Hindawi
Scientific Programming
Volume 2017, Article ID 7809047, 8 pages
https://doi.org/10.1155/2017/7809047

Research Article

WASTK: A Weighted Abstract Syntax Tree Kernel Method for Source Code Plagiarism Detection

Deqiang Fu,[1,2] Yanyan Xu,[1] Haoran Yu,[2] and Boyang Yang[2]

[1] *School of Information Science and Technology, Beijing Forestry University, No. 35 Qinghuadong Road, Haidian District, Beijing 100083, China*
[2] *Jisuan Institute of Technology, Beijing Judao Youda Network Technology Co. Ltd., No. 18 Suzhoujie St., Room 1204, Haidian District, Beijing 100080, China*

Correspondence should be addressed to Yanyan Xu; xuyanyan@bjfu.edu.cn

Received 28 September 2016; Revised 23 November 2016; Accepted 21 December 2016; Published 13 February 2017

Academic Editor: Michele Risi

In this paper, we introduce a source code plagiarism detection method, named WASTK (Weighted Abstract Syntax Tree Kernel), for computer science education. Different from other plagiarism detection methods, WASTK takes some aspects other than the similarity between programs into account. WASTK firstly transfers the source code of a program to an abstract syntax tree and then gets the similarity by calculating the tree kernel of two abstract syntax trees. To avoid misjudgment caused by trivial code snippets or frameworks given by instructors, an idea similar to TF-IDF (Term Frequency-Inverse Document Frequency) in the field of information retrieval is applied. Each node in an abstract syntax tree is assigned a weight by TF-IDF. WASTK is evaluated on different datasets and, as a result, performs much better than other popular methods like Sim and JPlag.

1. Introduction

Due to the advancement of the Internet, source code plagiarism has become a big issue in the field of computer science education [1]. Some students usually try to copy source code from their classmates or search similar source code from the Internet as their assignments without modifying. Plagiarism diminishes the quality of education seriously. Students are deprived of the abilities to make innovations and think independently, which may also cause academic dishonesty.

As what happened in traditional offline computer science education, online education suffers from plagiarism, too. Moreover, it is even harder for online education platforms to find a method against source code plagiarism when the number of submissions from students goes much greater than the traditional offline cases. Therefore, detecting source code plagiarism becomes heavy load of responsible instructors' daily work [2].

In order to detect the source code plagiarism automatically, three problems have already been considered by researchers in this field [3–5].

Problem 1. Computer programs are structured and hierarchical. Looking for a reasonable method to measure the similarity between a pair of programs needs to be carefully treated.

Problem 2. The modifications on programs for the plagiarism purpose are almost changeless. Common distortions, for example, comment alteration, whitespace padding, identifier renaming, code reordering, and algebraic expressions, can be found with high confidence.

Problem 3. The comparison between each pair of programs takes a long time and leads to inefficiency.

Additionally, there exist some other facts which need to be considered.

Problem 4. Instructors may provide frameworks of programming assignments for students to start with. The provided frameworks will contribute a lot to the similarity between each pair of programs.

Problem 5. Solutions for simple problems may be alike. For example, a hundred students, without any communication, may produce very similar programs for the task of calculating the sum of two variables.

To solve these two problems, in this paper, two accurate methods for source code plagiarism detection are presented, named ASTK (Abstract Syntax Tree Kernel) and WASTK (Weighted ASTK). Since computer programs are structured, in this work, they are presented as abstract syntax trees [6]. In ASTK, a method called tree kernel [7] is used for measuring the similarity between a pair of programs. Additionally, different from traditional tree kernel methods, to highlight the significance of nontrivial parts and reduce impacts caused by short sample source code and source code provided by instructors, WASTK gives weights to every tree node. Inspired by the idea of TF-IDF [8], lower weights are given to the part of common code and code provided by the framework of coding assignments, and higher weights are assigned to those distinguished parts of code.

The rest of this paper is organized as follows. Section 2 introduces related previous work. Section 3 illustrates the methods of ASTK and WASTK. We show the experiment results and corresponding analyses in Section 4. Finally, in Section 5, we conclude this work and give some discussions about possible future work.

2. Related Work

There exist some plagiarism detection measures. According to the categories identified by Mozgovoy in 2006 [9], there are mainly fingerprint based and content comparison based approaches. Content comparison techniques have three subcategories including string-matching algorithms, parameterized matching algorithms, and parse tree comparison algorithms.

MOSS (Measure of Software Similarity), proposed by University of California-Berkeley in 2003 [10], as a proposed fingerprint based measure, divides each program into k-grams. Each gram is made of some substrings of length k. The possibility of plagiarism is determined according to the number of overlapped fingerprints generated by hashing each gram.

