A Source Code Plagiarism Detecting Method Using Alignment with Abstract Syntax Tree Elements

Hiroshi Kikuchi*, Takaaki Goto*, Mitsuo Wakatsuki*, Tetsuro Nishino*
* Graduate School of Informatics and Engineering,
The University of Electro-Communications,
1-5-1 Chofugaoka, Chofu, Tokyo, 182-8585, Japan
Email: { kikuchi, gototakaaki, wakatsuki.mitsuo, nishino }@uec.ac.jp

***Abstract*—Learning to program is an important subject in computer science courses. During programming exercises, plagiarism by copying and pasting can lead to problems for fair evaluation. Some methods of plagiarism detection are currently available, such as `sim`. However, because `sim` is easily influenced by changing the identifier or program statement order, it fails to do enough to support plagiarism detection.**

In this paper, we propose a plagiarism detection method which is not influenced by changing the identifier or program statement order. We also explain our method's capabilities by comparing it to the `sim` plagiarism detector. Furthermore, we reveal how our method successfully detects the presence of plagiarism.

I. Introduction

Programming education is one of the most important subjects in computer science courses. In programming exercises, plagiarism caused by copying and pasting leads to problems for fair evaluation. In this study, plagiarism is defined as someone handing in a report or documentation as his or her own original work which was, in fact, written or created by someone else.

Plagiarism is a serious problem. It is difficult for teachers to detect plagiarism with the huge volume of reports they review and, therefore, they are unable to give fair evaluations. Furthermore, students cannot improve their skills if they plagiarize. In addition, teachers must be able to detect plagiarism in order to prevent dishonesty among their students. However, comparing source code or reports manually is very time-consuming.

Teachers would benefit from automatic plagiarism detection, which would allow them more time to focus on their students and to prepare new materials. It is also important to reveal the capabilities of plagiarism detection systems to discourage cheaters. Therefore, techniques for automatic plagiarism detection play an important role in education.

Some methods of plagiarism detection are currently available, such as `sim` [1]. However, because `sim` is easily influenced by changing the identifier or program statement order, it fails to provide enough support for plagiarism detection.

In this paper, we propose a plagiarism detection method which is not thwarted by changing the identifier or program statement order. We also explain our method's capabilities by comparing it to `sim`.

II. Background

A. Sequence Alignment

Sequence alignment is a method to calculate a correspondence relationship among strings by adding a space or shifting the alphabetic positions. Strings obtained after alignment are also called sequence alignment. The similarity between two strings can be described by a score. In our method, we first obtain tokens by lexical analysis, and then calculate the sequence alignment and compare the obtained similarity scores.

Here is an example. Let s and t be strings. Sometimes the lengths of s and t are different. We insert a gap symbol "-" in order to line up the strings ' lengths. This process is called alignment. Fig. 1 illustrates an example of calculating the alignment between "masters" and "stars".

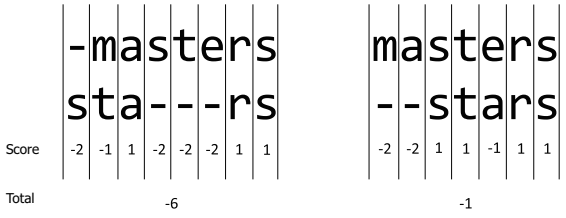

Fig. 1. Example of a sequence alignment.

As described in Fig. 1, it is possible to have several patterns of alignment. Usually, sequence alignment includes a minimum number of gap symbols and fewer different characters in the two strings. After obtaining the alignment, we then compare each character in the same position in the two strings. If two characters are the same, we score it as m point. If two characters are different, we score it as d. Moreover, if one or both characters include a gap symbol, then we score it as g. We can obtain an alignment score by adding the scores of every character. If $m \geq d \geq g$ or $m \geq g \geq d$, the alignment with the maximum number of total scores is the optimal alignment. If we assume $m = 1, d = -1, g = -2$, then we can obtain alignment scores for the left- and right-hand sides of Fig. 1 as -6 and -1, respectively. It can be seen that the right-hand alignment is appropriate for this example.

Usually, calculating an alignment score can be done using dynamic programming. The Needleman-Wunsch algorithm [2] is an efficient algorithm based on dynamic programming.

