analytics

MDPI

Article

Code Plagiarism Checking Function and Its Application for Code Writing Problem in Java Programming Learning Assistant System †

Ei Ei Htet [1], Khaing Hsu Wai [1], Soe Thandar Aung [1], Nobuo Funabiki [1,*], Xiqin Lu [1], Htoo Htoo Sandi Kyaw [2] and Wen-Chung Kao [3]

[1] Graduate School of Natural Science and Technology, Okayama University, Okayama 700-8530, Japan; pukr6nfs@s.okayama-u.ac.jp (E.E.H.); pjsu9uam@s.okayama-u.ac.jp (K.H.W.); soethandar@s.okayama-u.ac.jp (S.T.A.); pch55zhl@s.okayama-u.ac.jp (X.L.)
[2] Department of Computer and Information Science, Tokyo University of Agriculture and Technology, Tokyo 184-8588, Japan; htoohtoosk@go.tuat.ac.jp
[3] Department of Electrical Engineering, National Taiwan Normal University, Taipei 106, Taiwan; jungkao@ntnu.edu.tw
* Correspondence: funabiki@okayama-u.ac.jp
† This paper is an extended version of our paper published in 2023 11th International Conference on Information and Education Technology (ICIET), 18–20 March 2023, Fujisawa, Japan.

Abstract: A web-based *Java programming learning assistant system (JPLAS)* has been developed for novice students to study *Java programming* by themselves while enhancing *code reading* and *code writing* skills. One type of the implemented exercise problem is *code writing problem (CWP)*, which asks students to create a *source code* that can pass the given *test code*. The correctness of this answer code is validated by running them on *JUnit*. In previous works, a Python-based *answer code validation program* was implemented to assist teachers. It automatically verifies the source codes from all the students for one test code, and reports the number of passed test cases by each code in the CSV file. While this program plays a crucial role in checking the correctness of code behaviors, it cannot detect *code plagiarism* that can often happen in programming courses. In this paper, we implement a *code plagiarism checking function* in the *answer code validation program*, and present its application results to a Java programming course at Okayama University, Japan. This function first removes the whitespace characters and the comments using the *regular expressions*. Next, it calculates the *Levenshtein distance* and *similarity score* for each pair of source codes from different students in the class. If the score is larger than a given threshold, they are regarded as *plagiarism*. Finally, it outputs the scores as a CSV file with the student IDs. For evaluations, we applied the proposed function to a total of 877 source codes for 45 CWP assignments submitted from 9 to 39 students and analyzed the results. It was found that (1) CWP assignments asking for shorter source codes generate higher scores than those for longer codes due to the use of test codes, (2) proper thresholds are different by assignments, and (3) some students often copied source codes from certain students.

Keywords: Java programming learning; JPLAS; JUnit; code writing problem; plagiarism; Levenshtein distance; Python

Citation: Htet, E.E.; Wai, K.H.; Aung, S.T.; Funabiki, N.; Lu, X.; Kyaw, H.H.S.; Kao, W.-C. Code Plagiarism Checking Function and Its Application for Code Writing Problem in Java Programming Learning Assistant System. *Analytics* **2024**, *3*, 46–62. https://doi.org/10.3390/analytics3010004

Academic Editors: Ping-Feng Pai and Qingshan Jiang

Received: 5 November 2023
Revised: 16 December 2023
Accepted: 15 January 2024
Published: 17 January 2024

1. Introduction

For decades, *Java* has been widely used in a variety of practical application systems, including large enterprise systems in large companies as well as compact systems such as embedded ones. Therefore, there has been a strong need for engineers who have high *Java programming* skills in IT companies. To meet this demand, a lot of universities and professional schools have provided *Java programming* courses.

To help *Java programming* education, we have developed a web-based *Java programming learning assistant system (JPLAS)* for novice students to study *Java programming* by themselves

while enhancing *code reading* and *code writing* skills. *JPLAS* provides various exercise problems at different difficulty levels to support the learning of students at different learning stages.

For *JPLAS*, the *answer platform* has been implemented to help self-studies of students at home [1]. It is implemented on *Node.js* [2], and will be distributed to students using *Docker* [3]. The correctness of an answer to any exercise problem is verified automatically and instantly by running the automatic marking functions.

In programming education, novice students should start by solving easy problems to develop *code reading* skills. These problems may have short and simple codes that can help students learn the language rules and basic programming ideas. After gaining solid *code reading* skills, they can move to *code writing* study. If students are not able to understand source codes written by others, they will have difficulty in writing their own programs correctly. The learning goals should be gradually progressed as the understanding levels of students are improved.

To assist this progressive programming learning by novice students, *JPLAS* offers the following types of exercise problems. The *grammar-concept understanding problem (GUP)* requests to answer the important words, such as reserved words and common libraries in the programming language, in the given source code by giving the questions that describe their concepts. The *value trace problem (VTP)* requests the current values of the important variables and the output messages in the given source code. The *element fill-in-blank problem (EFP)* requests to fill in the blank elements in the given source code so that the original source code is gained. The *code completion problem (CCP)* requests to correct and complete the given source code that has several blank elements and incorrect ones. The *code writing problem (CWP)* requests to write a source code that will pass the tests in the given *test code*, where the *code testing* is applied by running both codes on *JUnit*. In every exercise problem, the correctness of any answer from a student is verified automatically in the *answer platform*. The *string matching* with the correct answer is adopted in *GUP*, *VTP*, *EFP*, and *CCP*. The *unit testing* of the answer code is applied in *CWP*.

Among these problem types, *CWP* is designed to study writing source codes from scratch that will satisfy the requested specifications in the assignment. The *answer platform* automatically runs *JUnit* with the given *test code* and the submitted source code for *code testing* when the answer submission button is clicked [1].

Previously, we implemented the *answer code validation program* in Python to help a teacher assign and mark a lot of CWP assignments to many students in their Java programming course in a university or professional school [4]. This program automatically verifies the source codes from all the students for each CWP assignment and reports the number of passed test cases by each code in the CSV file. By checking the summary of the test results of all the students in the CSV file, the teacher can easily grasp the learning progress of the students and grade them. However, although this program plays a crucial role in evaluating the correctness of code behaviors, it cannot detect *code plagiarism* that can often happen in programming courses.

In this paper, we implement a *code plagiarism checking function* in the *answer code validation program*, and present its application results to a Java programming course at Okayama University, Japan. First, this function removes the whitespace characters, such as spaces or tabs, and the comment lines using the *regular expressions*. Next, it calculates the *Levenshtein distance* and the *similarity score* for each pair of source codes from different students in the class. If the score is larger than a given threshold, these two source codes are regarded as *plagiarism*. Finally, it outputs the scores as a CSV file with the student IDs. The function will enhance the functionality of the *answer code validation program* and contribute to comprehensive and effective *Java programming* studies of novice students.

