EPI International Journal of Engineering
Volume 5, Number 2, August 2022, pp. 81-85
DOI: 10.25042/epi-ije.082022.01

*p*ISSN 2615-5109
*e*ISSN 2621-0541

Application of JavaScript Code Similarity Detection for Assessment of Web Programming Assignment

Muhammad Niswar[a,*]

[a]Department of Informatics, Faculty of Engineering, Universitas Hasanuddin. Email: niswar@unhas.ac.id

Abstract

Students tend to copy programming assignments from their classmates in programming courses. Students copy codes in various ways, such as changing variable names and code structure order. Lecturers spend much time checking programming assignments, especially when the number of students enrolled in the course is large. They must check whether students have completed their programming assignments individually or copied their classmates' assignments. We developed a JavaScript code similarity detection application for web programming coursework using lexical analysis and Jero Winkler's Algorithm. Our application can detect the level of the students' programming assignment similarity and assist the lecturer in deciding on plagiarism.

Keywords: Similarity detection; javascript; tokenizer; Jaro-Winkler algorithm; web programming

1. Introduction

Lecturers spend much time checking the students' programming assignments for programming courses, especially if the number of students enrolled in the class is large. They must check whether students have completed their programming assignments independently or copied their classmates' assignments. Students burdened with many tasks from other courses usually tend to copy and modify the source code of their classmates so that plagiarism is not detected. Moreover, the nature of many computer science assignments is that there is an ideal solution for each question; consequently, the best answers will be highly similar [1]. To reduce student cheating in programming courses, the author in [2] proposed to change the grading policy by reducing the weight of the assessment of the programming assignment and increasing the weight of the quiz assessment. This solution may burden the lecturers with other assessments, such as quizzes and presentations, to determine whether students do their programming assignments individually.

Generally, students modify the source code by changing the lexical and the code structure. There has been some research on attempts to detect programming code similarities to assist lecturers in checking programming assignments. Reference [3] proposed a tool called CODESIGHT to detect the similarity of programming source code using modified Greedy String Tiling algorithms. The CODESIGHT analyzes a source code collection and identifies the fragments' similarities at the lexical and syntactic levels. Reference [4] proposed similarity detection using the Karp-Rabin Greedy-String-Tiling algorithm and the Winnowing algorithm for Java source code. The proposed method can detect the similarity when various lexical or structural modifications are applied to plagiarized source code. Reference [5] proposed a cross-language source similarity detection (CLCSD) based on a code flowchart and compared it with the standardized code flowchart (SCFC).

Reference [6] proposes a similarity detection technique that uses richer structural information than normal while maintaining a reasonable execution time. The technique generates the syntax trees of program code files, extracts directly connected n-gram structure tokens from them, and performs the subsequent comparisons using an algorithm from information retrieval, cosine correlation in the vector space model. Reference [7] discusses a system designed to test the independence of source codes submitted by students participating in programming competitions. It highlights the challenges in programming education and the benefits of systematic programming and competition participation. The article also addresses the issue of plagiarism and suggests an algorithm utilizing the Levenshtein edit distance and similarity to detect plagiarized code.

Reference [8] presents a method for detecting similarities in language independent source code using standard Unix filter. Reference [9] introduces an approach to identify plagiarism by analyzing the sequence of code submission made by a single student. References [10] examines several name matching techniques and provides a comparative analysis of their effectiveness. Reference [11] introduces Deckard, a tree-based approach for detecting code clones.

*Corresponding author. Tel.: +62-852-5642-8572
Jalan Poros Malino km. 6, Bontomarannu, Gowa
Sulawesi Selatan, Indonesia

Reference [12] presents a novel approach called WASTK (Weighted Abstract Syntax Tree Kernel for detecting source code plagiarism in compter science education. The approach involves converting source code into abstract syntax trees and calculating the tree kernel to determine similarity between two abstract syntax trees. Reference [13]focuses on identifying code fragments that exhibit similar API usage patterns, which can indicate potential code clones. The authors propose an efficient technique that leverages API call sequences to detect such clones without relying on detailed syntax or semantics of the code.

