Copyright ⓒ 2012 KSII

Plagiarism Detection among Source Codes using Adaptive Methods

Yun-Jung Lee[1], Jin-Su Lim[2], Jeong-Hoon Ji[3], Hwaun-Gue Cho[4] and Gyun Woo[4]
[1]Center of U-Port IT Research and Education, Pusan National University
Busandaehak-ro 63beon-gil, Geumjeong-gu, Busan 609-735 Korea
[2]HA Control R&D Lab in LG Electronics
Seongsan-gu, Changwon-si, Gyeongsangnam-do 642-711 Korea
[3] Korean Intellectual Property Office
Seo-Gu Daejeon 302-701 Korea
[4]Dept. of Computer Science and Engineering, Pusan National University
Busandaehak-ro 63beon-gil, Geumjeong-gu, Busan 609-735 Korea
[e-mail: { leeyj01, elphy, hgcho,woogyun}@ pusan.ac.kr, jhji@kipo.go.kr]
*Corresponding author: Gyun Woo

Received June 29, 2011; revised April 23, 2012; accepted May 17, 2012; published June 25, 2012

Abstract

We propose an adaptive method for detecting plagiarized pairs from a large set of source code. This method is adaptive in that it uses an adaptive algorithm and it provides an adaptive threshold for determining plagiarism. Conventional algorithms are based on greedy string tiling or on local alignments of two code strings. However, most of them are not adaptive; they do not consider the characteristics of the program set, thereby causing a problem for a program set in which all the programs are inherently similar. We propose adaptive local alignment—a variant of local alignment that uses an adaptive similarity matrix. Each entry of this matrix is the logarithm of the probabilities of the keywords based on their frequency in a given program set. We also propose an adaptive threshold based on the local outlier factor (LOF), which represents the likelihood of an entity being an outlier. Experimental results indicate that our method is more sensitive than JPlag, which uses greedy string tiling for detecting plagiarism-suspected code pairs. Further, the adaptive threshold based on the LOF is shown to be effective, and the detection performance shows high sensitivity with negligible loss of specificity, compared with that using a fixed threshold.

Keywords: Plagiarism, program plagiarism detection, adaptive local alignment, similarity measurement, software similarity, local outlier factors.

A preliminary version of this paper appeared in ICUIMC 2011, February 21-23, Seoul, Korea. This version includes an adaptive threshold base on the LOF for determining the plarigiarism.

http://dx.doi.org/10.3837/tiis.2012.06.008

1. Introduction

Plagiarism of general documents is becoming a major problem in society. It is also becoming a very serious problem in universities. According to one survey, more than 5% of college students have had plagiarism-related experiences [1]. Recently, the President of Hungary resigned as it was alleged that his dissertation had been plagiarized[1]. Further, an IOC member of Korea who was elected as a member of parliament, has been investigated for suspected plagiarism of his thesis[2]. Since plagiarism occurs frequently in universities, it is almost impossible for an instructor to accurately appraise all students' work manually.

The problem of detecting plagiarism is becoming more serious in the area of computer programming. Compared to general text plagiarism, it is very hard for instructors to detect plagiarism in source codes. If the number of programs to inspect is large, it is nearly impossible to compare every pair of programs manually. For example, if there are more than 1,000 source codes in a program set, the number of comparisons required is about 500,000. A lot of effort is required for a human inspector to examine all the code pairs in a large program and detect the plagiarized ones. Consequently, many automatic systems such as JPlag, YAC3, and MOSS are used to detect plagiarism in general texts and source codes [2][3][4][5][6][7].

In common with the detection of plagiarism in general text documents, there are a number of requirements for the detection of plagiarism in program source codes. First, the plagiarism detection should be quantitative and specific enough with respect to the similar regions and to the plagiarizing techniques. A plagiarism detection system should indicate what parts of the two programs are similar and how they are plagiarized. Since the detection system is not a human expert, it may fail to detect the plagiarism on a semantic level such as data structures and algorithms. In fact, most conventional detection systems largely detect syntax-level plagiarism and are being extended to cope with structure-level plagiarism. However, the key requirement is that the detection system should be sensitive enough to detect the suspicious region as much as possible.

Second, it is hard to collect actual examples of plagiarized source code pairs included in a large program set with the same functional behavior. To verify the effectiveness of a plagiarism detection system, many plagiarized source code pairs are needed. However, since plagiarism itself is illegal, plagiarized programs are not easily secured. Although plagiarized source codes can be made artificially, they can hardly be regarded as actual plagiarized data in a strict sense. This may be a problem as programs with the same functional behavior can be inherently similar to one another.

Finally, there is no standard reference model regarding whether one of two similar programs plagiarized the other. For example, if two students are assigned to write bubble sort programs, then the resulting programs will inevitably be similar. Lim et al. defined this case as pseudo-plagiarism that is not a result of plagiarism, but comes from a strong functional requirement [8]. The distribution of the similarity score of a program group is dependent on the program set itself. If we use a fixed threshold score of similarity to determine plagiarism-suspected pairs, it may cause over detection or misdetection. Therefore, an

[1] F. Facsar, "Hungary's president quits over alleged plagiarism," CNN, April 2, 2012 (http://edition.cnn.com/2012/04/02/world/europe/hungary-president-resigns).
[2] P. Hersh, "Another plagiarist on IOC?" Chicago Tribune, April 13, 2012 (http://www.chicagotribune.com/sports/globetrotting/chi-another-plagiarist-on-ioc-20120412,0,7137694.column).

adaptive threshold that takes into consideration the similarity distribution of the program set is necessary to handle pseudo-plagiarism cases.

In this paper, we propose a new, adaptive method for detecting the blocks in given program pairs that are similar. Our method is a variant of the local alignment [9]; in which we changed the similarity matrix to adaptively reflect the keyword frequencies for a given program group. We extended this basic idea [10] and constructed an adaptive similarity matrix from the probabilities of keyword occurrences for a given program set.

Further, our new method determines the cut-off threshold adaptively. It based on the idea that the code pairs that have a high similarity score regarded as outliers. This implies that the plagiarism detection problem reduced to that of outlier detection. Specifically, this paper proposes a new criterion based on the local outlier factor (LOF) to determine the cut-off threshold of similarity. The LOF indicates the degree of outlierness of each object in a dataset [11]. By calculating the LOF of each code pair using their similarity score, the cut-off threshold for detecting the plausible plagiarism can be adaptively determined using the LOF.

From experiments using program groups including artificially plagiarized source codes, we show that adaptive local alignment is especially effective in detecting plagiarism of source codes: in particular, it is superior to JPlag. The experimental results indicate that adaptive local alignment is more sensitive than greedy string tiling (GST). Further, adaptive threshold based on the LOF is found to be more effective than its static counterpart, which implies that the detection of plagiarized code pairs determined by the adaptive threshold is more sensitive, regardless of the program groups, than methods using the fixed threshold.

This paper is organized as follows. Chapter 2 discusses related work on the detection of plagiarism of source codes. Chapter 3 gives an overview of our proposed method and explains each procedure for detecting plagiarized code pairs in detail. Chapter 4 outlines the implementation of our system. Chapter 5 describes some experimental results. Chapter 6 concludes this paper.

2. Related Work

In this section, we will briefly review previously released plagiarism detection systems and the algorithms that were adopted in those systems. We will also explain the local alignment algorithm that is being used in computational biology.

2.1 Plagiarism Detection Systems and Algorithms

A plagiarized program can be defined as a program that has been produced from another one without a thorough understanding of the source code [12]. There are many previously released systems for detecting plagiarism. They can be classified into two categories: (1) systems for general text documents and (2) systems for program source codes. Since plagiarism in plain-text prevails widely compared to plagiarism in software, the detection of plagiarism in plain-text documents has been studied for a long time in information retrieval and document processing disciplines. Recently, concerns about program plagiarism have been increasing due to the many clever plagiarism tools and advanced Internet search technologies currently available. **Table 1** shows the application domains and the detection algorithms that the previously released systems were based on.

As shown in **Table 1**, there are some detection systems for program source code. JPlag is a stable system for detecting plagiarism. It finds pairs of similar programs among a given set of programs [4]. It is written in Java and analyzes program source code written in Java, Scheme,

C, or C++. JPlag takes as input a set of programs, compares those programs pairwise (computing for each pair a total similarity value and a set of similarity regions), and provides as output a set of HTML pages that allow for exploring and understanding the similarities found in detail. **Fig. 1** shows the JPlag result display page. The algorithm used in JPlag to compare program pairs is GST.