There are some famous string-matching algorithms, including Plague, YAP3 (the third version of Yet Another Plague), JPlag, and FDPS (Fast Plagiarism Detection System) [11]. These methods all firstly convert programs into corresponding token sequences. Then they use similarity as the evidence of plagiarism by comparison among token sequences generated from different programs. Plague, proposed by Whale in 1988, is the first one converting a file into a token sequence and using a string-matching technique for comparison [12]. YAP3, proposed by Wise in 1996, performs better than Plague due to a faster converting method using Running Karp-Rabin Greedy String Tiling (RKR-GST) [13]. JPlag, proposed by Malpohl in 2006, runs even faster than YAP3 by defining a minimum-matching length [14]. FDPS, as another algorithm similar to YAP3, pays more attentions on efficiency which is proposed by Mozgovoy et al. in 2007 [15]. It uses an indexed data structure to store matches which support faster searching.

Dup tool, proposed by Baker in 1995, is a parameterized matching algorithm. It firstly uses a lexical analyzer to scan a program. Then it modifies all identifiers and constants into same symbols and output the transformed program together with a list of parameter candidates. The similarity is determined according to p-matches between two transformed files, where p means a parameter [16].

Sim and Brass are based on parse tree comparison algorithms. Sim, proposed by Gitchell and Tran in 1999, gets a token sequence by a lexical analyzer and calculates the sequence alignment as the similarity [17]. In order not to be influenced by identifiers or statement orders, in 2014, Kikuchi et al. proposed a modified method which uses syntactic elements for tokens with lexical elements and the method does not include identifier names or literal values in the tokens [4]. Brass, proposed by Belkhouche et al. in 2004, is an application of parse tree comparison algorithms. It represents each file as a binary tree and a symbol table (the data dictionary), containing the variables and data structures used in the file [18].

3. Proposed Approach

Definition 1 (abstract syntax tree). An abstract syntax tree is an abstract syntactic structure of a piece of source code in the form of tree.

Figure 1 shows an example of an abstract syntax tree created from short sample code. The left part is the source code in two lines and the right part is its corresponding abstract syntax tree after lexical and syntax analysis. It is easy to find that a short piece of source code has a large abstract syntax tree and only leaf nodes show the symbols of source code. The in-order traversal result of all the leaf nodes will get the source code.

Definition 2 (tree kernel). Tree kernel is an algorithm for computing tree structures and measuring the similarity between two contents in Natural Language Processing (NLP), which is firstly proposed by Collins and Duffy in 2001 [7].

Between two trees T_1 and T_2, a kernel $K(T_1, T_2)$ can be represented as an inner product between two vectors:

$$K(T_1, T_2) = \mathbf{h}(T_1) \cdot \mathbf{h}(T_2). \quad (1)$$

Each tree T can be represented as a vector $\mathbf{h}(T) = (h_1(T), h_2(T), \dots, h_n(T))$, where $h_i(T)$ is the number of occurrences of the ith subtree in T.

Definition 3 (TF-IDF). In information retrieval, TF-IDF is a numerical statistic model that is intended to reflect how important a word means to a document in a collection of documents [8].

TF-IDF includes two parts: TF is the abbreviation for "Term Frequency" and IDF is the abbreviation for "Inverse Document Frequency." TF(t, d) denotes the frequency of word t in document d. Similarly, DF(t, D) is the frequency