In this research, we developed an application to detect the similarity of JavaScript code to determine plagiarism. JavaScript is a programming language used in building web applications. Initially, Javascript was intended to build front-end applications, but now JavaScript is also used to build back-end applications, i.e., node.js. We use the JavaScript programming language to teach internet and web programming courses. In this course, we give students a programming assignment that takes much time to review to ensure that the students completed the programming assignment correctly and individually. Therefore, we developed an application to assist the lecturers in detecting the similarity of students' programming assignments.

2. Methods

We developed an application that allows students to conduct unit testing of their programming assignment before submission, and the lecturer can detect ad classify the similarity of students' Javascript programming assignments using the Jaro-Winkler algorithm. Our proposed solution uses the ESPRIMA [14] library for lexical analysis (tokenizing) and the Jaro Winkler Algorithm to check the level of similarity. Generally, programming tasks have ideal solutions so that the solutions for student programming tasks have high similarity. Therefore, we assume the student has committed plagiarism when the similarity is more than 90%. This application aims to assist lecturers in evaluating students' programming assignments.

The workflow of this application consists of four stages, as shown in Fig. 1. First, the application retrieves student assignments from the database. Each student's assignment is compared with one another.

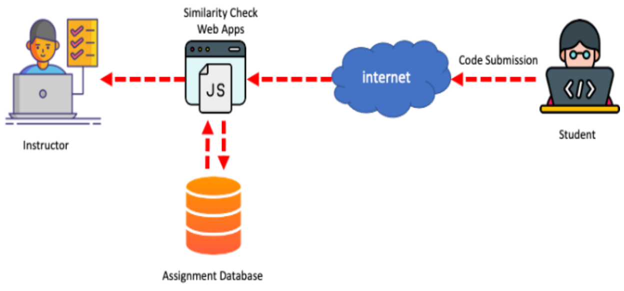


Figure 1. Code similarity check application

The application carries out a lexical analysis using the ESPRIMA method. Then it compares the results of ESPRIMA with the Jaro-Winkler algorithm and, finally, groups the data by the system. Lexical analysis and similarity detection algorithm will be explained as follows:

2.1. Lexical analysis (Tokenizer)

Lexical analysis also referred to as tokenization, transforms a series of characters, such as programming code or web pages, into a series of tokens. Tokens are strings that are identified and carry specific meanings within the context. We use ESPRIMA, a tool used to perform syntactic analysis and lexical analysis in JavaScript programs. The main function of ESPRIMA is to parse the Javascript program code. ESPRIMA will take a string value that contains a valid JavaScript program, and then from the program, and code will be made a syntax tree (syntax tree), an orderly tree that describes the syntactic structure of the program. From the results of this decomposition, the resulting syntax tree can be used for various purposes, ranging from program transformation to static program analysis.

2.2. Similarity detection algorithm

In our application, we used the Jaro-Winkler algorithm to detect the similarity of source codes. According to [9], the Jaro-Winkler algorithm performs better than other algorithms in personal name matching. Jaro-Winkler distance is an extension of the Jaro distance metric, an algorithm to measure the similarity between two strings. Usually, this algorithm is used in duplicate detection. It measures the similarity between two strings by considering both the number of matching characters and the positions of those characters. It provides a score between 0 and 1, where 0 indicates no similarity and 1 indicates an exact match. The Jaro-Winkler distance algorithm has a time complexity of quadratic runtime complexity, which is very effective on short strings and can work faster than the edit distance algorithm.

