Engineering Applications of Artificial Intelligence 26 (2013) 1911–1918

Contents lists available at SciVerse ScienceDirect

Engineering Applications of Artificial Intelligence

journal homepage: www.elsevier.com/locate/engappai

An application for plagiarized source code detection based on a parse tree kernel

Jeong-Woo Son, Tae-Gil Noh, Hyun-Je Song, Seong-Bae Park *

School of Computer Science and Engineering, Kyungpook National University, 80, Daehakro, Bukgu, Daegu, Republic of Korea

ARTICLE INFO

Article history:
Received 17 November 2012
Received in revised form
12 June 2013
Accepted 14 June 2013
Available online 4 July 2013

Keywords:
Plagiarism detection
Software plagiarism
Parse tree kernel
Tree kernel

ABSTRACT

Program plagiarism detection is a task of detecting plagiarized code pairs among a set of source codes. In this paper, we propose a code plagiarism detection system that uses a parse tree kernel. Our parse tree kernel calculates a similarity value between two source codes in terms of their parse tree similarity. Since parse trees contain the essential syntactic structure of source codes, the system effectively handles structural information. The contributions of this paper are two-fold. First, we propose a parse tree kernel that is optimized for program source code. The evaluation shows that our system based on this kernel outperforms well-known baseline systems. Second, we collected a large number of real-world Java source codes from a university programming class. This test set was manually analyzed and tagged by two independent human annotators to mark plagiarized codes. It can be used to evaluate the performance of various detection systems in real-world environments. The experiments with the test set show that the performance of our plagiarism detection system reaches to 93% level of human annotators.

1. Introduction

Plagiarism, defined as “using someone else's work as their own without reference to original sources” (Maurer et al., 2006), has received much attention in diverse fields. The attention to plagiarism is constantly increasing due to the growth of information technology. Internet, digital documents, and file sharing systems have made it easy to access more and more information including program source codes. Plagiarism is considered as one of the most severe problems in education, since students can submit their course assignments without any understanding of the subject by plagiarizing someone else's work. According to Evans (2006), over 30% of students have experience of copying some text or even an entire paper without reference. Therefore, there is much demand for methods to deal with students' plagiarism.

This paper focuses on detecting structured text plagiarism, especially in program source code. Parker and Hamblen (1989) defined program source code plagiarism as a program that has been produced from another program with a small number of routine changes. One key part of plagiarism detection systems is the similarity measure. According to White and Joy (2004), plagiarism detection tools can be regarded as programs that compare documents in order to identify similarities and discover submissions that might be plagiarized.

The similarity measure for plagiarism detection systems should reflect the characteristics of the program source code. Compared with normal text, source code has some unique characteristics:

- Source code is composed of a large set of rarely occurring user-defined words and a small set of frequently occurring reserved words like *for*, *while*, and *if*.
- Even though a pair of source code can be greatly different in terms of string similarity, they can achieve the same functionality.
- Source code has a structure that is determined by the reserved words.

The first and second characteristics imply that character-level comparison that is often adopted in plagiarism detection for normal text is not suitable for program plagiarism detection. A clue for a way to achieve good program plagiarism detection can be found in the third characteristic. When plagiarizing a source code, it is much harder to change the structure of the source code. Converting user-defined vocabulary is easy, and can be done without a proper understanding of the source code. However, redefining the structure of a program is very tricky and often as hard as writing the module from the scratch. This means that the program structure is an important feature for plagiarism detection. Some previous program plagiarism detection systems attempted

* Corresponding author. Tel.: +82 53 950 7574.
E-mail address: seongbae@knu.ac.kr (H.-J. Song).

0952-1976/$ - see front matter © 2013 Elsevier Ltd. All rights reserved.
http://dx.doi.org/10.1016/j.engappai.2013.06.007

to design similarity measures that reflected structural information to some extent. However, since most of them defined their structural features on the lexical level, their ability to compare entire structures is fairly limited. Structural information of a source code can be presented by the parse tree of the source code. Thus, a software plagiarism detection system should be able to use parse trees to incorporate structural information.

Defining a metric for parse trees is not a trivial task, since it is generally harder to define a metric for structured data. One of the prominent methods for comparing structured data is the kernel functions. Haussler (1999) first defined a mathematically profound way to define a kernel function for structured data, the so-called *R-convolution kernel*. An R-convolution kernel defines a kernel space with infinitely high dimensions where each dimension corresponds to each possible substructure. By comparing two given structures in the kernel space (without explicitly generating infinite dimensions – the kernel trick), a convolution kernel can make structural comparison without manually selected structural features.

In this paper, an effective plagiarism detection system for the Java language is proposed. The proposed system adopts a parse tree kernel (Collins and Duffy, 2001), which is an R-convolution kernel for tree structures. The parse tree kernel computes the similarity value between a pair of parse trees. Thus, the structural information of the source code is fully reflected in the proposed system. Parse tree kernels have been successfully used in natural language processing (NLP). However, the parse tree kernel used in NLP does not perform well for program source code due to two issues. The first issue is the asymmetric influence of node changes. In previous tree kernels, changes near a root node have larger influence than changes near leaf nodes. When the tree depth is small, this effect is not serious. However, the parse trees of program sources are much larger than that of natural language sentences, and this unwanted influence greatly affects tree comparison. The second issue is the sequence of subtrees. Previous tree kernels count sequence of subtrees. Unlike natural language sentences, the sequence of two substructures (like the order of two methods in a Java class) has little information in program source codes. We identified these two issues, and propose a new parse tree kernel for program source code.