Currently, code plagiarism has become serious due to the great progress of generative AI tools such as *ChatGPT*. For CWP, students can obtain an answer source code for each assignment by submitting the test code. It is expected that *code plagiarism checking function* will detect the source codes generated by AI tools by collecting the possible ones.

For evaluations, we applied the proposed function to a total of 877 source codes for 45 CWP assignments submitted from 9 to 39 students and analyzed the results. It was found that (1) CWP assignments asking for shorter source codes generate higher scores than those for longer codes due to the use of test codes, (2) proper thresholds are different by assignments, and (3) some students often copied source codes from certain students.

The rest of this paper is organized as follows: Section 2 discusses related works in literature. Section 3 reviews our previous works of the *code writing problem*. Section 4 presents the implementation of the *code plagiarism checking function*. Section 5 discusses application results. Finally, Section 6 concludes this paper with future works.

2. Literature Review

In this section, we discuss related works in literature to this study.

2.1. Programming Education Learning Tools

Numerous studies have explored the domain of educational tools and platforms designed to enrich programming education.

In [5,6], Ala-Mutka et al. and Konecki discussed typical issues faced by beginners in programming and reviewed the current efforts and methods being used to teach programming. Many tools have been suggested to assist students in overcoming challenges in learning programming. *ToolPetcha* is an example of such a tool, serving as an automated helper for programming tasks [7].

In [8], Li et al. introduced a game-based learning environment designed to help beginners in programming. This environment uses the task of creating games to simplify the understanding of basic programming. It also incorporates methods for visualizing ideas, allowing students to interact with game objects to grasp key programming concepts.

In [9], Ünal et al. created a collaborative learning environment with a technology-focused curriculum for dynamic websites in order to learn what the students thought about the learning environment. It was a qualitative research. They gathered student opinions on the dynamic online learning environment, which promoted problem-solving cooperation through semi-structured interviews. According to their research, dynamic web page technology may enhance the learning environment in a community college context, and collaborative learning techniques focused on problem-solving can be successful.

In [10], Zinovieva et al. conducted a comparative analysis of various online platforms used for programming instruction. The study focused on selecting engaging assignments from the educational website hackerrank.com. The goal was to identify the key features of different online platforms suitable for teaching programming to aspiring computer scientists and programmers through distance learning. The researchers examined user experiences on online coding platforms (OCP) and suggested incorporating online programming simulators into computer science lessons. This recommendation takes into consideration functionality, students' readiness levels, and expected learning outcomes.

In [11], Denny et al. introduced and assessed *CodeWrite*, a web-based application that provides drills and practices for Java programming. Students have responsibility for creating activities that can be shared with their classmates. Because the tool does not employ a testing tool such as *JUnit*, validations through program testing are limited.

In [12], Shamsi et al. developed a graph-based grading system for beginner Java programming classes, termed *eGrader*. Each submitted program is dynamically analyzed using *JUnit*, and statically analyzed based on the program's graph. Experiments were conducted to validate the correctness.

In [13], Edwards et al. shared their experiences of using *test-driven development (TDD)* in conjunction with an automated grader. This paper explored the benefits and challenges of implementing *TDD* in a computer science education context and evaluated the effectiveness of using an automated grading system. The authors discussed how *TDD*, combined with automated grading, enhances student learning and provides valuable feedback on their programming assignments.

In [14], Tung et al. implemented *Programming Learning Web (PLWeb)*. It offered an *integrated development environment (IDE)* for instructors to compose exercises and a user-friendly editor for students to submit solutions. The system also included features such as visualized learning status and a plagiarism detection tool to support the teaching and learning process.

These initiatives span a wide spectrum, encompassing innovative solutions such as *ToolPetcha* [7], an automated programming assistant aiming to aid learners in navigating programming complexities. Additionally, researchers propose game-based learning environments [8], leveraging engaging tasks to simplify fundamental programming concepts. Collaborative learning environments powered by dynamic web technologies [9] offer a communal approach to problem-solving, enhancing students' understanding through collective efforts. Alongside these, platforms like *hackerrank.com* [10] serve as hubs for practical assignments, enhancing learning experiences through engaging coding challenges. Complementing these platforms are web-based tools like *CodeWrite* [11], novel grading systems such as *eGrader* [12], and *Programming Learning Web (PLWeb)* [14], each tailored to provide targeted exercises and structured assessments, contributing to a multifaceted landscape of educational aids and platforms in programming education.

2.2. Plagiarism Detection and Assessment Tools

Another cluster of research concentrates on advancing plagiarism detection and assessment tools within programming education.

In [15], Rani et al. focused on Levenshtein's edit distance, a method for comparing text documents and measuring the effort needed to change one document into another. The paper tries to make Levenshtein's edit distance algorithm work better by leaving out common words when calculating the transformation effort. The proposed system improved the execution time by removing stop words.

In [16], Ihantola et al. examined the latest advancements in automatic assessment tools for programming exercises. They discussed key features and approaches, encompassing programming languages, learning management systems, testing tools, restrictions on resubmissions, manual assessments, security measures, distributions, and specialized considerations.

In [17], Duric et al. proposed various source code similarity detection systems, including the *source code similarity detection system (SCSDS)*. They were evaluated on their abilities to detect plagiarism despite complex modifications. *SCSDS* stands out due to its customizable algorithm weights, providing users with flexibility. While promising results were observed, concerns about processing speed were noted. This study emphasizes the importance of considering code context in plagiarism detection. Future research should focus on optimizing processing speed and improving user interfaces while exploring the impact of code contexts on detection accuracy.

In [18], Ahadi et al. investigated the degree of agreement among the three popular plagiarism detection tools, namely, *Jplay*, *MOSS*, and *Sim* upon the students' C++ program source codes in a data structure and algorithms course. *SIM* has higher precision than the other two tools. It was found that integrating *SIM* and *MOSS* will be more effective for dealing with code similarity.

In [19], Novak et al. reviewed plagiarism detection tools and analyzed the effectiveness of each tool using comparison metrics and obfuscation methods with data sets for quantitative analysis and categorizations. It is described that the results will be helpful for teachers finding the right tools for similarity detection and also useful for researchers for improvements and future research.

In [20], Karnalim et al. discussed the way to improve the accuracy of code similarity detection by excluding the code segments that are unlikely to indicate plagiarism. By analyzing and identifying the code segments that can be excluded from various programming assignments, this paper aimed to enhance the accuracy of plagiarism detection in programming assignments.

In [21], Kustanto et al. proposed *Deimos*, a tool to detect plagiarism in programming assignments. Its innovative approach combines tokenization and the *Running Karp–Rabin Greedy String Tiling* algorithm, providing instructors with an efficient and language-independent tool. *Deimos* not only detects plagiarism but also contributes to improving programming education.