978-1-4799-5604-3/14/$31.00 copyright 2014 IEEE
SNPD 2014, June 30-July 2, 2014, Las Vegas, USA

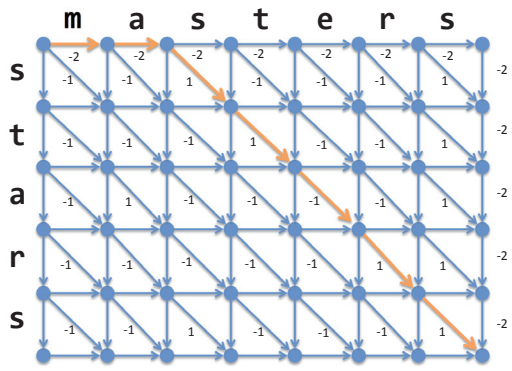

Fig. 2. Example of an alignment graph.

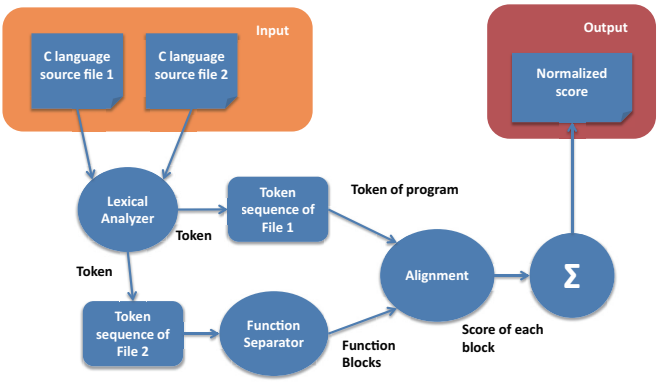

Fig. 3. Outline of process flow for sim.

Calculating the alignment score by using dynamic programming is done as follows [3]. Let Σ be a set of characters. Let s and t be strings, and $|s|$ and $|t|$ indicate the length of each string. $s[i]$ means the ith character in a string. We also use "$-$" as a gap symbol. Note that the gap symbol is not a member of Σ, and it cannot appear in the same position of the alignments.

Here we show a definition of score function which calculates scores by checking consistent and inconsistent gaps for each character in the strings. $w(x, y)$ is a function which means $(\Sigma \cup \{-\}) \times (\Sigma \cup \{-\}) \mapsto \mathbb{R}$, it holds $w(x, y) = w(y, x)$ under $x, y \in \Sigma \cup \{-\}$. In this paper, we use the following definition:

$$w(x,y) = \begin{cases} m(if\ x = y) \\ d(if\ x \neq y\ and\ y \neq -) \\ g(if\ x \neq y\ and\ y = -) \end{cases} \quad (1)$$

where $x \in \Sigma$, $y \in \Sigma \cup \{-\}$.

Let D be a matrix of $|s| \times |t|$. Matrix D stores the intermediate results of the alignment scores, which can be calculated by the following recursive formula:

$$D(i,0) = i \times g \quad (2)$$

$$D(0,j) = j \times g \quad (3)$$

$$D(i,j) = \max \begin{cases} D(i-1,j-1) + w(s[i], t[j]) \\ D(i-1,j) + g \\ D(i,j-1) + g \end{cases} \quad (4)$$

This formula calculates the maximum weighted path on the alignment graph as shown in Fig. 2.

B. Related Works

Gitchell et al. proposed a similarity index "`sim`" using sequence alignment for detecting plagiarism in a program's source code [1]. `sim` calculates the similarity between two source codes using a normalized value between 0.0 and 1.0. `sim`'s score can be obtained by the following steps (refer to Fig. 3):

1) Tokenize source code S and T
2) Tokens of T are divided into each function unit
3) Calculate alignments by comparing a sequence of token S (all tokens of S) and each function blocked tokens.
4) Summation of the alignment score is repeated by the number of functions
5) Normalize

At step 1, `sim` receives two source codes S, T as inputs to the system. These source codes are converted into token sequences by a parser. Gitchell et al. used `flex`, which is a major compiler-compiler, in order to obtain the parser. `sim` uses tokens such as operator, identifier, separator, and so on. For example, the following code can be converted into the token sequence below:

```
for (i = 0; i < max; i++)

TKN-FOR TKN-LPAREN TKN-ID-I TKN-EQUALS
        TKN-ZERO ...
```

In step 2, a token sequence containing one input source code is divided into function base units. In this method, the comparison source (in this case, the token sequence of source code S) is not divided into function base units. Only the token sequence of the comparison target is divided (in this case, the token sequence of source code T).