The Jaro-Winkler algorithm uses several formulas to calculate the similarity score between two strings. First, Jaro-Similarity score is calculated between two strings, s1 and s2. It calculates the length of the strings s1 and s2 and then finds the number of matching characters in the two strings being compared. It also calculates the number of transpositions, i.e., the number of adjacent characters that are out of order or swapped between two compared strings. Jaro's algorithm defines matching character as a character in both strings that are the same and characters are no exceeds the value of the following equation:

$$\left\lfloor \frac{max(|s_1|,|s_2|)}{2} \right\rfloor - 1 \tag{1}$$

Jaro's Algorithm calculate the similarity score using the following equation:

$$d_j = \frac{1}{3} \times \frac{m}{|s_1|} + \frac{m}{|s_2|} + \frac{m-t}{m} \quad (2)$$

where,
m = the matching characters of the two strings being compared
s_1 = string length 1
s_2 = string length 2
t = number of transposition
d_j = Jaro distance score between string 1 and string 2

Jaro-Winkler distance uses a prefix scale (p) which gives a higher level of assessment, and a prefix length (l) which states the length of the prefix, which is the length of the same character from the string being compared until an inequality is found. If the strings s1 and s2 are compared, then the Jaro-Winkler distance (d_w) is:

$$d_w = d_j + \left(lp\left(1 - d_j\right)\right) \quad (3)$$

where,
d_j = Jaro distance for strings s_1 and s_2
l = the length of the common prefix at the beginning of the string, the maximum value is four characters (the length of the same character before the inequality is found, max 4)
p = constant scaling factor. The standard value for this constant, according to Winkler, is p = 0.1
d_w = Jaro Winkler Distance score

For instance, let's compare two strings "HELLO" and "HLELO" using the Jaro-Winkler algorithm.

- Number of matching characters = 4 ("H","L","L", and "O")
- Number of transpositions = 1 ("E" and "L").
- Length of string1 = 5 and string2 = 5

So, we can calculate the Jaro similarity score = (4/5 + 4/5 + (4 - 1)/2) / 3 = 0.867. Then we can obtain the Jaro distance = 1 – 0.867 = 0.133. After that, it calculates the prefix scale factor by counting the number of matching characters at the beginning of the strings until a specified prefix length. The default prefix length in the Jaro-Winkler algorithm is 4. Here, the matching characters at the beginning are "H" and "L". Prefix scale factor. (p) = 0.1 * (number of matching characters at the beginning) = 0.1 * 2 = 0.2. Now, we can calculate the Jaro-winkler similarity score = 0.867 + (1 - 0.133) * 0.2 = 0.926. Therefore, the Jaro-Winkler similarity score between "HELLO" and "HLELO" is 0.926. This score indicates a relatively high similarity between the two strings, despite the transposition of the "E" and "L" characters.

2.3. Web application for similarity detection

We developed a web application for similarity detection using Hackathon Starter Pack Framework [15] to help instructor to assess the students' web programming assignments. It provides a basic foundation and structure for building web applications using JavaScript as the programming language. The Hackathon Starter Pack Framework is built using JavaScript frameworks and libraries such as Node.js, Express.js, and MongoDB. It includes pre-configured settings, file structures, and example code to help developers kickstart their projects without having to set up everything from scratch.

Algorithm 1 and 2 show the pseudocode of calculating Jaro and Jaro-Winkler Similarity score, respectively. We implemented the Jaro and Jaro-Winkler algorithms into JavaScript code. Algorithms 3 shows the pseudocode of similarity check function. In this implementation, the *similarity_check* function takes an array of student objects as input. It iterates over the students and compares the exercises' code using the Jaro-Winkler algorithm. The result is stored in the *similarTask* array, which contains objects specifying the names of the two students and their similarity scores.