Table 1. Plagiarism detection tools for plain texts and for program source codes.

System Name	Domain	Base Method	Usage	Cost
Plagiarism.org	Plain text	Fingerprint	On-line	Free
IntegriGuard	Plain text	Unknown	On-line	$4.95/Month
EVE2	Plain text	Unknown	Stand-alone	$19.99
CopyCatch	Student reports	Lexical matching	Stand-alone	Free
MatchDetectReveal	Plain text	Suffix Tree matching	DB Service	Free
SCAM	Digital library	Vector-space model	DB Service	Free
YAP3	Software	greedy-string-tiling	Stand-alone	Free
Clonechecker	Software	Unknown	Stand-alone	Commercial
MOSS	Software	Winnowing	Web Service	Free
JPlag	Software	greedy-string-tiling	Web Service	Free
SID	Software	Data Compression	Web Service	Free
SIM	Software	Local Alignment	Stand-alone	Free
CodeMatch	Software	Fingerprint	Stand-alone	Commercial
Viper	Plain text	Unknown	On-line	Free
PlagiarismDetect.com	Webpage	Unknown	Web Service	$4.95
Copyscape	Webpage	Unknown	Web Service	$0.05/Search

MOSS is an automatic system for determining the similarity of programs, and is a widely-used plagiarism detection service available on the Internet since 1997 [7]. The system is based on winnowing, a local fingerprinting algorithm, and can analyze program source code written in C, C++, Java, C# and so on. Since fingerprinting is a relatively simple method, the range of programming languages supported can be wider than other methods.

Three methodologies for source code plagiarism detection are widely used. One methodology is based on software metric comparison [13][14][15]. For example, Halstead's software metric [16] used to check the similarity of two programs. Fingerprinting is also a popular methodology that used in the early stages of a detection system. Fingerprinting is a procedure that extracts information from source code such as frequency of keywords and unique symbol count. Fingerprints are easy to compute, but the effectiveness of this approach is not very good.

Another group of methods compares program structures. This approach is less sensitive to plagiarism attack techniques (i.e., techniques that are used to defeat plagiarism detection methods). Structure-based plagiarism detection methods generally consist of two steps. The first step is the construction of other forms of objects that can be easily compared, from the given programs. These generated objects are typically token strings. The second step is the comparison of these two token strings by some sort of string matching algorithms. Not only the algorithms for finding common intervals [17], but other clever methods such as greedy-string-tiling [4], local alignment [3], and parse tree comparison [18][19] are also widely used for structural comparison [20][21]. For comparing the parse trees, a tree matching algorithm is used instead of string matching algorithms.

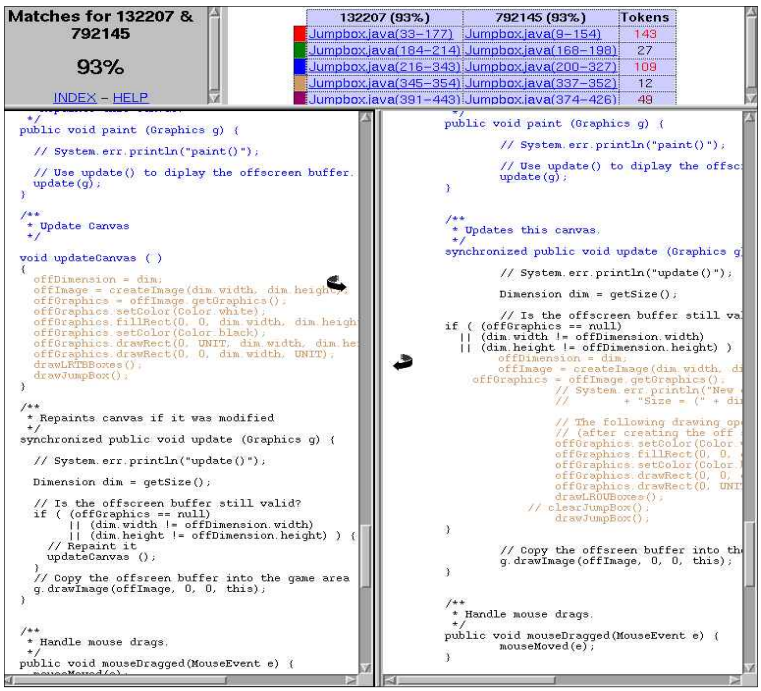


Fig. 1. A snapshot of JPlag results displaying a pair of programs to be examined [4].

The last category of methods for measuring the similarity of two programs is based on the Kolmogorov complexity of information theory [22]. The Kolmogorov complexity of a given string is defined by the length of the minimum string that is required to represent the program. This length represents the amount of information included in that string. In reality, this minimum length is evaluated by compressing the given string using a compression algorithm, say RSA. If two programs, say A and B, are very similar, the size of the compressed result of the concatenation of A and B will be similar to that of compressing A or B.

Although the Kolmogorov complexity based method is effective for comparing two documents, it is hard to locate the regions that are very similar in the given documents. Furthermore, programs may contain unreachable codes, but this method is not sensitive to this kind of attack. In spite of this weakness, this method can be effectively used for narrowing the candidate documents by selecting the documents that are similar before applying a discreet comparison method to them.

2.2 Local Alignment

Local alignment was defined by Smith and Waterman [9] in 1981, and is usually called the Smith-Waterman algorithm. The Smith-Waterman algorithm was originally developed to find similar regions in two nucleotide or protein sequences. Local alignment adopts the dynamic programming technique, which constructs the optimal solution of a problem from the optimal solutions of subproblems that are usually cached in a table to avoid recomputation.

Local alignment focuses on comparing two linearized sequences in order to find the longest subsequences that closely match. The score of a sequence block is the sum of the individual scores, and the optimal alignment score is the score of the highest-scoring sequence block. Formally, two sequences of P and Q are given, where p_i is an element of P and q_j is an element of Q ($i \leq |P|, j \leq |Q|$), where the individual score of $score(p_i, q_j)$ is defined as follows:

$$score\left(p_i, q_j\right) = \begin{cases} m, & \text{if } p_i \neq q_j \\ d, & \text{if } p_i = q_j \end{cases}$$

where m is the score for a match and d for a mismatch; m is generally positive whereas d is negative. The optimal alignment score $score(i, j)$ is computed using the following function:

$$score(i,j) = max \begin{Bmatrix} score\ (i, j-1) + g \\ score\ (i-1, j-1) + score\ (p_i, q_j) \\ score\ (i-1, j) + g \\ 0 \end{Bmatrix}$$

Here, g is the penalty for matching using a gap. The above equation indicates that the alignment score up to i and j is the maximum of four cases: gap insertion, matching (including mismatching), gap deletion, and no matching.

3. The Adaptive Local Alignment Algorithm

3.1 Procedural Overview

In this section, we describe a new model that is required to adaptively measure the similarity between two programs. Our algorithm that detects plagiarism using local alignment operates in three phases: the program linearization phase, the local alignment phase, and the similarity measurement phase. **Fig. 2** shows the plagiarism detection procedure used by adaptive local alignment.

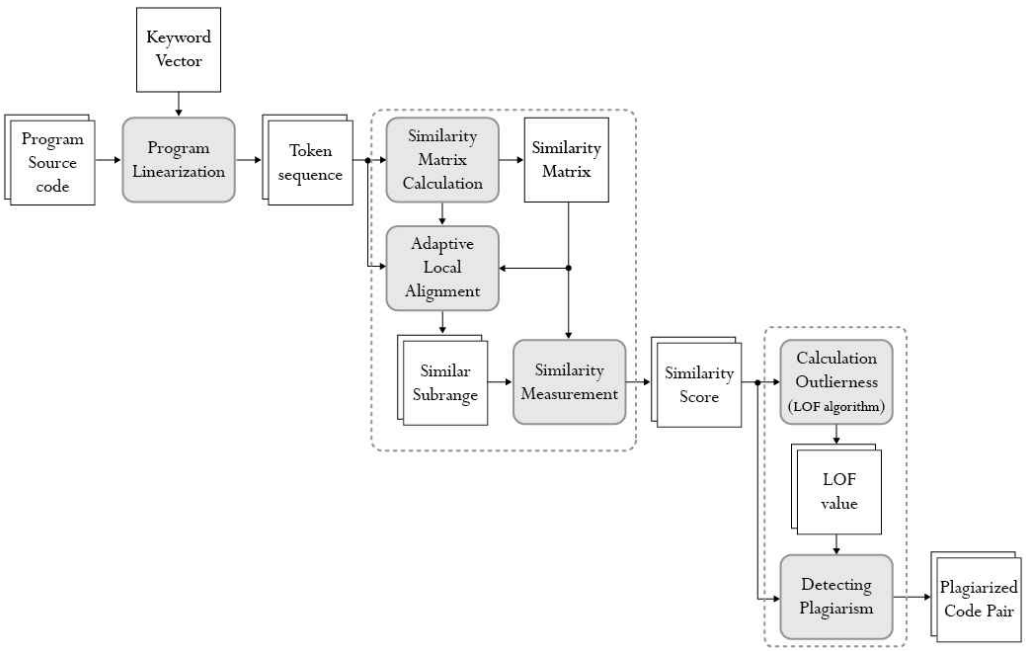


Fig. 2. The similarity measurement procedure using the adaptive methods

The dashed boxes in **Fig. 2** show the adaptive parts of our proposed method. The first box indicates the procedure for adaptive local alignment, while the second indicates the procedure for producing the adaptive threshold. These two steps highlight the basic idea underlying our

adaptive method.

