An Implementation of Code Plagiarism Checking Function for Code Writing Problem in Java Programming Learning Assistant System

March, 2024

Ei Ei Htet

Graduate School of
Natural Science and Technology

(Doctor's Course)
Okayama University

Dissertation submitted to
Graduate School of Natural Science and Technology
of
Okayama University
for
partial fulfillment of the requirements
for the degree of
Doctor of Philosophy.

Written under the supervision of

Professor Nobuo Funabiki

and co-supervised by
Professor Satoshi Denno
and
Professor Yasuyuki Nogami

Okayama University, March 2024.

To Whom It May Concern

We hereby certify that this is a typical copy of the original doctor thesis of
EI EI HTET

Signature of

the Supervisor

Prof. Nobuo Funabiki

Seal of

Graduate School of

Natural Science and Technology

Abstract

As the reliable and portable object-oriented programming language, *Java programming* has been extensively used to implement a variety of practical systems. A large number of universities and professional schools are offering *Java programming* courses to meet these needs. To assist Java programming education in schools, we have developed the Web-based *Java Programming Learning Assistant System (JPLAS)*. JPLAS offers various types of exercise problems at different levels such that the correctness of any answer from a student is automatically verified. Among them, the *code writing problem (CWP)* asks a student to write a source code to pass the given *test code* where the correctness is verified by running them on *JUnit*. Unfortunately, some students copied the answer source codes made by other students and submitted them to the teacher. This code plagiarism needs to be found to avoid them.

In this thesis, I implemented the *code plagiarism checking function* for the *code writing problem (CWP)* in *Java Programming Learning Assistant System*. 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 the CSV file with the student IDs. The results confirm the effectiveness of *Code Plagiarism Checking Function* in detecting the answers of the students who attempt to solve CWP problems while studying Java programming after solving many simple *JPLAS* problems.

Recently, *Python programming* has gained the popularity for use in various groundbreaking fields in experiments, prototyping, embedded systems, and data sciences, due to rich libraries and short coding features. A lot of people in both IT and non-IT fields will start using it to achieve their needs. Then, high-quality learning tools of *Python programming* have been highly demanded, especially for self-study, since many people have no opportunities of taking the courses at schools.

Therefore, in the thesis, I implemented the *Grammar-Concept Understanding Problem (GUP)* to be able to understand the basic grammar keywords with their respective definitions for novice students who are learning *Python Programming*. A GUP instance asks for the knowledge and understanding of the keywords with their intended definitions. A GUP instance consists of a source code and a set of questions describing the definitions of grammar or library keywords appearing in the code. The correctness of the answer from a student is marked through string matching with the correct keyword. The goal of GUP is to give students opportunities to profoundly code reading with a clear understanding of its grammar meaning. The results show that the proposal is effective in improving the performance of the students who are novices in *Python Programming*.

In future works, I will apply the *code plagiarism checking function* to other programming languages such as C and Python. I will also generate a variety of GUP questions for the advanced topics, useful libraries and apply them to students.

Acknowledgements

It is my great pleasure to thank those people who have supported and encouraged me throughout this Ph.D. study at Okayama University, Japan. It would not be possible to complete this thesis without their help. I want to say many things, but I can hardly find the proper words. Therefore, I will just say that you are the greatest blessing in my life.

I owe my deepest gratitude to my supervisor, Professor Nobuo Funabiki, who has supported me throughout my thesis with his patience and knowledge. I am greatly indebted to him, whose encouragement, advice, and support from the beginning to the end enabled me to proceed with this study, not only in scientific issues but also in life. He gave me wonderful advice, comments, and guidance when formulating problems, writing papers, and presenting them. Thanks for making me who I am today.

I am deeply grateful to my co-supervisors, Professor Satoshi Denno and Professor Yasuyuki Nogami, for their continuous support, guidance, mindful suggestions, and proofreading of this work.

I also want to express my gratitude to the course teachers during my Ph.D. study for enlightening me with wonderful knowledge. I would like to acknowledge the Monbukagakusho Honors Scholarship (JASSO Scholarship) for financially supporting my Ph.D. study. I would like to thank my friends and colleagues who helped me in this study, including Dr. Htoo Htoo Sandy Kyaw, Dr. San Hay Mar Shwe, Dr. Hein Htet, Ms. Soe Thandar Aung, Ms. Khaing Hsu Wai, Ms. Shune Lae Aung and Mr. Lynn Htet Aung all the FUNABIKI Lab's members. Thank you for your support during my tough time during this study and thank you for sharing the thoughts and experiences with me.

Finally, I am eternally grateful to my beloved family and teachers, who always encourage and support me throughout my life. Your support and understanding gave me the strength and inspiration to overcome any difficulty in my life.

Ei Ei Htet
Okayama, Japan
March 2024

List of Publications

Journal Papers

1. **Ei Ei Htet**, Khaing Hsu Wai, Soe Thandar Aung, Nobuo Funabiki, Xiqin Lu, Htoo Htoo Sandi Kyaw, and Wen-Chung Kao, "Code Plagiarism Checking Function and Its Application for Code Writing Problem in Java Programming Learning Assistant System", Analytics, vol. 3, no. 1, pp. 46-62, January 2024.

International Conference Papers

2. **Ei Ei Htet**, San Hay Mar Shwe, Soe Thandar Aung, Nobuo Funabiki, Evianita Dewi Fajrianti, and Sritrusta Sukaridhoto, "A study of grammar-concept understanding problem for Python programming learning," 2022 IEEE 4th Global Conference on Life Sciences and Technologies (LifeTech 2022), pp. 245-246 (Osaka, Japan, 2022).

List of Figures

List of Tables

Contents

Chapter 1

Introduction

1.1 Background

As the reliable and portable object-oriented programming language, *Java programming* has been extensively used to implement a variety of practical systems. The usage of Java has involved mission-critical systems for both large-scale enterprises and small-sized embedded systems. Therefore, there have been strong demands in IT companies for the cultivation of Java programming engineers. A large number of universities and professional schools are offering *Java programming* courses to meet these needs.

To assist Java programming education in schools, we have developed the web-based *Java programming learning assistant system (JPLAS)* as a self-study tool for learning Java programming. The web server of JPLAS adopts *Node.js* for the web application server, JavaScript, Java, and Python for application programs. The file system is used for handling the problem data and the students' data. The user can access to JPLAS through a web browser. The programs of JPLAS will be distributed to students using *Docker*. By installing it in their computers, students can access to JPLAS even if no Internet access is available.

In JPLAS, there are several types of exercise problems, such as the *grammar concept understanding problem (GUP)* for studying keyword definitions [1], the *value trace problem (VTP)* for reading value changes of important variables in the source code [2], the *element fill-in-blank problem (EFP)* for studying partial source code writing [3], and the *code writing problem (CWP)* for studying full source code writing [4], which can cover various students at different learning levels. The correctness of any answer is automatically checked in the system, and the result will be automatically returned to the learner so that he/she can instantly correct them. Among them, the *code writing problem (CWP)* [4] asks a student to write a source code to pass the given *test code* where the correctness is verified by running them on *JUnit*.

Unfortunately, some students copied the answer source codes made by other students, and submitted them to the teacher. This *code plagiarism* needs to be found to avoid them. Previously, we implemented the *answer code validation program* in Python to help teachers. It automatically verifies the source codes for one test code from all the students, 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 the *code plagiarism* that can often happen in programming courses.

1.2 Contributions

In this thesis, as the first contribution of the thesis, I propose the *code plagiarism checking function* for the *code writing problem* in JPLAS. This function first removes the whitespace characters and the comments using the regular expressions. Next, it calculates the *Levenshtein* distance between the two source codes from each pair of the students in the class. Third, it calculates the similarity score from the distance. If the score is larger than a given threshold, they are regarded as plagiarism. Finally, it outputs the scores in the 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 $9-39$ 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.

In the second contribution, I implement the *grammar-concept understanding problem (GUP)* for studying basic grammar concepts of *Python Programming* by novice students. A GUP instance asks for the knowledge and understanding of the keywords with their intended definitions. A GUP instance consists of a source code and a set of questions describing the definitions of grammar or library keywords appearing in the code. The correctness of the answer from a student is marked through string matching with the correct keyword.

For evaluations, we generate 24 GUP instances with 139 questions and assign them to 9 students in Okayama University. The results show that the proposal is effective in revealing the student understanding levels.

1.3 Contents of This Dissertation

The remaining part of this thesis is organized as follows: Chapter 2 reviews the overview of Java programming learning assistant system with the server platform, the software architecture, and the implemented problem types. Chapter 3 presents the code plagiarism checking function for the *code writing problem (CWP)* in JPLAS. Chapter 4 presents the *grammar-concept understanding problem (GUP)* for Python programming. Chapter 5 reviews relevant works in literature. Finally, Chapter 6 concludes this thesis with some future works.