These studies encompass a wide array of endeavors, including comprehensive evaluations of automatic assessment tools [16] and the proposal of sophisticated source code similarity detection systems like *SCSDS* [17]. Comparative analyses among popular plagiarism detection tools [18] highlight their varying precision levels and advocate for synergistic integrations to enhance code similarity examinations. Moreover, studies explore strategies to bolster the accuracy of code similarity detection, focusing on segment exclusion methodologies [19,20]. The introduction of innovative tools such as *Deimos* [21], integrating tokenization and advanced algorithms, underscores the ongoing efforts to develop robust and efficient tools addressing the evolving challenges of detecting code plagiarism, pivotal for upholding academic integrity in programming education.

These related studies underscore the significance of understanding and addressing the unique challenges faced by programming novices, advocating for versatile and effective teaching methodologies to cultivate a robust programming education landscape. This categorization highlights the various facets of education explored in these studies, encompassing challenges in learning programming, innovative learning environments, technological platforms for teaching, collaborative learning techniques, and advancements in assessment tools and plagiarism detection methods within the programming education domain.

2.3. Discussion and Implications

This subsection critically examines the study's findings. It includes a comparative analysis of experimental results against prior research, discusses theoretical implications regarding the methodology, and explores practical implications for educators and system developers in the domain of programming education.

2.3.1. Comparative Analysis

The literature review reveals a variety of approaches to code plagiarism detection and programming education tools. In [17], the authors emphasize customizable algorithm weights in plagiarism detection systems, offering flexibility, but highlighting concerns regarding processing speed. Conversely, in [18], the authors delve into a comparative study of popular plagiarism detection tools in a C++ programming course, highlighting the effectiveness of specific tools when integrated. Moreover, in [20], the authors focus on enhancing plagiarism detection accuracy by excluding unlikely segments, emphasizing the importance of refining detection methods. These studies collectively underscore the need for adaptable and accurate detection methods in combating code plagiarism.

2.3.2. Theoretical Implications

Theoretical insights from the reviewed literature suggest multifaceted considerations in code plagiarism detection systems. In [17], the authors emphasize the pivotal role of customizable algorithm weights in detecting complex modifications, urging further exploration of code context's impact on detection accuracy. Conversely, in [21], the authors introduce *Deimos*, employing tokenization and the *Running Karp–Rabin Greedy String Tiling* algorithm, demonstrating an innovative, language-independent approach to detections. These studies underscore the significance of customizable algorithms and innovative methodologies in enhancing the theoretical underpinnings of plagiarism detection systems.

2.3.3. Practical Implications

Practical insights from the reviewed literature highlight diverse approaches to address the practical challenges of plagiarism detection and programming education tools. In [19],

the authors present the integration of different detection tools as a practical solution to enhance code similarity examinations in programming courses. Additionally, in [20], the authors emphasize the need to exclude segments unlikely to indicate plagiarism, offering a pragmatic approach to refining detection accuracy. Moreover, in [21], the authors introduce *Deimos*, a tool with practical implications for instructors, providing efficient and language-independent plagiarism detections, and enhancing programming education. These studies collectively emphasize the significance of pragmatic strategies and innovative tools in practical implementations within programming education.

By examining these papers, our proposal introduces a simple and unique method for detecting code plagiarism by utilizing *regular expressions* to streamline source codes and employing the *Levenshtein* distance for similarity scoring. We applied the proposal to a Java programming course at Okayama University, demonstrating its practicality and effectiveness in a real-world educational context. Although the *Levenshtein* distance is a useful metric for detecting plagiarism, the current proposal may have some weaknesses, such as not considering syntax or grammar.

3. Previous Works of Code Writing Problem

In this section, we discussed an overview of the *code writing problem (CWP)* and the answer platform using *Node.js* in JPLAS.

3.1. Code Writing Problem

The *code writing problem (CWP)* assignment contains a statement accompanied with *test code*, both provided by the teacher. Students are tasked with writing a source code that successfully passes all the test cases described in the test code. The correctness of the source code from a student is validated through *code testing*, utilizing *JUnit* to execute the *test code* with the source code. In order to write the correct source code, each student should refer to the detailed specifications given in the *test code*.

To generate a new assignment for *CWP*, the teacher needs to perform the following steps:

1. Create the problem statement with specifications for the assignment.
2. Make or collect the model source code for the assignment and prepare the input data.
3. Run the model source code to obtain the expected output data for the prepared input data.
4. Make the *test code* that has proper test cases using the input and output data, and add messages there to help implement the source code.
5. Register the test code and the problem statement as the new assignment.

3.2. JUnit for Unit Testing

In order to facilitate *code testing*, an open-source Java framework *JUnit* that has been designed with a user-friendly style for Java, is utilized, aligning with the *test-driven development (TDD)* approach [22]. *JUnit* helps the automatic *unit test* of a source code. Performing a test on *JUnit* is simple by using a proper "assert" method in the library. For example, the "assertEquals" method compares the output by the source code with its expected output for the given input data, and shows the result in the standard output.

3.3. Example Test Code

A *test code* is written by using the *JUnit* library. Here, the *BubbleSort* class in Listing 1 [23] is used to explain how to write the corresponding test code. This *BubbleSort* class contains the "sort(int[] a)" method for performing the *bubble sort* algorithm on the integer input array "a" and returns the sorted array.

Listing 1. Source Code 1.

```
1   package CWP;
2   public class BubbleSort {
3   public static int[] sort(int[] a) {
4           int n = a.length;
5           int temp = 0;
6           for(int i=0; i < n; i++){
7               for(int j=1; j < (n-i); j++){
8                   if(a[j-1] > arr[j]){
9                       temp = a[j-1];
10                      a[j-1] = a[j];
11                      a[j] = temp;
12               }
13            }
14       }
15          return a;
16      }
17  }
```

The *test code* in Listing 2 tests the *sort* method.

Listing 2. Test Code 1.

```
1   package CWP;
2   import static org.junit.Assert.*;
3   import org.junit.Test;
4   import java.util.Arrays;
5   public class BubbleSortTest {
6       @Test
7       public void testSort() {
8           BubbleSort bubbleSort = new BubbleSort();
9           int[] codeInput1 = {7,5,0,4,1,3};
10          int[] codeOutput = bubbleSort.sort(codeInput1);
11          int[] expOutput = {0,1,3,4,5,7};
12          try {
13              assertEquals("1:One input case:",Arrays.toString(
                    expOutput),Arrays.toString(codeOutput));
14          } catch (AssertionError ae) {
15              System.out.println(ae.getMessage());
16          }
17      }
18  }
```

This *test code* includes the three *import* statements for the *JUnit* packages at Lines 2, 3, and 4. It also declares the *BubbleSortTest* class at Line 5, which contains one test method annotated with "@Test" at Line 6. This annotation indicates that the following lines represent a test case that will be executed on *JUnit* as the following procedure:

1. Generate the *bubbleSort* object of the *BubbleSort* class in the source code.
2. Call the *sort* method of the *bubbleSort* object with the arguments for the input data.
3. Compare the output *codeOutput* of the *sort* method with the expected one *expOutput* using the *assertEquals* method.