3.2 Program Linearization

In the first step of the procedure depicted in **Fig. 2** (Program Linearization), the system generates the token sequences from a given program set. Program linearization is the first step that extracts the sequence of predefined tokens from each program. The keyword vector prescribes the set of tokens that should be extracted from the given programs. Therefore, the keyword vector is defined according to the host programming language. In addition to keywords, the keyword vector also includes the operator symbols since the aim of the keyword vector is to reflect the structural characteristics of programs—such as control flow, subprograms, code blocks, and so on.

The linearization procedure of our system has a novel feature—namely, static tracing—which cannot be found in other systems. Static tracing is a technique that executes a program statically at the syntax level to generate the token sequence in that order. In order to execute a program syntactically, the syntax tree is constructed prior to the tracing. **Fig. 3** shows an example of the result of program linearization utilizing static tracing.

Source Code	Token Sequence

1 int main()
2 {
3 int num1 = 100;
4 int num2 = 200;
5 swap(&num1, &num2);
6 printf(num1 = %d,
num2 = %d", num1, num2);
7 }
8 void swap(int *m, int *n)
9 {
10 int *temp;
11 temp = m;
12 m = n;
13 n = temp;
14 }
	1 FUNC_CALL, # main()
	2 BLOCK_START
	3 INT, ASSIGNMENT
	4 INT, ASSIGNMENT
	5 REFERENCE, REFERENCE
	8 FUNC_CALL, # swap(int*, int*)
	9 BLOCK_START
	10 INT, PTR
	11 ASSIGNMENT
	12 ASSIGNMENT
	13 ASSIGNMENT
	14 BLOCK_END, # swap
	6 UNREACHABLE_FUNC, # printf
	7 BLOCK_END, # main

Fig. 3. A simple source code and its token sequence generated from the static tracing.

The left half of **Fig. 3** shows a simple C program that swaps the contents of two integer variables using the function `swap`. The right half of **Fig. 3** is the corresponding token sequence that is generated by the linearization procedure using static tracing. The effect of static tracing is found around the function call `swap`. When the linearization procedure encounters a function call and the function is a user-defined one that has not been traced yet, the linearization procedure records the function call, `FUNC_CALL`, and continues to trace the body of that function. Tracing the called function statically, it returns to the calling site and continues. When it encounters system-defined functions, tracing is not performed and `UNREACHABLE_FUNC` is all that is recorded.

3.3 Adaptive Local Alignment

Adaptive local alignment is a variant of the original local alignment with respect to the similarity matrix. We compute the similarity of two programs depending on the set of programs that contains the subject programs. The rationale is that the similarity of two

programs should not be determined solely on the basis of the programs themselves, but rather the characteristics of the program group that they are involved in should be taken into consideration.

The basic strategy of adaptive local alignment is that the matching score of a keyword should reflect the frequencies of keywords. More specifically, we attribute matching scores to keywords in inverse proportion to their frequencies; we attribute high scores to low frequency keywords and low scores to high frequency keywords. The same rule is also applied as the penalty for mismatches. Since it is rare to see low frequency keywords being used by two programs at the same time, two programs that both use keywords of low frequency together should be considered quite similar. In this respect, our approach is more natural than the original local alignment.

The adaptive approach is immune to typical plagiarizing attacks such as those involving the insertion or deletion of meaningless or dummy keywords. **Table 2** shows some of the highest and lowest frequency keywords in the program group ICPC06-3, which consists of 157 programs (See Table 3). As shown in **Table 2**, assignments (‘=’) and block delimiters (‘{’and ‘}’) are the most frequently used keywords. Since the adaptive approach weakens the penalties for mismatching of these keywords, inserting or deleting these keywords has less effect. In contrast, inserting or deleting low frequency keywords such as ‘`struct`’ or ‘`switch`’ has a lot of influence on the overall similarity computed.

Table 2. The keywords with high and low frequencies in a typical program group. The total number of keywords extracted from this group is 13,104.

High frequency keyword	Frequency	Low frequency keyword	Frequency
Assignment ‘=’	12.64%	‘`struct`’	0.01%
Block Start ‘{’	10.26%	‘`delete`’	0.01%
Block End ‘}’	10.26%	‘`bool`’	0.02%
Equal ‘==’	6.40%	Assignment ‘-=’	0.02%
‘`if`’	6.28%	‘`switch`’	0.02%

The crucial part of the adaptive local alignment is the similarity matrix. Just like the original local alignment, the similarity matrix M is an $(r+1) \times (r+1)$ matrix if the kinds of keywords considered is r because the special gap symbol takes part as the $(r+1)$-th column and row index of the matrix. Each element $M(k_i, k_j)$ of the similarity matrix represents the score or the penalty for matching or mismatching: it represents a matching score if $k_i = k_j$ and a mismatching penalty if $k_i \neq k_j$. For gap columns and gap rows, the elements represent penalties for inserting or deleting gap (inserting gap to the other side) symbols.

In order to determine the adaptive similarity matrix, the frequencies of keywords should be computed beforehand. Let the entire set of programs that consist of n programs be $\mathrm{P}=\{p_1, p_2, \ldots, p_n\}$ and assume that $occur(p, k)$ denotes the number of occurrences of keyword k in program p. Then, the total number of occurrences of k in program group $\mathbb{P}$ can be defined as $occur(\mathbb{P}, k)=\sum_{p\in\mathbb{P}} occur(p, k)$. Based on this definition, the frequency $f_i^{\mathbb{P}}$ of a keyword k_i in a program group $\mathbb{P}$ is defined as follows:

$$f_i^{\mathbb{P}} = occur(\mathbb{P}, k_i) \Big/ \sum_{j=1}^{r} occur(\mathbb{P}, k_j)$$

Since the denominator $\sum_{j=1}^{r} occur(\mathbb{P}, k_j)$ denotes the sum of the number of occurrences of all the keywords, the frequency $f_i^{\mathbb{P}}$ of a keyword k_i lies between 0 and 1 ($0 \leq f_i^{\mathbb{P}} \leq 1$).

Using the keyword frequencies defined above, the adaptive similarity matrix $M^{\mathbb{P}}$ can be defined as follows:

$$M^{\mathbb{P}}(k_i, k_j)=\begin{cases} -\alpha \cdot \log_2 (f_i^{\mathbb{P}} \cdot f_j^{\mathbb{P}}) & \text{if } k_i=k_j \\ \beta \cdot \log_2 (f_i^{\mathbb{P}} \cdot f_j^{\mathbb{P}}) & \text{if } k_i \neq k_j \\ 4\beta \cdot \log_2 f_i^{\mathbb{P}} & \text{if } k_j \text{ is a gap and } k_i \text{ is not} \\ 4\beta \cdot \log_2 f_j^{\mathbb{P}} & \text{if } k_i \text{ is a gap and } k_j \text{ is not} \\ -\infty & \text{if both } k_i \text{ and } k_j \text{ are gaps} \end{cases}$$

Here, α and β are tuning parameters, and the sum of these parameters is currently set to 1 (that is α + β = 1). We can adjust the relative weights for a matching score and a mismatching penalty using these parameters. Adjusting α to be greater than β makes the matching score more significant in the final similarity score, while the reverse makes the mismatching penalty more significant. After tuning these parameters, α is set to 0.65 and β is 0.35.