Chapter 2

Review of Java Programming Learning Assistance System (JPLAS)

In this chapter, we introduce the outlines of Java Programming Learning Assistant System (JPLAS) and Python Programming Learning Assistant System (PyPLAS).

2.1 JPLAS Overview

Firstly, we overview the server platform, the software architecture, and the implemented problem types of JPLAS.

2.1.1 Server Platform

Originally, *JPLAS* was implemented using *JSP* with *Java* 1.6.2 as the web application on a server. It adopts *Ubuntu-Linux 10.04* as the operating system running on *VMware* for portability. *Tomcat 6.0.26* is used as the web application server to run *JSP* source codes that is a script language with embedding *Java* codes within *HTML* codes. *Tomcat* returns the dynamically generated web pages to the client web browser. *MySQL 5.0.27* is adopted for managing the data in *JPLAS*. Figure 2.1 illustrates the server platform of *JPLAS* [6].

2.1.2 Software Architecture

The software architecture of *JPLAS* follows the MVC model as the common architecture of the web application system. It basically uses *Java* for the Model (M), *HTML/CSS/JavaScript* for the View (V), and *JSP* for the controller (C).

The system implements the logic functions of *JPLAS* using *Java*. For the independence from the view and controller, any input/output to/from the model uses a string or its array that does not contain *HTML* tags. *Servlet* is not used to avoid the possible redundancy that could happen between *Java* codes and *Servlet* codes where the same function may be implemented. A design pattern called *Responsibility Chain* is adopted to handle marking functions of the student answers, and the specific functions for the database access are implemented such that the controller does not handle them.

The view implements the user interfaces of *JPLAS* by using a *CSS* framework to provide integrated interfaces using *Cascading Style Sheet (CSS)* in the web standard. The user interface is

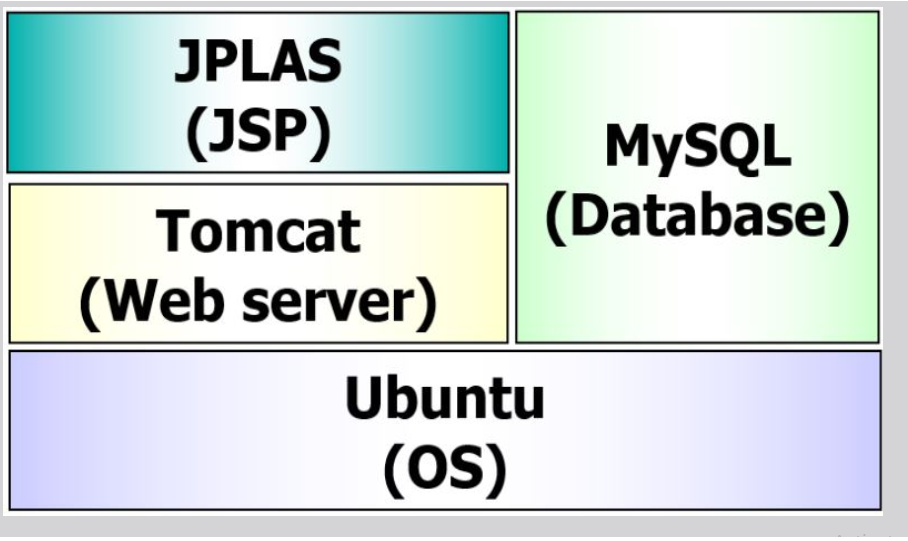

Figure 2.1: Server platform for JPLAS.

dynamically controlled with *Ajax* to reduce the number of *JSP* files.

For the control architecture, the control in *JPLAS* is implemented by *JSP*. When it receives a request from the view, it sends it to *Java* in the model and requests the corresponding process. When *Java* in the model returns the processing results by strings, the control changes the format for the view use *HTML*. The procedure is elaborated as follows:

1) to show the assignment list in the view, *JSP* in the control receives the list with strings in the two dimensional array, changes them into the table format in *HTML*, and sends them to *JavaScript* in the view,

2) to demonstrate the selected assignment in the view, *JSP* receives the details with strings, changes them into the table in *HTML*, and sends it to *JavaScript*, and

3) to mark the answers from the student, *JSP* receives them from *JavaScript* in the view and sends them to *Java* in the model. After completing the marking in the model, *JSP* receives the marking results from *Java*, changes them into the table format in *HTML*, and sends it to *JavaScript* in the view [5].

The overall software architecture in *JPLAS* can be seen in Figure 2.2.

2.1.3 Implemented Problem Types

Currently, *JPLAS* has several types of exercise problems to accommodate a variety of students at difference learning levels. Problem types in *JPLAS* are as follows:

1) *Grammar Concept Understanding Problem (GUP)*: This problem instance consists of a source code and a set of questions on grammar concepts or behaviors of the code. Each answer can be a number, a word, or a short sentence, whose correctness is marked through string matching with the correct one. The algorithm is implemented to automatically generate a *GUP* instance from a given source code by 1) extracting the registered keywords in the source code, 2) selecting the registered question corresponding to each extracted keyword, and 3) detecting the data required in the correct answer from the code [1].

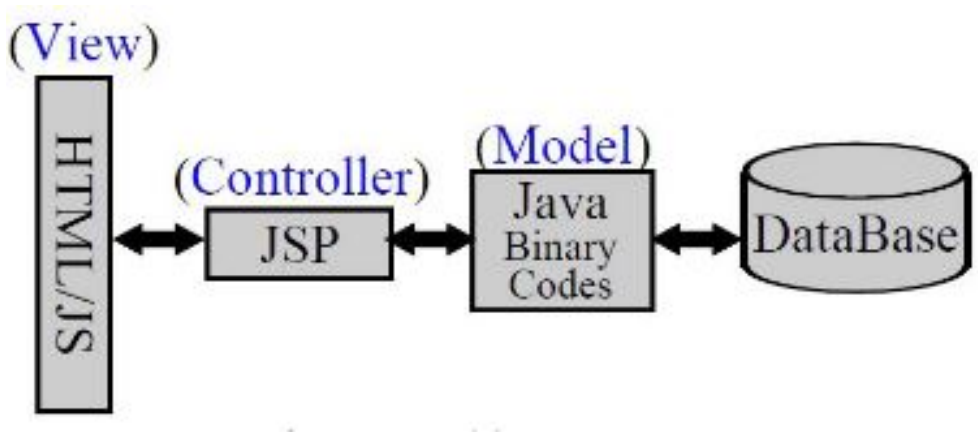

Figure 2.2: Software architecture for JPLAS.

2) *Value Trace Problem (VTP)*: This problem requires students to trace the actual values of important variables in a code when it is executed. The correctness of the answers is also marked by comparing them with their correct ones stored in the server [2].

3) *Element Fill-in-blank Problem (EFP)*: This problem requires students to fill in the blank elements in a given *Java* code. The correctness of the answers is marked by comparing them with their original elements in the code that are stored in the server. The original elements are expected to be the unique correct answers for the blanks [7]. To help a teacher to generate a feasible element fill-in-blank problem, the blank element selection algorithm has been proposed [8].

4) *Statement Fill-in-blank Problem (SFP)*: This problem asks students to fill in the blank statements in a code. The correctness of the code is marked by using the test code on *JUnit* that is an open source software for the *test-driven development (TDD)* method [**?**]. To help a teacher select blank statements from a code, the *program dependency graph (PDG)* has been used [10].

5) *Code Writing Problem (CWP)*: This problem asks students to write a whole code from scratch that satisfies the specifications described in the test code [6]. The correctness of the code of students is also marked by the test code.

6) *Code Amendment Problem (CAP)*: In this problem type, a source code that has several missing or error elements, called a problem code, is shown to student. A student needs to identify the locations of missing or error elements in the code, and to fill in them or correct them with the correct elements. The correctness of any answer will be marked through string matching of the whole statement with the corresponding original one in the code [11].

7) *Code Completion Problem (CCP)*: In this problem, a source code with several missing elements is shown to the students without specifying their existences. Then, a student needs to locate the missing elements in the code and fill in the correct ones there. The correctness of the answer from a student is verified by applying string matching to each statement in the answer to the corresponding original statement in the code. Only if the whole statement is matched, the answer for the statement will become correct. Moreover, merely one incorrect element will result in the incorrect answer [12].

2.2 Service Functions in JPLAS

There are two services functions in *JPLAS*, namely, teacher service functions, and student service functions.