The proposed plagiarism detection is performed in three steps: First, parse trees are generated from the source codes. Second, all the parse trees are compared via the proposed parse tree kernel, and similarity values between all pairs are obtained. Finally, the groups of source codes that are most likely to be plagiarized are selected according to the similarity values.

The performance of the proposed system is evaluated with two evaluation sets. In the first experiment, our system is evaluated by using a synthesized data set. In this data set, several well-known plagiarism methods are simulated (like replacing variable names, and inserting redundant code) to generate different types of plagiarized source codes. In the second experiment, a real-world Java source code collection from a college programming class is used as the data set. The experimental results show that the proposed system can successfully detect real-world program plagiarism on 93% level of human annotators.

This paper is organized as follows. Section 2 presents related work and previous plagiarism detection systems. Section 3 describes the proposed system. Section 4 details the two experiments and the paper is concluded in Section 5.

2. Related work

The characteristics of intellectual properties are important in detecting partial copying of the properties. For written texts, various anti-plagiarism systems have been introduced including COPS (Brin et al., 1995), SCAM (Shivakumar and García-Molina, 1995), and CHECK (Si et al., 1997). Oberreuter and Velásquez (2013) recently proposed a text mining technique to detect plagiarized documents. In their work, a document is divided into segments by using a sliding window of a certain length over the document. Then, it is determined whether a document is plagiarized or not by deviation of the writing styles of the segments. The deviation in the writing styles was defined as difference of term frequencies in the segments. Bravo-Marquez et al. (2011) suggested a Web Document Similarity Retrieval Tool (WDSRT) that receives a document as an input and retrieves Web documents similar to the input document. Since WDSRT uses a similarity between a pair of documents to find appropriate Web documents, the authors insisted that WDSRT can be applied to detecting plagiarized documents. WDSRT measures a similarity between documents with n-grams of words. Like two systems explained above, most anti-plagiarism systems for written texts compare documents in their lexical level like n-grams of words (Sojka et al., 2006; Nawab et al., 2013). Since lexical level information is enough to compare written texts, these anti-plagiarism systems have been successfully applied to plagiarism detection for written texts. However, they are not appropriate for comparing structured documents such as source codes, since comparison between source codes should consider the structural level information (Durić and Gašević, 2013).

Program plagiarism detection systems are divided into two categories. The first one is the so-called attribute-counting-metric type. This type represents a program as a vector of elements such as the number of operators, the number of operands, etc. Halstead (1977) first proposed a plagiarism detection system using a software science metric. Berghel and Sallach (1984) and Grier (1981) reported another systems by defining different attributes. The major problem of these systems is that they show relatively poor performance compared to the systems that consider program structures.

The second type of plagiarism detection systems does not only use attribute-counting metrics but also compares the structure of source code. These systems represent the source code as a string that contains certain structural information (Gitchell and Tran, 1998). Chen et al. (2004) suggested the SID system using Kolmogorov complexity. Prechelt et al. (2002) proposed the JPlag system based on a greedy string tiling algorithm. One of the state-of-the-art and well-known plagiarism detection systems, CCFinder proposed by Kamiya et al. (2002) falls on this type. In CCFinder, a source code is transformed into a set of normalized token sequences by transformation rules. The transformation rules are manually constructed for each language such as C++ or Java considering the structural characteristics of a language. Thus, CCFinder removes lexical variations in variable names, function names, and so on, using the structural information. Then, the normalized tokens are compared to reveal clone pairs in source codes. Even if they show good performance and use structural information to some degree, it cannot be said that structural information is fully reflected within their metric. The systems in the second type do not use parse trees, but use just string representations of parsed source codes for their structural information. Thus, there is still room to improve in reflecting structural information for program plagiarism detection.

To express structural information of a program source code efficiently, the systems should use the parse trees which contain the structural information of the source code. In addition, in order to apply machine learning algorithms or statistical methods to this task, the parse tree should be mapped into a feature space including the entire set of structural information. However, it is difficult to translate a tree into a vector. An effective way to do this

is a kernel method. Kernel methods can handle complex data in high dimensional spaces and can calculate the inner product of the representations. Collins and Duffy (2001) first proposed a parse tree kernel for parse trees in Natural Language Processing (NLP). A parse tree kernel is derived from the convolution kernel (Haussler, 1999) that was proposed to handle discrete structures like trees and graphs.

As far as we know, there is only one report on a program plagiarism detection system that utilizes parse trees with kernels (Son et al., 2006). This is our previous work, and it describes a preliminary version of the plagiarism detection system presented in this paper. Compared to our previous work, there are two major improvements in the current system. First, we clearly defined and described the parse tree kernel for program source code – the major contribution of this paper. Previous tree kernels, like those used in Collins and Duffy (2001) and Son et al. (2006), are not adequate to compare highly modularized structures like program source code. Our previous work did not reflect such characteristic of source codes in the parse tree kernel. Second, our previous work is evaluated only with a small set of synthesized data. While the synthesized data are useful to probe the effects of specific attacks, it cannot show how the proposed system would perform in real-world situation. In this paper, a large collection of real-world plagiarism data set is prepared and used to evaluate the effectiveness of the proposed system.

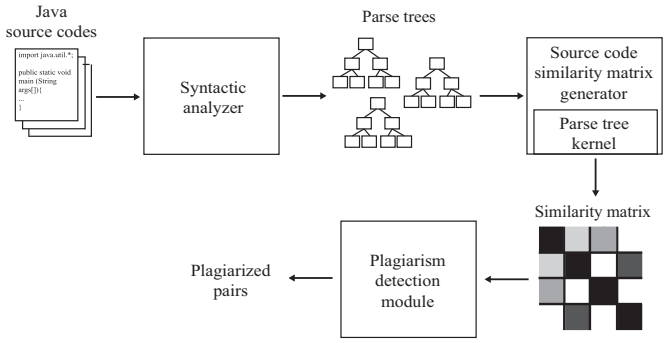

Fig. 1. Plagiarism detection process of the proposed system.