3.4. CWP Answer Platform for Students

To assist students in solving CWP assignments efficiently, we have implemented the *answer platform* as a web application system using *Node.js*. Figure 1 illustrates the software architecture. It is noted that the OS can be *Linux*, *Windows* or *Mac*.

This platform follows the *MVC model*. For the *model (M)* part, *JUnit* is used where *Java* is used to implement the programs. The *file system* is used to manage the data where all data are provided by a file. For the *view (V)* part of the browser, *Embedded JavaScript (EJS)* is used instead of the default template engine of *Express.js*, to avoid the complex syntax structure. For the *control (C)* part, *Node.js* and *Express.js* are adopted together, where *JavaScript* is used to implement the programs.

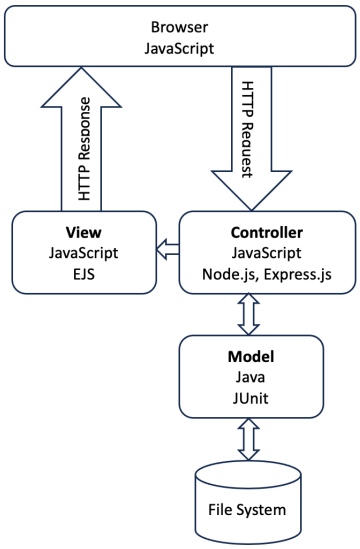

Figure 1. CWP software architecture.

Figure 2 illustrates the answer interface to solve a CWP assignment on a web browser. The right side of the interface shows the test code of the assignment. The left side shows the input form for a student to write the answer source code. A student needs to write the code to pass all the tests in the test code while looking at it. After completing the source code, the student needs to submit it by clicking the "Submit" button. Then, the code testing is immediately conducted by compiling the source code and running the test code with it on *JUnit*. The test results will appear on the lower side of the interface. It is noted that Figures 1 and 2 are adopted from a previous paper [4].

Figure 2. CWP answer interface.

3.5. Answer Code Validation Program for Teachers

The implementation of the *answer code validation program* for CWP in *JPLAS* has been implemented to help teachers. This program allows automatic testing of all the source codes from students stored in one folder for one assignment with the same test code by the following procedure:

1. Download the zip file containing the source codes for each assignment using one *test code*. It is noted that a teacher usually uses an e-learning system such as *Moodle* in the programming course.
2. Unzip the zip file and store the source code files in the appropriate folder under the "student_codes" folder within the project path.
3. Store the corresponding *test code* in the "addon/test" folder within the project directory.
4. Read each source code in the "student_codes" folder, run the test code with the source code on *JUnit*, and save the test result in the text file within the "output" folder. This process is repeated until all the source codes in the folder are tested.
5. Generate the summary of the test results for all the source codes by the CSV file and save it in the "csv" folder. The example of folder structure and related files are illustrated in Figure 3, which was adapted from [4].

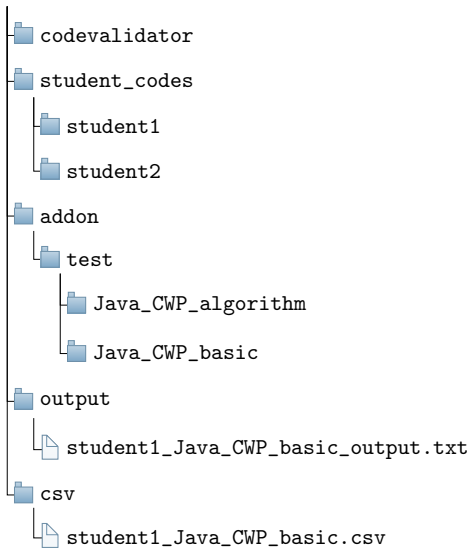

Figure 3. Example of file structures with folder hierarchy.

4. Code Plagiarism Checking Function

In this section, we present the implementation of the *code plagiarism checking function* in the *answer code validation program* for the *code writing problem* in *JPLAS*. The current program cannot detect *code plagiarism* that can often happen in programming courses. The *code plagiarism checking function* detects the code duplication or copy by calculating the *similarity score* using the *Levenshtein distance* for every pair of two source codes from different students.

4.1. Levenshtein Distance

The *Levenshtein distance*, also known as the *edit distance*, indicates the measure of the similarity between two strings or their sequences [24]. It represents the minimum number of single-character edits by insertions, deletions, or substitutions that are required to transform one string into another. The smaller the Levenshtein distance, the more similar these strings are. Then, the *similarity score* is calculated by the following equation:

$$\text{similarity score} = \left(1 - \frac{\text{Levenshtein Distance}}{\max(\text{length of string1}, \text{length of string2})}\right) \times 100 \quad (1)$$

where max(length of string1, length of string2) represents the larger length between two strings string1 and string2.

4.2. Procedure of Code Plagiarism Checking Function

The *code plagiarism checking function* that will compare the similarity between pairs of source code files and generate a CSV file containing the results, will be described in the following procedure.

1. Import the necessary Python libraries to calculate the *Levenshtein distance*, *CSV* output, and *regular expressions*.
2. Read the two files for source codes, and remove the whitespace characters such as spaces and tabs and the comment lines using the *regular expression* to make one string.
3. Calculate the *Levenshtein distance* using the *editops* function.
4. Compute the *similarity score* from the *Levenshtein distance*.
5. Repeat Steps 2–4 for all the source codes in the folder.
6. Sort the pairs in descending order of *similarity scores* using the *sorted* function and output the results in the CSV file.

4.3. Example Result

An example result by the proposed function is shown here using the source codes for *HelloWorld* class submitted by Student 1 and by Student 2. The *similarity score* for this pair is 83%.

By Student 1

```
01: package p1;
02: public class HelloWorld{
03:     public static void main(String[] args) {
04:         System.out.println("Hello World!");
05:    }
06: }
```

By Student 2

```
01: package p1;
02: public class HelloWorld{
03:  public static void main(String[] args){
04:         System.out.print ("Hello World!");
05:     }
06: }
```

4.4. Computational Complexity Analysis of Code Plagiarism Checking Function

The code plagiarism checking function implemented in this study employs the *Levenshtein* distance, which represents a measure of the similarity between two sequences, to detect code duplications or copying among student submissions. Here, we analyze the computational complexity and the efficiency of the proposed algorithm.