2.2.1 Teacher Service Functions

The teacher service functions include the problem generation, the registration and management of assignments, the creation of projects, and analyzing student performance by checking student's answers and viewing the number of submissions for individual problem by each student to evaluate the difficulty of assignments, and compressions of students. If most of the students did a lot of submissions for an assignment, the teacher need to consider that it will be too difficult for the learners and if it is necessary, the teacher needs to change or replace that problem with easier one. Sometimes, the teacher can implement hint functions, and recommendation functions for the assignments to assist the students for better understanding. In addition, the teacher can also create the references for each topic. On the other hands, if the teacher finds a student who submitted the answers many times whereas other students did so fewer times, in this case, the teacher needs to carefully instruct that student and check that student's performing extraordinarily. Moreover, the teacher can create the project assignment by summarizing all important concepts that was previously learned by the students to verify and evaluate the students' situations about the corresponding topics.

2.2.2 Student Service Functions

The student service functions include the view of the assignments, solving project assignments, and the submissions of their answers for the assignments. For code writing problem type, the student needs to write a source code for an assignment by reading problem statement, and the test code where the student must use the class/method names, the types, and the other specifications to satisfy the given test code. The answer from a student is generally processed at the *JPLAS* by the following steps:

1) When a student accesses to *JPLAS*, the list of the assigned problems to the student is displayed.

2) When a problem is selected by the student, the corresponding problem text in the database is displayed.

3) The student writes the answers in the corresponding forms.

4) The answers submitted by the student are marked in the server, and both the answer and the marking results will be saved in the database.

5) *JPLAS* offers feeds back to the student.

6) If necessary, the student could repeat the steps from (3).

The utilization procedure for both *JPLAS* functions by a teacher and a student are given as follows:

1) A teacher generates a new problem, and registers it to the database.

2) A teacher generates a new assignment by selecting proper problems in the database and registers it to the database.

3) A student selects an assignment to be solved.

4) A student selects a problem in the assignment to be solved.

5) A student solves the questions in the problem and submits the answers to the server.

6) The server marks the answers and returns the marking results.

7) A student modifies the incorrect answers and resubmits them to the server, if necessary.

8) A student refers to his/her solution results of the assignments.

9) A teacher refers to the solution results of all the students of the assignments.

2.3 Desktop-version JPLAS

As the previously mentions, *JPLAS* has been developed as a web application system. However, it has been found that the online system can be used only in Internet-available environments and it will be difficult in some areas where the Internet connection can't access at all or may not be stable due to the weak network infrastructure and the frequent power shortage, particularly in developing countries. To avoid those difficulties of the online *JPLAS*, we have implemented the offline *Desktop-version JPLAS (D-JPLAS)* as an efficient solution for schools and homes with the poor Internet accesses. Unlike to the *online JPLAS*, *D-JPLAS* runs on the client PC only, without the server access through the Internet. It keeps all the programs and data including the problems and the student answers in the file system of the user's PC, where it does not use the database.

Basically, the usage flow of offline *D-JPLAS* can be process through the following steps. Firstly, the teacher needs to create and assign the programming assignment. After that, he/she distributes the created assignment problems to the students, who are learning programming languages for improving their skills. Students can solve the assignments repeatedly on their own PC on offline until they can get the correct answers. After that, the students need to submit their answer files to the teacher who stores the files in the respective folder for each problem type and the student. Finally, the teacher will manage and analyze the submitted answers on his/her own PC using the answer analysis function, and give feedbacks to the students. The file exchange between the teacher and student will be done through the USB memories, the file servers, or emails if the Internet is accessible. Figure 2.3 illustrates the usage flow of *Desktop-version JPLAS* and Section 2.4 will discuss the offline answering functions in *JPLAS*.

2.4 Offline Answering Functions in JPLAS

As mentioned in Section 2.3, in addition to the online platform, the offline answering function has been implemented to allow students to answer the problems in *JPLAS* even if the students cannot access to the *JPLAS* server when the Internet is unavailable. Therefore, this function is very useful and actually inevitable in applying *JPLAS*. For solving the problem instances in this offline *JPLAS*, the problem assignment delivery and answer submission can be accomplished with a USB. There are three mainly functions: operation flow, file generation, and cheating prevention in *offline JPLAS*.

Figure 2.3: Usage flow of Desktop-version JPLAS.

2.4.1 Operation Flow

The operation flow of the offline answering function is as follows:

1) Problem instance download: a teacher accesses to the *JPLAS* server, selects the problem instances for the assignment, and downloads the required files into the own PC on online.

2) Assignment distribution: the teacher distributes the assignment files to the students by using a file server or USB memories.

3) Assignment answering: the students receive and install the files on their PCs, and answer the problem instances in the assignment using Web browsers on offline, where the correctness of each answer is verified instantly at the browsers using the *JavaScript* program.

4) Answering result submission: the students submit their final answering results to the teacher by using a file server or USB memories.

5) Answering result upload: the teacher uploads the answering results from the students to the *JPLAS* server to manage them.

2.4.2 File Generation

Table 2.1 shows the necessary files with their specifications for the offline answering function in *JPLAS*. These files are designed for the problem view, the answer marking, and the answer storage.

2.4.3 Cheating Prevention

In *offline JPLAS*, the correct answers need to be distributed to the students so that their answers can be verified instantly on the browser. To prevent disclosing the correct answers, they will be distributed after taking hash values using *SHA256* [20]. In addition, to avoid generating the same hash values for the same correct answers, the assignment ID and the problem ID are concatenated

with each correct answer before hashing. Then, the same correct answers for different blanks are converted to different hash values, which ensure the independence among blanks [12].

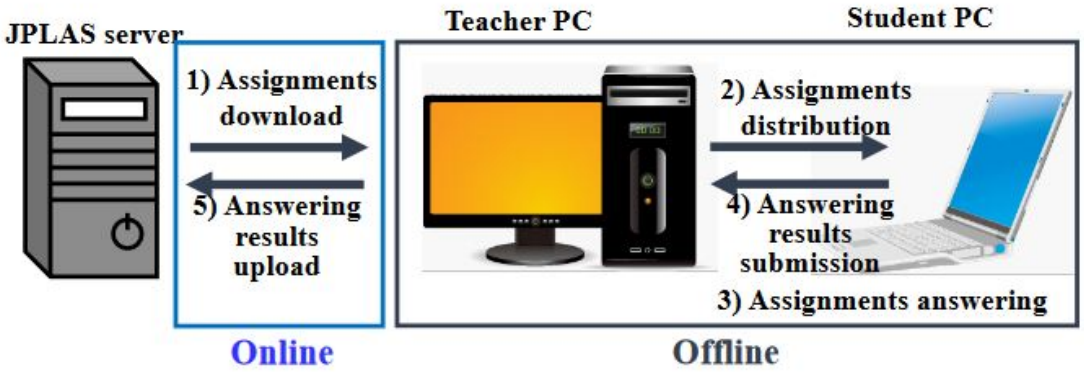

Figure 2.4: Operation flow for offline answering function.

Table 2.1: Files for distribution.

File name	Outline
css	CSS file for Web browser
index.html	HTML file for Web browser
page.html	HTML file for correct answers
jplas2015.js	js file for reading the problem list
distinction.js	js file for checking the correctness of answer
jquery.js	js file for use of jQuery
sha256	js file for use of SHA256
storage.js	js file for Web storage

2.5 Implementation of JPLAS Platform Using Node.js and Docker

Besides the *online JPLAS* and *offline JPLAS*, we implemented the *JPLAS* platform with the newly designed software architecture using *Node.js* and *Docker* without the internet connection, to avoid the redundancy and improve the portability of previous implementations [13].

The students can solve the *JPLAS* problems without Internet connection by installing all the system in their PCs. *Node.js* [14] is adopted as the popular web application server, where application programs on both the server and client can be made using *JavaScript*. Besides, *Express.js* [15] is used together as the framework to reduce the implementation cost of this platform. Furthermore, the user interface is dynamically controlled with *EJS* that can avoid the complex syntax structure.

To avoid the software version problem on a PC when we distribute the system to the students, we use *Docker* which provides the flexibility and portability for running various software in different platforms. *Docker* [16]is adopted to make students easily install the platform software in their own PCs, so that they can solve exercises in *JPLAS* without the Internet connections. *Docker* has been designed to make it easier to create, deploy, and run an application program on various

platforms using the container. The *Docker* container [17] allows an application developer to combine all the necessary software required to run the application program, such as the libraries, the middleware, the parameters, and the other dependencies, into one package file called the container image, to be shipped out. The *Docker container image* is a lightweight, standalone, and executable package of all the software needed to run the application program. It may include the source codes, the runtime environments, the system tools, the system libraries, and the settings.