3. Source code plagiarism detection

The proposed system detects plagiarism by using the parse trees of a piece of source code. Fig. 1 illustrates the proposed system that operates in three steps. In the first step, the system extracts parse trees from the source codes with a syntactic analyzer. Then, the pair-wise similarities between all pairs of the source code are calculated using the parse tree kernel. This results in an all-pair similarity matrix. Finally, in the third step, plagiarized pairs are detected by selecting pairs of source code that are above a certain threshold in terms of similarity.

3.1. Extracting parse trees

A parse tree contains all the syntactic structural information of a program source code. A parser is needed to get a complete parse tree from the source code. We use ANTLR (Parr and Quong, 1995) as our parser. ANTLR is a parser generator tool that provides a framework for constructing a recognizer, compiler and translator from grammatical descriptions. It also has a tree parser to translate a source code into a parse tree and generate output that can easily be used in other applications. Fig. 2 shows an example of a simple parse tree generated by ANTLR.

3.2. Parse tree kernel

In this section, we give a brief overview of the parse tree kernel proposed by Collins and Duffy (2001). Then, the modifications needed for the kernel to be used on program source code are introduced in the next section,

A parse tree kernel is a kernel that is designed to compare data structures in a tree form including parse trees of natural language sentences. In the kernel, a parse tree is mapped onto a space

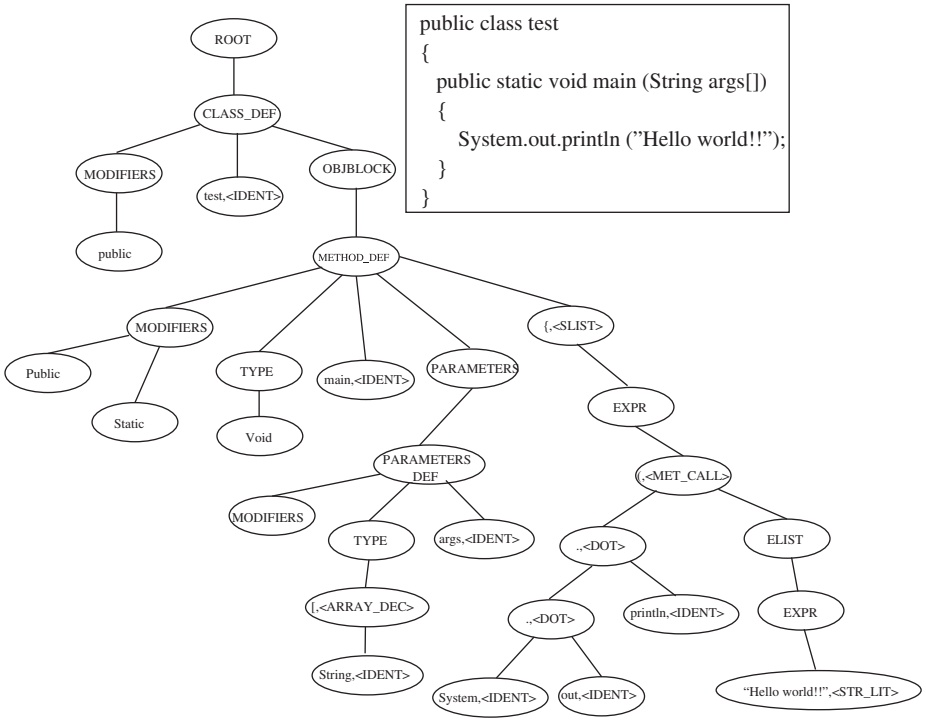

Fig. 2. An example of extracted parse tree from a simple code.

spanned by all subtrees that could possibly appear in the parse tree. For example, when a parse tree shown in Fig. 3(a) is given, it is represented as a vector whose elements are its subtrees as shown in Fig. 3(b). The value of each element is the frequency of the corresponding subtree.

The explicit enumeration of all subtrees is computationally problematic since the number of subtrees increases exponentially as the size of tree grows. Collins and Duffy proposed a method to compute the inner product of two trees without having to enumerate all subtrees. Let $subtree_1, subtree_2, \ldots$ be all of the subtrees in a parse tree T. Then, T can be represented as a vector

$$V_T = (\#subtree_1(T), \#subtree_2(T), \ldots, \#subtree_t(T)),$$

where $\#subtree_i(T)$ is the frequency of $subtree_i$ in the parse tree T. The kernel function between two parse trees T_1 and T_2 can be defined as $k(T_1, T_2) = V_{T_1}^T V_{T_2}$ and determined as

$$\begin{aligned} k(T_1, T_2) &= V_{T_1}^T V_{T_2} \\ &= \sum_i \#subtree_i(T_1) \cdot \#subtree_i(T_2) \\ &= \sum_i \left(\sum_{n_1 \in N_{T_1}} I_{subtree_i}(n_1) \right) \cdot \left(\sum_{n_2 \in N_{T_2}} I_{subtree_i}(n_2) \right) \\ &= \sum_{n_1 \in N_{T_1}} \sum_{n_2 \in N_{T_2}} C(n_1, n_2), \end{aligned}$$

where N_{T_1} and N_{T_2} are all the nodes in trees T_1 and T_2. The indicator function $I_{subtree_i}(n)$ is 1 if $subtree_i$ is rooted at node n and 0 otherwise. $C(n_1, n_2)$ is a function which is defined as

$$C(n_1, n_2) = \sum_i I_{subtree_i}(n_1) \cdot I_{subtree_i}(n_2).$$

This function can be calculated in polynomial time using the following recursive definition.

