Finding Plagiarisms with Evidence in Java Programming Courses

Ushio Inoue

Abstract—**This paper proposes a new method to find plagiarisms with evidence among a set of student programs that are submitted for programming assignments in elementary Java programming courses. In such an assignment, each student program tends to be similar to others because it is small and structurally simple. Existing plagiarism detection methods are not useful and yield many false-positive errors. The proposed method detects plagiarisms with evidence based on the nature of students who are not accustomed to Java programming. The evaluation results using real assignments showed that analyzing white spaces and peculiar patterns in source codes are very powerful to find plagiarisms with reliable evidence.**

Index Terms—**Plagiarism detection, java programming, n-gram similarity, longest common subsequence.**

I. Introduction

In programming courses of universities and colleges, students are often given assignments to write programs based on some specifications by the instructor. Such assignments are very helpful to learn programming languages and to acquire coding skills for the students. However, the evaluation of these student programs is very troublesome for the instructor if many students study at the same time.

There may be plagiarisms that are unfair copies of programs written by other students. To find such plagiarisms is a very painful task of the instructor, because it is not a productive effort. The cost of finding plagiarisms is basically proportional to the square number of submitted programs. The task becomes more difficult if camouflage is done by modifying the code.

Many research activities have been done to develop automatic plagiarism detection methods and tools, and some of them are freely available on the Internet. Surveys and comparisons of these activities have been done [1], [2]. However, existing methods and tools are very likely to yield many false positive errors among student programs in elementary courses. The reason is that each program tends to be similar because it is small and structurally simple.

The author proposed a method that automatically finds plagiarisms in elementary courses [3]. The method uses three types of similarities: code, comment, and inconsistency similarities. The code similarity reflects the common expressions and statements in source codes. The comment similarity reflects the common words and phrases in explanatory comments. The inconsistency similarity reflects strange expressions, which are syntactically correct but rarely used by experienced programmers. This method showed higher precision and recall ratio than that of other methods.

This paper proposes a new method, which is an extension of the previous work to find the evidence of plagiarisms. Finding reliable evidence is important when giving a warning or penalty to plagiarists, because they can easily defend themselves by saying that they wrote their programs independently and the resemblance was caused by accident.

The rest of the paper is organized as follows; Section II describes related work. Section III defines the target and goal of the research. Section IV explains the proposed method in detail. Section V presents evaluation results by using actual student programs. Section VI concludes the discussion.

II. Related Work

Plagiarism detection methods can be classified into two categories: static and dynamic. Static methods compare a pair of source files by means of text, tokens, or logical structures. Moss [4], [5] normalizes text in source files, composes fingerprints by using an n-gram technique, and finds the longest common sequence on the fingerprints. In the text normalization, white spaces are removed and identifier names are replaced. JPlag [6], [7] converts text into strings of canonical tokens and compares the token strings by using the Greedy String Tiling algorithm. White spaces and identifier names are not included in the token strings. Koschke, *et al.* [8] proposed a method based on an abstract syntax tree that is a representation of the logical structure of the program. The trees are created by a language-specific parser.

On the other hand, dynamic methods compare the behavior of object files (i.e. class files in Java). Schuler, *et al.* [9] proposed a method that observes how a program uses objects provided by the Java Standard API. Lu, *et al.* [10] proposed a birthmark based on opcodes, which are executable instructions in the program. The similarity of birthmarks is evaluated by employing the theory of the probability and statistic. Fukuda and Tamada [11] proposed a new dynamic birthmark based on the runtime behavior of Java VM.

Our method belongs to the very first category: static and text based. The reason is the size of our target files. Since they are relatively small, every single character in the files should be used for accurate plagiarism detection. Another reason is that the evidence of plagiarisms should be given clearly to the plagiarists by using actual text in the source files. The other reason is that some of student programs may cause errors in compiling or execution time. Plagiarisms should be detected against such buggy programs.

Manuscript received August 21, 2013; revised October 10, 2013.

Ushio Inoue is with the Department of Information and Communication Engineering, the School of Engineering, Tokyo Denki University, Tokyo, 120-8551 Japan (e-mail: inoue@c.dendai.ac.jp).