The next step (step 3) is to calculate the alignments between the token sequence of source code S and each function based unit of source code T. Gitchell et al. claim that, because of this mechanism, their method can detect plagiarism even if the function sequence of the target source code T is changed.

Alignment is calculated using the Needleman-Wunsch algorithm arranged by the Smith-Waterman algorithm. Let D be a matrix which stores the intermediate result of the alignment score. Here, let $D(i, j)$ be a partial token sequence indicating the score of the optimal alignment score, where $s[i]$ indicates the i-th element of token sequence s ($1 \leq i \leq |s|$), and $t[j]$ ($1 \leq j \leq |t|$) means the j-th element of token sequence t. Each element of D is calculated by the following recursive formula:

$$D(1,i) = i * g \quad (5)$$

$$D(j,1) = j * g \quad (6)$$

$$D(i,j) = \max \begin{cases} 0 \\ D(i-1,j-1) + w(s[i],t[j]) \\ D(i-1,j) + g \\ D(i,j-1) + g \end{cases} \quad (7)$$

The evaluation function $w(s[i],t[j])$ returns values whether or not one token corresponds to another token. The values returned by the evaluation function are as follows:

- Returns 2 if two tokens are the same and both token types are identifiers, returns 1 if two tokens are the same and token types are not identifiers.
- Returns -2 if two tokens are not the same and one of the tokens is a gap symbol.
- Returns 0 if two tokens are not the same and both token types are identifiers, returns -2 if two tokens are not the same and both token types are not identifiers.

In step 4, the total summation of optimal alignment score $D(i,j)$ between the token sequence of source code S and the token sequence of source code T, which is divided into each function unit, is calculated. The obtained value is called $score(S,T)$.

In step 5, we can obtain a normalized value (between 0.0 - 1.0) by the formula below. We call this value the "`sim` score".

$$s = \frac{2 \times score(S,T)}{score(S,S) + score(T,T)} \quad (8)$$

III. Similarity Index ilar

A. Outline of ilar

We propose a similarity index called `ilar` that improves upon the plagiarism detection capability of previous works. `ilar` outputs similarities between two source codes. `ilar` targets source codes developed using C or C++ language. The range of the similarity is between $0 \leq s \leq 1$. Fig. 4 illustrates the outline of the process flow for `ilar`.

`ilar` is computed as follows. Here, let S_1, S_2 be two input source code files for `ilar`.

1) Tokenize source code S_1, S_2. Obtained tokens of S_1, S_2 are divided into function base units.
2) Select one token sequence divided in function units from S_1, and calculate the alignment between the selected token sequence and each token sequence of S_2. Continue calculating the alignments between each left token sequence and each token sequence of S_2. Then, normalize using the obtained alignment score.
3) For each alignment score between methods S_1 and S_2, find the combination that has the maximum score. Then calculate the average score for the obtained value of the combination.

In Step 1, we obtain token sequences which are divided into function base units for two source codes, S_1, S_2 via the following procedure. First, we parse two source codes and obtain the abstract syntax trees. Then we extract all subtrees of the function parts from the obtained abstract syntax trees. For each subtree, we traverse it in a pre-ordered manner. Finally, we obtain token sequences consisting of token types. Fig. 5 shows an example of obtaining a token sequence.