If the productions at n_1 and n_2 are different

$$C(n_1, n_2) = 0$$

Else if both n_1 and n_2 are pre-terminals

$$C(n_1, n_2) = 1 \tag{1}$$

Else

$$C(n_1, n_2) = \prod_i^{nc(n_1)} (1 + C(ch_i(n_1), ch_i(n_2))), \tag{2}$$

where $nc(n_1)$ is the number of children of node n_1 in the tree. Since the productions at n_1 and n_2 are the same, $nc(n_1)$ is also equal to $nc(n_2)$. Here, $ch_i(n_1)$ denotes the i-th child node of n_1.

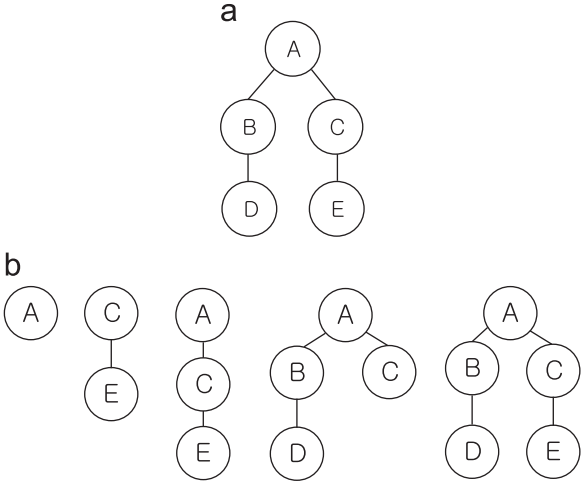

Fig. 3. An example of a parse tree and its subtrees. (a) A parse tree. (b) Subtrees which possibly appears in the tree.

This recursive algorithm is based on the fact that all subtrees rooted at a certain node can be constructed by combining the subtrees rooted at each of its children.

3.3. Modified parse tree kernel

To apply the parse tree kernel to plagiarism detection, two major modifications are needed to cope with the problems that come from program source codes. The first problem is related with the depth of parse trees generated from source codes. Compared to parse trees of natural language sentences, parse tree of program sources tends to be much larger and deeper. This is due to the fact that the length of a natural language sentence is usually much shorter than the length of a source code. The width of a large tree is not a problem, but the depth of a large tree can be problematic. When a change occurs at a high level (near the root) of a tree, it is much more influential than the same change occurring at a low level (near leaves). Since the parse tree kernel uses all subtrees, the changes near the root node are reflected more often than the changes near the leaf nodes.

Consider that we have a tree with depth n, and two nodes from the tree: a near-root node A and a leaf node B. When we generate all possible subtrees with a depth k from the tree, as the number k approaches n, the number of subtrees that include node A grows more rapidly than the number of subtrees that include node B. For example, consider the tree of Fig. 2, and compare node A (root node) with node E (leaf node). Every subtree with depth-3 must have the root node, but it is possible to generate depth-3 subtrees without including the leaf node E (like two middle ones in Fig. 3 (b)). Likewise, among five possible depth-2 subtrees, three of them have the root node while only one of them includes node E. In parse tree kernels, each sub-structure is corresponding to a dimension of a tree vector. This means that the changes at near-root node affect more elements of the vectors than those of leaf nodes. When the depth of a tree is relatively small, this is not a great factor. However, this effect is significant when the depth is greater.

This problem can be overcame by introducing the decay factor λ ($0 < \lambda \leq 1$). The decay factor scales the relative importance of subtrees by their size. That is, when a subtree has a large depth, the kernel value of the subtree is penalized for its depth as much as λ^{size}, where *size* is the depth of the subtree. Then, the effect of large subtrees is reduced.

The decay factor controls the effect of large subtrees with their sizes efficiently. However, large subtrees still participate in constructing a tree vector, and their effects exist in the kernel value. Thus, in this paper, the maximum depth of possible subtrees is explicitly limited with a pre-defined threshold value Δ. By restricting the depth, the modified kernel can calculate the similarity without the unwanted disproportionate influence of near-root level changes. With the decay factor λ and the threshold value Δ, the recursive rules of the parse tree kernel are modified as

If the productions at n_1 and n_2 are different

$$C(n_1, n_2) = 0$$

Else if both n_1 and n_2 are pre-terminals or the current depth equals the predefined threshold value Δ

$$C(n_1, n_2) = \lambda$$

Else

$$C(n_1, n_2) = \lambda \prod_i^{nc(n_1)} (1 + C(ch_i(n_1), ch_i(n_2))). \tag{3}$$

This modification corresponds to a kernel

$$k(T_1,T_2)=\sum_{subtree_i\in ST_\Delta}\lambda^{size_i}\left(\sum_{n_1\in N_{T_1}} I_{subtree_i}(n_1)\right)\cdot\left(\sum_{n_2\in N_{T_2}} I_{subtree_i}(n_2)\right) \quad (4)$$

where ST_Δ is a set of subtrees whose depth is less than Δ.