As the software architecture, *Mac OS* is adopted for the operating system in the server platform. *Node.js* is used as a web application server together with the *Express.js* framework. *EJS* is used for the template engine. Any database system is not installed for managing the data. JUnit [18] is used for testing the answer source codes in code writing problemre [6]. *Visual Studio Code (VS Code)* IDE is used for editing the source codes as a popular development environment. In our architecture, *Java* is used for the model (M) to run *JUnit*, *JS/CSS/JavaScript* are for the view (V), and *JavaScript (Node/Express)* is for the controller(C). It is a compact web application server and can create both the client and server side of the application using only *JavaScript*. It can make application program is easy, simple and reduce by using a flexible framework as *Express.js* that provides ready-made components for a web application. *Node.js*. The overall architecture can be seen in Figure 2.5.

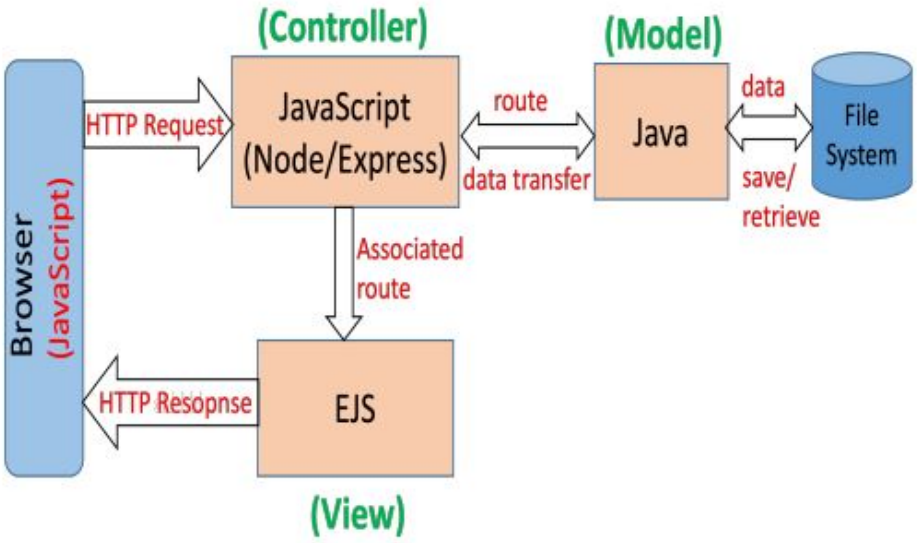

Figure 2.5: MVC model in JPLAS using Node.js.

2.6 Elaboration of PyPLAS

In today's technology landscape, *Python* becomes a widely used as the highly versatile programming language. It has gained immense popularity and importance in various domains. *Python* has the following features:

1. Easy to Learn and Readability
2. Wide Range of Applications
3. Large Standard Library and Third-Party Packages

4. Cross-Platform Compatibility
5. Strong Community and Support
6. Data Science and Machine Learning
7. Scripting and Automation
8. Integration and Extensibility

Python has been used by professionals, programmers, and application developers. The following list represents a few of the careers where *Python* is a key skill:

- Back-end developer (server-side)
- Front-end developer (client-side)
- Full-stack developer (both client and server-side)
- Web designer
- Back-end developer (*Python* developer)
- Machine learning engineer
- Data scientist
- Data analyst
- Data engineer
- DevOps engineer (development operations)
- Software engineer
- Game developer
- Statistician
- SEO specialist
- And more... [19]

Due to the *Python's* importance, we considered to extend *JPLAS* to *Python programming* learning, called *Python Programming Learning Assistant System (PyPLAS)*.

2.6.1 Problem Types in PyPLAS

Currently, we have implemented various types of problems with automatic marking functions to cover self-studies of *Python programming* at different levels by novice students. They include *Grammar Concept Understanding (GUP)*, *Value Trace Problem (VTP)*, *Comment Insertion Problem (CIP)*, *Code Modification Problem (CMP)*, and *Code Writing Problem (CWP)*. Among them, this thesis focuses on GUP. The answers of students are marked by string matching with correct codes or by unit testing using test codes.

2.7 Summary

In this chapter, we reviewed *JPLAS*, including the functions in *JPLAS* and desktop versions of *JPLAS*. Also, this chapter reviewed tge implementation of the *JPLAS* platform using *Node.js* and *Docker*, and the elaboration of *PyPLAS* as the extension of *JPLAS*.

Chapter 3

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

This chapter presents the code plagiarism checking function and its application for the code writing problem in Java programming learning assistant system.

3.1 Introduction

To assist *Java programming* learning of novice students, our group has developed the web-based *Java Programming Learning Assistant System (JPLAS)*. *JPLAS* provides various exercise problems at various levels to cultivate *code reading* and *code writing* skills of students. Among them, the *code writing problem (CWP)* asks a student to write a source code to pass the given *test code* where the correctness is verified by running them on *JUnit*.

Previously, our group implemented the *answer code validation program* in Python to help teachers. It automatically verifies the source codes for one test code from all the students, 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 chapter, I present the implementation of the *code plagiarism checking function* in the *answer code validation program*.

3.2 Previous Works of Code Writing Problem

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

3.2.1 Code Writing Problem

The *code writing problem (CWP)* assignment contains a statement accompanying 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.

3.2.2 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. *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.

3.2.3 Example Test Code

A *test code* is written by using the *JUnit* library. Here, the *BubbleSort* class in **Listing** 3.1 [24] 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 3.1: **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** 3.2 tests the *sort* method.