DOI: 10.7763/IJCTE.2014.V6.843

III. Definition

A. Target

The goal of the research is to develop a method that detects plagiarisms with evidence among student programs in elementary programming courses. Since most of the students in these courses are beginners, programming assignments are very simple, such as "Write a program that finds the least common multiple (LCM) number of three natural numbers." Consequently, student programs are small and tend to be similar to others. Therefore, it is difficult to judge a program as a plagiarism or not. Plagiarists may also camouflage their copies in order to make the detection more difficult. Although there are many levels of the camouflage, they do not use complicated techniques, because they are immature in programming skills and do not want to spend much time for their programs. Typical camouflages are limited to easy tricks such as changing the indentation style and replacing identifier names.

B. Template File

The instructor distributes a template to students for his/her easy manipulation of student programs. The basic structure of the template is shown in Fig. 1.

```
/* Instructor Comment:
 * Defines specifications and requirements of
 * the program.
 */

/* Student Comment:
 * Describes algorithms or procedures used in
 * the program.
 */

class Assignment {
    public static void main(String[ ] args) {
        // Replace "ID" and "NAME" by your own!
        System.out.println("/*** ID NAME ***/Yn");
        // Write your code from here!
        Student Code:
    }
}
```

Fig. 1. The template file for an assignment.

The template consists of three parts: instructor comment, student comment, and student code. The instructor comment defines the specifications and requirements to the program. A typical size of the instructor comment is 5 lines. The student comment explains the algorithm or solution in their natural language. The size of student comments may vary much but typically 3 to 10 lines. The student code consists of a sequence of source code, which follows an identification statement that prints the student ID and name on the standard output device (console). This is used to identify the student in the source code and the execution results.

IV. Proposed Method

A. Basic Consideration

Plagiarisms can be classified into several levels according to the techniques of camouflage that change the code of the original program. Table I shows seven levels of plagiarisms, which are based on the definition by Faidhi, *et al.* [12].

TABLE I: Classification of Plagiarisms

Level	Definition
0	No changes (exact copy)
1	Changes in comments and indentation
2	Changes of level 1 and in identifiers
3	Changes of level 2 and in declarations (e.g., sequence)
4	Changes of level 3 and in modules (e.g., new methods)
5	Changes of level 4 and in statements (e.g., "for"/"while")
6	Changes of level 5 and in decision logic (e.g., expression)

In elementary programming courses, most of plagiarisms are between the level 0 and 2. To find the level 1 plagiarism, deleting all comments and white spaces from the source file is very effective. For the level 2, replacing identifiers (user-defined variable names) to fixed ones brings good results. For the level 3 and over, using a comparison technique that is not sensitive to the positions of statements is necessary. Multidimensional n-gram vectors have a desirable feature. Table II shows the difference of two trigram vectors, which are generated from two code fragments that are semantically same but differ in the order of appearance.

TABLE II: Difference of N-Gram Vector Elements

Source code fragment	Trigram vector elements
int x = 1; int y = 2;	"int" "ntx" "tx=" "x=1" "=1;" "1;i" ";in" "int" "nty" "ty=" "y=2" "=2;"
int y = 2; int x = 1;	"int" "nty" "ty=" "y=2" "=2;" "2;i" ";in" "int" "ntx" "tx=" "x=1" "=1;"

A plagiarist usually copies a source file of another student in a digital manner. This means that styles of writing a program and peculiar patterns of the code are exactly reproduced on the copy. Fig. 2 shows the main part of an actual student program (In order to improve the readability, the indentation and string values are modified). The red circles in Fig. 2 denote peculiar patterns: "x =stdIn" should be "x = stdIn," "nextInt( )" should be "nextInt()," and "ans + """ should be "ans." These patterns can be extracted with a basic text matching technique and provide the reliable evidence of the plagiarism.

```
public static void main(String[] args) {
   Scanner stdIn = new Scanner(System.in);
   System.out.print("N1:"); int x =stdIn.nextInt( );
   System.out.print("N2:"); int y =stdIn.nextInt();
   System.out.print("N3:"); int z =stdIn.nextInt();
   int ans=0;
   for(int a=1;a<=100;a++) {
      for(int b=1;b<=100;b++){
         for(int c=1;c<=100;c++){
            int d=x*a;int e=y*b;int f=z*c;
            if(d==e)
               if(e==f)
                  ans = d;
            if(ans!=0) break;
         }
         if(ans!=0) break;
      }
      if(ans!=0) break;
   }
   System.out.println("LCM=" + ans + "");
}
```

Fig. 2. A student program with peculiar code patterns.