The second problem is that the parse tree kernel compares the orders (sequence) of subtrees. In natural language sentences, comparing orders of child nodes is meaningful and often corresponds to a specific piece of grammar. However, program structures corresponding to subtrees are often without order. For example, consider two methods within a class. The two methods are subtrees of the larger tree that represents the class. In this case, the order of the two subtrees (methods) is nothing to do with the functionality of the class. To solve this problem, the first and second recursive rules are modified as

If n_1 *and* n_2 *are different*

$C(n_1,n_2)=0$

Else if both n_1 and n_2 are *terminals* or the current depth equals the predefined threshold value Δ

$C(n_1,n_2)=\lambda$

In the modified rules, the kernel does not compare the ordered sequence of child nodes.

When the modified recursive rules are applied to the kernel, Eq. (3) does not work any more since the number of child nodes can be different in n_1 and n_2. Therefore, Eq. (3) must be modified. There are two ways to modify it:

1. *Maximum node value* (*MNV*):

$$C(n_1,n_2)=\lambda\prod_{i}^{nc(n_1)}\left(1+\max_{ch\in ch_{n_2}} C(ch_i(n_1),ch)\right) \quad (5)$$

2. *Mean value* (*MV*):

$$C(n_1,n_2)=\lambda\prod_{i}^{nc(n_1)}(1+E_{ch\in ch_{n_2}}[C(ch_i(n_1),ch)]), \quad (6)$$

where ch_{n_2} is a set of child nodes of n_2.

MNV computes the value of $C(n_1,n_2)$ with maximum similarity between child nodes of n_1 and n_2, while MV uses the mean value of similarities between child nodes. Plagiarism detection is the task to capture a set of source code pairs with high similarity values. Thus, MNV seems to be more appropriate for plagiarism detection, since it returns the value of the most similar node among the child nodes. In this paper, MNV is adopted in the modified parse tree kernel rather than MV. Parse tree kernels with the modified C functions actually do not satisfy Mercer's condition. However, many functions that do not satisfy Mercer's condition work yet in computing similarity (Moschitti and Zanzotto, 2007).

The value of the kernel function itself is not normalized and depends greatly on the size of the trees. Thus, a normalization is needed to use the kernel function as a similarity measure. Let s and s' be the source codes to be compared and the function $k(s,s')$ returns the similarity using the modified parse tree kernel. Then, the normalized similarity $N(s,s')$ is defined as

$$N(s,s')=\frac{k(s,s')}{\sqrt{k(s,s)\cdot k(s',s')}}.$$

This similarity between s and s' is bounded from 0 to 1 ($0\le N(s,s')\le 1$).

With the source code similarity N and a threshold, it is possible to determine plagiarized source code pairs. Let $\mathcal{S}$ be a set of source codes. Plagiarism detection aims to generate a list of plagiarized source codes based on the similarity between $s\in\mathcal{S}$ and $s'\in\mathcal{S}$. All source codes in $\mathcal{S}$ (except s) that are similar to s more than a predefined threshold θ are regarded as plagiarized codes. In other words, for each source code $s\in\mathcal{S}$, a set of plagiarized source codes $\mathcal{S}'$ is defined as

$$\mathcal{S}'=\{s'\in\mathcal{S}|N(s,s')\ge\theta \text{ and } s\neq s'\}.$$

Table 1
Average similarity between original codes and plagiarized codes.

Attack type	General tree kernel	Source code tree kernel	JPlag	CCFinder
1	0.50	0.73	0.58	**0.78**
2	0.59	**0.90**	0.77	0.50
3	0.55	**0.85**	0.68	0.66
4	0.56	0.68	0.52	**0.75**

4. Experiments

The proposed system is evaluated using two data sets: a synthesized data set and a real-world data set. The goal of the first experiment is to evaluate the effectiveness of the proposed system against specific "plagiarism attacks". The second experiment aims to evaluate the proposed system in a real-world environment. In all experiment, the threshold for subtree depth Δ is set as 3 and the decay factor λ is 0.3 heuristically.

4.1. Experiments on synthesized data set

The goal of the first experiment is to measure the effectiveness of detection system against specific plagiarism attacks, especially in terms of reduced similarity between the original code and the plagiarized codes. The synthesized data set was prepared by four programmers. Each programmer wrote a Java program to find the n-th triangular number, and then generated four copies of his program with the following four attacks.

1. Modify variables, functions, or class names.
2. Change expressions such as '*for*', '*do ~while*', and '*if*' to the synonymous expressions (for example, substituting *for* to *while* and *if* to *case*).
3. Merge or separate functions and classes.
4. Add useless lines into a source code (60, 120, 600 lines).

Table 1 summarizes the results of the first experiment. The table compares four systems: our previous detection system that is described in Son et al. (2006) (general tree kernel), the proposed system (source code tree kernel), JPlag (Prechelt et al., 2002),[1] and CCFinder (Kamiya et al., 2002).[2] The similarity value of 1.0 means that the two codes are identical, and the similarity value 0 means the two codes are completely different. Values in the table are average similarity values between the plagiarized pairs, and higher values indicate more reliable detection. The proposed kernel is strongest on structural attacks (types 2 and 3), while JPlag is strong on attack type 2 (replacement of for/while), but relatively weak on type 4 (padding attack). The performance of the general tree kernel is mediocre for all attacks. CCFinder records the highest similarity on attack type 4. However, it also shows the lowest similarity on attack type 2. In CCFinder, structural information is used to resolve lexical variations. However, when reserved tokens

[1] https://www.ipd.uni-karlsruhe.de/jplag/.
[2] http://www.ccfinder.net.

such as '*for*' or '*while*' are changed to the synonymous tokens, CCFinder fails in detecting such variations.

