Hindawi Publishing Corporation
Mathematical Problems in Engineering
Volume 2015, Article ID 429807, 12 pages
http://dx.doi.org/10.1155/2015/429807

Research Article

Computation of Program Source Code Similarity by Composition of Parse Tree and Call Graph

Hyun-Je Song, Seong-Bae Park, and Se Young Park

School of Computer Science and Engineering, Kyungpook National University, 80 Daehakro, Bukgu, Daegu 702-701, Republic of Korea

Correspondence should be addressed to Seong-Bae Park; sbpark@sejong.knu.ac.kr

Received 30 September 2014; Accepted 15 December 2014

Academic Editor: Jianjun Jiao

This paper proposes a novel method to compute how similar two program source codes are. Since a program source code is represented as a structural form, the proposed method adopts convolution kernel functions as a similarity measure. Actually, a program source code has two kinds of structural information. One is syntactic information and the other is the dependencies of function calls lying on the program. Since the syntactic information of a program is expressed as its parse tree, the syntactic similarity between two programs is computed by a parse tree kernel. The function calls within a program provide a global structure of a program and can be represented as a graph. Therefore, the similarity of function calls is computed with a graph kernel. Then, both structural similarities are reflected simultaneously into comparing program source codes by composing the parse tree and the graph kernels based on a cyclomatic complexity. According to the experimental results on a real data set for program plagiarism detection, the proposed method is proved to be effective in capturing the similarity between programs. The experiments show that the plagiarized pairs of programs are found correctly and thoroughly by the proposed method.

1. Introduction

Many real-world data resources such as web tables and powerpoint templates are usually represented in a structural form for providing more organized and summarized information. Since this type of data contains various and useful information in general, they are often an information source for data mining applications [1]. However, there exist a number of duplications of the data nowadays due to the characteristics of digital data. This phenomenon makes data mining from such data overloaded. Program source code is a data type that is duplicated frequently like web tables and powerpoint templates. Therefore, plagiarism of program source codes becomes one of the most critical problems in education of computer science. Students can submit their program assignments by plagiarizing someone else's work without any understanding of the subject or any reference of the work [2]. However, it is highly impractical to detect all plagiarism pairs manually, especially when the size of source codes is considerably large. Thus, a method to remove such duplicates is required, and the first step to do it is to measure the similarity between two source codes automatically.

Program source code is one of the representative structured data. Although a program source code looks like continuous strings, it is easily represented as a structural form, that is to say, a parse tree. A source code is represented as a tree structure after compiling it with a grammar for a specific programming language. Therefore, the similarity between program source codes has to take their stuctures into consideration. Many previous studies have proposed similarity measures for program source code comparison, and most of them reflect structural information to some extent [3, 4]. However, there are some shortcomings of the similarity measures. First, they cannot reflect entire structure of a source code, because they represent the structural information of the source code as structural features defined at lexical level. In order to overcome this problem, some studies consider the structural information at a structure level such as parse tree [5] and function-call graph [6]. Second, there is no study to consider a parse tree and a call graph at the same time even though each structure provides a different structural view on the program source code. A parse tree gives a relatively local structural view, while a call graph provides a high and global

level structural view. Since both views are useful to detect plagiarized pairs of program source codes, the similarity measure for program source code comparison should reflect both kinds of structural information simultaneously.

This paper proposes a novel method to calculate the similarity between two program source codes. The proposed method adopts two kinds of structural information based on kernel functions. A kernel function is one of the prominent methods for comparing structured data [7]. It can be used as a similarity measure, since it calculates an inner product of two elements [8]. The proposed method reflects the syntactic structure of a program source code using a parse tree kernel [9]. A parse tree kernel computes the similarity between a pair of parse trees. Thus, the syntactic structural information of a source code is fully reflected into the proposed method by it. The proposed method consider also a dynamic structure of a source code by adopting a graph kernel [10]. The graph kernel in the proposed method computes the similarity value between a pair of function-call graphs. Since these two kernels are instances of R-convolution kernels by Haussler [7], they compare trees and graphs efficiently without explicit feature enumeration, respectively.

Each kernel produces its own similarity based on its own structural view. The proposed method incorporates both kinds of structural information into program source code comparison by composing the parse tree kernel and the graph kernel into a composite kernel. Since the proposed composite kernel is based on a weighted sum composition, optimizing the weights of base kernels is a crucial issue. In this paper, the weights are automatically determined with the consideration of the complexity of source codes. Thus, if any two program source codes are given, the proposed method can compute their similarity.

The proposed method is evaluated on source code plagiarism detection with a real data set used in the work [5]. Our experiments show three important results. First, the similarity measure based on the parse tree kernel is more reliable than that based on the graph kernel in terms of overall performance. Second, the more complicated a source code, the more useful the similarity based on the graph kernel in detecting plagiarized pairs. Finally, the proposed method which combines the parse tree kernel and the graph kernel detects plagiarism of real-world program source codes successfully. These results prove that global-level structural information is an important factor for comparing programs and that the proposed similarity measure that combines syntactic and dynamic structural information results in good performance for program source code plagiarism detection.

In summary, we draw the following contributions in this paper.

(1) We design and implement a source code similarity measure for plagiarism detection based on two kinds of structural information: syntactic information and dependencies of function calls. From the fact that the change of the structure of the source code is harder than the one of the user-defined vocabulary, the proposed method is robust for detecting plagiarism pairs.

(2) In order to make use of two kinds of structural information simultaneously, we design new combination method based on a complexity of source code. This makes the proposed method work more robustly even if we compare between complicated source codes.

The rest of the paper is organized as follows. Section 2 is devoted to related studies on program source code comparison and program source code plagiarism detection. Section 3 introduces the problems of source code plagiarism detection. The similarity measure based on parse tree kernel and functional-call graph kernel are given in Sections 4 and 5, respectively. Section 6 proposes the composite kernel that combines the parse tree kernel and the graph kernel. Section 7 explains the experimental settings and the results obtained by the proposed method. Finally, Section 8 draws the conclusion.

2. Related Work

Measuring the similarity between two objects is fundamental and important in many scientific fields. For instance, in molecular biology, it is often required to measure the sequence similarity between protein pairs. Thus, many similarity measures have been proposed such as distance-based measurements including Euclidean distance and Levenshtein distance, mutual information [11], information content using wordNet [12], and citation-based similarity [13]. In addition, the measures have been applied to various applications such as information retrieval [14, 15] and clustering [16] as their core part.

The similarity measure for source codes have been of interest for a long time. Most early studies are based on attribute-counting similarity [17, 18]. The similarity represents a program as a vector of various elements such as the number of operators and operands. Then, a vector similarity is normally used to detect plagiarized pairs. However, the performance of this approach is relatively poor compared to other methods that consider structure of source codes, since this approach uses only the abstract-level information.