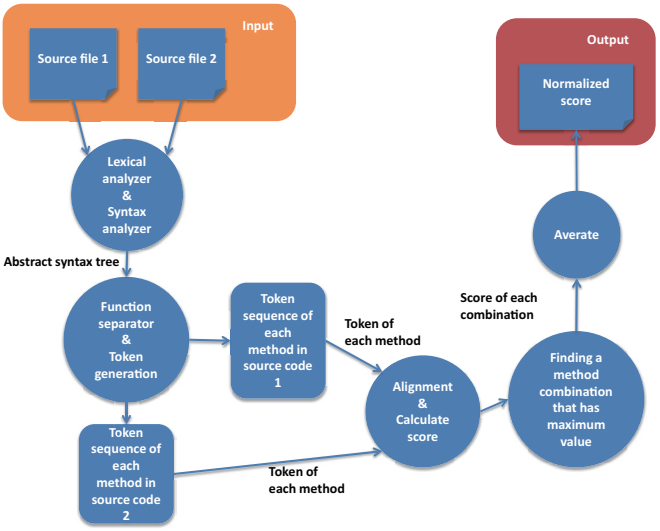

Fig. 4. Outline of process flow for ilar.

Here are the differences between the tokenizing processes of `sim` and `ilar`:

- `sim` uses lexical elements for tokens, while `ilar` uses syntactic element for tokens.
- `sim` includes the identifier name or literal value in the tokens. `ilar` does not include such information.

Tokens obtained by `ilar`'s methodology are more abstract compared to those of `sim`, which means the tokens of `ilar` are unaffected by identifier names or literal values in the source codes. `ilar` can detect plagiarism that uses changed variable or function names.

In step 2, we calculate the alignment score and normalize. We write $score(M_{i,j}, M_{x,y})$ as an optimal alignment score for token sequences between arbitrary functions $M_{i,j}$ and $M_{x,y}$. We calculate the normalized similarity index $s_m(M_{i,j}, M_{x,y})$ between two methods by the formula:

$$s_m(M_{i,j}, M_{x,y}) = \frac{1}{2}\left(1 + \frac{2 \times score(M_{i,j}, M_{x,y})}{score(M_{i,j}, M_{i,j}) + score(M_{x,y}, M_{x,y})}\right) \quad (9)$$

Fig. 6 shows a comparison of tokens located in the same positions. The algorithm deems a "match '' if the tokens in the same position are the same, and a "not match '' they are different. If there is no matching token, a gap symbol is inserted in the token sequences. The optimal alignment can be calculated by the Needleman-Wunsch algorithm. Parameters m, d, g of a score function w are required restrictions such that $m = -g$ and $m \leq d \leq g$.

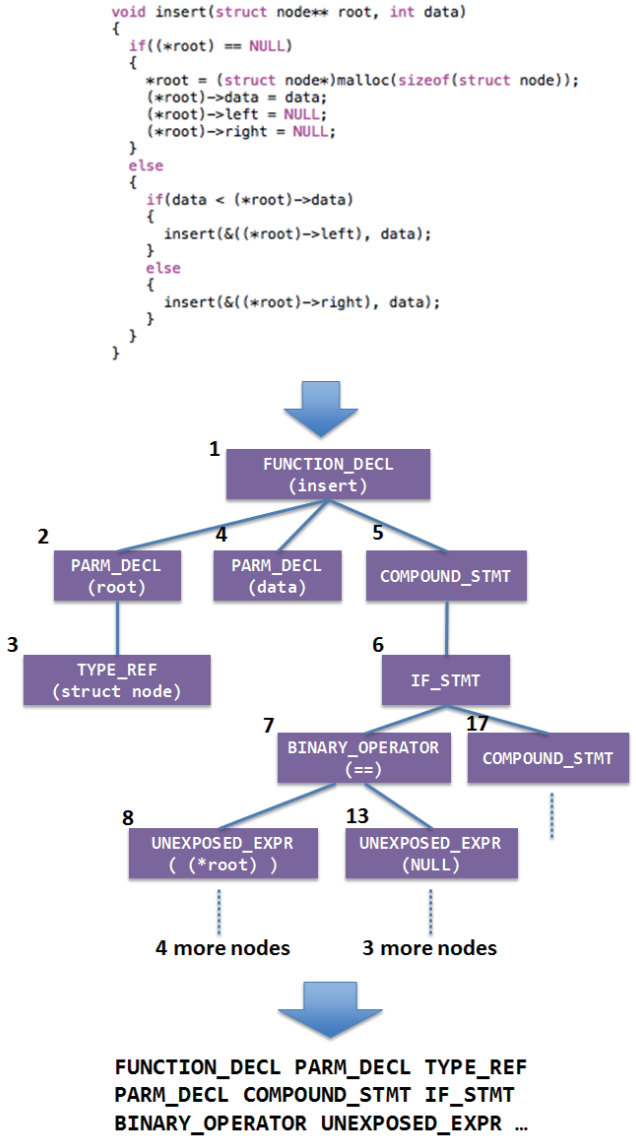

Fig. 5. Example of generating a token sequence based on `ilar`'s methodology.