All four systems successfully found the synthesized plagiarized pairs. The average similarity values among non-plagiarized pairs (between different programmers) were 0.35 for the general tree kernel, 0.54 for the proposed tree kernel, and 0.28 for CCFinder. JPlag does not report similarity values among non-plagiarized source codes. The similarity values given by the proposed kernel are generally higher. This is mainly because of the second modification given to the proposed kernel. All three systems showed a sufficient margin between the plagiarized pairs and original pairs. The gap is 0.20 for general tree kernel, 0.25 for the proposed kernel, and 0.38 for CCFinder. Thus by setting a proper threshold value, all systems can detect plagiarized pairs from the synthesized source codes reliably.

4.2. Experiments on real world data set

To evaluate the system in a real-world environment, we collected and prepared a data set that consists of actual programming assignments submitted by college students. We collected assignments in nine Java programming classes from the years 2005 to 2009. Table 2 shows some simple statistics of the data set. The total number of assignments is 36. We have 555 pieces of source code. The average number of source codes per assignment is 15.42, and the average number of lines per code is 305.07.

Two annotators created the gold standard for this data set. They independently investigated the source codes and marked plagiarized pairs manually. In order to measure the reliability and validity of the annotators, we used Cohen's kappa agreement. The kappa agreement of the annotators is $\kappa = 0.93$, which falls on the category of 'almost perfect agreement'. Only the pairs that are annotated by both judges are used as the gold standard answer of plagiarized pairs. In total, 175 pairs are marked as plagiarized pairs, and this is about 31% of the submitted assignments.

Three metrics are used for the evaluation: precision, recall and F-1 measure. They are calculated as follows:

$$Precision = \frac{\#\text{ of correctly detected plagiarized pairs}}{\#\text{ of detected plagiarized pairs}}$$

$$Recall = \frac{\#\text{ of correctly detected plagiarized pairs}}{\#\text{ of true plagiarized pairs}}$$

$$\text{F}-1\ measure = 2 \cdot \frac{\text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}}$$

The proposed system is again compared with three baselines: our previous plagiarism detection system based on the general tree kernel, JPlag, and CCFinder.

Fig. 4 shows the performance of the proposed system in a precision/recall graph. In this experiment, the proposed system shows the best performance of 0.93 in F-1 measure at threshold $\theta = 0.95$. The precision and recall curves in Fig. 4 also show the strength of the proposed system. As the threshold increases, the precision is rapidly increasing while the recall is only slightly decreasing. Even at the point of highest precision, the recall is only 20% lower than the best recall. With lower threshold values, the system can achieve high recall at the cost of some precision. The drop of precision is, however, quite stable and achieves relatively high precision even at near 1.0 recall. This result implies that the proposed system can efficiently distinguish plagiarized codes from non-plagiarized ones.

Table 2
A simple statistics on the real data set.

Information	Value
Number of total assignments	36
Number of submitted source codes	555
Number of marked plagiarism pairs	175
Average number of submitted codes per assignment	15.42
Average number of lines per source code	305.07

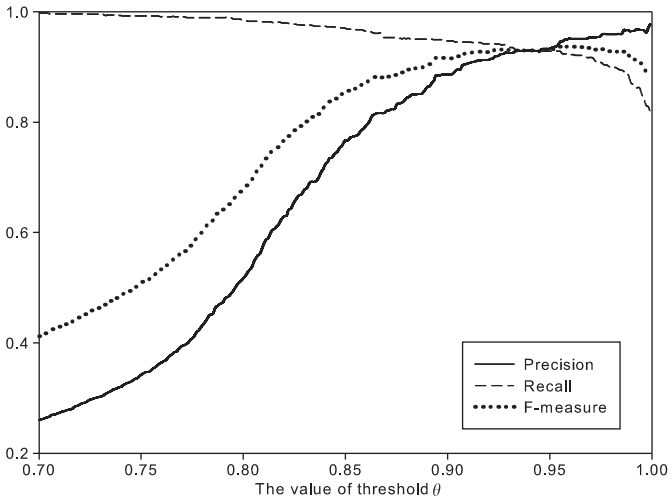

Fig. 4. Performance of the proposed system on real-world data set.

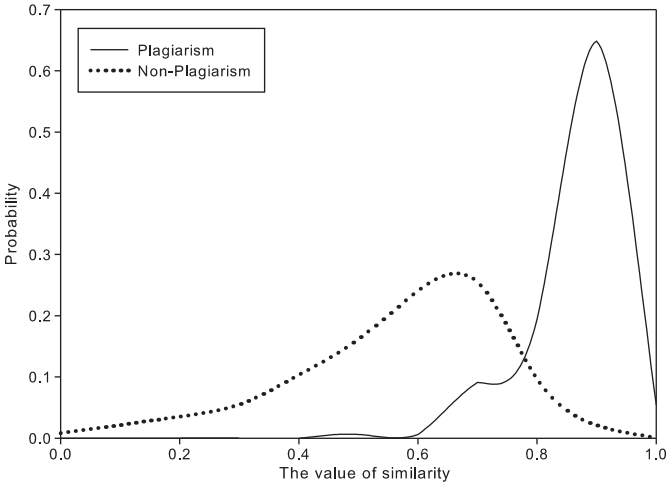

Fig. 5. Distribution of similarities in the space of the proposed kernel.