The core of the code plagiarism checking function is the *Levenshtein* distance algorithm. This algorithm calculates the minimum number of single-character edits of insertions, deletions, or substitutions that are required to change one string into another.

Before computing the *Levenshtein* distance, this function preprocesses the given source codes. This preprocessing involves removing the whitespace and comments, accomplished using their *regular expressions*. While the time complexity of the preprocessing varies, it generally operates in linear time relative to the length n of the input string.

Then, the code plagiarism checking function computes the *Levenshtein* distance between the strings of each pair of the source codes. The computational complexity of the

Levenshtein distance computation is given by $O(nm)$, where n and m represent the lengths of the two source codes. Therefore, the complexity of each computation depends on the length of the files being compared. However, the source codes to be checked were made by the students for the same assignment. Thus, it is possible to assume that every code has n characters. As a result, the complexity for each code pair checking would be $O(n^2)$.

The number of source code pairs is given by $k(k-1)/2$ when k students submit source codes. Therefore, the final computational complexity of the function is given by $O(k^2n^2)$.

In addition, in the revised paper, we measure the CPU time for applying the code plagiarism checking function to all the source codes for each assignment in Section 5.1. The PC environment consists of an Intel® Core™ i5-7500K CPU @ 3.40 GHz with a 64-bit Windows 10 Pro operating system. The function was implemented by Python 3.9.6.

5. Analysis of Application Results

In this section, we applied the *code plagiarism checking function* to a total of 877 source codes that were submitted from 9 to 39 students for each of the 45 CWP assignments in a Java programming course in Okayama University, Japan, and analyzed the results.

5.1. CWP Assignments

The 45 CWP assignments can be categorized into five groups, namely, *basic grammar*, *data structure*, *object-oriented programming*, *fundamental algorithms*, and *final examination*. Basically, they have different levels. Table 1 shows the group topic, the assignment title, the number of students who submitted answer source codes, lines of code (LOC), and CPU time for each assignment.

Table 1. CWP assignments for evaluations.

Group Topic	ID	Assignment Title	Number of Students	LOC	CPU Time (s)
basic grammar	1	helloworld	33	6	1.13
	2	messagedisplay	33	8	0.27
	3	codecorrection1	32	11	0.23
	4	codecorrection2	32	12	0.25
	5	ifandswitch	32	27	0.25
	6	escapeusage	32	6	0.23
	7	returnandbreak	32	18	0.25
	8	octalnumber	32	8	0.23
	9	hexadecimal	32	9	1.38
	10	maxitem	32	11	1.02
	11	minitem	31	11	1.05
data structure	12	arraylistimport	19	35	0.20
	13	linkedlistdemo	18	28	0.19
	14	hashmapdemo	17	26	0.22
	15	treesetdemo	17	32	0.11
	16	que	16	17	0.06
	17	stack	16	17	0.06
object-oriented programming	18	animal	16	18	0.06
	19	animal1	16	20	0.08
	20	animalinterfaceusage	16	29	0.41
	21	author	16	34	0.13
	22	book	16	43	0.55
	23	book1	16	24	0.08
	24	bookdata	16	40	0.11
	25	car	16	21	0.09
	26	circle	16	22	0.09
	27	gameplayer	16	13	0.27
	28	methodoverloading	16	13	0.31
	29	physicsteacher	16	25	0.08
	30	student	16	17	0.27

Table 1. *Cont.*

Group Topic	ID	Assignment Title	Number of Students	LOC	CPU Time (s)
fundamental algorithms	31	binarysearch	12	12	0.16
	32	binsort	11	20	0.19
	33	bubblesort	11	21	0.22
	34	bubblesort1	11	16	0.17
	35	divide	11	8	0.09
	36	GCD	11	19	0.13
	37	LCM	11	18	0.16
	38	heapsort	10	38	0.14
	39	insertionsort	10	23	0.16
	40	shellsort	10	28	0.19
	41	quicksort1	9	38	0.28
	42	quicksort2	9	25	0.11
	43	quicksort3	9	30	0.13
final examination	44	makearray	39	25	0.34
	45	primenumber	39	20	0.27

5.2. Analysis Results of Individual Assignments

First, we analyze the solution results of the individual assignments by the students.

5.2.1. Results for Basic Grammar

Figure 4 shows the average *similarity score* and the percentage of pairs whose *similarity score* is 100% as the identical code pair among all the source code pairs for *basic grammar*. Assignment at ID = 1 has a high average *similarity score* of 84.45%. It indicates that the source codes of most students are similar. Assignments at ID = 2 and ID = 6 also have relatively high similarity scores, which are higher than 70%. The reason is that the source codes for the assignments are short and simple and their class and method names are fixed in the test codes. Thus, variations of source codes are very limited.

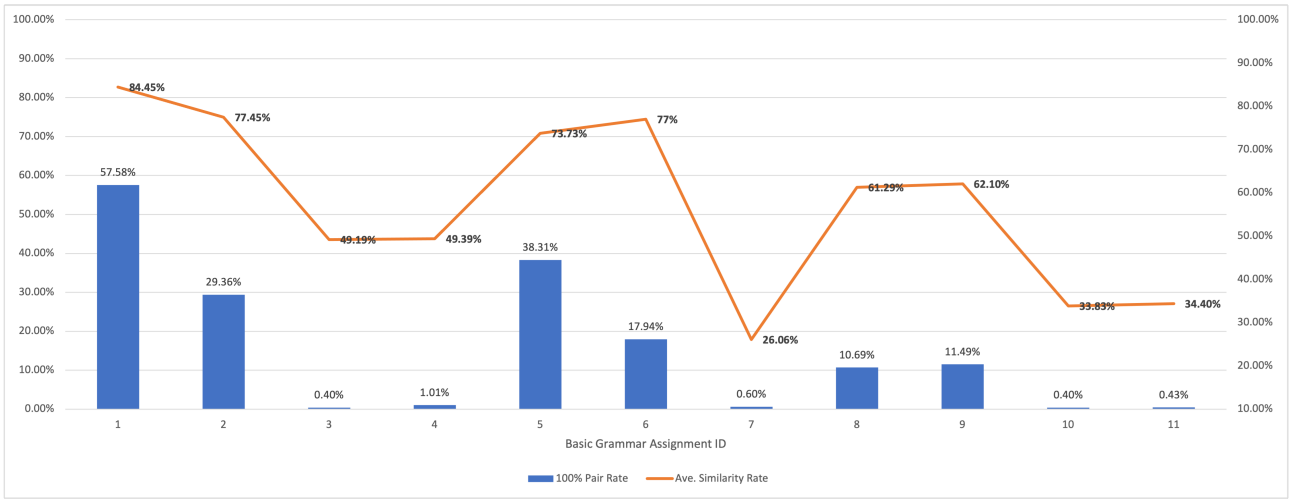

Figure 4. Results for *basic grammar*.