In order to overcome the weaknesses of the attribute-counting-metric approach, some studies incorporate the structural information of source code into their similarity measure. In general, the structure of source codes is a tree or a graph. From the fact that a source code is compiled into a syntactic structure as described by the grammar of a programming language, some studies used a tree matching algorithm to calculate the similarity between source codes [4, 19]. However, the algorithm represents a source code as a string that contains certain structural information, so that it fails in reflecting an entire structure of a source code into a similarity measure. On the other hand, some other studies used the knowledge that comes from the topology of source codes. Horwitz first adopted graph structures for comparing two programs [3] and determined which components are changed from one to another based on the program dependency graph [20]. Liu et al. also used the program dependency graph to represent a source code and adopted the relaxed subgraph isomorphism testing to compare two source

codes efficiently [21]. Kammer built a plagiarism detection tool for Haskell language [6]. He extracted a call-graph first from a source code. The nodes in the graph are functions and an edge indicates that one function calls another function. Then, he transformed the graph into a tree to compare source codes efficiently. Finally, he applied the A^*-based tree edit distance and the tree isomorphism algorithm for the comparison of source codes. However, this approach loses much information lying on a graph. since the graph is transformed into a tree. Lim et al. proposed a method of detecting plagiarism of Java programs through analysis of flow paths of Java bytecodes [22]. Since a flow path of a program is a sequence of basic blocks upon execution of the program, they tried to align flow paths using a semiglobal alignment algorithm and then detected program source code plagiarism pairs. Chae et al. also tried to detect binary program (executable file) plagiarism [23]. They constructed first *A-CFG* (API-labeled control flow graph) that is the functional abstraction of a program and then generated a vector of a predefined dimension from the A-CFG using Microsoft Development Network Application Programming Interface (MSDN API) to avoid computational intractability. Finally, they used a random walk (page-rank) algorithm to calculate the similarity between programs. Unfortunately, this approach cannot be applied to other languages that do not have MSDN API. Recently, some studies have used stopword n-grams [24] and topic model [25] to measure the similarity.

Several program source code plagiarism detection tools are available online. Most of them use a string tokenization and a string matching algorithm to measure the similarity between source codes. Prechelt et al. proposed JPlag. It is a system that can be used to detect plagiarism of source codes written in C, C++, Java, and Scheme [26]. It first extracted tokens from source codes and then compared tokens from each source code using Karp-Rabin Greedy String Tiling algorithm. Another widely-used plagiarism detection system is *MOSS* (Measure Of Software Similarity) proposed by Aiken [27]. It is also based on a string-matching algorithm. It divides programs into k-grams, where a k-gram is a contiguous substring of length k. Then, the similarity is determined by the number of same k-grams shared by the programs. One of the state-of-the-art and well-known plagiarism detection systems is CCFinder proposed by Kamiya et al. [28]. It uses both attribute-counting-metric and structure information. A source code is transformed into a set of normalized token sequences by its own transformation rules. The transformation rules are constructed manually for each language to express structural characteristics of languages. Then, the normalized tokens are compared to reveal clone pairs in source codes. They showed relatively good performance and used structural information to some degree, but it does not reflect the structural information of source codes into its similarity measure fully.

The proposed method in this paper extends the kernel-based method proposed by Son et al. [5]. They compared the structure of source codes using a kernel function directly. They used the parse tree kernel especially [9], a kind of R-convolution kernels [7], to compare tree structure of source codes. Compared to this work, the proposed method incorporates function-call information additionally. The functional calls are one of the important structural information in comparing source codes. The main problem of Son et al. is that they focused only on syntactic structural information that is local and static. On the other hand, the function-call information provides a global view on source code execution. Therefore, the plagiarized pairs of source codes are detected more accurately by considering not only the syntactic structure but also the function-call information.

3. Program Source Code Plagiarism Detection

Plagiarism detection for program source codes, also known as programming plagiarism detection, aims to detect plagiarized source code pairs among a set of source codes. The source code plagiarism detection normally consists of three steps as illustrated in Figure 1. The first step is a preprocessing step that extracts features such as tokens and parse trees from source codes. The second step calculates pair-wise similarity with the extracted features and a similarity measure. Therefore, the similarity values among all pairs are recorded into a similarity matrix. Finally, the groups of source codes that are most likely to be plagiarized are selected according to their similarity values.

Formally, let S be a set of source codes. Plagiarism detection aims to generate a list of plagiarized source codes based on a similarity $\mathrm{sim}(s, s')$ between $s \in S$ and $s' \in S$. If the similarity of a pair is higher than a predefined threshold θ, the pair is determined as a plagiarized one. Therefore, for a source code $s \in S$, a set of plagiarized source codes S' is defined as

$$S' = \left\{ s' \in S \mid \mathrm{sim}\left(s, s'\right) \geq \theta, s \neq s' \right\}. \quad (1)$$

The similarity measure, sim( ), is decided by information type extracted from source codes. A source code has two kinds of information: lexical and structural information. Lexical information corresponds to variables and the reserved words like *public*, *if*, and *for*. This vocabulary is composed of a large set of rarely occurring user-defined words (variables) and a small set of frequently occurring words (reserved words). On the other hand, structural information corresponds to a structure that is determined by reserved words. Among them, structural information is a more important clue for detecting plagiarism, since tokens can be easily converted into other tokens without understanding the subject of a source code. Therefore, this paper focuses on structural information. Note that a source program has two kinds of structural information. One is syntactic structure that is usually expressed as a parse tree and the other is function-call graph structure.

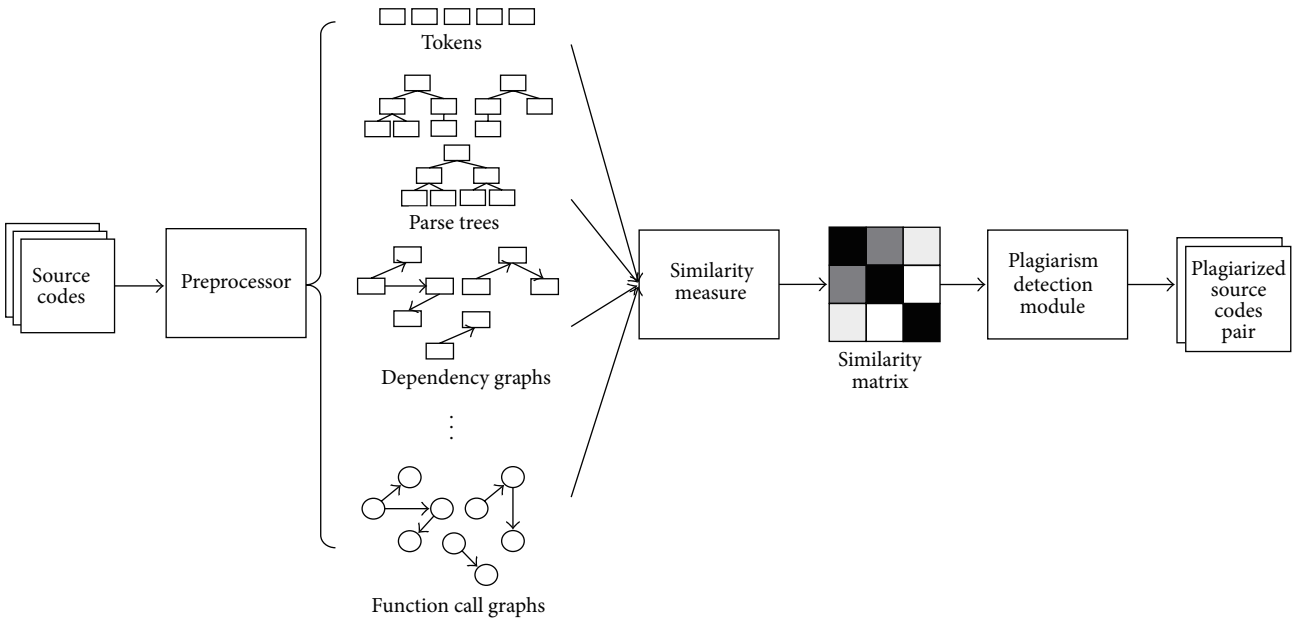

Figure 1: The overall process of program source code plagiarism detection.