The effectiveness of the proposed kernel and its kernel space can be observed from the viewpoint of distributions. Fig. 5 shows the distribution of similarities. In this figure, the *X*-axis is the similarity value determined by the proposed system and the *Y*-axis denotes the probability of observing a source code pair with the similarity value of *X*. The solid line in the graph denotes the distribution of similarities between plagiarized source codes, while the dotted line is that of non-plagiarized codes. According to this figure, the distribution of plagiarized source codes is approximately a Gaussian distribution with a mean of 0.93. On the other hand, the distribution of non-plagiarized source codes follows a Gaussian distribution with a mean of about 0.63. The difference between the two distributions can be measured with Kullback–Leibler divergence, a distance measure between distributions

$$D_{KL}(P\|Q) = \frac{\int_0^{1.0} p(x)\log\frac{p(x)}{q(x)}\,dx + \int_0^{1.0} q(x)\log\frac{q(x)}{p(x)}\,dx}{2},$$

where P and Q are the distributions to be measured. The Kullback–Leibler (KL) divergence between the two distributions is 2.01, and this value indicates that the two similarity distributions are significantly different. The gap between the means of the two distributions is 0.30. The large gap between the means and the significant KL divergence shows that the kernel space is discriminating well the two groups: non-plagiarized source pairs and plagiarized source pairs. This means that the kernel is very effective at detecting plagiarized codes from original codes.

Even though the similarity between plagiarized source codes is significantly different from non-plagiarized ones, the mean of the non-plagiarized similarity distribution, 0.63, is still high. This is due to the fact that some assignments simply require students to extend programs from the textbook, instead of composing a program from scratch. As a result, pair similarities among source codes in such assignments are higher than others, and it raised the mean of the non-plagiarized similarity distribution. Even in such a situation, the proposed system shows reliable performance.

The effect of the two modifications to the kernel (described in Section 3.3) can be seen in the precision/recall graph of the system based on the general tree kernel. Fig. 6 shows the graph of the baseline. The performance of this baseline is quite good in mid-range recall situations, and achieves a score of about 0.82–0.83 in terms of F-1 measure. However, there is a sudden drop of precisions around a recall value of 0.90. This greatly hurts the performance of the baseline system in high recall situations.

The plagiarism detection results are summarized in Table 3. It shows the precisions of the detection output on four recall points. The JPlag does not permit users to scale the threshold value, and it outputs only a single set of results. Thus, the precisions of the first three recall points (0.80, 0.85 and 0.90) are empty for JPlag. Due to the same reason with JPlag, the performance of CCFinder is also represented for the recall point 0.96. The output of JPlag scored 0.56 in F-1 measure (recall of 0.96 and precision of 0.39). The low precision of JPlag is mainly due to the large number of false positives. CCFinder shows the precision of 0.56 at the recall of 0.96. Since CCFinder utilizes more structural information than JPlag, it achieves 25% better performance than JPlag in terms of F-1 measure. However, CCFinder adopts the structural information just to resolve lexical variations in source codes. Thus, the influence of the structural information is limited in plagiarism detection. On the other hand, the proposed system records a relatively high precision of 0.79 at the recall of 0.96 by handling entire structure of source codes. The baseline system of the general kernel achieves a very low precision of 0.17 at the same recall.

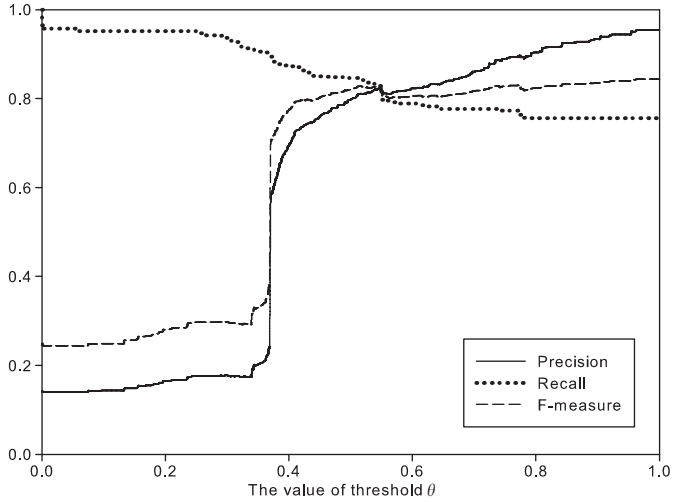

Fig. 6. Performance of the general tree kernel on the real-world data set.

Table 3
Plagiarism detection results with an R-precision on recall of 0.80, 0.85, 0.90 and 0.96.

	General tree kernel	Source code tree kernel	JPlag	CCFinder
Precision at recall 0.80	0.81	0.97	–	–
Precision at recall 0.85	0.75	0.96	–	–
Precision at recall 0.90	0.25	0.95	–	–
Precision at recall 0.96	0.17	0.79	0.39	0.56
F-1 measure at recall 0.96	**0.29**	**0.86**	**0.56**	**0.70**

The result shows that the proposed system based on a source code tree kernel performs much better than other well-known plagiarism detection systems that use structural information partially. Previous tree kernel used in Son et al. (2006) performs adequately in the mid recall range, but its performance is drastically worsen in high recall situations. Since plagiarism detection systems often serve as a first filter for possible plagiarized pairs in semi-automated detection, the performance in high recall situations is very important. These results show that the two modifications of the tree kernel proposed in the paper are crucial in enhancing the plagiarism detection performances. The proposed system obtained an F-1 measure score of 0.93 at its peak performance, and it achieves relatively high precision even in the extremely high coverage case with recall of 0.96.

5. Conclusions

In this paper, we proposed an automatic program plagiarism detection system. The proposed system compares source codes with a specialized tree kernel for parse trees of source codes. Parse tree kernels used in the NLP domain are not appropriate to program source codes due to their characteristics. Compared to the parse trees of natural language sentences, the parse trees of program source codes are much larger and deeper, and the order of subtrees is not informative. We proposed a specialized parse tree kernel for program source codes: first, asymmetric effects of the changes near-root node have been reduced by limiting depth of subtrees. Second, subtrees are compared without regarding the sequence among them, by adopting a new recursive rules.