Table 2 shows the number of student pairs that had a 100% similarity score for each number of assignments for *basic grammar*. It suggests that one pair submitted the identical source codes for all of the 11 assignments, and another pair did the same for 10 assignments. With the high probability, these pairs submitted copied source codes. Some students often copied the source codes from certain students.

Table 2. Number of student pairs with identical codes.

Number of Assignments with Identical Codes	Number of Student Pairs
11	1
10	1
6	3
5	8
4	31
3	71
2	132
1	182

5.2.2. Results for Data Structure

Figure 5 shows the average *similarity score* and the percentage of pairs whose *similarity score* is 100% as the identical code pair among all the source code pairs for *data structure*. Assignment at ID = 15 has a high average *similarity score* of 51.18%. It indicates that the source codes of most students are similar. Assignments at ID = 17 also have relatively high similarity scores in identical code pairs. The reason is that as this data structure topic is more advanced than *basic grammar*, the assignments were challenging or the students struggled to find unique solutions.

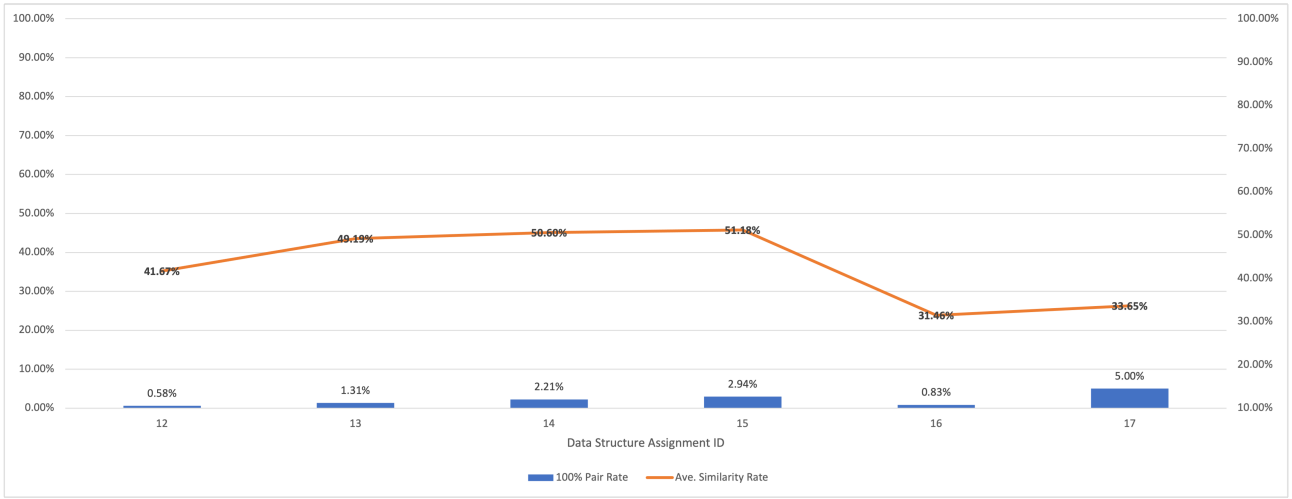

Figure 5. Results for *data structure*.

Table 3 shows the number of student pairs that had a 100% similarity score for each number of assignments for *data structure*. It suggests that one pair submitted the identical source codes for five assignments, and another pair did the same for four assignments. With the high probability, these pairs submitted copied source codes. Some students often copied the source codes from certain students.

Table 3. Number of student pairs with identical codes.

Number of Assignment with Identical Codes	Number of Student Pairs
5	1
4	1
1	8

5.2.3. Results for Object-Oriented Programming

Figure 6 shows the average *similarity score* and the percentage of pairs whose *similarity score* is 100% as the identical code pair among all the source code pairs for *object-oriented programming*. Assignment at ID = 23 has a high average *similarity score* of 64.57%. It indicates that the source codes of most students are similar. Assignments at ID = 18 and ID = 30 also have relatively high similarity scores, which are higher than 60%. The reason is that a significant portion of students submitted very similar solutions for these assignments. The absence of identical submissions in most assignments is a positive sign that students tried different source codes.

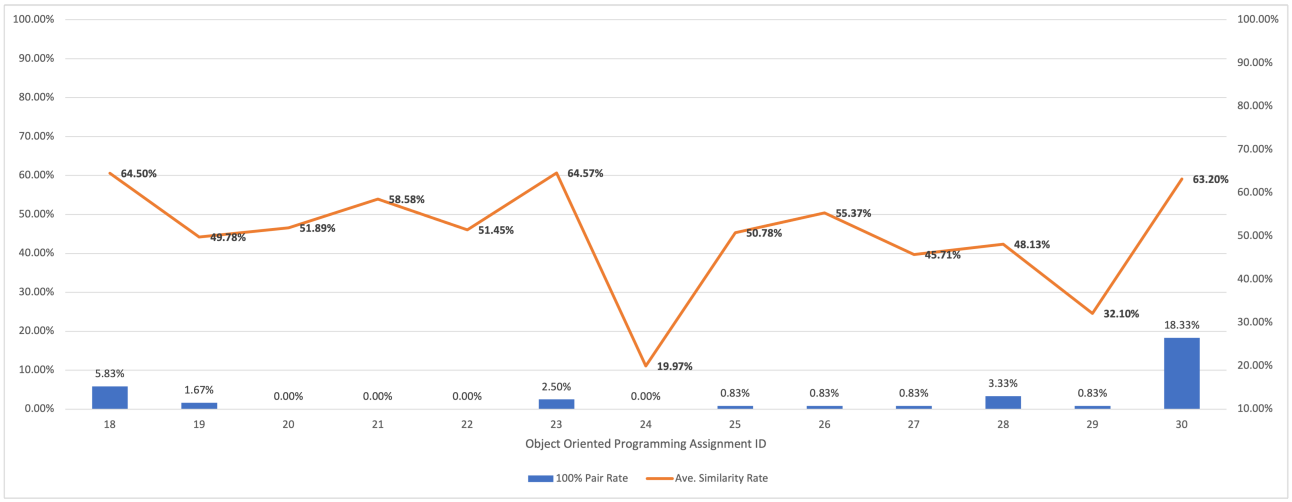

Figure 6. Results for *object-oriented programming*.

5.2.4. Results for Fundamental Algorithms

Figure 7 shows the average *similarity score* and the percentage of pairs whose *similarity score* is 100% as the identical code pair among all the source code pairs for *fundamental algorithms*. Assignment at ID = 35 has a high average *similarity score* of 49.58%. It indicates that the source codes of most students are similar. Although fewer students submitted these assignments, the low similarity rates and absence of identical submissions in most assignments suggest that students likely tackled these fundamental algorithm problems independently. These assignments may have been sufficiently challenging, encouraging diverse solutions.