In step 3, we find a combination that has a maximum value of $s_m(M_{1,i}, M_{2,j})$. Here we assume that $M_{1,i}$ and $M_{2,j}$ are elements of S_1, $M_{2,j}$, respectively. Fig. 7 illustrates a combinations of methods in S_1 and S_2.

Each method appears in one combination only; that is, there is no method which belongs to two combinations. A total of l similarity scores are obtained. The average scores of obtained combinations are calculated by dividing the sums of the scores by l. The obtained average score is a similarity index $s_s(S_1, S_2)$ of two source codes. We call this index the "`ilar` score".

B. Evaluation experiment of ilar (1)

We now describe our evaluation experiment of `ilar` using a fictitious exercise. We evaluate our method by comparing it to a previous work, `sim`. Our proposed method is implemented using Python (2.7.6). We also use `clang` [4] for parsing the source code.

We created a fictitious exercise. For this exercise, we prepared three solutions A, B, C for each part of the exercise. Solutions A and C were created without plagiarism, while B does contain plagiarized code. We assume that the developer who wrote code B (developer B) behaved in the following manner:

- Developer B is not familiar with programming. He/she took a programming class because it is compulsory. Developer B wants to get an easy credit, so he/she decides to plagiarize.
- Developer B has obtained a program from an earlier exercise or a friend's report. Here we assume that solution A is the obtained program. Developer B makes a solution for each exercise 1 to 4 using obtained program A.
- Developer B has to change the source code in order to ensure that the plagiarism is not detected. He/she modifies the obtained program without a compile error or unexpected behavior in order to obtain a good grade.

Developer B modifies solution A into solution B as follows:

- Function names other than the main function are renamed as far as possible.
- Names of variables, constants, and macros are changed.
- Changes are made to the statement order, such as function and variable declarations, to avoid errors and bugs.

We calculated the similarities among the solutions using `sim` and `ilar`, and compared the obtained values. The fictitious exercises are as follows:

- (Exercise 1) Make a variable-length list which stores strings by using a linked list.
- (Exercise 2) Make a reverse polish notation calculator by using a stack.
- (Exercise 3) Sort integers by using a queue.
- (Exercise 4) Sort integers in ascending order by using a binary search tree.

Tables I and II indicate similarities between source codes A and B, and A and C, respectively. Fig. 8 and Fig. 9 illustrate its graph.

TABLE I. SIMILARITIES BETWEEN SOURCE CODES A AND B

	ilar	sim
Exercise 1	0.928070175	0.267774147
Exercise 2	0.85955384	0.488528902
Exercise 3	0.974926254	0.393674724
Exercise 4	0.810762165	0.129985229

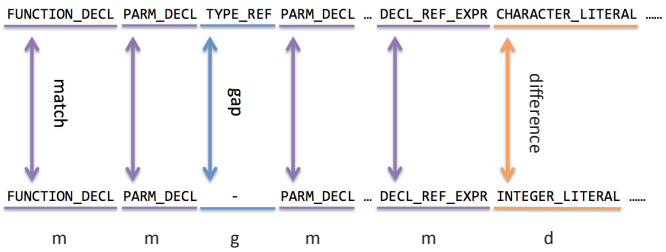

Fig. 6. Example of a comparison of token alignment for ilar.

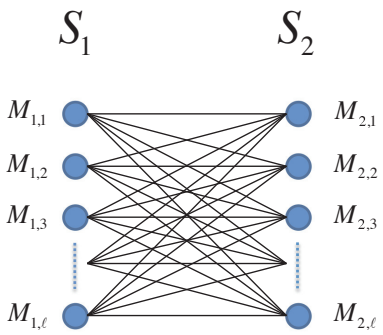

Fig. 7. Combination of methods between two source codes S_1 and S_2.