A plagiarist is likely to modify the student comment

because he/she thinks that it is very conspicuous in the source file. However, if the modification is a cheap trick, a text matching technique also perceives such a trick and provides the evidence of the plagiarism.

B. Process of Finding Plagiarisms

The proposed method first finds suspect programs, then inspects the suspects in detail. Fig. 3 shows a block diagram of the method at a conceptual level.

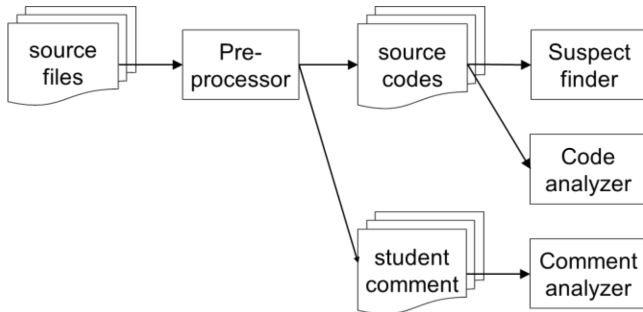

Fig. 3. Block diagram of the proposed method.

First of all, the preprocessor splits every source file into the source code and the student comment. Then, the suspect finder normalizes the source code and evaluates the similarity of every pair of the normalized codes. Then the code analyzer evaluates the similarity of every pair of the raw source codes. Finally, the comment analyzer evaluates the similarity of the student comments. Following subsections describe each process in more detail.

C. Preprocessor

As mentioned in Section III, each student creates his/her source file from the common template file. First, the preprocessor finds the instructor comment and removes it. Then, the preprocessor detects the student comment, saves it in a separate file, and leaves only source code in the file. The preprocessor uses special patterns of text inside the instructor and student comments as clues for the detection.

D. Suspect Finder

The role of the suspect finder is to extract questionable programs that should be inspected in detail by both the code analyzer and comment analyzer. The suspect finder normalizes the source code by replacing every identifier to a fixed symbol "#" and by removing every white space character in the source code. Then it generates a trigram vector from ASCII printable (i.e. $0x21$-$0x7E$) characters only. Then, it calculates the similarity S_{NOR}, which is the cosine similarity of a trigram vector pair, X and Y, by using the following equation:

$$similarity(S_{NOR}) = \frac{X \cdot Y}{\|X\| \|Y\|} \quad (1)$$

Then, the suspect finder judges the source code to be suspect if the similarity is higher than a threshold that will be discussed later.

E. Code Analyzer

The code analyzer finds evidence of plagiarisms among the suspected source codes in two levels: global and local. The term "global" means the overall style of the code, and "local" means specific small patterns in the code.

In the global level, trigram vectors are generated again. This time, however, white spaces in the source code are not discarded. The reason is that the usage of white space characters reflects a tendency of the programmer. For example, one inserts nothing before the open braces '{', but another inserts a single space, and the others insert a new line and tabs. These styles of writing code are exactly reproduced in the plagiarist's code. Therefore, if the cosine similarity of trigram vectors including white spaces, S_{RAW}, is extremely high, it will be the reliable evidence.

At the local level, over 40 peculiar or strange patterns are extracted. Table III shows typical patterns. Each pattern is assigned a unique token. The code analyzer creates a sequence of tokens from patterns discovered in the source code. Then, it finds the LCS (Longest Common Subsequence) of every pair of the sequences. For example, suppose P_1 = "ABCABC" and P_2 = "ACAAB," where P_1 and P_2 are sequences from two different source codes, and "A", "B", and "C" are tokens assigned to the patterns, respectively. The LCS of P_1 and P_2 is "ACAB." If the length of the LCS, S_{PTN}, is long, it will be the reliable evidence.