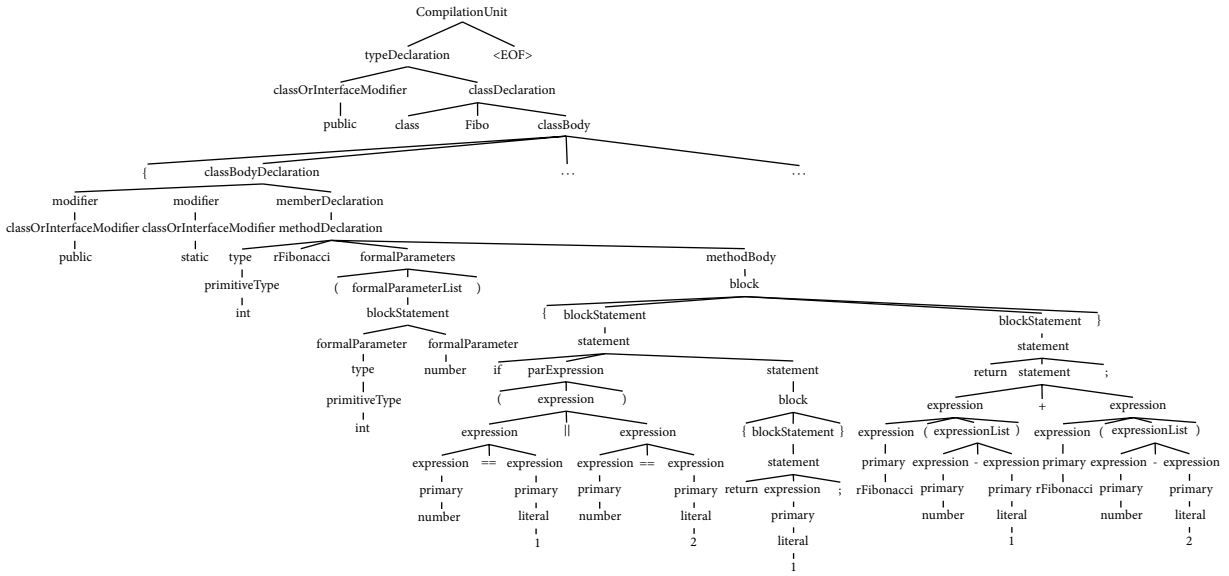

Figure 2: A parse tree extracted from the source code in Box 1.

4. Similarity Measure for Source Codes Based on Parse Tree Kernel

4.1. Source Code as a Tree. The program source code can be naturally represented as a parse tree of which each node denotes variables, reserved words, operators, and so on. Figure 2 shows an example parse tree extracted from a Java code in Box 1 (this parse tree is slightly different from the parse tree used in Son et al. [5]. This is because a more recent version of Java grammar is used in this paper). The Java code in Box 1 implements the Fibonacci sequence. Due to the lack of width of paper, only one function, `rFibonacci`, is shown in Figure 2, while there exist five functions in Box 1. As shown in this algorithm, a parse tree from a simple source code can be very large and deep-rooted.

In this paper, we use *ANTLR* (another tool for language recognition) (http://www.antlr.org/) to extract a parse tree from a source code. *ANTLR*, proposed by Parr and Quong, is a language tool that provides a framework for constructing recognizers, interpreters, compilers, and translators from grammatical descriptions [29]. With *ANTLR* and a language grammar, a tree parser that translates a source code into a parse tree can be easily constructed.

```
public class Fibo {
        public static int rFibonacci(int number) {
                if (number == 1 || number == 2) {
                        return 1;
                }
                return rFibonacci(number-1) + rFibonacci(number-2);
        }
        private static int sum(int value1, int value2) {
                return value1 + value2;
        }
        public static int iFibonacci(int number) {
                if (number == 1 || number == 2) {
                        return 1;
                }
                int fibo1 = 1, fibo2 = 1;
                int fibonacci = initOne();
                for (int i = 3; i <= number; i++) {
                        fibonacci = sum(fibo1, fibo2);
                        fibo1 = fibo2;
                        fibo2 = fibonacci;
                }
                return fibonacci;
        }
        private static int initOne() {
                return 1;
        }
        public static void main(String[] args) {
                int rFibo = Fibo.rFibonacci(7);
                int iFibo = Fibo.iFibonacci(7);
                System.out.println(rFibo);
                System.out.println(iFibo);
        }
}
```

Box 1: An example of Java source code.

Since parse tree has syntactic structural information, a metric for parse tree that reflects entire structural information is required. The parse tree kernel is one of such metrics. It compares parse trees without manually designed structural features.

4.2. Parse Tree Kernel. Parse tree kernel is a kernel that is designed to compare tree structures such as parse trees of natural language sentences. This kernel maps a parse tree onto a space spanned by all subtrees that can appear possibly in the parse tree. The explicit enumeration of all subtrees is computationally infeasible, 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 [9].

Let $\text{subtree}_1, \text{subtree}_2, \ldots$ be all of the subtrees in a parse tree T. Then, T can be represented as a vector

$$V_T = \left\langle \#\text{subtree}_1(T), \#\text{subtree}_2(T), \ldots, \#\text{subtree}_t(T) \right\rangle, \tag{2}$$

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

$$\begin{aligned} K_{\text{tree}}(T_1, T_2) &= V_{T_1}{}^T V_{T_2} \\ &= \sum_i \#\text{subtree}_i(T_1) \cdot \#\text{subtree}_i(T_2) \\ &= \sum_i \left(\sum_{n_1 \in N_{T_1}} I_{\text{subtree}_i}(n_1) \right) \\ &\quad \cdot \left(\sum_{n_2 \in N_{T_2}} I_{\text{subtree}_i}(n_2) \right) \\ &= \sum_{n_1 \in N_{T_1}} \sum_{n_2 \in N_{T_2}} C(n_1, n_2), \end{aligned} \tag{3}$$

where N_{T_1} and N_{T_2} are all the nodes in trees T_1 and T_2. The indicator function $I_{\text{subtree}_i}(n)$ 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

$$C(n_1, n_2) = \sum_i I_{\text{subtree}_i}(n_1) \cdot I_{\text{subtree}_i}(n_2). \quad (4)$$

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

(i) If the productions at n_1 and n_2 are different,

$$C(n_1, n_2) = 0. \quad (5)$$

(ii) If both n_1 and n_2 are preterminals,

$$C(n_1, n_2) = 1. \quad (6)$$

(iii) Otherwise, the C function can be defined as follow:

$$C(n_1, n_2) = \prod_i^{\text{nc}(n_1)} (1 + C(\text{ch}_i(n_1), \text{ch}_i(n_2))), \quad (7)$$

where $\text{nc}(n_1)$ is the number of children of node n_1 in the tree.