Listing 3.2: **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:",
14             Arrays.toString(expOutput),Arrays.toString(codeOutput));
15         } catch (AssertionError ae) {
16             System.out.println(ae.getMessage());
17         }
18     }
19 }
```

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.

3.2.4 CWP Answer Platform for Students

To assist students in solving CWP assignments efficiently, our group has implemented the *answer platform* as a web application system using *Node.js*. Figure 3.1 illustrates the software architecture. It is noted that OS can be *Linux* or *Windows*.

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 every data is provided by a file. For the *view (V)* part on the browser, *Embedded JavaScript (EJS)* is used instead of the default template engine *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.

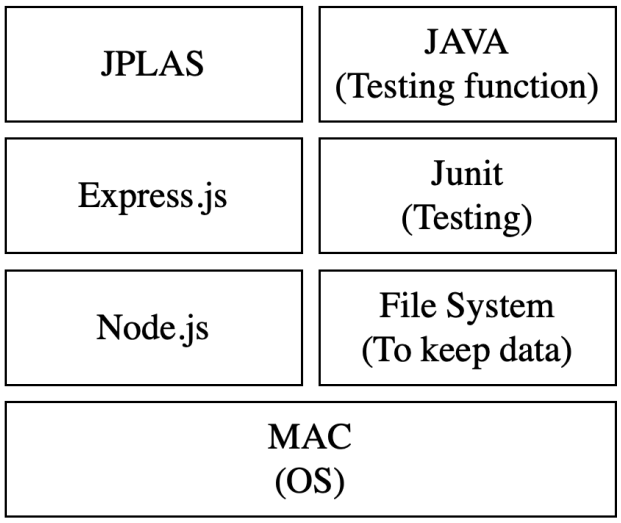

Figure 3.1: CWP software architecture.

Figure 3.2 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 at the lower side of the interface. It is noted that Figure 3.1 and Figure 3.2 are adopted from the previous paper [21].

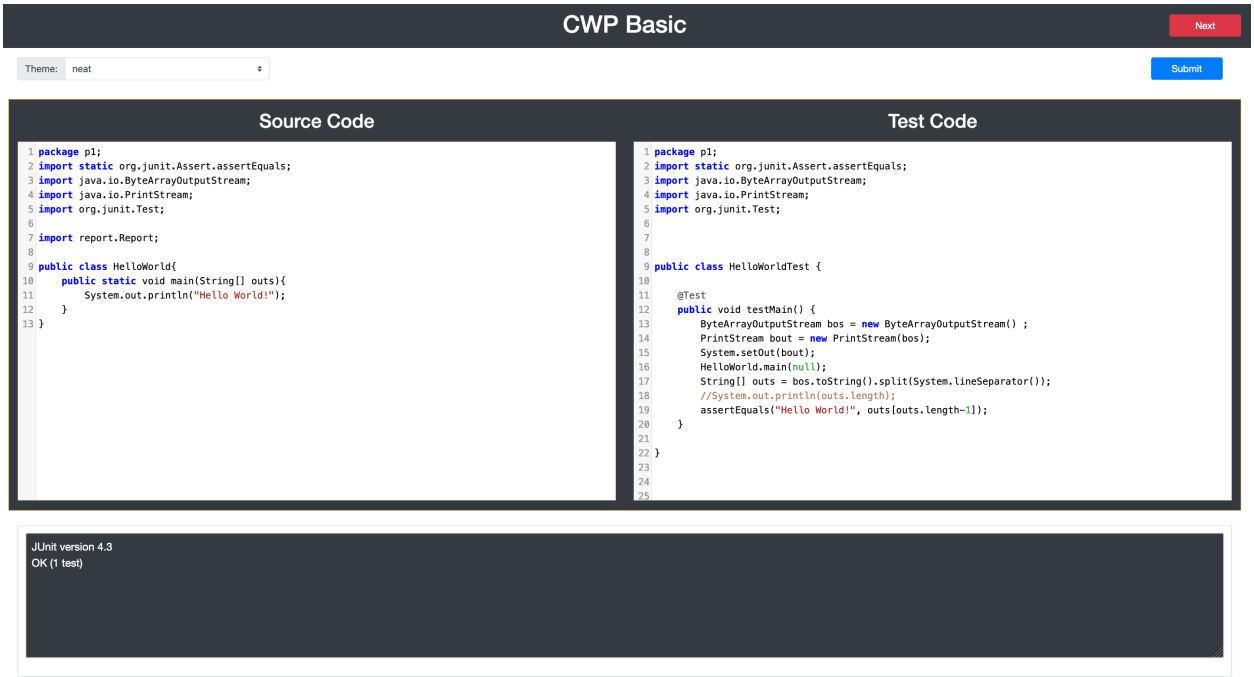

Figure 3.2: CWP answer interface.

3.2.5 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 "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 **??**, which was adapted from [21].

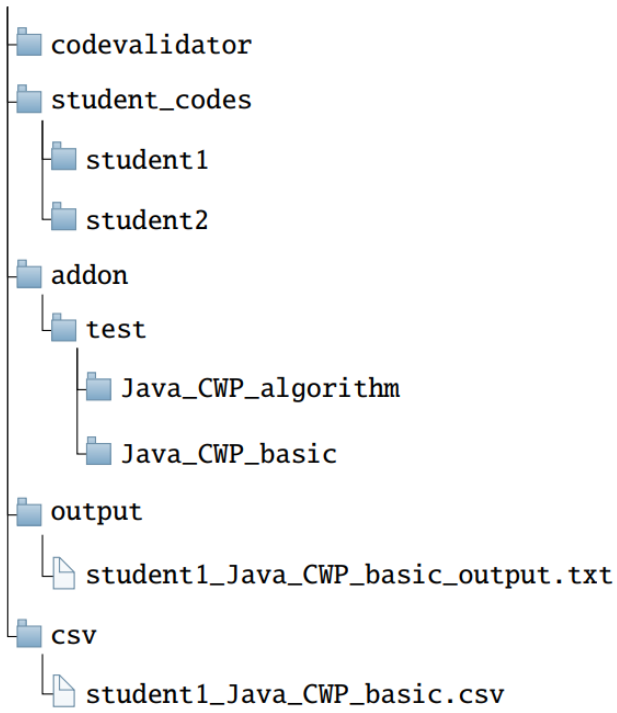

Figure 3.3: Example of file structures with folder hierarchy

3.3 Code Plagiarism Checking Function

In this section, I 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.

3.3.1 Levenshtein Distance

The *Levenshtein distance*, also known as the *edit distance*, indicates the measure of the similarity between two strings or their sequences. 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:

$$\text{similarity score} = \left(1 - \frac{\text{Levenshtein Distance}}{\max(\text{length of string1}, \text{length of string2})}\right) \times 100 \tag{3.1}$$

where $\max(\text{length of string1}, \text{length of string2})$ represents the larger length between two strings *string1* and *string2*.

3.3.2 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.

3.3.3 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: }
```

3.3.4 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, I 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 submitted source codes. Therefore, the final computational complexity of the function is given by $O(k^2n^2)$.

In addition, in the revised paper, I measure the CPU time for applying the code plagiarism checking function to all the source codes for each assignment in Section 3.4.1. 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.

3.4 Analysis of Application Results

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

3.4.1 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 3.1 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 3.1: CWP assignments for evaluations.

group topic	ID	assignment title	# 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
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

3.4.2 Analysis Results of Individual Assignments

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

3.4.2.1 Results for Basic Grammar

Figure 3.4 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 the 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.

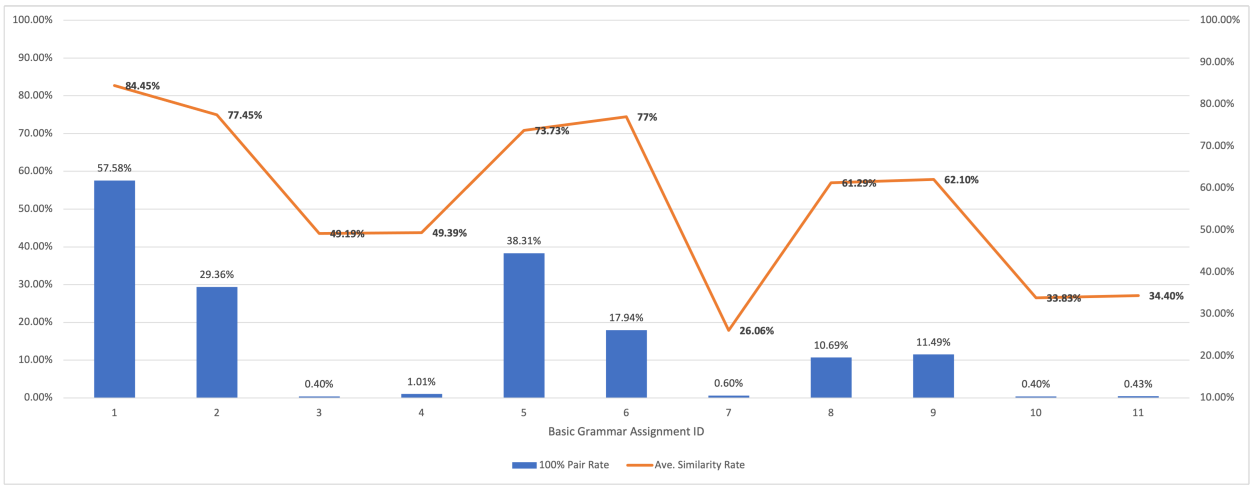

Figure 3.4: Results for *basic grammar*.

Table 3.2 shows the number of student pairs that had 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 for 10 assignments. With the high probability, these pairs submitted copied source codes. Some students often copied the source codes from certain students.

3.4.2.2 Results for Data Structure

Figure 3.5 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 the 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.

Table 3.2: Number of student pairs with identical codes.

# of assignments with identical codes	# of student pairs
11	1
10	1
6	3
5	8
4	31
3	71
2	132
1	182

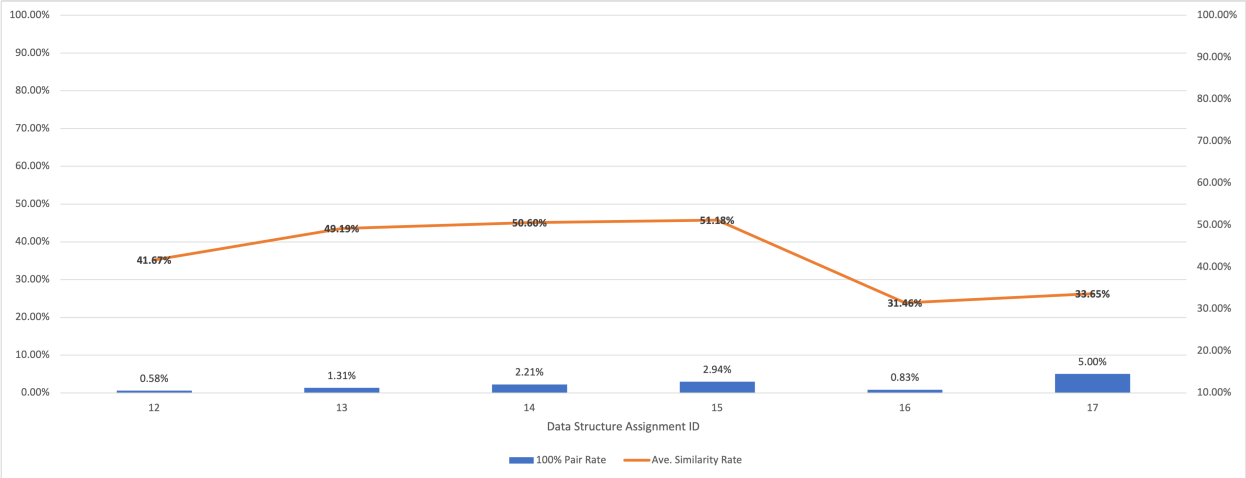

Figure 3.5: Results for *data structure*.