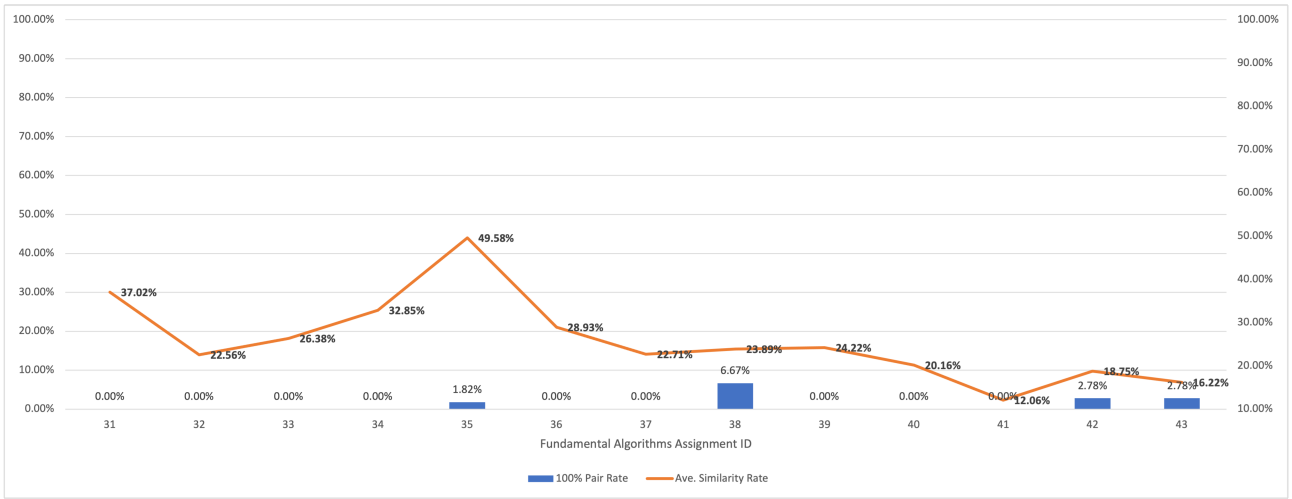

Figure 7. Results for *fundamental algorithms*.

5.2.5. Results for Final Examination

Figure 8 shows the average *similarity score* and the percentage of pairs whose *similarity score* is 100% as the identical code pair among all the source code pairs for *final examination*. Assignment at ID = 45 has the average *similarity score* of 26.89%. It indicates that the source codes of most students are similar. Assignment at ID = 44 has a high average *similarity score* of 21.37%. Both final examination assignments have relatively low average similarity rates. It indicates that students' solutions to these assignments were not highly similar. Moreover, the 0.0% in the identical code pair shows that there were no identical submissions for either of these assignments, which is a positive sign in a final examination.

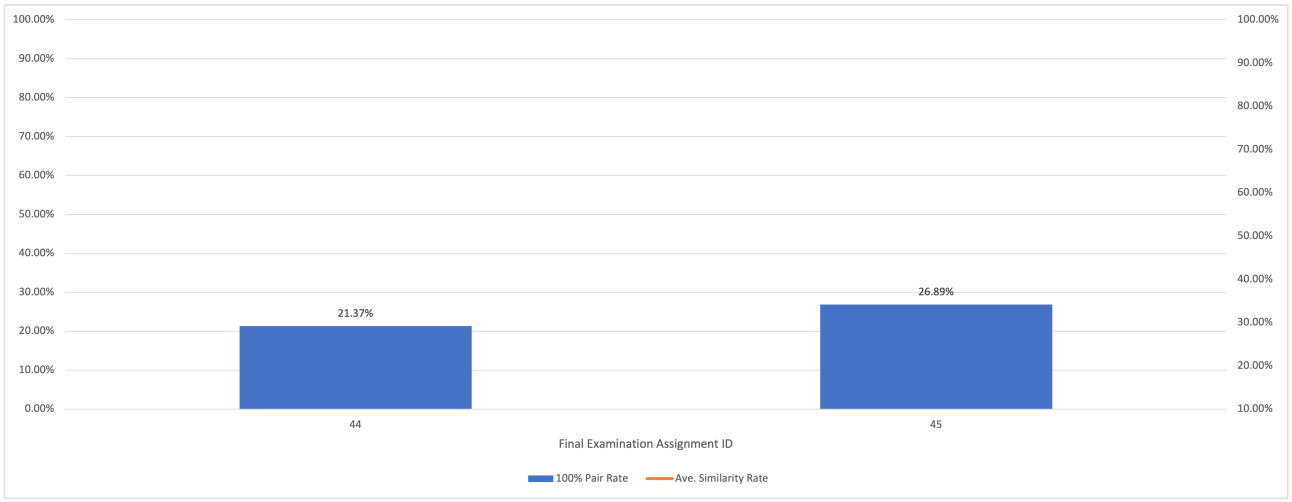

Figure 8. Results for *final examination*.

5.3. Analysis Results of Assignment Group

Next, we analyze the solution results by each group. Table 4 shows the total number of source code submissions, the total number of assignments, the average similarity score, and the identical code percentage among all the student pairs in each group. It indicates that *basic grammar* has the highest average similarity score of 57.17%, and *final examination* has the lowest one. The assignments in *basic grammar* ask for short and simple source codes. The assignments in *final examination* ask for more complex and long source codes.

Fortunately, the rate of identical source codes is very low in the four groups other than *basic grammar*. It becomes zero in *final examination*, which suggests no cheating was done in this online examination. Basically, most of the students seriously solved the assignments by themselves.

When the source codes among the assignments are compared, it can be found that the ones with high similarity scores do not need to use conditions or loops. Since the class names, the method names, and the data types are basically fixed by the given test codes, the answer source codes can be identical or highly similar to each other. Therefore, for the automatic detection of code plagiarism by the proposed function, the threshold needs to be adjusted properly by considering the features of each assignment. The formula will be in future works.

The *CPU time* for each group will be also discussed in Table 4. The CPU time for each section seems to correlate more with the number of submissions and assignments rather than the complexity of the tasks themselves. This suggests that the volume of data plays a significant role in the computational resources required for plagiarism detection and analysis in this study.

Table 4. Number of source codes and results in each group.