Since the productions at n_1 and n_2 are the same, $\text{nc}(n_1)$ is also equal to $\text{nc}(n_2)$. Here, $\text{ch}_i(n_1)$ denotes the ith child node of n_1. 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.

4.3. Modified Parse Tree Kernel. The parse tree kernel has shown good performance for parse trees of natural language, but it does not work well for program source code comparison due to two issues. The first issue is asymmetric influence of node changes. The parse tree from a source code tends to be much larger and deeper than that from a natural language sentence. Therefore, the changes near root node have been reflected more often than the changes near leaf nodes. The second issue is the sequence of subtrees. The original parse tree kernel counts the sequence of subtrees by considering their order. However, the order of two substructures in a source code is meaningless in programming languages.

Son et al. proposed a modified parse tree kernel to cope with these issues [5]. In order to solve the first issue, they introduced a decay factor λ_{tree} and a threshold Δ that control the effect of large subtrees. The decay factor scales the relative importance of subtrees by their size. As the depth of a subtree increases, the kernel value of the subtree is penalized by $(\lambda_{\text{tree}})^{\text{size}}$, where size is the depth of the subtree. In addition, the limitation of the maximum depth of possible subtrees is set as Δ, so that the effect of large subtrees could be reduced. The second issue is solved by changing C function in (7) to ignore the order of two nodes.

With a decay factor λ_{tree} and a threshold Δ, the recursive rules of the parse tree kernel is modified as follows.

(i) If n_1 and n_2 are different,

$$C(n_1, n_2) = 0. \quad (8)$$

(ii) If both n_1 and n_2 are terminals or the current depth is equal to Δ,

$$C(n_1, n_2) = \lambda_{\text{tree}}. \quad (9)$$

Equation (7) cannot be used with these new recursive rules, since the number of child nodes can be different in n_1 and n_2. Thus, we adopt the maximum similarity between child nodes. As a result, the C function in (7) becomes

$$C(n_1, n_2) = \lambda_{\text{tree}} \prod_i^{\text{nc}(n_1)} \left(1 + \max_{\text{ch} \in \text{ch}_{n_2}} C(\text{ch}_i(n_1), \text{ch})\right), \quad (10)$$

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

The parse tree kernel with the modified C function, K_{mpt}, does not satisfy Mercer's condition. However, many functions that do not satisfy Mercer's condition [30, 31] work well in computing similarity [32]. Finally, this parse tree kernel is used as the similarity measure sim( ) in (1) for syntactic structural comparison of source codes.

5. Similarity Measure for Source Codes Based on Graph Kernel

5.1. Source Code as a Graph. Recently, program source codes are written with object-oriented concepts and several refactoring techniques, so that the codes are getting more and more modularized at functional level. Since a source code encodes program logic to solve a problem, the execution flow at function level is one of the important factors to identify the source code. Therefore, this function-level flow should be considered to compare source codes.

One possible representation for the function-level flow is a function-call graph which represents dependencies among functions within a program. Let s be a source code. Then, a *function calls graph* $G_s = (V_s, E_s)$ is a directed graph extracted from s, where $v \in V_s$ is a function in s. Thus, $|V_s|$ is the number of functions in s. E is a set of edges, and each edge represents a dependency relation between functions. That is, an edge $e_{ij} \in E_s$ that connects nodes v_i and v_j implies that a function v_i calls another function v_j. The weight of an edge is given by

$$w_{ij} = \begin{cases} 1 & \text{if } v_i \text{ calls } v_j \\ 0 & \text{otherwise.} \end{cases} \quad (11)$$

Figure 3 illustrates an example function-call graph extracted from the Java code in Box 1 This code contains five functions including `main`. First, `main` calls `rFibonacci` and `iFibonacci` in order. Since `rFibonacci` is a recursive function, it calls itself. `iFibonacci` calls two functions, `initOne` and `sum`, to initialize a variable and get a sum. Finally, `main` calls `println` to print out the results.

A rule-based approach is adopted to extract a call graph from a source code. Simple rules are used to find the caller-callee relationship from a parse tree. For instance, in Java, a rule "**if** '`expression (expresionList)`' is found, **then** `expression` is a called function name" is used to find subtrees from a parse tree. Then, function names and

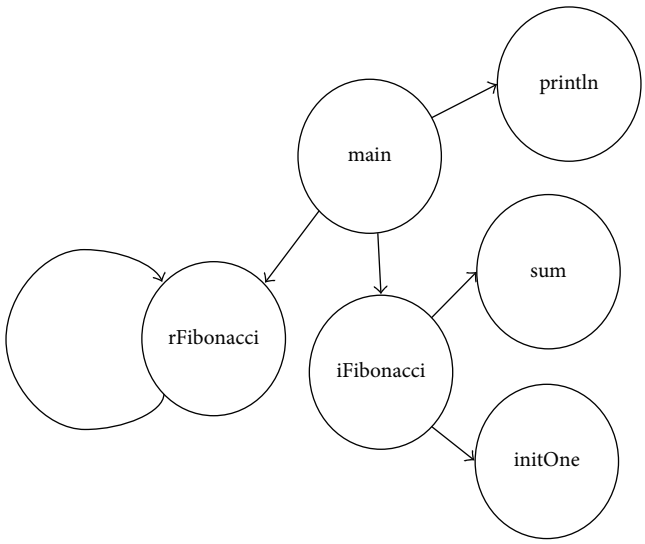

Figure 3: Example of extracted call graph from Java code.

parameters are extracted from the matched subtrees. The nodes for the extracted function names are connected to the caller node.

Function-call graph of a program depicts how the program executes at function level and how functions are related to one another. Since the flow of a program is quite unique according to the task, the similarity between two sources can be calculated using the flows of the programs. Since this flow is represented as a function-call graph, the graph kernel is the best method to compare function-call graphs. It showed good performance in several fields including biology and social network analysis.

5.2. Graph Kernel. Graph kernel is a kernel that is designed to compare graph structures. Like the parse tree kernel, a graph is mapped onto a feature space spanned by their subgraphs in the graph kernel. The intuitive feature of the graph kernel is graph isomorphism that determines the topological identity. According to Gärtner et al. [10], however, it is as hard as deciding whether two graphs are isomorphic to compute any complete graph kernel with an injective mapping function for all graphs, where graph isomorphism is a NP-complete problem [33]. Thus, most graph kernels focus on alternative feature representation of graphs.