The proposed system detects plagiarized source codes in three steps. The system first represents all source code as trees. Then, similarity values between all pairs are calculated via the proposed kernel. Finally, source code pairs that are more similar than a certain predefined threshold are detected as plagiarized pairs.

The effectiveness of the proposed system was evaluated in two experiments. When experimented with synthesized data, the proposed system is strong against structural attacks, and is more reliable than the other baseline systems. The experiments with the real-world college data showed that the performance of the proposed system reaches to 93% level of the human annotators. The system outperforms well-known program plagiarism detection systems by 0.16 points of F-1 measure.

Although the experiments were only conducted in Java, the detection system could be used with other languages such as C, C++ and Pascal. The only requirement of our system to work with such languages is parsers for the languages. Since the parser module used in the method (ANTLR) is a parser-generating tool that can produce various parsers, the proposed system can be easily used with other languages.[3]

[3] Until now, only Java tree-grammar is available for ANTLR.

Role of funding source

This work was supported in part by the Industrial Strategic Technology Development Program (10035348, Development of a Cognitive Planning and Learning Model for Mobile Platforms) funded by the Ministry of Knowledge Economy (MKE, Korea). The funding sources did not influence the research direction and submission decision.

References

Berghel, H., Sallach, D., 1984. Measurements of program similarity in identical task environments. SIGPLAN Not. 19, 65–75.

Bravo-Marquez, F., L'Huillier, G., Rios, S., Velásquez, J., 2011. A text similarity meta-search engine based on document fingerprints and search results records. In: Proceedings of the 2011 IEEE/WIC/ACM International Conferences on Web Intelligence and Intelligent Agent Technology, pp. 146–153.

Brin, S., Davis, J., García-Molina, H., 1995. Copy detection mechanisms for digital documents. In: Proceedings of the ACM SIGMOD International Conference on Management of Data, pp. 398–409.

Chen, X., Francia, B., Li, M., McKinnon, B., Seker, A., 2004. Shared information and program plagiarism detection. IEEE Trans. Inf. Theory 50, 1545–1551.

Collins, M., Duffy, N., 2001. Convolution kernels for natural language. In: Proceedings of the NIPS 2001, pp. 625–632.

Durić, Z., Gaševeć, D., 2013. A source code similarity system for plagiarism detection. Comput. J. 56, 70–86.

Evans, R., 2006. Evaluating an electronic plagiarism detection service. Active Learn. Higher Educ. 7, 87–99.

Gitchell, D., Tran, N., 1998. A utility for detecting similarity in computer programs. In: Proceedings of the 30th ACM Special Interest Group on Computer Science Education Technology Symposium, pp. 266–270.

Grier, S., 1981. A tool that detects plagiarism in pascal programs. In: Twelfth SIGCSE Technical Symposium, vol. 13, pp. 15–20.

Halstead, M., 1977. Elements of Software Science. Elsevier.

Haussler, D., 1999. Convolution kernels on discrete structures. Technical Report UCS-CRL-99-10 University of California at Santa Cruz.

Kamiya, T., Kusumoto, S., Inoue, K., 2002. CCFinder: a multilinguistic token-based code clone detection system for large scale source code. IEEE Trans. Software Eng. 28, 654–670.

Maurer, H., Kappe, F., Zaka, B., 2006. Plagiarism—a survey. J. Univ. Comput. Sci. 12, 1050–1084.

Moschitti, A., Zanzotto, F., 2007. Fast and effective kernels for relational learning from texts. In: Proceedings of the International Conference on Machine Learning, pp. 649–656.

Nawab, R., Stevenson, M., Clough, P., 2013. Comparing medline citations using modified n-grams. J. Am. Med. Inf. Assoc., http://dx.doi.org/10.1136/amiajnl-2012-001552, in press.

Oberreuter, G., Velásquez, J., 2013. Text mining applied to plagiarism detection: the use of words for detecting deviations in the writing style. J. Expert Syst. Appl. 40, 3756–3763.

Parker, A., Hamblen, J., 1989. Computer algorithms for plagiarism detection. IEEE Trans. Educ. 32, 94–99.

Parr, T., Quong, R., 1995. ANTLR: a predicated LL(k) parser generator. J. Software Pract. Exper. 25, 789–810.

Prechelt, L., Malphol, G., Philippsen, M., 2002. Finding plagiarisms among a set of programs with jplag. J. Univ. Comput. Sci. 8, 1016–1038.

Shivakumar, N., García-Molina, H., 1995. SCAM: a copy detection mechanism for digital documents. In: Proceedings of the 2nd Annual Conference on the Theory and Practice of Digital Libraries.

Si, A., Leong, H., Lau, R., 1997. CHECK: a document plagiarism detection system. In: Selected Areas in Cryptography, pp. 70–77.

Sojka, P., Kopeček, I., Pala, K., 2006. PPCnecker: plagiarism pattern checker in document copy detection. In: Text, Speech and Dialogue, Lecture Notes in Computer Science, vol. 4188, pp. 661–667.

Son, J.-W., Park, S.-B., Park, S.-Y., 2006. Program plagiarism detection using parse tree kernels. In: PRICAI 2006: Trends in Artificial Intelligence, Lecture Notes in Computer Science, vol. 4099, pp. 1000–1004.

White, D., Joy, M., 2004. Sentence-based natural language plagiarism detection. J. Educ. Resour. Comput. 4.