Group Topic	Number of Source Codes	Number of Assignments	Ave. Similarity Score	100% Pair Rate	CPU Time (s)
basic grammar	353	11	57.17	15.29	6.29
data structure	103	6	42.96	2.15	0.84
object-oriented programming	208	13	50.46	2.69	2.53
fundamental algorithms	135	13	25.79	1.08	2.13
final exam	78	2	24.13	0.00	0.61

6. Conclusions

This paper presented the *code plagiarism checking function* in the *code validation program*. It removes the whitespace characters and the comment lines using *regular expressions*, and calculates the *similarity score* from the *Levenshtein distance* between every pair of two source codes from students. If the score is larger than a given threshold, they are regarded as *plagiarism*. The results are output in the CSV file. For evaluations, we applied the proposal to a total of 877 source codes for 45 CWP assignments from 9 to 39 students and analyzed the results. The results confirm the validity and effectiveness of the proposal.

We also applied this *code plagiarism checking function* to this year's Java programming class. Although we informed the students to avoid copy from each other, we still found that 4 students submitted copied source codes for some assignments among 55 students. In future works, we will assign new assignments to students in Java programming courses, and apply the proposal to them. We will also study the *coding rule checking function* to improve the readability and efficiency of the codes.

Author Contributions: Methodology, S.T.A.; Validation, X.L.; Writing—original draft, E.E.H. and K.H.W.; Writing—review and editing, E.E.H., K.H.W., N.F. and H.H.S.K.; Supervision, N.F. and W.-C.K. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Data Availability Statement: All data is contained within the manuscript.

Conflicts of Interest: The authors declare no conflicts of interest.

References

1. Aung, S.T.; Funabiki, N.; Aung, L.H.; Htet, H.; Kyaw, H.H.S.; Sugawara, S. An implementation of Java programming learning assistant system platform using Node.js. In Proceedings of the International Conference on Information and Education Technology, Matsue, Japan, 9–11 April 2022; pp. 47–52.
2. Node.js. Available online: https://nodejs.org/en (accessed on 4 November 2023).
3. Docker. Available online: https://www.docker.com/ (accessed on 4 November 2023).
4. Wai, K.H.; Funabiki, N.; Aung, S.T.; Mon, K.T.; Kyaw, H.H.S.; Kao, W.-C. An implementation of answer code validation program for code writing problem in java programming learning assistant system. In Proceedings of the International Conference on Information and Education Technology, Fujisawa, Japan, 18–20 March 2023; pp. 193–198.
5. Ala-Mutka, K. *Problems in Learning and Teaching Programming. A Literature Study for Developing Visualizations in the Codewitz-Minerva Project*; Tampere University of Technology: Tampere, Finland, 2004; pp. 1–13.
6. Konecki, M. Problems in programming education and means of their improvement. In *DAAAM International Scientific Book*; DAAAM International: Vianna, Austria, 2014; pp. 459–470.
7. Queiros, R.A.; Peixoto, L.; Paulo, J. PETCHA—A programming exercises teaching assistant. In Proceedings of the ACM Annual Conference on Innovation and Technology in Computer Science Education, Haifa, Israel, 3–5 July 2012; pp. 192–197.
8. Li, F.W.-B.; Watson, C. Game-based concept visualization for learning programming. In Proceedings of the ACM Workshop on Multimedia Technologies for Distance Learning, Scottsdale, AZ, USA, 1 December 2011; pp. 37–42.
9. Ünal, E.; Çakir, H. Students' views about the problem based collaborative learning environment supported by dynamic web technologies. *Malays. Online J. Edu. Tech.* **2017**, *5*, 1–19.

10. Zinovieva, I.S.; Artemchuk, V.O.; Iatsyshyn, A.V.; Popov, O.O.; Kovach, V.O.; Iatsyshyn, A.V.; Romanenko, Y.O.; Radchenko, O.V. The use of online coding platforms as additional distance tools in programming education. *J. Phys. Conf. Ser.* **2021**, *1840*, 012029. [CrossRef]
11. Denny, P.; Luxton-Reilly, A.; Tempero, E.; Hendrickx, J. CodeWrite: Supporting student-driven practice of Java. In Proceedings of the ACM Technical Symposium on Computer Science Education, Dallas, TX, USA, 9–12 March 2011; pp. 471–476.
12. Shamsi, F.A.; Elnagar, A. An intelligent assessment tool for student's Java submission in introductory programming courses. *J. Intelli. Learn. Syst. Appl.* **2012**, *4*, 59–69.
13. Edwards, S.H.; Pérez-Quiñones, M.A. Experiences using test-driven development with an automated grader. *J. Comput. Sci. Coll.* **2007**, *22*, 44–50.
14. Tung, S.H.; Lin, T.T.; Lin, Y.H. An exercise management system for teaching programming. *J. Softw.* **2013**, *8*, 1718–1725. [CrossRef]
15. Rani, S.; Singh, J. Enhancing Levenshtein's edit distance algorithm for evaluating document similarity. In Proceedings of the International Conference on Computing, Analytics and Networks, Singapore, 27–28 October 2018; pp. 72–80.
16. Ihantola, P.; Ahoniemi, T.; Karavirta, V.; Seppälä, O. Review of recent systems for automatic assessment of programming assignments. In Proceedings of the 10th Koli Calling International Conference on Computing Education Research, New York, NY, USA, 28 October 2010 ; pp. 86–93.
17. Duric, Z.; Gasevic, D. A source code similarity system for plagiarism detection. *Comput. J.* **2013**, *56*, 70–86. [CrossRef]
18. Ahadi, A.; Mathieson, L. A comparison of three popular source code similarity detecting student plagiarism. In Proceedings of the Twenty-First Australasian Computing Education Conference, Sydney, Australia, 29–31 January 2019; pp. 112–117.
19. Novak, M.; Joy, M.; Keremek, D. Source-code similarity detection and detection tools used in academia: A systematic review. *ACM Trans. Comp. Educ.* **2019**, *19*, 1–37. [CrossRef]
20. Karnalim, S.O.; Sheard, J.; Dema, I.; Karkare, A.; Leinonen, J.; Liut, M.; McCauley, R. Choosing code segments to exclude from code similarity detection. In Proceedings of the Working Group Reports on Innovation and Technology in Computer Science Education, Trondheim, Norway, 17–18 June 2020; pp. 1–19.
21. Kustanto, C.; Liem, I. Automatic source code plagiarism detection. In Proceedings of the 10th ACIS International Conference on Software Engineering, Artificial Intelligences, Networking and Parallel/Distributed Computing, Daegu, Republic of Korea, 27–29 May 2009; pp. 481–486.
22. JUnit. Available online: https://en.wikipedia.org/wiki/JUnit (accessed on 4 November 2023).
23. Bubble Sort. Available online: https://en.wikipedia.org/wiki/Bubble_sort (accessed on 4 November 2023).
24. Levenshtein Distance. Available online: https://en.wikipedia.org/wiki/Levenshtein_distance (accessed on 4 November 2023).