To define the similarity matrix, we take the log of the keyword frequencies. Log odds are generally accepted in the information theory discipline. For instance, taking log odds is generally adopted when comparing the intensity of two signals, especially when the ratio of the intensity is sufficiently large. As shown in **Table 2**, the ratio of the frequency of the most frequent keyword and the least frequent one is in the thousands. Therefore, we adopt log odds here.

Disregarding the tuning parameters, the matching score is basically set to the log of the product of the frequencies of the keywords involved. This is the same for mismatching except that the mismatching penalties are set to be negative. Gap insertions or deletions should also be treated as penalties. Incidentally, according to the original local alignment, the penalty for gaps is twice that for mismatches. Reflecting the philosophy of the original local alignment, we make the penalty for gaps twice as large for mismatches: $2\beta \cdot \log_2(f_i^{\mathbb{P}} \cdot f_i^{\mathbb{P}})=4\beta \cdot \log_2 f_i^{\mathbb{P}}$.

The last case of two gap symbols should not occur because the aligned region can be enlarged to an arbitrary length if this is permitted. Hence, we set the score to be $-\infty$ to prevent this anomaly. This strategy is the same as the strategy that was used in the original local alignment.

3.4 Similarity Measurement

Using the similarity matrix, we can compute the similarity score of an aligned region between two programs. As a matter of fact, the aligned region is determined when the alignment score is being computed. However, it is convenient to assume that the conceptual alignment process takes place before the computation of the similarity score of two programs.

Let us assume that *align* is a function that produces a pair of aligned blocks from two programs, A and B. Say the aligned block taken from A is $\langle a_1, a_2, \ldots, a_m \rangle$ and the corresponding block from B is $\langle b_1, b_2, \ldots, b_m \rangle$, and let *align* produce the vector of pairs of corresponding keywords:

$$align(A, B)= \langle (a_1, b_1), (a_2, b_2), \ldots, (a_m, b_m) \rangle$$

Using the function *align* defined above, the absolute similarity score of two programs can be defined as follows:

$$SIM_{abs}(A, B) = \sum_{(a, b) \in align(A, B)} M^{\mathrm{P}}(a, b)$$

The absolute similarity score of two programs is the sum of the individual similarity score of the corresponding keywords (including gaps) in the aligned region.

In order to compare one similarity with another similarity, a normalized similarity measure is needed. Since SIM_{abs} largely depends on the lengths of the subject programs, it is not adequate to compare the similarities themselves. For instance, if two programs A and B are given, where the length of keyword sequence of A is larger than that of B, the absolute similarity $SIM_{abs}(A, A)$ will be greater than $SIM_{abs}(B, B)$.

One way to normalize the absolute similarity is to divide the similarity score by the sum of the self similarities. This definition of normalized similarity seems to be generally accepted [4][22] and the corresponding similarity function of $SIM_{sum}(A, B)$ can be defined as follows:

$$SIM_{sum}(A, B) = \frac{2SIM_{abs}(A, B)}{SIM_{abs}(A, A) + SIM_{abs}(B, B)}$$

Notice that the absolute similarity score of A and B (the numerator) is doubled in order to make the normalized similarity equal to 1.0 when A is identical to B.

Another way to normalize the similarity score is to divide it by the minimum value of the self similarities. If a program is made up of only the crucial parts of a program and the length of the original program is quite long compared to the plagiarized program, the difference in the length of the two programs may cause the similarity to be less than expected. Since the plagiarized program consists purely of the copied segment of the original program, the similarity may seem to be 100%, but SIM_{sum} is not. Conversely, if the plagiarized program has a lot of dummy statements that are not copied from the original program, then SIM_{sum} can also be lower than expected. The following new similarity SIM_{min} can be an alternative way to overcome these shortcomings in SIM_{sum}:

$$SIM_{min}(A, B) = \frac{SIM_{abs}(A, B)}{min\{SIM_{abs}(A, A),\ SIM_{abs}(B, B)\}}$$

In this paper, we normalize the similarity score using SIM_{min} to prevent bias due to the large difference in program sizes.

3.5 Adaptive cut-off threshold based on LOF

It is generally considered that the higher the similarity score a program pair has, the more the possibility of plagiarism exists. As a result, many methods for detecting plagiarism regard program pairs having similarity score more than a predefined threshold as plagiarized pairs. As mentioned above, because the similarity distribution of program groups is dependent on the programming environment or the restrictions on the problem, it is important that an adaptive threshold for plagiarism be defined.

We consider the code pairs that have relatively higher similarity score than others as outliers. Thus, the problem of detecting plagiarized code pairs can be replaced by the outlier detection problem. The outlier detection algorithm aims to find a small number of entities in a data set that appear to deviate markedly from other members of the set. This algorithm has been proposed for numerous applications, including credit card fraud detection, voting irregularity analysis, data cleansing, network intrusion, severe weather prediction, geographic information systems, athletic performance analysis, and other data-mining tasks [23].

Outlier detection methods can be divided into classes such as parametric (statistical) methods, nonparametric methods, and clustering based methods [24]. Statistical parametric methods either assume a known underlying distribution of the observations, or they are based on statistical estimates of unknown distribution parameters [25]. Within the class of non-parametric outlier detection methods, one can set apart the data-mining methods (also called distance-based methods). These methods are usually based on the local distance measures and are capable of handling large databases [11][26]. Clustering based methods consider the cluster of small sizes as clustered outliers. There are many clustering techniques—such as K-means clustering, K-nearest neighbor clustering, and support vector machine (SVM)—and they are used in a variety of applications [27][28][29].

From among these algorithms, we use the local outlier factor (LOF) algorithm to define a criterion for plagiarism [11]. The LOF algorithm is a density-based outlier detection algorithm that utilizes the concept of a local outlier that captures the degree to which an object is an outlier based on the density of its local neighborhood. In this method, each entity in the data set is assigned an LOF value that represents the likelihood of that object being an outlier. High LOF values are used to identify data objects that are potential outliers, whereas low LOF values indicate normal data objects [30]. In addition, the LOF algorithm does not need any assumption on the similarity distribution of the data set.

In our method, the similarity values between program source codes are the data objects. To get the LOF value of a data object, we first calculate the k−distance(A), which is defined as the distance of the object A to the k nearest neighbor. We then calculate reachability−distance of A from B, which is the true distance between two objects. The reachability-distance is defined as follows:

$$\text{reachability-distance}_k(A, B) = \max\{k\text{-distance}(B), d(A, B)\}$$

Here, $d(A, B)$ represents the Euclidean distance between A and B. Using this distance measure, we can calculate the local reachability density of an object A; that is, the quotient of the average reachability−distance of the object A from its neighbors.

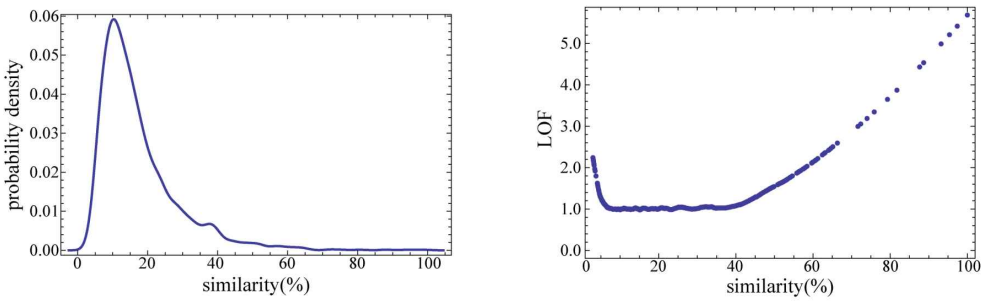
$$\text{lrd}(A) = \frac{|N_k(A)|}{\sum_{B \in N_k(A)} \text{reachability-distance}_k(A, B)}$$

Here, $N_k(A)$ means the sets of k nearest neighbors. Thus, the LOF of object A can be calculated as follows:

$$\text{LOF}_k(A) = \frac{\sum_{B \in N_k(A)} \text{lrd}(B)}{|N_k(A)|} \Big/ \text{lrd}(A)$$

An LOF value close to one indicates that the corresponding object is comparable to its neighbors (and thus not an outlier). An object whose LOF is less than one is in a denser region (which would make it an inlier), while the objects whose LOF values are significantly larger than one indicate outliers.