The random walk graph kernel is one of the most widely used graph kernels. It uses all possible random walks as features for graphs. Let S be a set of all possible random walks and $\mathscr{W}_n(G)$ denotes the set of all possible walks with n edges in graph G. For each random walk $s \in S$ whose length is n, the corresponding feature mapping function of a graph G is given as

$$\mathbf{\Phi}_s(G) = \sqrt{\lambda_{\text{graph}}^n}\left|\{w \in \mathscr{W}_n(G), \forall i : l(s_i) = l(w_i)\}\right|, \quad (12)$$

where λ_{graph}^n is a weight for the length n, and $l(s_i)$ and $l(w_i)$ are the ith label of the random walk s and w, respectively. The kernel function between two graphs G_1 and G_2, denoted by $K_{\text{graph}}(G_1, G_2)$, can be defined as

$$K_{\text{graph}}(G_1, G_2) = \sum_s \mathbf{\Phi}_s(G_1) \cdot \mathbf{\Phi}_s(G_2). \quad (13)$$

Gärtner et al. proposed an approach to calculate all random walks within two graphs without explicit enumeration of all random walks [10]. A direct product graph of two graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$, denoted by $G_1 \times G_2 = (V_\times, E_\times)$ where $V_\times$ is its node set and $E_\times$ is its edge set, is defined as follows:

$$\begin{aligned} V_\times(G_1 \times G_2) &= \{(v_1, w_1) \in V_1 \times V_2 \mid l(v_1) = l(w_1)\}, \\ E_\times(G_1 \times G_2) &= \{((v_1, w_1), (v_2, w_2)) \in V_\times(G_1 \times G_2) \mid \\ &\qquad (v_1, v_2) \in E_1,\ (w_1, w_2) \in E_2, \\ &\qquad l(v_1, v_2) = l(w_1, w_2)\}, \end{aligned} \quad (14)$$

where $l(v)$ is the label of a node v and $l(a, b)$ is the label of an edge between node a and node b. Based on the direct product graph, the random walk kernel can be calculated. Let $A_\times \in \mathbb{R}^{|V_\times| \times |V_\times|}$ denote an adjacency matrix of the direct product $G_1 \times G_2$. With a weighting factor $\lambda_{\text{graph}} \geq 0$, K_{graph} in (13) can be rewritten as

$$K_{\text{graph}}(G_1, G_2) = \sum_{i,j=1}^{|V_\times|} \left[\sum_{n=0}^{\infty} \lambda_{\text{graph}}^n A_\times^n\right]_{ij}. \quad (15)$$

This random walk kernel can be computed in $O(n^3)$ using Sylvester equation or Conjugate Gradient method where n is the number of nodes [34].

5.3. Modified Graph Kernel. When a graph kernel is used to compare source codes, good performance is not expected due to the fact that the graph kernel measures similarities between walks with an identical label. Since the labels (function names) of nodes within the function-call graph are decided by human developers, they are seldom identical even if the source codes are simple. Therefore, the graph kernel has to consider nonidentical labels.

Borgwardt et al. modified the random walk kernel to compare nonidentical labels by changing the direct product graph to include all pairs of nodes and edges [35]. Assume that nodes are compared by a node kernel K_{node} and edges are compared by an edge kernel kedge K_{edge}. That is, $K_{\text{node}}(v, w)$ calculates the similarity between two labels from the nodes v and w, and $K_{\text{edge}}((v_i, v_{i+1}), (w_i, w_{i+1}))$ computes the similarity between two edges (v_i, v_{i+1}) and (w_i, w_{i+1}). With these two kernels, the random walk kernel between two function call graphs G_1 and G_2 is now defined as

$$K_{\text{mg}}(G_1, G_2) = \sum_{i=1}^{n-1} K_{\text{step}}((v_i, v_{i+1}), (w_i, w_{i+1})), \quad (16)$$

where

$$K_{\text{step}}\left(\left(v_i, v_{i+1}\right), \left(w_i, w_{i+1}\right)\right) = K_{\text{node}}\left(v_i, w_i\right) \cdot K_{\text{node}}\left(v_{i+1}, w_{i+1}\right) \cdot K_{\text{edge}}\left(\left(v_i, v_{i+1}\right), \left(w_i, w_{i+1}\right)\right). \quad (17)$$

If this modified random walk kernel is used for the comparison of source code, the node kernel K_{node} and the edge kernel K_{edge} should be defined. Note that the labels of edges in function-call graph are binary values by (11). Thus, K_{edge} is simply designed to compare binary values. The simplest form for $K_{\text{node}}(v, w)$ is a function that returns 1 when v and w have similar string patterns, 0 otherwise. That is, it returns 1 if a distance between v and w is smaller than a predefined threshold. In this paper, we simply use Levenshtein distance as the distance and set the threshold as 0.5.

The modified random walk kernel K_{mg} can be also computed using (15). However, the adjacency matrix $A_\times$ of the direct product $G_1 \times G_2$ should be modified as

$$\left[A_\times\right]_{(v_i,w_i),(v_j,w_j)} = \begin{cases} K_{\text{step}}\left(\left(v_i, v_{i+1}\right), \left(w_i, w_{i+1}\right)\right), & \\ \quad \text{if } \left(\left(v_i, v_{i+1}\right), \left(w_i, w_{i+1}\right)\right) \in E_\times & \\ 0 \quad \text{otherwise,} & \end{cases} \quad (18)$$

where $E_\times$ is a short form of $E_\times(G_1 \times G_2)$, and the edges (v_i, v_{i+1}) and (w_i, w_{i+1}) belong to E_1 and E_2, respectively. As in parse tree kernel, this modified graph kernel is used to compare source codes as similarity measure sim( ) in (1).

6. Similarity Measure for Source Codes Based on a Composite Kernel

The modified parse tree kernel manages syntactic structural information, whereas the modified graph kernel considers high-level topological information of source codes. In order to make use of both kinds of information, the composition of the two kernels is required. Cristianini and Shawe-Taylor proved that a new kernel can be obtained by combining existing several kernels with some closure properties such as weighted sum and multiplication [36]. Among various closure properties, this paper adopts the weighted sum, since it is simple and widely used.

Before combining two kernels, the kernels should be normalized since the modified parse tree kernel K_{mpt} and the modified graph kernel K_{mg} are not bound. Therefore, one kernel can dominate other in their composition. In order to remove this effect, the kernels are first normalized. When a kernel $K(s, s')$ is given, its normalized one, $K'(s, s')$, is defined as

$$K'\left(s, s'\right) = \frac{K\left(s, s'\right)}{\sqrt{K\left(s, s\right) \cdot K\left(s', s'\right)}}. \quad (19)$$

Therefore, $K'(s, s')$ is bounded between 0 and 1. That is, $0 \leq K'(s, s') \leq 1$.

Our composite kernel K_{co} is composed of the normalized modified parse tree kernel K'_{mpt} and the normalized modified graph kernel K'_{mg}. That is, the composite kernel, K_{co}, for given two source codes s and s' is defined as

$$K_{\text{co}}\left(s, s'\right) = \left(1 - \gamma\right) \cdot K'_{\text{mpt}}\left(T_s, T_{s'}\right) + \gamma \cdot K'_{\text{mg}}\left(G_s, G_{s'}\right), \quad (20)$$

where γ is a mixing weight between two kernels. T_s and $T_{s'}$ are parse trees extracted from source codes s and s', respectively, and G_s and $G_{s'}$ are call graphs from s and s', respectively. The larger γ is, the more significant K_{mpt} is. On the other hand, as the value of γ gets small, the graph kernel, K_{mg}, is more significant than the parse tree kernel, K_{mpt}. This composite kernel is used as our final similarity measure sim( ) in (1).