TABLE III: EXAMPLES OF PECULIAR PATTERNS

<table>
<tr><th>Regular Expression</th><th>Example</th><th>Description</th></tr>
<tr><td>[^\s]\s+$</td><td>; \n</td><td>Extra spaces before newline</td></tr>
<tr><td>[^\s]\s+;</td><td>a = b ;</td><td>Extra spaces before semicolon</td></tr>
<tr><td>(^\s*;)|(;\s*;)</td><td>; ;</td><td>Empty statement</td></tr>
<tr><td>\"\"</td><td>a + ""</td><td>Empty string</td></tr>
<tr><td>\(\s+\)</td><td>( )</td><td>Extra spaces in parentheses</td></tr>
<tr><td>\s\.</td><td>a .b</td><td>Extra spaces before period</td></tr>
<tr><td>\(\s.*\w\)</td><td>(a)</td><td>Parentheses /w imbalanced content</td></tr>
<tr><td>\s=[^=\s]</td><td>a =b</td><td>Assignments /w imbalanced operand</td></tr>
<tr><td>\s\+\+[\s\);]</td><td>a ++ b</td><td>Extra spaces before incrementor</td></tr>
<tr><td>\s==[^\s]</td><td>a ==b</td><td>Comparison /w imbalanced operand</td></tr>
<tr><td>\s\+[^\+=\s]</td><td>a +b</td><td>Arithmetic /w imbalanced operand</td></tr>
<tr><td>(^\s+$)(^\s+$)</td><td>\n\n</td><td>Consecutive empty lines</td></tr>
</table>

F. Comment Analyzer

The comment analyzer finds the evidence of plagiarisms from the student comment. Plagiarists may replace words/phrases with synonyms, add/delete adverbs/adjectives, and join/divide sentences. However, the order of key words is not changed especially for their descriptions about procedures. Therefore, the comment analyzer finds again the LCS of every pair of the student comment strings, X and Y. Then, the comment analyzer normalizes the LCS length with the following equation so that the length of the student comment does not affect the results. If the normalized length of the LCS S_{COM} is long, it will be the reliable evidence.

$$similarity(S_{COM}) = \frac{LCS_length(X, Y)}{min_length(X, Y)} \quad (2)$$

V. Evaluation Results

A. Test Datasets

This section evaluates the proposed method by using five real programming assignments used in 2013. Each test dataset was created from all student programs submitted for the assignments excluding incomplete programs. The assignments are as follows:

A_1: Create an LCM class that finds the least common multiple number of three natural numbers.

A_2: Create a StandardScore method that transforms student testing scores into the standard scores given by an array.

A_3: Create a DatePeriod class that has instance variables of start/end days and an instance method to get the number of days.

A_4: Create an OrderList class that prints an invoice of product items which are instantiated from a given Item class.

A_5: Create a MonthlyCalendar class that prints a calendar for any month of the year, which is derived from a given Date class.

The assignments are arranged in order of difficulty; A_1 is the easiest, and A_5 is the most difficult. From our past experience, the ratio of plagiarisms increases as the difficulty of assignments increases. Table IV shows a summary of each dataset. The number of lines (#Lines) of student code includes every line from the starting declaration of a class/method to the ending brace symbol.

TABLE IV: Assignments Used for the Evaluation

Assignment	#Source files	#Lines of student code (average)	#Lines of student comment (average)
A_1	120	39.5	4.0
A_2	105	21.0	4.1
A_3	100	49.4	4.4
A_4	90	49.3	5.9
A_5	75	115.7	8.1

B. Finding Suspects

The suspect finder calculates the similarity S_{NOR}, which is the cosine similarity of trigram vectors made of normalized source codes. The distribution of S_{NOR} for each dataset is shown in Fig. 4.