Table 3.3 shows the number of student pairs that had 100% similarity score for each number of assignments for *data structure*. It suggests that one pair submitted the identical source codes for 5 assignments, and another pair did for 4 assignments. With the high probability, these pairs submitted copied source codes. Some students often copied the source codes from certain students.

Table 3.3: Number of student pairs with identical codes.

# of assignment with identical codes	# of student pairs
5	1
4	1
1	8

3.4.2.3 Results for Object Oriented Programming

Figure 3.6 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 the 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.

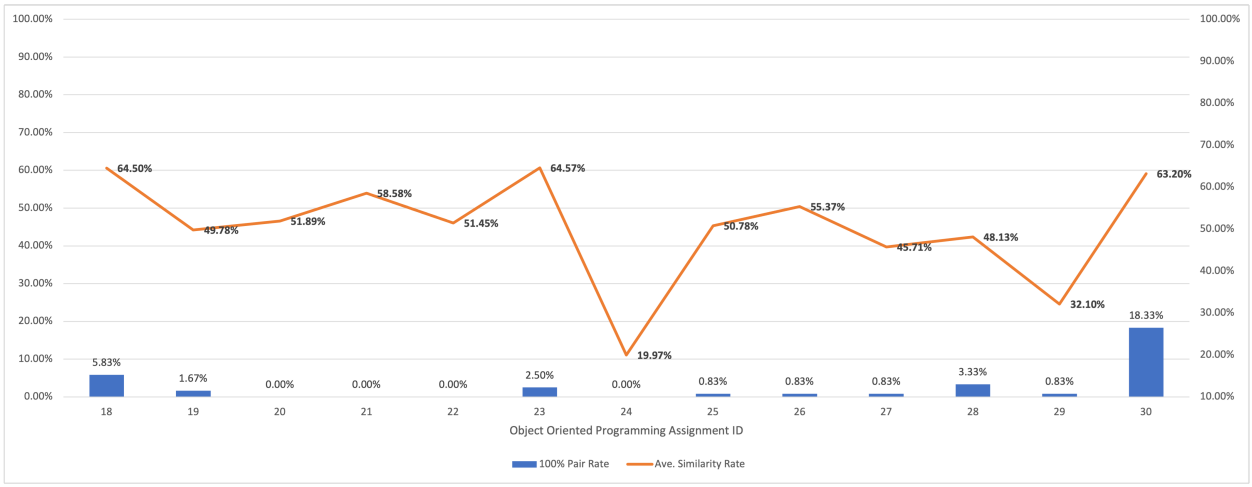

Figure 3.6: Results for *object oriented programming*.

3.4.2.4 Results for Fundamental Algorithms

Figure 3.7 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 the 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.

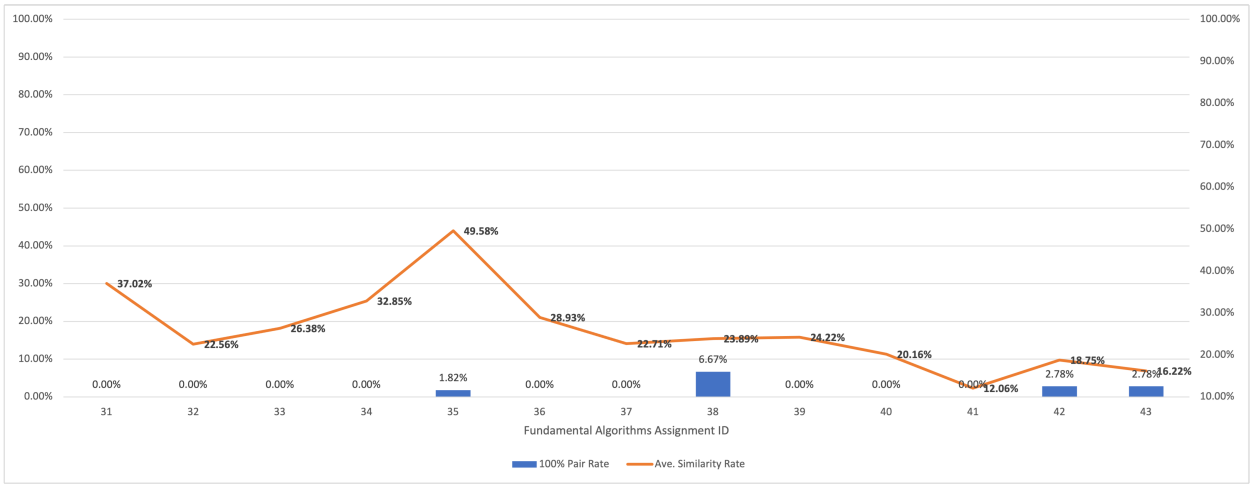

Figure 3.7: Results for *fundamental algorithms*.

3.4.2.5 Results for Final Examination

Figure 3.8 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.

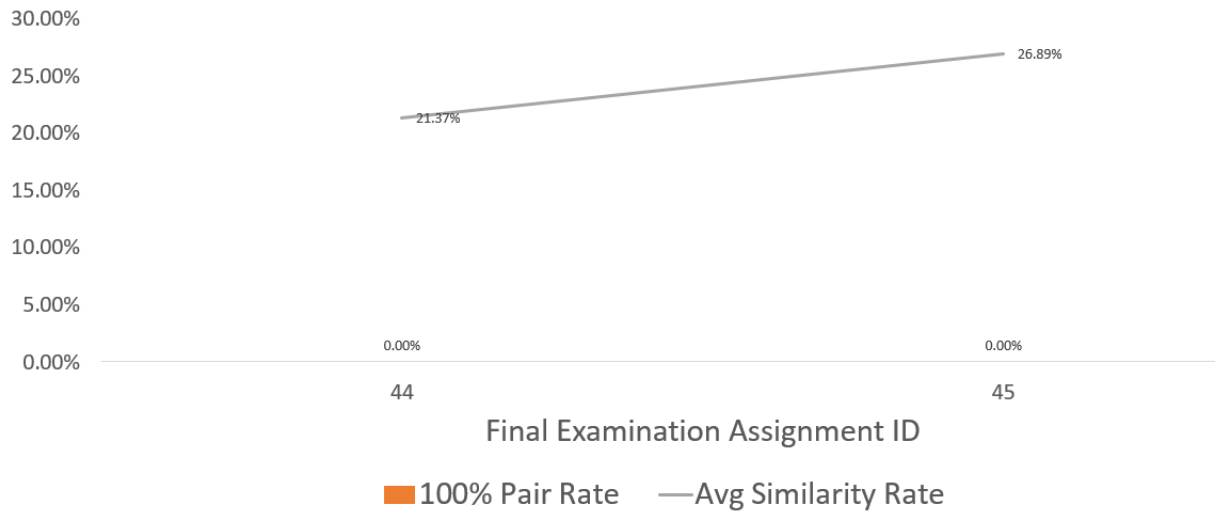

Figure 3.8: Results for *final examination*.

3.4.3 Analysis Results of Assignment Group

Next, I analyze the solution results by each group. Table 3.4 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 made 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 the code plagiarism by the proposed function, the threshold needs to be adjusted properly by considering the feature of each assignment. The formula will be in future works.

The *CPU time* for each group will be also discussed in Table 3.4. 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 3.4: Number of source codes and results in each group.

group topic	# of source codes	# 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

3.5 Summary

In this chapter, I 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, I applied the proposal to a total of 877 source codes for 45 CWP assignments from 9 – 39 students and analyzed the results. The results confirm the validity and effectiveness of the proposal.

Chapter 4

A Study of Grammar-Concept Understanding Problem for Python Programming Learning

In this chapter, I present the grammar-concept understanding problem for the Python programming learning assistance system.

4.1 Introduction

Recently, *Python programming* has gained the popularity for use in various groundbreaking fields in experiments, prototyping, embedded systems, and data sciences, due to rich libraries and short coding features. A lot of people in both *IT (information technology)* and non-IT fields will start using it to achieve their needs. However, a lot of students are suffering from studying it due to the formality nature in programming. To assist self-studies of *Python programming*, our group has developed *Python Programming Learning Assistant System (PPLAS)* that offers several types of exercise problems with different levels, by extending our works of JPLAS for *Java programming* [23].

To learn programming effectively, it is suggested that students should firstly solve simple problems on grammar concepts while reading source codes. Then, they can solve harder problems step-by-step before practicing code writing from scratch. In JPLAS, our group proposed the *Grammar-concept Understanding Problem (GUP)* as the first-step problem for novice students [24]. A GUP instance consists of a source code, a set of questions, and the correct answers. Each question describes a basic grammar concept in Java programing in the source code, and requests to answer the corresponding keyword in the code. Any answer is marked by *string matching* with the correct one. The *GUP generation algorithm* is implemented to automatically generate a GUP instance from a given code to help teachers.