Fig. 4 shows the similarity distribution of the program group and the relation between the similarity score and the LOF value of each code pair belonging to that group. The distribution of the similarity score of a program group seems to follow the Log-Normal distribution in which most code pairs have similarity score in the 5–40% range, as shown in **Fig. 4(a)**. The portion of the similarity score where most code pairs are concentrated may vary depending on the program groups.



(a) The PDF graph of similarity scores (b) The LOF distribution

Fig. 4. The similarity distribution of a program group and the relation between the similarity scores and the LOF values of code pairs.

From **Fig. 4(b)**, it can be seen that the LOF values of similarity from about 5−40% have approximately 1.0 and high similarity scores have high LOF values. If we define the similarity threshold of plagiarism using LOF values, the detection can be resilient to changes in the similarity distribution. This is the basic idea underlying the adaptive threshold.

4. Implementation

In this section, we describe the implementation details of our proposed adaptive plagiarism detection system. The system was developed using Visual Studio 2005 with Extreme Toolkit ver 9.3, and is currently able to detect plagiarism in C, C++, and Java source codes. In addition, it can be extended to support other languages once the parser for those languages are available.

The system comprises three modules: (1) program linearizer, (2) similarity analyzer, and (3) viewer modules. The program linearizer was implemented by modifying the parsing module of OpenC++, a metacompiler for C++ [31]. The similarity analyzer computes the similarity score for every pair of programs using adaptive local alignment and the LOF algorithm. Finally, the viewer module for the system was implemented using the user interface library of Extreme toolkit. **Fig. 5** depicts screenshots of the system.

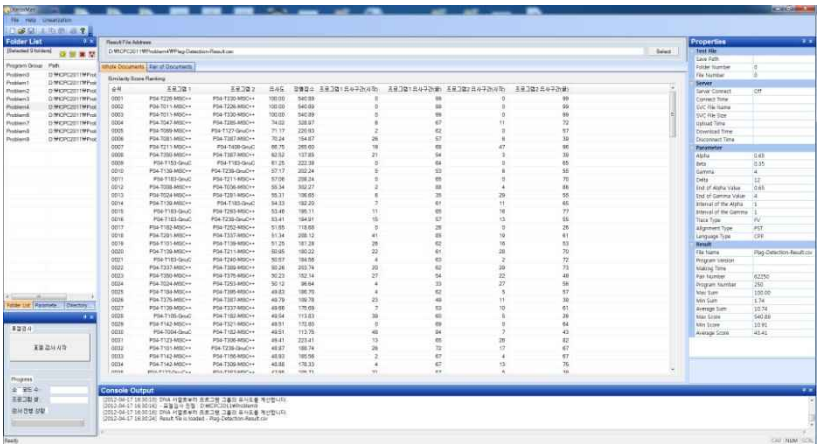


Fig. 5. The similarity table summarizing the similarity scores for the pairs in a program group

The set of programs to be analyzed is normally stored in a folder and given as input to the system. The similarity scores of all program pairs are calculated and stored as a score table. The screenshots in **Fig. 5** show the similarity table analyzed for a given set of programs. The right pane of the figure shows the set of control parameters, where the actual values can be modified.

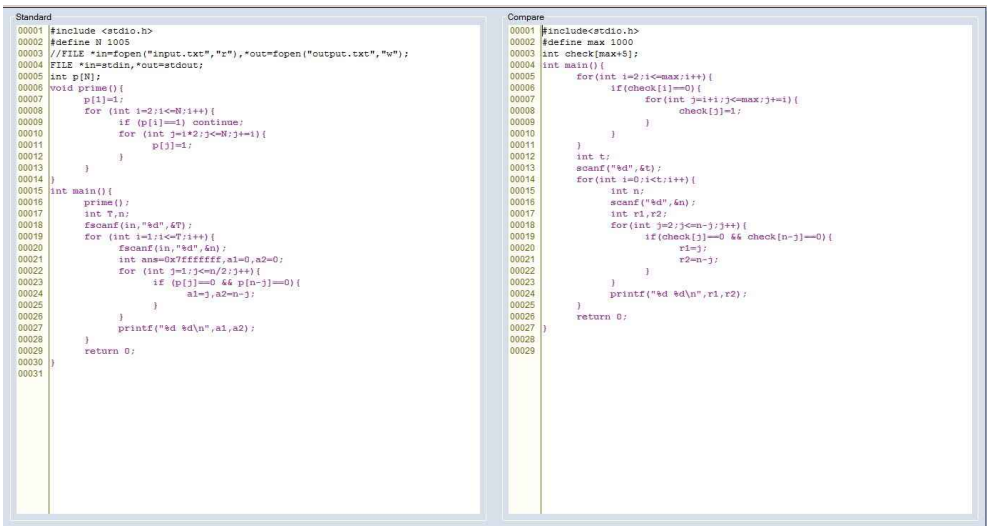


Fig. 6. The source code view that highlights the region of similar blocks respectively from the pair of source codes.

Fig. 6 depicts the source codes comparison window, which is shown when the user clicks on a row in the similarity table. The user can inspect the pair of source codes side by side in this window, where the regions of similarity in the code pairs are highlighted. Although static tracing is used to compare the programs, the comparison window shows the programs in the original order rather than the traced order since the original order is more natural to human inspectors than the traced order.

5. Experiment

We tested our proposed adaptive local alignment for detecting plagiarized source codes with 14 sets of test programs. We also compared the result obtained from local alignment using the static similarity scoring matrix to that obtained using our proposed adaptive scoring matrix. (The static scoring matrix (+1 for match, −1 for mismatch, and −2 for gap) is applied in most of the plagiarism detection systems previously mentioned.) We also compared the performance of our proposed plagiarism detection system to that of JPlag in terms of sensitivity and specificity analysis. JPlag is a well-known system and is at present one of the most reliable systems for finding plagiarized source codes [12].

5.1 Experimental Data

In order to evaluate the performance of our proposed system, we collected a set of test programs from a programming contest—specifically, the ACM International Collegiate Programming Contest (ICPC). The test programs used were all those submitted in the East-Asia preliminary and final rounds of the ICPC. All of the submitted programs were

written in the C/C++ language. **Table 3** summarizes the statistics of the program groups in the experiment.

Column N indicates the number of submitted programs in each program group. The number of program pairs was N(N–1)/2, shown in the Pairs column. Most of the submitted programs were under 100 lines of code (LOC). In the program group ICPC11-4, the maximum length of the programs was 1,015 lines, but most source codes other than the maximum were found to be useless, i.e., not a solution to the problem. The last two columns in the table denote the average (**μ**) and the standard deviation (**σ**) of LOC, respectively.

5.2 Comparison of Plagiarism-Suspected Program Codes

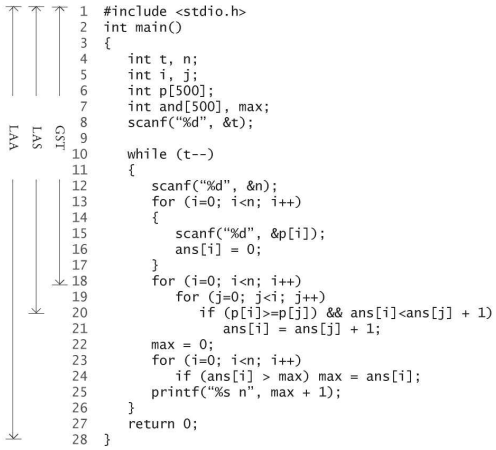
To find the plagiarism-suspected source codes, we first calculated the similarity of all the program pairs in each groups. The result of our first experiment enabled us to find a few plagiarism-suspected programs in the ICPC05 and ICPC11 groups. In the case of the ICPC11 group, the source codes of three program pairs were exactly the same, and it was eventually confirmed by the contestants who had cheated.