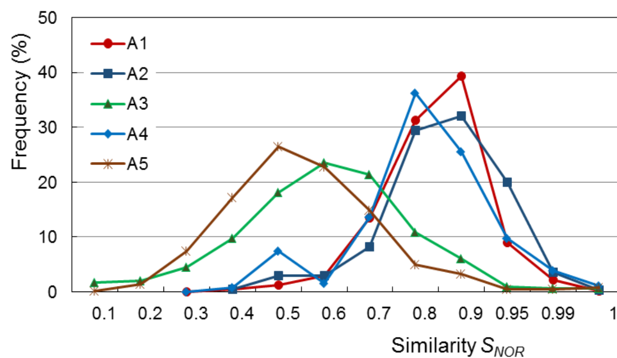

Fig. 4. Distribution of S_{NOR}.

According to our experience, most of plagiarism pairs satisfy $S_{NOR} > 0.95$, and most of pairs satisfying $S_{NOR} > 0.99$ are plagiarisms. The distribution was different for each assignment, but the pairs satisfying $S_{NOR} > 0.99$ were less than 1% for all of the assignments.

C. Analyzing Code

The code analyzer calculates the similarity S_{RAW}, which is the cosine similarity of trigram vectors made of raw source codes including white spaces. Fig. 5 shows the distribution of S_{RAW}. This figure looks like Fig.4, but the distributions of similarity are wider and shift toward the lower similarity. The pairs satisfying $S_{RAW} > 0.9$ were less than 5% for all of the assignments.

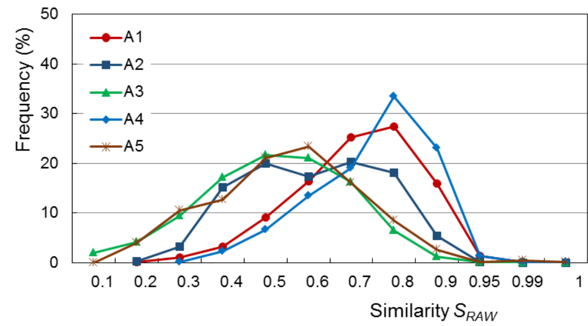

Fig. 5. Distribution of S_{RAW}.

Then, the code analyzer creates a sequence of peculiar patterns in each source code, and finds the LCS for each pair of codes. Fig. 6 shows the length of the subsequence, S_{PTN}. Peculiar patterns may match accidentally once or twice in pairs of source codes. The pairs satisfying $S_{PTN} > 3$ were less than 5% for most of the assignments.

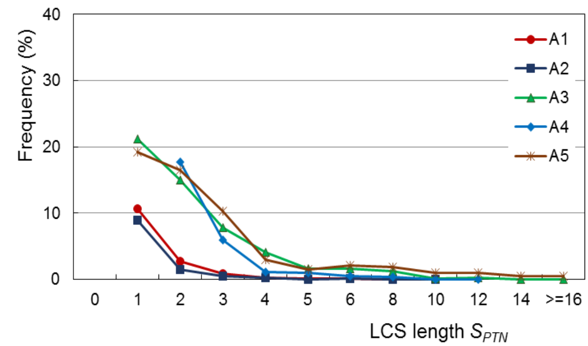

Fig. 6. Distribution of S_{PTN}.

D. Analyzing Comment

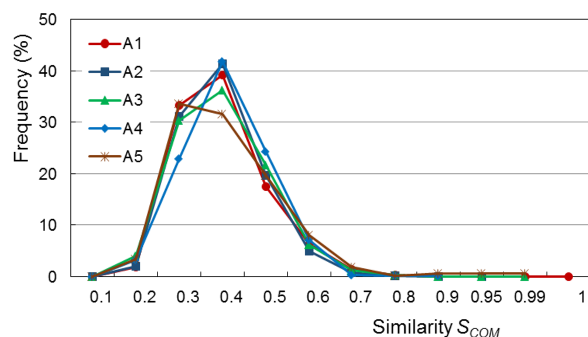

Fig. 7. Distribution of S_{COM}.