The parse tree kernel compares source codes with *local-level* view since it is based on subtree comparison. Most plagiarized source codes change a small portion of the original source code. Thus, the parse tree kernel has shown good performances in general. However, it does not reflect the flow of the program, which is dynamic structural information. The graph kernel, on the other hand, calculates the similarity in terms of dynamic *high-level* view. Thus, when source codes consist of a number of functions, the graph kernel achieves reasonable performance. As a result, γ should be determined by the complexity of source codes, since it is a parameter to control the relative importance between the parse tree kernel and the graph kernel.

There are many methods that measure the complexity of a source code. One widely-used method is the cyclomatic complexity proposed by McCabe [37]. The cyclomatic complexity is a graph-theoretic quantitative metric and measures the number of paths within a source code. It is simply calculated using a control flow graph of a source code where the nodes of the graph correspond to entities of the source code and an (directed) edge between two nodes implies a dependency relation between entities. Given the control flow graph of a source code, $M(s)$, the cyclomatic complexity of source code s, is defined as

$$M\left(s\right) = E - N + 2P, \quad (21)$$

where E is the number of edges of the graph, N is the number of nodes, and P is the number of connected components. The larger M is, the more complicated source code is.

In this paper, we measure the complexity of a source code using its function-call graph. Since a function-call graph represents dependencies among functions within a program, it can be considered as a kind of control flow graphs where entities of the source code are the function in the source code and the edge implies dependencies between functions.

Let $M(s)$ and $M(s')$ be the cyclomatic complexities of two source codes s and s', respectively. Since γ is the weight of two normalized kernels, it has to normalize between 0 and 1. The sigmoid function is defined for all real input values and returns a positive value between 0 and 1. Thus, the sigmoid function is adopted for γ of (20), and γ is defined as

$$\gamma = \frac{1}{1 + e^{-(\min(M(s), M(s')) - 25)}}, \quad (22)$$

Table 1: Simple statistics on the real data set.

Information	Value
Number of total assignments	36
Number of submitted source codes	555
Average number of submitted codes per assignment	15.42
Minimum number of lines in source code	49
Maximum number of lines in source code	2,863
Average number of lines per source code	305.07
Minimum number of nodes in source code	12
Maximum number of nodes in source code	447
Average number of nodes in source code	64.29
Number of marked plagiarism pairs	175

where $\min(a, b)$ returns the minimum value between a and b. According to (22), as the cyclomatic complexity gets larger, γ also increases. γ is to be 0.5 when the cyclomatic complexity of source code is 25. This indicates that when the cyclomatic complexity of source code is 25, the parse tree kernel and the graph kernel have an equal importance in the composite kernel. A number of source code analysis applications regard source codes whose cyclomatic complexity is more than 25 as complicated codes (http://msdn.microsoft.com/en-us/library/ms182212.aspx). Thus, we set 25 as the equal point of the importance between the parse tree kernel and the graph kernel.

7. Experiments

7.1. Experimental Settings. For experiments, the same data set in the work of Son et al. [5] is used. This data set is collected from actual programming assignments of Java classes submitted by undergraduate students from 2005 to 2009. Table 1 shows simple statistics of the data set. The total number of programming assignments is 36 and the number of submitted source codes is 555 for the 36 assignments. Thus, the average number of source codes per an assignment is 15.42.

Figure 4 shows the histogram of the source codes per lines. The x-axis is the number of program lines, and the y-axis represents the number of source codes. As shown in this figure, about 75% of source codes are written with less than 400 lines. The minimum number of lines of a source code is 49 and the maximum is 2,863. The average number of lines per code is 305.07.

In our data set, the minimum number of functions within a program is 12, whereas the maximum number is 447. The programs with larger number of programs are paint programs with any buttons. In the paint programs, students are required to set a layout manually with raw functions such as `setBounds`. Thus, paint programs have a number of functions. The average number of functions is 64.27.

Two annotators created the gold standard for this data set. They investigated all source codes and marked plagiarized pairs manually. In order to measure the reliability and validity of the annotators, Cohen's kappa agreement [38] is measured. The kappa agreement of the annotators is $\kappa = 0.93$, which falls

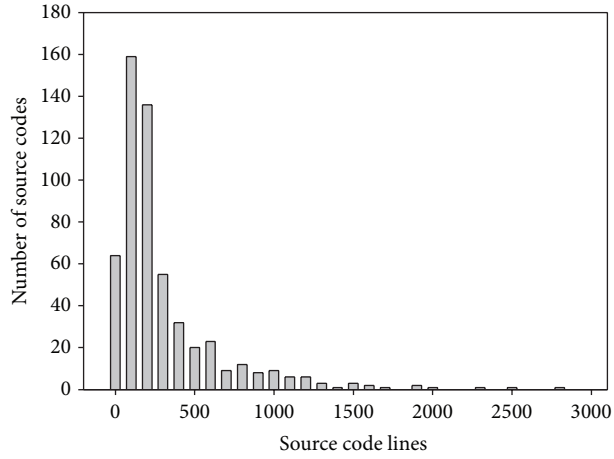

Figure 4: Histogram of source lines per code.

on the category of "almost perfect agreement." Only the pairs judged as plagiarized pairs by both annotators are regarded as actual plagiarized pairs. In total, 175 pairs are marked as plagiarized pairs.

Three metrics are used as evaluation measure: precision, recall, and $F1$-measure. They are calculated as follows:

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

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

$$F1\text{-measure} = 2 \cdot \frac{\text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}}. \tag{23}$$

In order to evaluate the proposed method, several baseline systems are used. It is compared with JPlag and CCFinder. In all experiments, for the parse tree kernel, the threshold for subtree depth Δ is set as 3 and the decay factor λ_{tree} is 0.1. The decay factor λ_{graph} for graph kernel is set to be 0.1 empirically. P in (21) is set to be 1 because each source code in our data set is a single program.

7.2. Experimental Results. Before evaluating the performance of plagiarism detection, we first examine relatedness between the number of source code lines and the cyclomatic complexity. This examination tries to show that (22) is feasible. Since γ is determined with cyclomatic complexity, it is expected to be proportional to cyclomatic complexity. Figure 5 shows scatter plot between the number of lines and cyclomatic complexity. As shown in this figure, they are highly correlated with each other in our data set. The Pearson correlation coefficient is 0.714. This result implies that it is feasible to set γ in (22) to be proportional to cyclomatic complexity.

In order to see the effect of threshold θ in (1) in our method, the performances are measured according to the values of θ. Figure 6 shows the performance of the proposed method for various θ. As θ increases, the precision

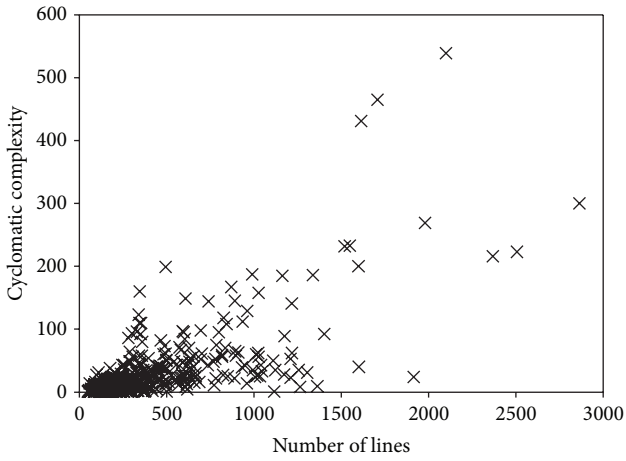

Figure 5: Scatter plot according to the number of lines of source codes and the corresponding cyclomatic compelxity.