Table 3. The experimental program sets: ICPC-2005 (ICPC05), ICPC-2006 (ICPC06), and ICPC-2011(ICPC11)

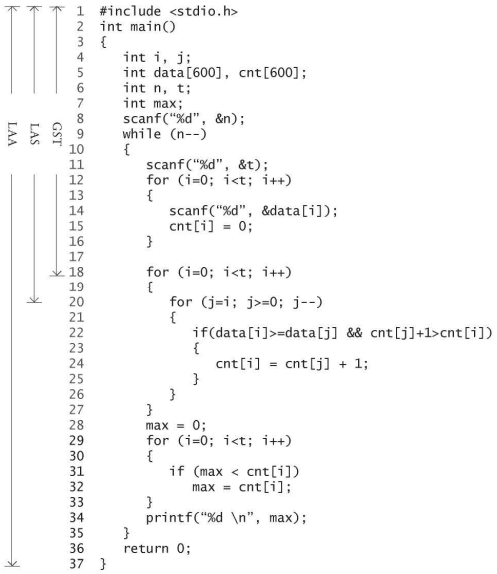
No.	Program Group	N	Pairs	LOC			
				Max.	Min.	μ	σ
1	ICPC05-1	153	11,628	144	21	44.46	15.85
2	ICPC05-2	109	5,886	139	24	65.44	22.86
3	ICPC06-1	179	15,931	216	19	47.60	20.86
4	ICPC06-2	174	15,051	180	19	43.78	20.46
5	ICPC06-3	157	12,246	234	18	54.29	23.84
8	ICPC06-4	66	2,145	110	25	59.98	35.66
9	ICPC06-5	58	1,653	225	29	66.12	28.91
10	ICPC06-6	60	1,770	227	38	78.43	65.89
11	ICPC11-1	161	12,880	233	34	77.16	30.34
12	ICPC11-3	50	1,225	367	42	92.02	47.67
13	ICPC11-4	263	34,453	1,015	18	56.60	67.37
14	ICPC11-8	42	861	159	35	65.14	25.67
15	ICPC11-9	38	703	161	36	69.45	22.72

We then compared the performance of three different methods: Local Alignment with Adaptive matrix (LAA), Local Alignment with Static matrix (LAS), and greedy string tiling (GST)—used in the JPlag system to determine the correctness of locating similar blocks in two independent programs. **Fig. 7** depicts a pair of programs that were suspected to have been plagiarized in program group ICPC05-1.

From **Fig. 7**, there is a certainty of substantial plagiarism between the two programs. First, it can be seen that variables *t* and *n* are swapped. Second, when compared with P_a, the block statements of `for` and `if` have been inserted into P_b. Finally, the looping structures in the two programs are quite similar. However, we note that the conditional expression of the `for` statements in lines 18–19 in P_a and in lines 18–20 in P_b have been modified. The two programs are similar enough to be suspected of being a pair of plagiarized codes.



(a) P_a which is suspected as a plagiarized source code



(b) P_b which is suspected as a plagiarized version of P_a.

Fig. 7. Plagiarism-suspected program pair obtained in the program group ICPC05-1.

Three methods, LAA, LAS, and GST, located the region of similar blocks respectively in the pairs of source codes in the test sets. In the following, let $Line_P[a{:}b]$ denote the consecutive lines (statements) between the a-th line and the b-th line inclusive of program P. In our test, GST suspected that $Line_{P_a}$[1:18] was plagiarized from $Line_{P_b}$[1:18]. Furthermore, local alignment using the static scoring matrix reported that $Line_{P_a}$[1:19] is suspected to have been plagiarized from $Line_{P_b}$[1:20].

LAS and GST did not effectively detect plagiarized blocks with inserted statements and the rewritten conditional expression of the `for` statement. It is worth noting that our LAA successfully reported that $Line_{P_a}$[1:28] is quite similar to $Line_{P_b}$[1:37]. In the experiment, the penalty score of our adaptive local alignment for the `for` loop (in $Line_{P_a}$[18:19] and in $Line_{P_b}$[18:20]) was determined to be a value that was less than the penalty score (−1), which is the static penalty value of a static local alignment. Though this case is typical, it implies that our algorithm is resilient to typical methods of attack such as variable name changing, operator changing, inserting/deleting short dummy statements, and rewriting logical expressions.

5.3 Comparison with JPlag

We also compared the general effectiveness of our detection technique to JPlag. Both our system and that of JPlag construct a sequence of tokens for the intermediate representation of source code, but a comparison of the algorithms revealed that the two systems are different. JPlag uses the greedy-string-tiling (GST) algorithm while our system uses the adaptive local alignment (LAA) algorithm. The main difference between the two methods is that local alignment is more suitable for locating the specific region of plagiarism, while GST is suitable for measuring the overall similarity of the two programs. Since plagiarism among codes happens in a few critical classes or functions, local alignment is more effective and efficient in detecting clever plagiarisms.

In order to compare the performance of different plagiarism detection systems, it is important that quantitative measures for the effectiveness of the systems be developed. To measure the effectiveness and accuracy of plagiarism detection systems, we computed two measures: *sensitivity* and *specificity*. These measures are commonly used in most experiment-based studies, such as information retrieval. In order to evaluate sensitivity and specificity, we counted the number of cases of true positives (*TP*), false positives (*FP*), true negatives (*TN*), and false negatives (*FN*). **Table 4** summarizes the possible testing outputs.

Table 4. Four different testing results of plagiarism detection *TP* (true positive), *FP* (false positive), *FN* (false negative), and *TN* (true negative)

		Actual condition	
		plagiarized	**Non-plagiarized**
Detection result	**Plagiarized**	*TP*	*FP*
	Non-plagiarized	*FN*	*TN*

TP signifies that a plagiarism pair (i.e., one program was plagiarized from the other) was detected as a plagiarism pair in our system. Based on these four cases, the sensitivity and specificity are defined as follows:

$$sensitivity(\theta) = \frac{|TP|}{|TP|+|FN|}, \quad specificity(\theta) = \frac{|TN|}{|TN|+|FP|}$$

Here, θ is a user-defined cut-off threshold similarity for plagiarism. If a detection system S shows high sensitivity, then it means S is not likely to miss a real plagiarized code pair. If a detection system S shows high specificity, then it means S is not likely to suspect any innocent code as a cheated one. The most desirable case is that both sensitivity and specificity of the system are high, but generally speaking, the specificity usually contradicts the sensitivity.

In order to compute sensitivity and specificity, we prepared some artificially plagiarized codes, (since it was hard to collect more than 10 "real" plagiarized codes in practice). We asked 10 students to plagiarize the given source codes that were selected from programs submitted in the ICPC06, resulting in 10 different plagiarized programs from an ICPC06 source code. This experiment limited the working time for plagiarizing the source codes to at most 2 h, as was usually done by dishonest students. **Table 5** shows the artificially generated test set of 40 plagiarized source codes.

In the experiment we compared our system to JPlag in terms of sensitivity and specificity according to the cut-off threshold θ. If $Sim(A, B)$ is greater than a threshold θ, then we report that A is certainly plagiarized from B.

Table 5. Overview of experiment programs including artificially plagiarized programs.

Program Group	**submitted files**	**plagiarized files**	**pairs**	**LOC**			
				Max.	**Min.**	**μ.**	**σ**
ICPC06-3	157	10	13,861	234	18	54.29	27.10
ICPC06-4	66	10	2,850	110	25	57.89	20.56
ICPC06-5	58	10	2,278	214	29	69.12	28.94
ICPC06-6	60	10	2,415	227	38	76.21	27.70

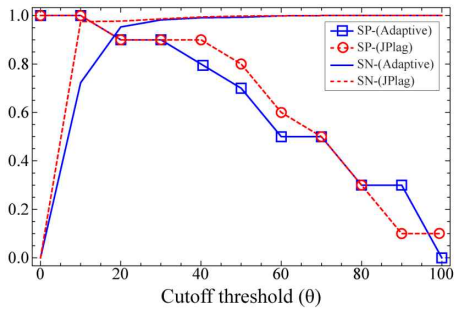
Fig. 8 shows the sensitivity and specificity for groups ICPC06-3, ICPC06-4, ICPC06-5,

and ICPC06-6. In each graph, the solid line and the dashed line indicate the result for our proposed adaptive system and that of JPlag, respectively. The lines that have a mark indicate the sensitivity graphs, while those without any mark indicates specificity graphs. The sensitivity graph in **Fig. 8** shows that our system was more sensitive than JPlag, especially for the range where the cut-off threshold θ was above 70%. Since the plagiarized code pairs hardly exist in the low similarity score range, the sensitivity in this portion was not very important in the detection of code plagiarism. Therefore our system is considered more practical than JPlag with respect to plagiarism detection.