The comment analyzer calculates the similarity S_{COM}, which is the ratio of LCS length over the total length of a student comment. For programs with the very short comment, S_{COM} was assumed to be 0. Fig. 7 shows the distribution of S_{COM}. The distribution of each assignment was almost same in this figure. The similarity S_{COM} tends to lower than the similarity S_{RAW} in general, because the representation of natural language in the student comment is more flexible than the Java language in the source code. The pairs satisfying $S_{COM} > 0.6$ were less than 5% for most of the assignments.

E. Plagiarism and Evidence

Based on the considerations mentioned above and the statistical significance levels $p < 0.05$ and $p < 0.01$ which are commonly used in the statistics, the following criteria were used to determine suspects and plagiarisms for every assignment in the evaluation:

- If $S_{NOR} > 0.99$, the pair of programs is suspect.
- If $S_{RAW} > 0.95$ or $S_{PTN} > 5$ or $S_{COM} > 0.7$, the suspect pair is indictable with strong evidence.
- If $S_{RAW} > 0.9$ or $S_{PTN} > 3$ or $S_{COM} > 0.6$, the suspect pair is indictable with weak evidence.

Table V shows the number of suspect and indictable pairs with strong and weak evidence in each dataset. In this table, "#distinct files" means the number of source files involved in the pairs. Note that it is not always double the number of pairs, because one particular file may participate in two or more pairs.

TABLE V: NUMBER OF PLAGIARISM PAIRS

(A) WITH STRONG EVIDENCE

	Suspect	Evidence			Indictable	
Assign-ment	S_{NOR} > 0.99	S_{RAW} > 0.95	S_{PTN} > 5	S_{COM} > 0.7	Strong evidence	#distinct files
A_1	6	1	0	0	1	2
A_2	14	1	0	1	1	2
A_3	32	4	4	1	5	10
A_4	41	9	3	2	13	22
A_5	22	15	18	10	19	15

(B) WITH WEAK EVIDENCE

	Suspect	Evidence			Indictable	
Assign-ment	S_{NOR} > 0.99	S_{RAW} > 0.9	S_{PTN} > 3	S_{COM} > 0.6	Weak evidence	#distinct files
A_1	6	5	0	3	6	9
A_2	14	4	0	2	5	7
A_3	32	8	5	1	10	16
A_4	41	16	6	2	21	26
A_5	22	16	21	14	22	20

In Table V (a), there was a suspect pair that satisfies $S_{RAW} > 0.95$ for assignment A_1 and A_2. By a manual inspection, the two files in each pair were almost same except some of identifiers (in Level 2 defined in Table I). For assignment A_3, there were 4 pairs that satisfy $S_{RAW} > 0.95$ and $S_{PTN} > 5$. Actually, 3 pairs satisfied the both, one satisfied only the former, and another satisfied only the latter. By a manual inspection, 2 of the first 3 pairs were identical (in Level 0) and the other was in Level 2. For assignment A_4, there were 9, 3, and 2 pairs satisfying $S_{RAW} > 0.95$, $S_{PTN} > 5$, and $S_{COM} > 0.7$, respectively. This means that only one pair satisfied two of the three conditions and the others satisfied only one. There were three identical programs having $S_{RAW} = 1$ and $S_{PTN} = 0$. The reason is that these programs were perfectly formatted. For assignment A_5, there were 18 pairs satisfying $S_{PTN} > 5$, which included all of 15 pairs satisfying $S_{RAW} > 0.95$. They also included 9 of 10 pairs satisfying $S_{COM} > 0.7$. Actually, there were two large plagiarist groups, 5 students joined in the former group and 4 students joined in the latter group. For this reason, the number of indictable pairs is larger than the number of distinct files.

In Table V (b), threshold values to find evidence were loosened, thus more evidences were found especially for assignments A_1 and A_2. In these assignments, the evidence of plagiarisms tends to be weak, because the number of lines of the student code is small. You may think that it is possible to make strong evidence by accumulating weak evidences, but this idea fails to work on the programs in the evaluation.

Based on the above considerations, the proposed method is effective to find many plagiarisms with evidence among elementary student programs. Although the effect depends on the assignments, the method can find many strong and weak evidences of plagiarisms.