ALGORITHM 1 : JARO DISTANCE

```
BEGIN PROCEDURE JARO DISTANCE (STR1, STR2)
  IF STR1 = STR2 THEN
     OUTPUT1
  ENDIF
  LEN1 ← LEN(STR1)
  LEN2 ← LEN(STR2)
  IF LEN1=0 OR LEN2=0 THEN
     OUTPUT0
ENDIF
max_dist ← floor(max(len1,len2) / 2) - 1
match ← 0
CREATE ARRAY HASH_STR1 HAVING SIZE = LEN(STR1)
CREATE ARRAY HASH_STR2 HAVING SIZE = LEN(STR1)
FOR I IN 0 TO LEN1
     FOR J IN MAX(0,I-MAX_DIST) TO MIN(LEN2, I+MAX_DIST+1)
        IF str1[I] = str2[J] and hash_str2 = 0 THEN
              hash_str1[I] ← 1
              hash_str2[I] ← 1
              BREAK
        ENDIF
     ENDFOR
ENDFOR
IF MATCH = 0 THEN
     OUTPUT0
ENDIF
T ← 0
Point ← 0
FOR I IN TO LEN1
   IF HAS_STR1[I] = 1 THEN
      WHILE(hash_str2[point] = 0)
         POINT ← POINT + 1
      DO
      IF STRING1[I] !=STRING2[POINT++] THEN
         T ← T + 1
      ENDIF
      T ← T/2
   ENDIF
ENDFOR
OUTPUT ((match / len1)+(match/len2)+((match-t)/ match)) / 3
END PROCEDURE
```

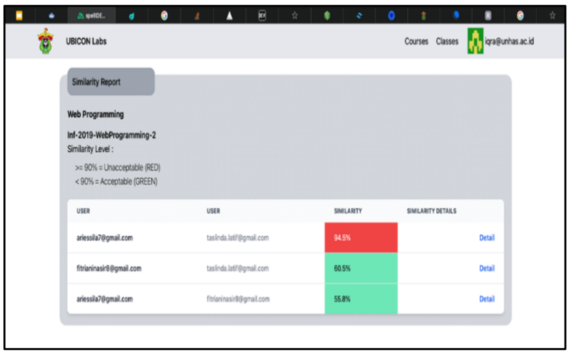
ALGORITHM 2 : JARO WINKLER

```
BEGIN PROCEDURE JAROWINKLER(STRI,STR2)
    JARO DIST JARO_DISTANCE(STR1,STR2)
    IF JARO_DIST> 0.7 THEN
        PREFIX -0
        FOR INOTO MIN(LEN(STRI), LEN(STR2))
            IF STRING] [I] = STRING2[I] THEN
                PREFIX - PREFIX+1
            ELSEIF
                BREAK
            ENDIF
        ENDFOR
        PREFIX - MIN(4,PREFIX)
        JARO_DIST - JARODIST+(0.1*PREFIX*(1-
        JARO_DIST))
    ENDIF
    OUTPUT JARO DIST
END PROCEDURE
```

ALGORITHM 3 : SIMILARITY CHECK

```
BEGIN PROCEDURE SIMILARITY CHECK(STUDENT)
    CREATE ARRAY SIMILARTASK
    FORU IN 0 TO LEN(STUDENT )
        FOR N IN (U +1) TO LEN(STUDENT )
            ANALYZECODE1 ←
            TOKENIZE(STUDENTS[U].EXERCISES)
            ANALYZECODE2 ←
            TOKENIZE(STUDENTS[N].EXERCISES)
            RESULT ←
            JARO_WINKLER(ANALYZECODE1,ANALYZ
            ECODE2)
        ENDFOR
END PROCEDURE
```

Determining the threshold of similarity at which two source codes are considered cheating is subjective and can vary depending on the context and specific guidelines set by the instructor. In this study, since the programming assignments have strict constraints and requirements that limit the possible solution approaches, the best answers will likely be more similar because they must adhere to the specified constraints. Therefore, we consider an acceptable similarity percentage is 90%. Anything beyond that is considered a high probability of cheating.