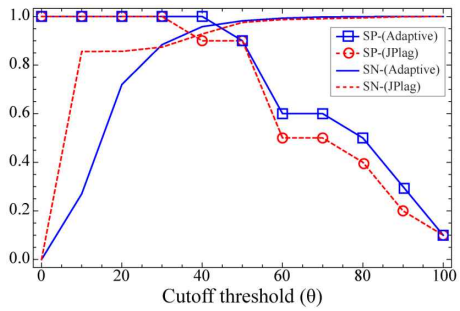
Was this gain in sensitivity obtained by sacrificing specificity? The specificity graph in **Fig. 8** shows that this was not the case. For a normal cut-off threshold θ of more than 70%, the specificity of JPlag was no better than our specificity. In fact, our system was slightly more specific than JPlag when the cutoff threshold θ was above 30% for three test sets (**Fig. 8(b), (c),** and **(d)**). We should point out that the specificity curves of our system are smoother than those of JPlag, which means that the specificity of our system is more stable than JPlag when the cutoff threshold is varying.

Since the specificity curve monotonically increases and the sensitivity curve monotonically decreases, a simple way to compare the performance of detection systems is to compare the *y*-axis value (throughput) of the intersection point of the specificity and sensitivity curves. The *x*-axis location of the intersection point of the specificity and sensitivity curves can be considered the *tradeoff threshold* balancing the minimization of false positives and false negatives. From the result, we can know that our adaptive method is more efficient and reliable than JPlag in terms of sensitivity and specificity. **Table 6** shows the exact *x* and *y* axis value of the tradeoff threshold points of **Fig. 8**.

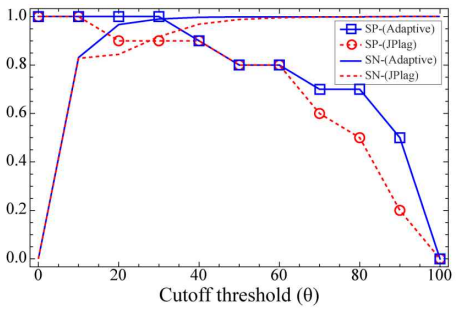
In all program groups but ICPC06-3, it can be seen that our system is more effective than JPlag in the sense that both the sensitivity and the specificity (throughput column in **Table 6**) of our system is higher than those of JPlag by 3.1% on average. The programming problem for ICPC06-3 has a relatively low level of difficulty compared with other problems. As mentioned above, the adaptive local alignment used in our system is more suitable for locating the specific region of plagiarism, while the algorithm used by JPlag (i.e., GST) is more suitable for measuring the overall similarity of the two programs. This implies that JPlag can be effective enough for simple programs such as those in program group ICPC06-3.



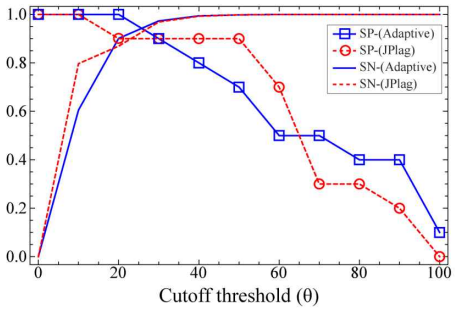
(a) ICPC06-3



(b) ICPC06-4



(c) ICPC06-5



(d) ICPC06-6

Fig. 8. The cut-off thresholds determined by the sensitivity and the specificity. Our adaptive system is superior to JPlag in that both the sensitivity and the specificity where the cut-off threshold θ is above 70%.

Table 6. Comparison of the tradeoff threshold points of our system and JPlag

	Our system		JPlag	
	tradeoff threshold	**throughput**	**tradeoff threshold**	**throughput**
ICPC06-3	18.4	91.6%	12.5	97.5%
ICPC06-4	43.3	96.6%	38.2	91.8%
ICPC06-5	30.9	99.1%	27.7	90.0%
ICPC06-6	25.8	94.2%	23.1	90.0%
average	29.6	95.4%	25.4	92.3%

5.4 Adaptively Deciding On The Cut-Off Threshold Based On LOF

From the above experiments, we found that the detection capabilities of the system varies depending on the cut-off threshold of the similarity score. We compared the detection performance for two cases: (1) using a fixed threshold and (2) using an adaptive threshold taking the LOF value into consideration. For the fixed threshold, we set the cut-off threshold at 70%, i.e., if the similarity score of a program pair was more than 70%, the pair would be detected as a plagiarism pair in our system. Further, for the adaptive threshold, we set the adaptive cut-off threshold based on the LOF value of the program pairs. The LOF value of a program pair depends on the distribution of similarity scores of its program group.

Table 7. Plagiarism detection performance of the cases using a fixed and an adaptive similarity threshold base on LOF value.

	adaptive threshold$_{LOF=3}$	**Sensitivity**			**Specificity**		
		fixed	**adaptive**	**gain**	**fixed**	**adaptive**	**gain**
ICPC06-3	57.31%	50.0%	70.0%	20.0%	100.0%	99.8%	-0.2%
ICPC06-4	72.36%	60.0%	60.0%	0.0%	99.8%	99.8%	0.0%
ICPC06-5	36.93%	70.0%	90.0%	20.0%	99.9%	99.6%	-0.3%
ICPC06-6	47.29%	50.0%	70.0%	20.0%	100.0%	99.7%	-0.3%
Average		57.5%	72.5%	15.0%	99.9%	99.7%	-0.2%

It means that although two program pairs that belong to different program groups have the same similarity score, their LOF values can be different.

For the test, if the LOF value of a program pair was more than three, the pair was considered a plagiarism pair. Under these conditions, we tested for the detection of plagiarism pairs with our data set ICPC06-3, ICPC06-4, ICPC06-5, and ICPC06-6. The detection results are summarized in **Table 7**.

For all test groups, the detection performance using an adaptive threshold was superior in sensitivity compared with that of using a fixed threshold with only negligible loss of the specificity (−0.2%). The second column (adaptive threshold$_{LOF=3}$) of **Table 7** indicates the similarity score when the LOF value of a code pair was set to 3. This value varies with each program group because program groups have different similarity distributions. Surprisingly, there is wide variance in this adaptive threshold depending on the program group. This means that we can choose a suitable threshold adaptively depending on the program group using the LOF value.

6. Conclusion and Future Work

In this paper, we proposed a new method of automatically detecting plagiarized programs in a given large program set. The contributions of our method are as follows. First, we proposed an adaptive local alignment that changes the similarity matrix depending on the frequencies of the keywords of the input program set. The basic idea underlying the adaptive local alignment is to attribute more weight to the important keywords of the program source code.

Second, we developed a method to adaptively determine the cut-off threshold based on the LOF of the similarity distribution. As the similarity distribution of a program set is dependent on the set of programs, it is important that the proper similarity threshold be adaptively determined according to the program set. The adaptive threshold can help a human inspector to cope with pseudo-plagiarism, in which most programs are similar but none is the result of a real plagiarism. Pseudo-plagiarism may occur when the problem is too simple or has a strong functional requirement. The adaptive thresholds for these cases will be set to a low value, producing lots of false positives. The human reader can eventually determine these cases as pseudo-plagiarism.

Third, we conducted experiments to evaluate the performance of our system. We prepared a set of testing programs from an actual programming contest, the ACM International Collegiate Programming Contest (ICPC). Experimental results indicate that our proposed adaptive local alignment is especially effective in detecting plagiarism in source codes; in particular, it is superior to JPlag in the portion where the similarity score is greater than 70%. Further, the adaptive threshold outperforms the fixed threshold. Specifically, it lifts the sensitivity up to more than 72.5% on average with negligible loss of specificity (0.2% on average).

We plan to extend our algorithm to further application of code analyses in the future. Currently, we are trying to reconstruct the phylogeny tree for a program set, which is based on the idea that a program can be considered as a form of artificial life. The code phylogeny tree will hopefully show the “evolution” of the source, which will help to identify the plagiarism groups more precisely. Also, the statistical method for determining the pseudo plagiarism may be effectively combined with our system; this will also be done in future work [32].