VI. CONCLUSION

This paper has presented a method to find plagiarisms with evidence among a set of relatively small and simple student programs. The method is based on the n-gram vector and the longest common subsequence. The notable features of the method are based on white spaces and peculiar patterns in the source codes to find evidence of plagiarisms. The effectiveness of the method has been demonstrated by using actual student programs. The method has been used and improved since 2009 at Tokyo Denki University, helping the instructor and teaching assistants.

The ability of finding evidence declines if students use code formatter tools before they submit their assignments. However, most of students in elementary courses prefer to write programs by their hands. Moreover, they cannot make good use of such tools. The current implementation runs as an offline batch processing, and is separated from a grading system that checks the functionality and structure of the student programs. The author is developing a new integrated online system, which accepts and tests student programs on the web, and gives a warning message immediately if a program is similar to another.

REFERENCES

[1] R. Koschke, "Survey of research on software clones," in *Proc. Dagstuhl Seminar on Duplication, Redundancy, and Similarity in Software,* 2007, no. 06301.

[2] U. Garg, "Plagiarism and detection tools: An overview," *Research Cell: Int. Journal of Engineering Sciences*, vol. 2, pp. 92-97, 2011.
[3] U. Inoue and S. Wada, "Detecting plagiarisms in elementary programming courses," in *Proc. 9th Int. Conf. on Fuzzy Systems and Knowledge Discovery (FSKD '12)*, 2012, pp. 2322-2326.
[4] S. Schleimer, D. Wilkerson, and A. Aiken, "Winnowing: Local algorithms for document fingerprinting," in *Proc. Int. Conf. on Management of Data (SIGMOD '03)*, ACM, 2003, pp. 76-85.
[5] A. Aiken. Moss: A system for detecting software plagiarism. [Online]. Available: http://theory.stanford.edu/~aiken/moss/
[6] L. Prechelt, G. Malpohl, and M. Philippsen, "Finding plagiarisms among a set of programs with JPlag," *J. Universal Computer Science*, vol. 8, no. 11, 2002, pp. 1016-1038.
[7] G. Malpohl. JPlag: Detecting software plagiarisms. [Online]. Available: https://www.ipd.uni-karlsruhe.de/jplag/
[8] R. Koschke, R. Falke, and P. Frenzel "Clone detection using abstract syntax suffix trees," in *Proc. 13th Working Conf. on Reverse Engineering (WCRD '06),* IEEE, 2006, pp. 253-262.
[9] D. Schuler, V. Dallmeier, and C. Lindig, "A dynamic birthmark for Java," in *Proc. 22nd Int. Conf. on Automated Software Engineering (ASE '07)*, IEEE/ACM, 2007, pp. 274-283.
[10] B. Lu, F. Liu, X. Ge, B. Liu, and X. Luo, "A software birthmark based on dynamic opcode n-gram," in *Proc. 1st Int. Conf. on Semantic Computing (ICSC '07)*, IEEE, 2007, pp. 37-44.
[11] K. Fukuda, and H. Tamada, "A dynamic birthmark from analyzing operand stack runtime behavior to detect copied software," in *Proc. 14th Int. Conf. on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD '13)*, IEEE/ACIS, 2013, pp. 505-510.
[12] J. Faidhi and S. Robinson, "An empirical approach for detecting program similarity and plagiarism within a university programming environment," *Computers & Education*, Elsevier, vol. 11, pp. 11-19, 1987.

Ushio Inoue received his B.Eng. and D.Eng. degrees from Nagoya University, Japan in 1975 and 1992 respectively. In 1975, he joined NTT Laboratories, where he was engaged in research of information retrieval and database management systems. In 1995, he was a senior research manager at NTT Data, where he developed distributed and multimedia information systems. Since 2004, he is a professor of Tokyo Denki University, Japan. Currently his research interests include geographic information systems, information retrieval and recommendation, and education support systems. Prof. Inoue is a member of ACM, IEICE, IPSJ, and GISA. He is currently a Vice Chair of ACM SIGMOD Japan.