TABLE II. SIMILARITIES BETWEEN SOURCE CODE A AND C

	ilar	sim
Exercise 1	0.667386289	0.091678506
Exercise 2	0.532739508	0.043478261
Exercise 3	0.377809619	0.006379585
Exercise 4	0.484219204	0.008836524

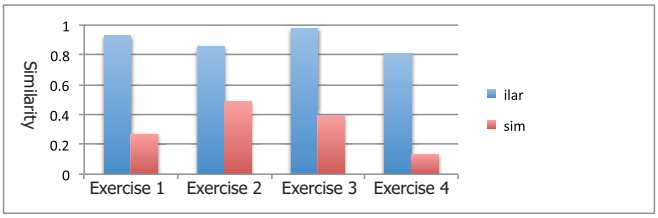

Fig. 8. Comparing similarities between sources code A and B.

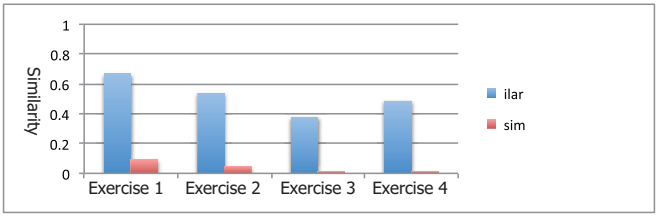

Fig. 9. Comparing similarities between source codes A and C.

C. Discussion evaluation experiment (1)

Here we discuss the evaluation experiment performed on a fictional exercise (experiment 1). In total, `ilar` scores are higher than the `sim` scores because `ilar` compares tokens using elements of abstract syntax trees, while `sim` compares tokens using lexical elements. Therefore, `ilar` obtains many corresponding tokens because it makes a rough comparison of token sequences.

Here we discuss the results of Fig. 8. `ilar`'s score for exercise 2 is lower than the score for exercise 1. On the other hand, `sim`'s score for exercise 2 is higher than the score for exercise 1. It is believed that source code A of exercise 1 has more readily-replaceable statements than source code A of exercise 2. Moreover, the source code of exercise 1 has only a few variables, while exercise 2 has many different identifiers. Therefore, `sim` did not obtain a higher score.

From the above consideration, we see that the `sim` score of exercise 1 was lower than for exercise 2. On the other hand, the `ilar` score will continue to rise even if the identifier name or exchanging statements change.

D. Evaluation experiment of ilar (2)

We performed an experiment to detect plagiarism in order to evaluate the superiority of `ilar` score by comparing its score to `sim`. In this experiment, we used some actual source codes used in a class. However, we could not show the source codes because of licensing policies.

First, we conducted a preliminary experiment to obtain a threshold value for judging plagiarism.

1) Preliminary experiment: Preliminary experiment: Source codes used in III-B were prepared for this purpose. Here we used some actual source codes used in a class. This course was an introduction to C language, so the submitted source codes were often similar. We chose one source code from four as a standard, and then calculated for similarities between it and other codes. Table III and Fig. 11 illustrate the results of the preliminary experiment.

TABLE III. SIMILARITY SCORE OBTAINED IN THE PRELIMINARY EXPERIMENT

ilar score	sim score
9.72E-01	5.48E-01
9.65E-01	3.78E-01
9.46E-01	3.22E-01

The four source codes were very similar to each other. They differed only by their identifiers. From the obtained results, we set threshold values for `ilar` and `sim` as $t_i = 0.8$ and $t_s = 0.4$, respectively.

2) Experiment: The experiment was conducted on source codes from a university lecture devoted to improving programming skills. One source code was selected out of 64 codes