References

[1] J. Carter, "Collaboration or plagiarism: What happens when students work together," in *Proc. of ITICSE'99*, pp.52-55, Jun.1999. Article (CrossRef Link)

[2] A. Knight, K. Almeroth, and B. Bimber, "An automated system for plagiarism detection using the internet," in *Proc. of* ED-MEDIA 2004, pp.3619-3625, Jun.2004. Article (CrossRef Link)

[3] D. Gitchell and N. Tran, "Sim: a utility for detecting similarity in computer programs," in *Proc. of SIGCSE'99*, pp.266-270, Mar.1999. Article (CrossRef Link)

[4] L. Prechelt, G. Malpohl, and M. Philippsen, "Finding plagiarisms among a set of programs with JPlag," *Journal of Universal Computer Science*, vol.8, no.11, pp.1016-1038, Nov.2002. Article (CrossRef Link)

[5] G. Whale, "Identification of program similarity in large populations," *The Computer Journal-Special issue on Procedural programming*, vol.33, no.2, pp.140-146, Apr.1990. Article (CrossRef Link)

[6] M. J. Wise, "Detection of similarities in student programs: Yap'ing may be preferable to plague'ing," *ACM SIGSCE Bulletin*, vol.24, no.1, pp.268-271, Mar.1992. Article (CrossRef Link)

[7] S. Schleimer, D.S. Wilkerson, and A. Aiken, "Winnowing: local algorithms for document fingerprinting," in *Proc. of the ACM SIGMOD 2003*, pp.76-85, Jun.2003. Article (CrossRef Link)

[8] JS. Lim, JH. Ji, HG. Cho, and G. Woo, "Plagiarism detection among source codes using adaptive local alignment of keywords," in *Proc. of ICUIMC'11*, pp.24-33, 2 Feb.2011. Article (CrossRef Link)

[9] T. F. Smith and M. S. Waterman, "Identification of common molecular subsequences," *Journal of Molecular Biology*, vol.147, pp.195-197, 1981. Article (CrossRef Link)

[10] JH. Ji, SH. Park, G. Woo, and HG. Cho, "Source code similarity detection using adaptive local alignment of keywords," in *Proc. of PDCAT 2007*, pp.179-180, Dec.2007. Article (CrossRef Link)

[11] M. M. Breunig, H. P. Kriegel, R. T. Ng and J. Sander, "LOF: Identifying Density-Based Local Outliers," in *Proc. of the ACM SIGMOD 2000*, pp.93-104, May.2000. Article (CrossRef Link)

[12] A. Parker and J. O. Hamblen, "Computer algorithms for plagiarism detection," *IEEE Trans. on Education*, vol.32, no.2, pp.94-99, May.1989. Article (CrossRef Link)

[13] S. Brin, J. Davis, and H. Garcia-Molina, "Copy detection mechanisms for digital documents," in *Proc. of the ACM SIGMOD 1995*, pp.398-409, May.1995. Article (CrossRef Link)

[14] J. H. Johnson, "Identifying redundancy in source coding using fingerprints," in *CASCON'93*, pp.171-183, 1993. Article (CrossRef Link)

[15] S. D. Stephens, "Using metrics to detect plagiarism (student paper)," *The journal of computing Sciences in Colleges*, vol.16, no.3, pp.191-196, Mar.2001. Article (CrossRef Link)

[16] M. H. Halstead, *Elements of Software Science (Operating and programming systems series),* Elsevier Science Inc., New York, 1977. Article (CrossRef Link)

[17] T. Schmidt and J. Stoye, "Quadratic time algorithms for finding common intervals in two and more sequences," in *Proc. of CPM 2004*, pp. 347-385, Jul.2004. Article (CrossRef Link)

[18] JW. Son, SB. Park, and SY. Park, "Program plagiarism detection using parse tree kernels," in *Proc. of PRICAI 2006*, pp.1000-1004, Aug.2006. Article (CrossRef Link)

[19] I. D. Baxter, A. Yahin, L. M. Moura, M. Sant'Anna, and L. Bier, "Clone detection using abstract syntax trees," in *Proc. of ICSM'98*, pp.368-377, Mar.1998. Article (CrossRef Link)

[20] K. L. Verco and M. J. Wise, "Software for detecting suspected plagiarism: Comparing structure and attribute-counting systems," in *Proc. of ACSE'96*, pp.81-88, Jul.1996. Article (CrossRef Link)

[21] M. J. Wise, "Neweyes: A system for comparing biological sequences using the running Karp-Rabin Greedy String-Tiling algorithm," in *Proc. of ISMB 1995*, pp.393-401, Aug.1995. Article (CrossRef Link)

[22] X. Chen, B. Francia, M. Li, B. McKinnon, and A. Seker, "Shared information and program plagiarism detection," *IEEE Trans. On Information Theory*, vol.50, no.7, pp.1545-1551, 2004. Article (CrossRef Link)

[23] J. Zhang and M. Zulkernine, "Anomaly based network intrusion detection with unsupervised outlier detection," in *Proc. of ICC'06*, vol.5, pp.2388-2393, Jun.2006. Article (CrossRef Link)

[24] O. Maimon and L. Rokach, "Data mining and knowledge discovery handbook", Springer-Verlag New York Inc, 2005. Article (CrossRef Link)
[25] J. Laurikkala, M. Juhola, and E. Kentala, "Informal identification of outliers in medical data," in *Proc. of IDAMAP 2000*, Aug.2000. Article (CrossRef Link)
[26] E. M. Knorr and R. T. Ng, "Algorithms for mining distance-based outliers in large datasets," in *Proc. of VLDB'98*, pp.392-403, Aug.1998. Article (CrossRef Link)
[27] Y. Zeng and T. M. Chen, "Classification of traffic flows into QoS class by unsupervised learning and KNN clustering," *KSII Trans. on Internet and Information Systems*, vol.3, no.2, pp.134-146, 2009. Article (CrossRef Link)
[28] SH. Song, CH. Lee, JH. Park, KJ. Koo, JK. Kim, and JS. Park, "enhancing location estimation and reducing computation using adaptive zone based K-NNSS algorithm," *KSII Trans. on Internet and Information Systems*, vol.3, no.1, pp.119-133, 2009. Article (CrossRef Link)
[29] JH. Yu, HS. Lee, YH. Im, MS. Kim, and DH. Park, "Real-time classification of internet application traffic using a hierarchical multi-class SVM," *KSII Trans. on Internet and Information Systems*, vol.4, no.5, pp.859-876, 2010. Article (CrossRef Link)
[30] M. Alshawabkeh, B. Jang, and D. Kaeli, "Accelerating the local outlier factor algorithm on a GPU for intrusion detection systems," in *Proc. of GPGPU-3*, pp.104-110, Mar.2010. Article(CrossRef Link)
[31] OpenC++ Homepage, http://opencxx.sourceforge.net/, lastly visited on Apr. 2012.
[32] JH. Ji, G. Woo, SH. Park, and HG. Cho, "An intelligent system for detecting source code plagiarism using a probabilistic graph model," in *Proc. of MLDM 2007*, pp.55-69, Jul.2007. Article(CrossRef Link)

Yun-Jung Lee received a B.S. degree from Pukyung National University, Republic of Korea, in 1995, a M.S. degree from Pukyung National University, Republic of Korea in 1999, and a Ph.D. degree from Pukyung National University, Republic of Korea, in 2008. She is currently with the Center for U-Port IT Research and Education in Pusan National University, Republic of Korea. Her research interests include weblog visualization, social network, program coding style visualization, and facial animation.

Jin-Su Lim received BS and MS degrees in computer science and engineering from the Pusan National University, Republic of Korea, in 2010 and in 2012, respectively. He is currently with the HA Control R&D lab in LG Electronics, Republic of Korea. His research interests are software plagiarism detection and software reuse.

Jeong-Hoon Ji received his B.S. degree in 2003 from Kyungsung University, Republic of Korea, a M.S. degree in 2005 from Kyungsung University, Republic of Korea, and a Ph.D. degree from Pusan National University, Republic of Korea in 2010. He is currently with the Korea Intellectual Property Office, Republic of Korea. His research interests are programming language and software plagiarism detection.

Hwan-Gue Cho received a B.S. degree from Seoul National University, Korea, and M.S. and Ph.D. degrees from Korea Advanced Institute of Science and Technology, Korea. Since 1990 he has been a Professor in Pusan National University, Korea. His research interests are graphics (visualization) and bioinformatics (sequence alignment and bionetwork analysis).

Gyun Woo is an associate professor at the School of Computer Science and Engineering, Pusan National University in Busan, Republic of Korea. He received B.S., M.S., and Ph.D. degrees in computer science from the Korea Advanced Institute of Science and Technology, in 1991, 1993, and 2000, respectively. He previously worked at Dong-A University as an assistant professor, then joined Pusan National University in September 2004. He is the co-author of *Playing with C* and *Playing with Java* (Kyobo Book Company). His areas of interest include implementation of programming languages, program analyses, functional languages, functional programming, software testing, swarm intelligence, robot control, and visualization of software.