Analysis: A Methods Sourcebook 3rd Edition*, 3rd ed. SAGE Publications, Inc, 2014.

[27] W. Wen, X. Xue, Y. Li, P. Gu, and J. Xu, “Code Similarity Detection using AST and Textual Information,” *Int. J. Performability Eng.*, vol. 15, no. 10, pp. 2683–2691, 2019, doi: 10.23940/ijpe.19.10.p14.26832691.

[28] T. Parr, *The Definitive ANTLR 4 Reference*. Dallas, Texas: The Pragmatic Bookshelf, 2014.

[29] Y. Y. Wang, R. K. Shen, G. J. Chiou, C. Y. Yang, V. R. L. Shen, and F. P. Putri, “Novel code plagiarism detection based on abstract syntax tree and fuzzy petri nets,” *Int. J. Eng. Educ.*, vol. 1, no. 1, pp. 46–56, 2019, doi: 10.14710/IJEE.1.1.46-56.

[30] K. Kredpattanakul and Y. Limpiyakorn, “Transforming javascript-based web application to cross-platform desktop with electron,” *Lect. Notes Electr. Eng.*, vol. 514, pp. 571–579, 2019, doi: 10.1007/978-981-13-1056-0_56.

[31] R. Ollila, N. Mäkitalo, and T. Mikkonen, “Modern Web Frameworks: A Comparison of Rendering

Performance," *J. Web Eng.*, 2022, doi: 10.13052/jwe1540-9589.21311.

[32] G. Manduchi, A. Luchetta, G. Moro, A. Rigoni, and C. Taliercio, "Web-based streamed waveform display using MDSplus events and Node.js," *Fusion Eng. Des.*, vol. 157, no. January, p. 111625, 2020, doi: 10.1016/j.fusengdes.2020.111625.

[33] X. Geng, X. Zeng, L. Hu, and Z. Guo, "An Novel Architecture and Inter-process Communication Scheme to Adapt Chromium Based on Docker Container," *Procedia Comput. Sci.*, vol. 107, no. Icict, pp. 691–696, 2017, doi: 10.1016/j.procs.2017.03.149.