4.2 GUP Instance Generation

In this section, I review the *GUP instance generation algorithm* to assist a teacher to generate a new GUP instance among the selected source code.

4.2.1 Input Files

To use the algorithm, a teacher needs to prepare the file of the source code that covers the grammar concepts to be studied by students through solving the GUP instance. Then, the algorithm will read this source code file, and generate the GUP instance file through the procedure in Section 4.2.2.

For this algorithm, the list of the keywords and the corresponding questions needs to be prepared beforehand. For the GUP instance generation, some of the questions are given as follows.

1) Which keyword can make the program jump out of the most inner block of statements? (break).

2) Which keyword is used for skipping the certain statements that are inside the loop? (continue).

3) Which keyword is used to mark the start of the function header? (def).

4) Which keyword is used for returning a value when exiting the function? (return)

5) Which keyword indicates the following lines may cause errors? (try).

This list should include any keyword that represents the basic grammar concept to be studied through solving GUP instances. The question to each keyword can describe the definition of the corresponding keyword, which can be unique. By reading the questions and replying the answer keywords, students are expected to study basic grammar concepts in *Python programming*.

4.2.2 GUP Generation Procedure

A GUP instance file is generated through the following procedure:

1) Read a Python source code file.

2) Extract the keywords in the *keyword list* from the source code.

3) Select the question in the *question list* that corresponds to each extracted keyword.

   3-1) If multiple questions are registered in the *question list* for the keyword, one of them is randomly selected.

   3-2) If the question needs to find the line number of the source code for the keyword, it is found and included in the question.

4) Find the element as the correct answer from the source code.

5) If the same pair of the question and the correct answer is selected, discard them as the duplicated question.

6) Output the GUP instance file of the source code, the questions, and the correct answers.

4.2.3 User Interface of GUP

Figure 4.1 illustrates the user interface for this instance. After filling in the answer form, the student clicks the ”Answer” button. If the answer is correct, the background color will stay in white. Otherwise, it will become red. The student can submit answer repeatedly until their answers are satisfied.

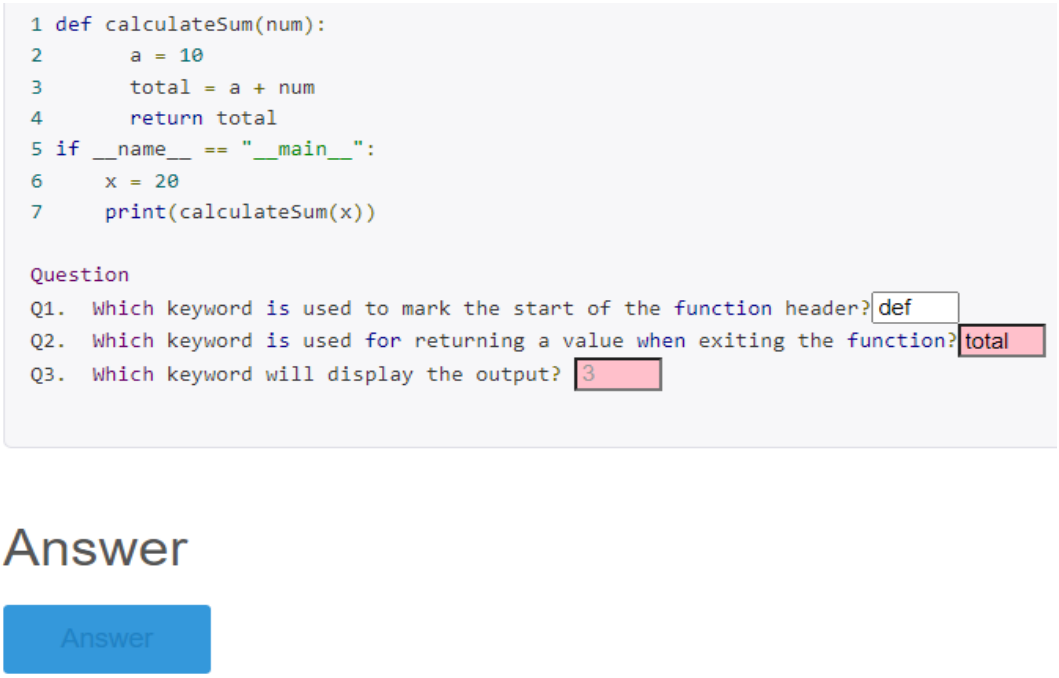

Figure 4.1: GUP User Interface

4.3 Evaluation

In this section, I evaluate the GUP for *PPLAS* through applications to 9 students in Okayama University. I generated 24 GUP instances with the total of 139 questions from source codes that cover basic grammar topics of the keywords. Then, the students solved them using the interface in [22]. Table 4.1 shows the average correct rate and submission times of the students.

4.3.1 Correct Rate Result

As shown in Table 4.1, all of the 9 students achieved 100% correct answer rates. It means that all of the students correctly understand the basic grammar concepts for *Python programming*. They are likely to well understand the further steps of *Python programming language*.

4.3.2 Submission Times Results

As more details of the solution results in Table 4.1, the students are divided into three groups according to their submission time ranges. First of all, it can be clearly seen that there are two students in group I who got 100% accuracy with the one time attempt. It means those two students may have either background programming knowledge or have studied *Python Programming*. Secondly, there are 6 students who submitted their answer with 2.54% average submission times until to get 100% accuracy. It is clear that these students learned and solved our GUP problems repeatedly to get the correct answer. Finally, in group III , one student submitted the answer again and again. Eventually, he got 100% correctness. By reviewing this student, he has no background knowledge of Python. With the help of grammar concept in our GUP problems, he got 100% correctness. It means he had understood the basic Python grammar concept in the end.

Table 4.1: Students performance for GUP.

Group	Average Correct Rate	Submission Time Range	# of Students
I	100%	24	2
II	100%	31-92	6
III	100%	127	1

4.4 Summary

This chapter presented the *grammar-concept understanding problem (GUP)* for PPLAS for the first-step study of *Python programming* by novice students. For evaluations, 24 GUP instances with 139 questions covering the keywords for basic grammar concepts were generated. The results confirmed that the proposal is effective in revealing the understanding levels. Their application results to 9 students in Okayama University confirmed the effectiveness of them, where every student achieved 100% correctness.

Chapter 5

Related Works in Literature

In this section, we introduce related works to this thesis.

In [25][26], the authors suggested common problems among programming novices, along with existing efforts and discussions of current methods used in teaching programming. Several tools have been proposed to help students to solve programming learning difficulties. Among them, *ToolPetcha* is the example tool that acts as an automated assistant in matters of programming [27].

In [28], the authors proposed a game-based learning environment to assist beginner students in learning programming. It uses game creation tasks to make basic programming easier to understand and includes idea visualization approaches to let students manipulate game objects to learn important programming concepts.

In [29], the authors developed a collaborative learning environment based on the problems powered by the technology for dynamic webs to investigate students' perceptions of the learning environment. This research was planned as a qualitative study. A semi-structured interview format was created to get the opinions of students about the learning environment that was supported by technologies for dynamic webs and was used for collaboratively solving issues. Their findings implied that collaborative learning techniques can focus on problems and the learning environment at a community college can benefit from technologies for dynamic web pages.

In [30], the authors made comparative evaluations of several online platforms for teaching programming and chose engaging assignments from the site used to educate students, named *hackerrank.com*. They investigated user experiences with *online coding platforms (OCP)* and contrasted features of various online platforms that should be utilized to teach programming to aspiring computer scientists and programmers via distance learning. In addition, they suggested the use of online programming simulators to enhance computer science instructions, taking into account functionality, as well as students' preparation levels and expected results of learning.

In [31], the authors presented and evaluated a web-based tool providing drills and practices for Java programming called *CodeWrite*. Students are responsible for developing exercises that will be shared among classmates. Because the tool does not adopt a testing tool such as *JUnit*, validations by program testing are limited.

In [32], the authors proposed a graph-based grading system for introductory Java programming courses called *eGrader*. The dynamic analysis of each submitted program is conducted on *JUnit*, and the static analysis is on the graph representation of the program. The accuracy was confirmed through experiments.

These initiatives span a wide spectrum, encompassing innovative solutions such as *ToolPetcha* [27], an automated programming assistant aiming to aid learners in navigating programming complexities. Additionally, researchers propose game-based learning environments [28], leveraging

engaging tasks to simplify fundamental programming concepts. Collaborative learning environments powered by dynamic web technologies [29] offer a communal approach to problem-solving, enhancing students' understanding through collective efforts. Alongside these, platforms like *hackerrank.com* [30] serve as hubs for practical assignments, enhancing learning experiences through engaging coding challenges. Complementing these platforms are web-based tools like *CodeWrite* [31] and novel grading systems such as *eGrader* [32], each tailored to provide targeted exercises and structured assessments, contributing to a multifaceted landscape of educational aids and platforms in programming educations.