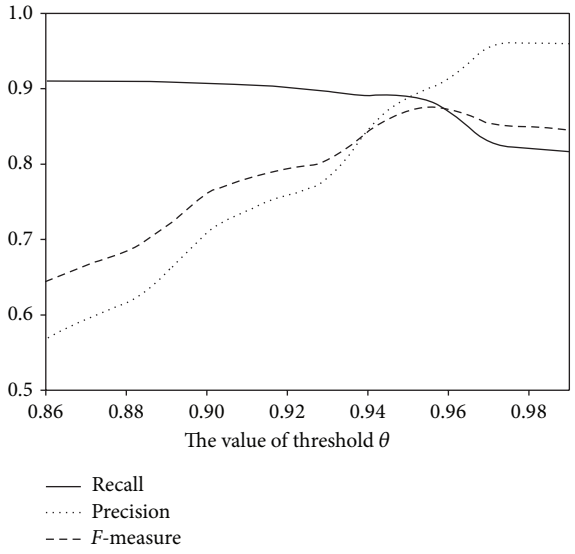

Figure 6: Performance of the proposed system for real-world data set.

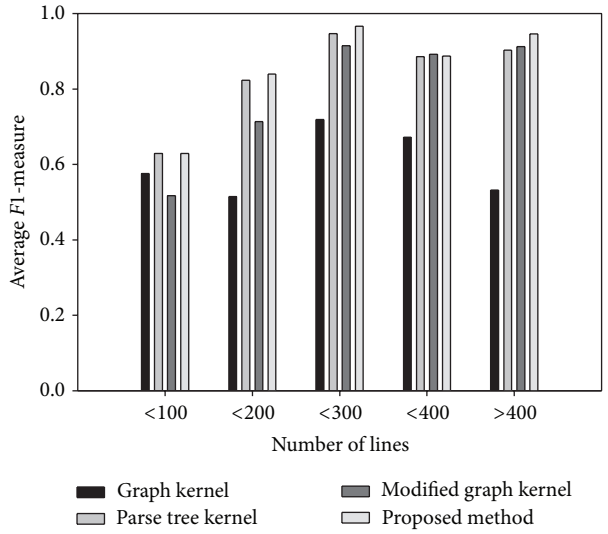

Figure 7: Average $F1$-measure according to the number of source code lines.

also increases, while the recall decreases slightly. The best performance is achieved at $\theta = 0.96$ with 0.87 of $F1$-measure. Thus, $\theta = 0.96$ is used at all the experiments below.

Figure 7 compares the proposed method with various kernels according to the number of source code lines. In this figure, the x-axis is the number lines of source code, and the y-axis represent the average $F1$-measure. As shown in this figure, the original graph kernel shows the worst performance. Since it uses only the graph structure of function calls, it often fails in calculating the similarity among source codes. For example, assume that there are two source codes. In one source code, `main` calls a function `add`, and `add` calls another function `multiply`. In the other source code, `main` calls `multiply`, and `multiply` calls `add`. These two source codes are the same under the graph kernel, since the label information is ignored by the graph kernel. Without the labels, these two graphs are identical. On the other hand, the modified graph kernel utilizes the label information. As a result, it achieves better performance than the graph kernel.

The parse tree kernel achieves higher performance than other methods for the source codes with less than 300 lines. When the number of lines in source codes is small, the plagiarized codes are often made by changing the original one locally. Thus, the parse tree kernel detects plagiarized pairs accurately for the codes with small number of lines. When source codes have more than 300 lines, the modified graph kernel shows slightly better performance than the parse tree kernel. This result implies that high-level structural information is another factor to compare (large) source codes and the modified graph kernel can reflect this structural information well.

The proposed method that combines the parse tree kernel and the modified graph kernel achieves the best performance for all source codes except those with 300 ~ 400 lines. Since the cyclomatic complexity of source codes with 300 ~ 400 lines is near 25, the proposed method reflects the parse tree kernel and the modified graph kernel equally. Thus, it achieves an average performance of the kernels. By the cyclomatic complexity of source codes, the proposed method is more influenced by the parse tree kernel when a source code is small. If a source code is large, the effect of graph kernel is larger than that of the parse tree kernel. From the results, it can be concluded that the proposed method does not consider only local-level structural information, but also high-level structural information effectively.

The final $F1$-measure of program source code plagiarism detection is given in Table 2. The proposed method shows the best $F1$-measure compared to other kernels or open source plagiarism systems. The difference of $F1$-measure is 0.29 against JPlag, 0.17 against CCFinder, 0.08 against the modified graph, and 0.05 against the modified parse tree kernel. This result implies that for source code plagiarism

Table 2: Final $F1$-measure of plagiarism detection.

Method	$F1$-measure
JPlag	0.56
CCFinder	0.70
Modified parse tree kernel ($\theta = 0.95$) [5]	0.84
Graph kernel ($\theta = 0.99$)	0.45
Modified graph kernel ($\theta = 0.97$)	0.82
Proposed method ($\theta = 0.96$)	**0.87**

detection, the similarity measure sim in (1) should consider not only the syntactic structural information but also the dynamic call structure simultaneously.

8. Conclusion

In this paper, we have proposed a novel method for program source code comparison. The proposed method calculates the similarity between two source codes with the composition of two kinds of structural information extracted from the source codes. That is, the method uses both syntactic information and dynamic information. The syntactic information which provides local-level structural view is included in the parse tree. In order to compare the parse trees, this paper adopts a specialized tree kernel for parse trees of source codes. The dynamic information, which is contained in the function-call graph, gives high and global level structural view. The graph kernel with the consideration function names is adopted to reflect the graph structure. Finally, the proposed method uses a composite kernel of the kernels to use both kinds of information. In addition, the weights of the kernels in the composite kernel are automatically determined with the cyclomatic complexity.

In the experiments of Java program source code plagiarism detection with real data set, it is shown that the proposed method outperformed existing methods in detecting plagiarized pairs. In particular, the experiments with the various number of lines show that the proposed method always works well regardless of the size of source codes.

One advantage of the proposed method is that it can be used with other languages such as C, C++, and Python even if the experiments were only conducted with Java. Since the proposed method requires only parse trees and function-call graphs of source codes, it can be applied to any other languages if a parser for the languages is available. All kinds of information of the proposed method are available at http://ml.knu.ac.kr/plagiarism.

Conflict of Interests

The authors declare that there is no conflict of interests regarding the publication of this paper.

Acknowledgments

This study was supported by the BK21 Plus project (SW Human Resource Development Program for Supporting Smart Life) funded by the Ministry of Education, School of Computer Science and Engineering, Kyungpook National University, Korea (21A20131600005), and by ICT R&D program of MSIP/IITP (10044494, WiseKB: Big data based self-evolving knowledge base and reasoning platform).