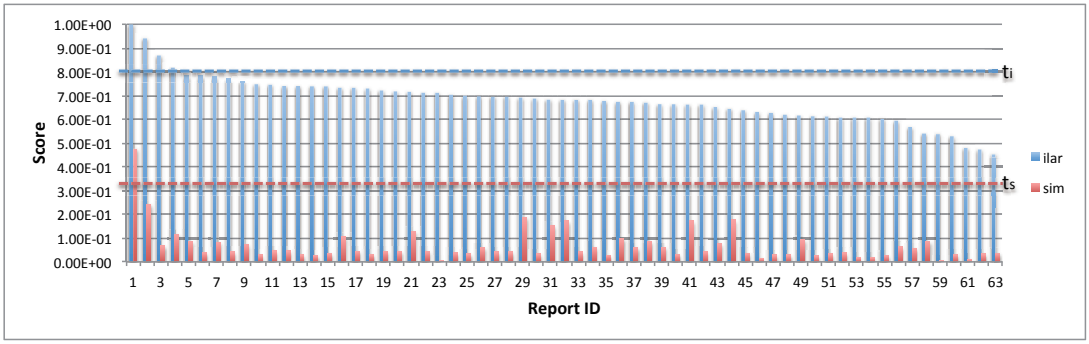

Fig. 10. Similarity between standard source code and other source code. Dotted upper line and lower line indicate t_i and t_s, respectively.

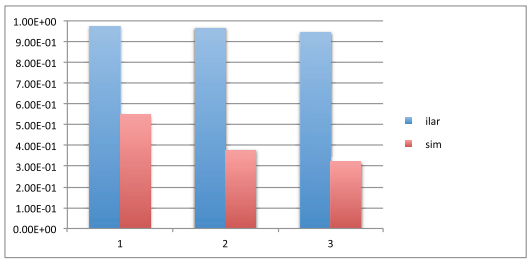

Fig. 11. Similarities in the preliminary experiment.

as a standard source code. The `sim` score was calculated by comparing the standard source code with each of the others. Fig. 10 shows the results of the experiment. In the figure , report IDs are sorted in descending order.

E. Discussion for evaluation experiment (2)

Here we discuss the evaluation experiment performed on actual source code (experiment 2). From the report of ID 2, it can be seen that the `ilar` score exceeded the thresholds. Value t_i, on the other hand, had `sim` scores that did not exceed the thresholds of value t_s. We can suspect plagiarism in the case of report ID 1 whose scores exceed t_i and t_s. However, report ID 2 only exceeded the `ilar` score for thresholds t_i; therefore, in this case, `ilar` detected plagiarism, but `sim` did not. The source code of report ID 2 had the following similarities and differences:

1) Similarities:

- This exercise targeted implementing functions using structures which indicate elements of a linked list based on specifications. Judging from the nature of the exercise, identifiers in the source code can be similar. In particular, function names are specified for the exercise; thus, each report has the same function name. Moreover, some names of constants and variables were used in common because of the example's implementation.
- The coding styles of each report were similar. For example, at a certain point in the source code, spaces were inserted around the equal sign, =, while in other places they were not. This was found in five out of eight functions.

2) Differences:

- The coding style of report ID 2 was different from that of the standard source code. In the standard source code, curly brackets ({ }) were not used if `while` or `if` statements had only one line of code. In report ID 2's source code, the { } were used in every case.
- A condition in one `if` statement was the opposite, and statements in `if` and `else` statements were exchanged.
- One literal notation out of four was different.
- The standard source code contained `output` statements, while the source code of report ID 2 did not contain any.
- One assignment statement order was transposed.
- One of seven `declare` names for the same purpose was different.

From the above, we can judge that report ID 2 is plagiarism by the operations described above.

IV. Conclusion

In this paper, we proposed a similarity index which cannot be defeated by changing identifier names or statement order. We performed an experiment for our proposed method using actual reports from a university programming class. Our method could reveal the presence of plagiarism.

In the future, we will work to improve the processing efficiency and accuracy in detecting plagiarism.

References

[1] D. Gitchell and N. Tran, "Sim: a utility for detecting similarity in computer programs," *SIGCSE Bull.*, vol. 31, no. 1, pp. 266–270, Mar. 1999. [Online]. Available: http://doi.acm.org/10.1145/384266.299783

[2] S. B. Needleman and C. D. Wunsch, "A general method applicable to the search for similarities in the amino acid sequence of two proteins," *Journal of Molecular Biology*, vol. 48, no. 3, pp. 443 – 453, 1970. [Online]. Available: http://www.sciencedirect.com/science/article/pii/0022283670900574

[3] Tatsuya Akutsu, *Mathematical Models and Algorithms in Bioinformatics*. KYORITSU SHUPPAN, 2 2007, pp. 17–28, (in Japanese).

[4] C. Project, "clang: a C language family frontend for LLVM," http://clang.llvm.org/.