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

In [33], the authors reviewed recent developments of automatic assessment tools for programming exercises and discussed their major features and approaches, including programming languages, learning management systems, testing tools, limitations on resubmissions, manual assessments, security, distributions, and specialties.

In [34], the authors proposed various source code similarity detection systems, including the *source code similarity detection system (SCSDS)*. They were evaluated in 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 researches should focus on optimizing processing speed and improving user interfaces while exploring the impact of code contexts on detection accuracy.

In [35], the authors 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 the higher precision than the other two tools. It was found that integrating *SIM* and *MOSS* will be more effective for dealing with the code similarity.

In [36], the authors 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 [37], the authors 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 [38], the authors proposed *Deimos* as 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.

Chapter 6

Conclusion

In this thesis, firstly, I presented the *code plagiarism checking function* in the *code validation program* for Java. 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, I applied the proposal to a total of 877 source codes for 45 CWP assignments from $9-39$ students and analyzed the results. The results confirm the validity and effectiveness of the proposal.

Secondly, I presented the *grammar-concept understanding problem (GUP)* for Python as the first-step study of *Python programming* by novice students. For evaluations, 24 GUP instances with 139 questions covering the keywords for basic grammar concepts were generated. The results confirmed that the proposal is effective in revealing the understanding levels. Their application results to 9 students in Okayama University confirmed the effectiveness of them, where every student achieved 100% correctness.

In future works, I will apply the *code plagiarism checking function* to other programming languages such as C and Python. I will also generate a variety of GUP questions for the advanced topics, useful libraries and apply them to students.

Bibliography

[1] S. T. Aung, N. Funabiki, Y. W. Syaifuddin, H. H. S. Kyaw, "A Proposal of Grammar-concept Understanding Problem in Java Programming Learning Assistant System," J. Adv. Inf. Tech., vol. 12, no. 4, pp. 342-350, November 2021.

[2] K. K. Zaw, N. Funabiki, and W.-C. Kao, "A proposal of value trace problem for algorithm code reading in Java programming learning assistant system," Inform. Eng. Exp., vol. 1, no.3, pp. 9-18, Sep. 2015

[3] N. Funabiki, Y. Matsushima, T. Nakanishi, and N. Amano,"A Java programming learning assistant system using test-driven development method," IAENG Int. J. Computer Science, vol. 40, no. 1, pp. 38-46, February 2013

[4] N. Funabiki, H. Masaoka, N. Ishihara, I-W. Lai, and W-C. Kao,"Offline answering function for fill-in-blank problems in Java programming learning assistant system," in Proc. IEEE ICCE-Taiwan, pp. 324-325, 2016.

[5] N. Ishihara, N. Funabiki, M. Kuribayashi, and W.-C. Kao, "A software architecture for Java programming learning assistant system," J. Comp. Soft. Eng., vol. 2, no. 1, Sept. 2017.

[6] N. Funabiki, Y. Matsushim, T. Nakanishi, K. Watanabe, and N. Amano, "A Java programming learning assistant system using test-driven development method," IAENG Int. J. Comput. Sci., vol. 40, no. 1, pp. 38-46, Feb. 2013.

[7] N. Funabiki, H. Masaoka, N. Ishihara, I-W. Lai, and W.-C. Kao,"Offline answering function for fill-in-blank problems in Java programming learning assistant system," in Proc. ICCE TW, pp. 324-325, May 2016.

[8] N. Funabiki, Tana, K.K. Zaw, N. Ishihara, and W.-C. Kao, "A graph- based blank element selection algorithm for fill-in-blank problems in Java programming learning assistant system. IAENG Int J Computer Science 44: 2.

[9] K. Beck, Test-driven development: by example, Addison-Wesley,2002.

[10] N. Ishihara, N. Funabiki, and W.-C. Kao, "A proposal of statement fill-in-blank problem using program dependence graph in Java programming learning assistant system," Info. Engr. Exp., vol. 1, no. 3, pp. 19-28, Sept. 2015.

[11] H.H.S. Kyaw, N. Funabiki, and W.-C. Kao, "A proposal of code amendment problem in Java programming learning assistant system," International Journal of Information and Education Technology (IJIET), vol. 10, No. 10, pp. 751-756, Oct. 2020.

[12] H.H.S. Kyaw, S.S. Wint, N. Funabiki, and W.-C. Kao, "A code completion problem in Java programming learning assistant system," IAENG International Journal of Computer Science (IJCS), vol. 47, No. 3, pp. 350-359, Sept. 2020.

[13] S. T. Aung, N. Funabiki, L. H. Aung, H. Htet, H. H. S. Kyaw, and S. Sugawara, " An Implementation of Java Programming Learning Assistant System Platform Using Node.js," ICIET International Conference of Information and Education Technology (ICIET), pp. 47-52, Apr.2022.

[14] D. Herron, Node.js web development, Birmingham, UK, 2016.

[15] "Express," Internet: https://expressjs.com/., Access June 20, 2023.

[16] R. McKendrick, Monitoring Docker, United Kingdom, 2015.

[17] A. Mouat, Using Docker: developing and deploying software with containers, USA, 2015.

[18] "JUnit 5," Internet: https://junit.org/junit5/, Access June 20, 2023.

[19] "What is Python The most versatile programming language," Internet: https://www.datacamp.com/blog/all-about-python-the-most-versatile-programming language/, Access June 20, 2023.

[20] "SHA-256 Cryptographic Hash Algorithm," Internet: http://www.movable discretionary-type.co.uk/scripts/sha256. html/, Access June 20, 2023.

[21] 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 International Conference on Information and Education Technology, Japan, 18-20 March 2023; pp. 193-198.

[22] N. Funabiki, H. Masaoka, N. Ishihara, I.-W. Lai, and W.-C. Kao, "Offline answering function for fill-in-blank problems in Java programming learning assistant system," in Proc. IEEE ICCE-TW 2016, pp. 324-325, May 2016.

[23] S. l. Ao et al. ed., IAENG Transactions on Engineering Sciences - Special Issue for the International Association of Engineers Conferences 2016 (Volume II), World Sci. Pub., pp. 517-530, 2018.

[24] S. T. Aung, N. Funabiki, Y. W. Syaifudin, and M. Kuribayashi, "A study of grammar-concept understanding problem for Java programming learning assistant system," IEICE Tech. Rep., ET2020-15, pp. 29-34, Sept. 2020.

[25] Ala-Mutka, K.; Problems in Learning and Teaching Programming. A literature study for developing visualizations in the Codewitz-Minerva project. 2004; pp. 1-13.

[26] Konecki, M.; Problems in programming education and means of their improvement. DAAAM Int. Sci. Book, 2014; pp. 459-470.

[27] Queiros, R. A.; Peixoto, L.; Paulo, J. PETCHA - a programming exercises teaching assistant, In Proceedings of ACM annual conference on Innovation and technology in computer science education, Haifa, Israel, 3-5 July 2012; pp. 192-197.

[28] Li, F. W.-B.; Watson, C.; Game-based concept visualization for learning programming. In Proceedings of ACM workshop on Multimedia technologies for distance learning, Scottsdale Arizona, USA, 1 December 2011; pp. 37-42.

[29] Ünal, E.; Çakir, H.; Students' views about the problem based collaborative learning environment supported by dynamic web technologies. *Malaysian Online J. Edu. Tech.* **2017**, *5(2)*, 1-19.

[30] 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*.

[31] Denny, P.; Luxton-Reilly, A.; Tempero, E.; Hendrickx, J. CodeWrite: supporting student-driven practice of Java. In Proceedings of ACM Technical Symposium on Computer Science Education, Dallas, USA, 9-12 March, 2011; pp. 471-476.

[32] 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.

[33] Ihantola, P.; Ahoniemi, T.; Karavirta, V.; Seppälä, O. Review of recent systems for automatic assessment of programming assignments. In Proceedings of Koli Calling, 2010; pp. 86–93.

[34] Duric, Z.; Gasevic, D.; A source code similarity system for plagiarism detection. *The Computer Journal* **2013**, *56(1)*, 70-86.

[35] 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, 2019; pp. 112-117.

[36] Novak, M.; Joy, M.; KERMEK, D.; Source-code similarity detection and detection tools used in academia: a systematic review. *ACM Trans. on Comp. Edu.* **2019**, *19(3)*, 1-37.

[37] S.; Karnalim, 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.

[38] 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, South Korea, May 2009; pp. 481-486.

[39] Bubble Sort. Available online: `https://www.javatpoint.com/bubble-sort-in-java`