References

[1] J.-W. Son and S.-B. Park, "Web table discrimination with composition of rich structural and content information," *Applied Soft Computing*, vol. 13, no. 1, pp. 47–57, 2013.

[2] D. L. McCabe, "Cheating among college and university students: a north American perspective," *International Journal for Educational Integrity*, vol. 1, no. 1, pp. 1–11, 2005.

[3] S. Horwitz, "Identifying the semantic and textual differences between two versions of a program," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 234–245, 1990.

[4] W. Yang, "Identifying syntactic differences between two programs," *Software: Practice and Experience*, vol. 21, no. 7, pp. 739–755, 1991.

[5] J.-W. Son, T.-G. Noh, H.-J. Song, and S.-B. Park, "An application for plagiarized source code detection based on a parse tree kernel," *Engineering Applications of Artificial Intelligence*, vol. 26, no. 8, pp. 1911–1918, 2013.

[6] M. L. Kammer, *Plagiarism detection in haskell programs using call graph matching [M.S. thesis]*, Utrecht University, 2011.

[7] D. Haussler, "Convolution kernels on discrete structures," Tech. Rep. UCS-CRL-99-10, University of California, Santa Cruz, Calif, USA, 1999.

[8] B. Schölkopf, K. Tsuda, and J.-P. Vert, *Kernel Methods in Computational Biology*, MIT Press, 2004.

[9] M. Collins and N. Duffy, "Convolution kernels for natural language," in *Advances in Neural Information Processing Systems*, pp. 625–632, 2001.

[10] T. Gärtner, P. Flach, and S. Wrobel, "On graph kernels: hardness results and efficient alternatives," in *Proceedings of the 16th Annual Conference on Learning Theory*, pp. 129–143, August 2003.

[11] D. Hindle, "Noun classification from predicate-argument structures," in *Proceedings of the 28th Annual Meeting on Association for Computational Linguistics (ACL '90)*, pp. 268–275, Stroudsburg, Pa, USA, June 1990.

[12] P. Resnik, "Using information content to evaluate semantic similarity in a taxonomy," in *Proceedings of the 13th International Joint Conference on Artificial Intelligence*, pp. 448–453, 1995.

[13] B. Gipp, N. Meuschke, and C. Breitinger, "Citation-based plagiarism detection: practicability on a large-scale scientific corpus," *Journal of the Association for Information Science and Technology*, vol. 65, no. 8, pp. 1527–1540, 2014.

[14] G. Varelas, E. Voutsakis, P. Raftopoulou, E. G. Petrakis, and E. E. Milios, "Semantic similarity methods in wordnet and their application to information retrieval on the web," in *Proceedings of the 7th Annual ACM International Workshop on Web Information and Data Management*, pp. 10–16, 2005.

[15] K. Williams, H.-H. Chen, and C. L. Giles, "Classifying and ranking search engine results as potential sources of plagiarism," in *Proceedings of the ACM Symposium on Document Engineering*, pp. 97–106, Fort Collins, Colo, USA, September 2014.

[16] R. A. Jarvis and E. A. Patrick, "Clustering using a similarity measure based on shared near neighbors," *IEEE Transactions on Computers*, vol. 22, no. 11, pp. 1025–1034, 1973.

[17] K. J. Ottenstein, "An algorithmic approach to the detection and prevention of plagiarism," *ACM SIGCSE Bulletin*, vol. 8, no. 4, pp. 30–41, 1976.

[18] M. Halstead, *Elements of Software Science*, Elsevier, 1977.

[19] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier, "Clone detection using abstract syntax trees," in *Proceedings of the IEEE International Conference on Software Maintenance (ICSM '98)*, pp. 368–377, November 1998.

[20] J. Ferrante, K. J. Ottenstein, and J. D. Warren, "The program dependence graph and its use in optimization," *ACM Transactions on Programming Languages and Systems*, vol. 9, no. 3, pp. 319–349, 1987.

[21] C. Liu, C. Chen, J. Han, and P. S. Yu, "Gplag: detection of software plagiarism by program dependence graph analysis," in *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pp. 872–881, 2006.

[22] H.-I. Lim, H. Park, S. Choi, and T. Han, "A method for detecting the theft of Java programs through analysis of the control flow information," *Information and Software Technology*, vol. 51, no. 9, pp. 1338–1350, 2009.

[23] D.-K. Chae, J. Ha, S.-W. Kim, B. J. Kang, and E. G. Im, "Software plagiarism detection: a graph-based approach," in *Proceedings of the 22nd ACM International Conference on Information & Knowledge Management (CIKM '13)*, pp. 1577–1580, Burlingame, Calif, USA, November 2013.

[24] E. Stamatatos, "Plagiarism detection using stopword n-grams," *Journal of the American Society for Information Science and Technology*, vol. 62, no. 12, pp. 2512–2527, 2011.

[25] G. Cosma and M. Joy, "An approach to source-code plagiarism detection and investigation using latent semantic analysis," *IEEE Transactions on Computers*, vol. 61, no. 3, pp. 379–394, 2012.

[26] 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, 2002.

[27] A. Aiken, "Moss: a system for detecting software plagiarism," 1998, http://theory.stanford.edu/~aiken/moss/.

[28] T. Kamiya, S. Kusumoto, and K. Inoue, "CCFinder: a multilinguistic token-based code clone detection system for large scale source code," *IEEE Transactions on Software Engineering*, vol. 28, no. 7, pp. 654–670, 2002.

[29] T. J. Parr and R. W. Quong, "ANTLR: a predicated-LL(k) parser generator," *Software: Practice and Experience*, vol. 25, no. 7, pp. 789–810, 1995.

[30] V. N. Vapnik, *The Nature of Statistical Learning Theory*, Springer, New York, NY, USA, 1995.

[31] R. Courant and D. Hilbert, *Methods of Mathematical Physics*, Interscience, New York, NY, USA, 1953.

[32] A. Moschitti and F. M. Zanzotto, "Fast and effective kernels for relational learning from texts," in *Proceedings of the 24th International Conference on Machine Learning (ICML '07)*, pp. 649–656, Corvallis, Ore, USA, June 2007.

[33] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W. H. Freeman, 1990.

[34] S. V. N. Vishwanathan, N. N. Schraudolph, R. Kondor, and K. M. Borgwardt, "Graph kernels," *Journal of Machine Learning Research*, vol. 11, pp. 1201–1242, 2010.

[35] K. M. Borgwardt, C. S. Ong, S. Schönauer, S. V. N. Vishwanathan, A. J. Smola, and H.-P. Kriegel, "Protein function prediction via graph kernels," *Bioinformatics*, vol. 21, supplement 1, pp. i47–i56, 2005.

[36] N. Cristianini and J. Shawe-Taylor, *An Introduction to Support Vector Machines*, Cambridge University Press, Cambridge,UK, 2000.

[37] T. J. McCabe, "A complexity measure," *IEEE Transactions on Software Engineering*, vol. 2, no. 4, pp. 308–320, 1976.

[38] J. Carletta, "Assessing agreement on classification tasks: the kappa statistic," *Computational Linguistics*, vol. 22, no. 2, pp. 249–254, 1996.