Web Programming

Inf-2019-WebProgramming-2

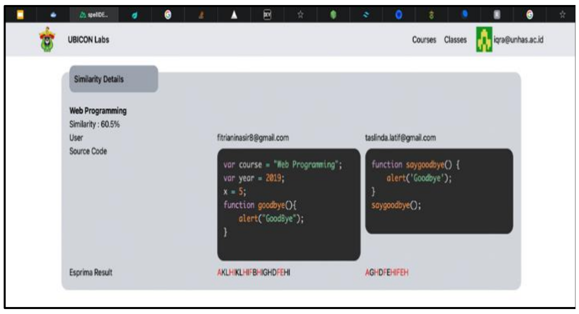
Similarity Level :

>= 90% = Unacceptable (RED)

< 90% = Acceptable (GREEN)

USER	USER	SIMILARITY	SIMILARITY DETAILS
ariessila7@gmail.com	taslinda.latif@gmail.com	94.5%	Detail
fitrianinasir8@gmail.com	taslinda.latif@gmail.com	60.5%	Detail
ariessila7@gmail.com	fitrianinasir8@gmail.com	55.8%	Detail

Figure 2. Web interface of similarity report



Web Programming

Similarity : 60.5%

User

fitrianinasir8@gmail.com

taslinda.latif@gmail.com

Source Code

```
var course = "Web Programming";
var year = 2019;
x = 5;
function goodbye(){
    alert("GoodBye");
}
```

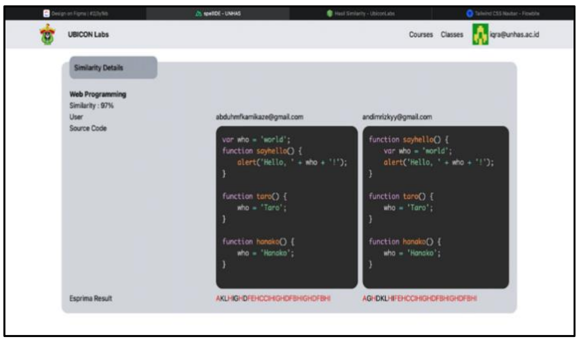
```
function saygoodbye() {
    alert('Goodbye');
}
saygoodbye();
```

Esprima Result

AKLHIKLHIFBHIGHDFEHI

AGHDFEHIFEH

Figure 3. Similarity Details with Similarity percentage of 60.5%



Web Programming

Similarity : 97%

User

abduhmfkamikaze@gmail.com

andimrizkyy@gmail.com

Source Code

```
var who = 'world';
function sayhello() {
    alert('Hello, ' + who + '!');
}

function taro() {
    who = 'Taro';
}

function hanako() {
    who = 'Hanako';
}
```

```
function sayhello() {
    var who = 'world';
    alert('Hello, ' + who + '!');
}

function taro() {
    who = 'Taro';
}

function hanako() {
    who = 'Hanako';
}
```

Esprima Result

AKLHIGHDFEHCCIHIGHDFBHIGHDFBHI

AGHDKLHIFEHCCIHIGHDFBHIGHDFBHI

Figure 4. Similarity Details with Similarity percentage of 97%

3. Results and Discussion

The application has tested on JavaScript programming assignments in a web programming class in Department of Informatics, Faculty of Engineering, Hasanuddin University. Figure 2 shows a user interface display that compares student assignments with one another and presents their similarities. Lectures can see the similarity of the code by pressing the detail button, which will display the complete code of the two students' assignments, as shown in Fig. 3 and 4. Figure 3 compares two JavaScript codes of student assignments with a similarity percentage of 60.5%. On the other hand, Figure 4 compares two JavaScript codes of student assignments with a 97% similarity percentage. These two students are considered plagiarizing if the similarity is above 90%. From the experiments, typically, students change the lexical and coding structures of the source code. Students alter variable names, function names, and comments to make the code appear different from the original. They use synonyms, abbreviations, or entirely different names for identifiers. Students might change the overall structure of the code, such as reordering or restructuring functions, loops, conditionals, or statements. This helps in making the code visually distinct from the original

4. Conclusions

The issue of students copying programming assignments from their classmates is a common occurrence in programming courses. With a large number of students enrolled in the course, manually checking each programming assignment becomes time-consuming and inefficient. To address this problem, we have developed a JavaScript code similarity detection application specifically designed for web programming coursework. Our application utilizes lexical analysis using the

ESPRIMA method and Jaro-Winkler Algorithm to assess the similarity level of students' programming assignments. By analyzing factors such as variable names and code structure order, the application can provide insights into potential cases of plagiarism. The primary objective of our application is to assist lecturers in making informed decisions regarding plagiarism. It offers a more efficient and reliable approach to identify instances of code similarity, enabling lecturers to focus their attention on potential cases that require further investigation. By automating the detection process, lecturers can allocate their time and resources more effectively, ensuring fairness and maintaining the integrity of the assessment process.

References

[1] R. Fraser and D. Cheriton, “Collaboration, Collusion, and Plagiarism in Computer Science Coursework,” *Informatics Educ.*, vol. 13, no. 2, pp. 179–195, 2014.

[2] J. Sukhodolsky, “How to Reduce Cheating in an Introductory Computer Programming Course?,” *Int. J. Comput. Sci. Educ. Sch.*, vol. 1, no. 4, 2017.

[3] A. Bejarano, L. García, and E. Zurek, “Detection of Source Code Similitude in Academic Environments,” *Comput. Appl. Eng. Educ.*, vol. 23, no. 1, pp. 13–22, 2015.

[4] Z. Đurić and D. Gasevic, “A Source Code Similarity System for Plagiarism Detection,” *Comput. J.*, vol. 56, pp. 70–86, 2013.

[5] Z. Feng, L. Guofan, L. Cong, and Q. Song, “Flowchart-Based Cross-Language Source Code Similarity Detection,” *Sci. Program.*, pp. 1–15, 2020.

[6] O. Karnalim and Simon, “Syntax Trees and Information Retrieval to Improve Code Similarity Detection,” in *ACE20: Proceedings of the Twenty-Second Australasian Computing Education Conference*, 2020, pp. 48–55.

[7] Z. Gniazdowski and M. Boniecki, “Detection of a Source Code Plagiarism in a Student Programming Competition,” *Zesz. Nauk. WWSI*, vol. 13, no. 21, pp. 74–94, 2018.

[8] J. Petrík, D. Chuda, and B. Steinmuller, “Source code plagiarism detection: The Unix way,” in *2017 IEEE 15th International Symposium on Applied Machine Intelligence and Informatics (SAMI)*, 2017.

[9] N. Tahaei and D. Noelle, “Automated Plagiarism Detection for Computer Programming Exercises Based on Patterns of Resubmission,” in *ICER18: Proceedings of the 2018 ACM Conference on International Computing Education Research*, 2018, pp. 178–186.

[10] P. Christen, “A Comparison of Personal Name Matching: Techniques and Practical Issues,” *Jt. Comput. Sci. Tech. Reports Ser.*, 2006.

[11] L. Jiang, Z. Zhang, and Z. Su, “Deckard: Scalable and accurate tree-based detection of code clones,” in *Proceedings of the 29th International Conference on Software Engineering (ICSE ’07)*, 2007, pp. 96–105.

[12] D. Fu, Y. Xu, H. Yu, and B. Yang, “WASTK: An Weighted Abstract Syntax Tree Kernel Method for Source Code Plagiarism Detection,” 2017.

[13] E. Kodhai and S. Kanmani, “Method-level code clone detection through LWH (Light Weight Hybrid) approach,” *J. Softw. Eng. Res. Dev.*, vol. 2, no. 12, pp. 1–29, 2014.

[14] Anonim, “ESPRIMA Release Master.” 2018. Accessed: Jan. 30, 2022. [Online]. Available: https://readthedocs.org/projects/esprima/downloads/pdf/latest/

[15] Sahat, “Hackathon Starter.” 2022. [Online]. Available: https://github.com/sahat/hackathon-starter.