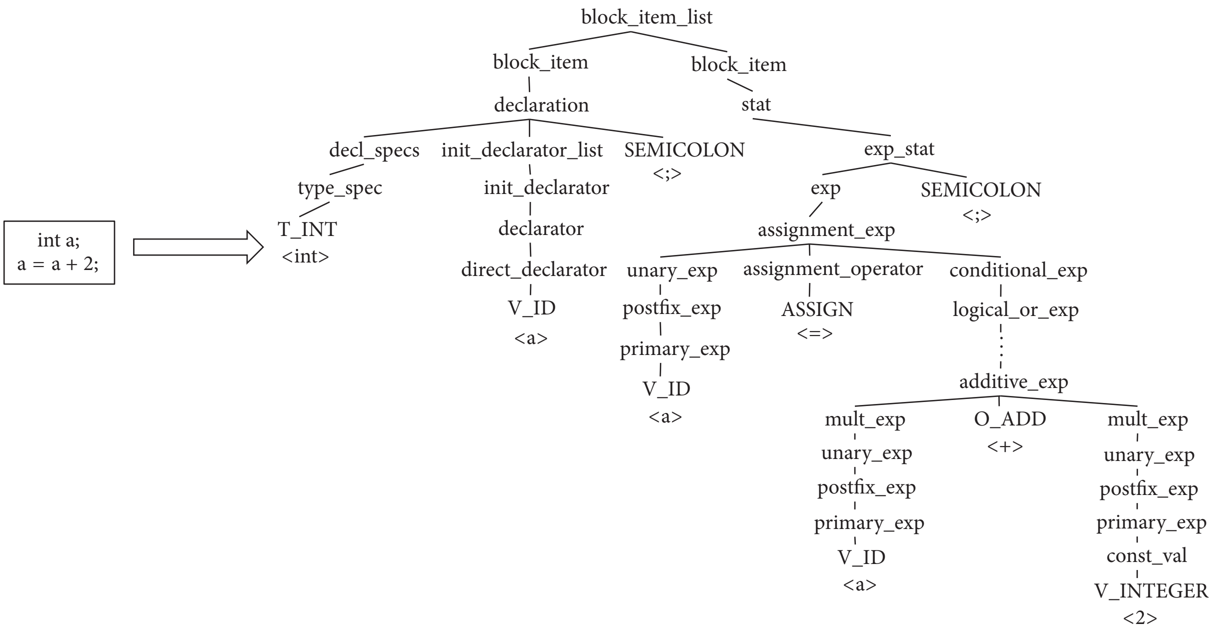


Figure 1: A piece of source code and its corresponding abstract syntax tree.

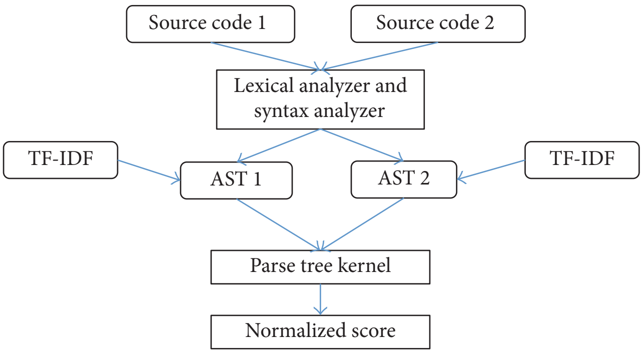


Figure 2: Weighted Abstract Syntax Tree Kernel.

of documents containing word t in the set of all documents D. IDF(t, D) is the reciprocal of DF(t, D). TF focuses on the importance of words in a document while IDF concerns the universality of words in all documents. TF-IDF can help to compute the importance of words for solving Problems 4 and 5.

The process of WASTK is shown in Figure 2. The proposed method firstly transforms programs into abstract syntax trees. Inspired by the idea of TF-IDF, weights are calculated for every subtree, giving expressions with high frequency in a document but low frequency in other documents higher weights and giving expressions that appear everywhere (in all documents) lower weights. Finally, tree kernel is applied with calculated weights on nodes to determine whether plagiarism happens. The details of the designed approach are shown in Figure 2.

3.1. Adjust the Structure of Abstract Syntax Tree. To solve Problem 1, programs are parsed into abstract syntax trees for further understanding the semantics.

Then some adjustments are applied to each abstract syntax tree. As mentioned in Problem 2, replacing variable names and size declarations of arrays are common ways to plagiarize. To solve this problem, all variable tokens are replaced with a unified token. The tokens for size declarations of arrays and the indices of array elements are unified as well.

Because rephrasing expressions into different expressions is not trivial, there is rarely a problem about plagiarism by modifying expressions. The structure information of expressions related to subtrees in the abstract syntax tree is abandoned. The resulting strings of the in-order traversal of leaf nodes on the subtrees are adopted as replacements of the subtrees. Besides, a simplified abstract syntax tree results in less time needed by running ASTK and WASTK, which helps to solve Problem 3.

After adjusting, the abstract syntax tree in Figure 1 is transformed into a new tree shown in Figure 3. All variable names are replaced with "temp." Because the node "exp" is a type of expression, it is adjusted to be a leaf node and represents the in-order traversal of leaf nodes on the previous subtree with the root "exp"; that is to say the dotted portion in Figure 3 is cut out from the original abstract syntax tree.

3.2. Determine Node Weights. In ASTK, the weights on all nodes are equal to 1. However, in WASTK, the weights of nodes depend on TF-IDF. Actually, this is the only difference between ASTK and WASTK. In WASTK, it is considered that weights on abstract syntax tree nodes reflect the significance of the corresponding subtree fragments of code. Taking Problems 4 and 5 into consideration, lower weights are given to the nodes that represent common expressions appearing in many other programs.

Different types of symbols and expressions frequently appear in the programs. They do not have abundant semantic

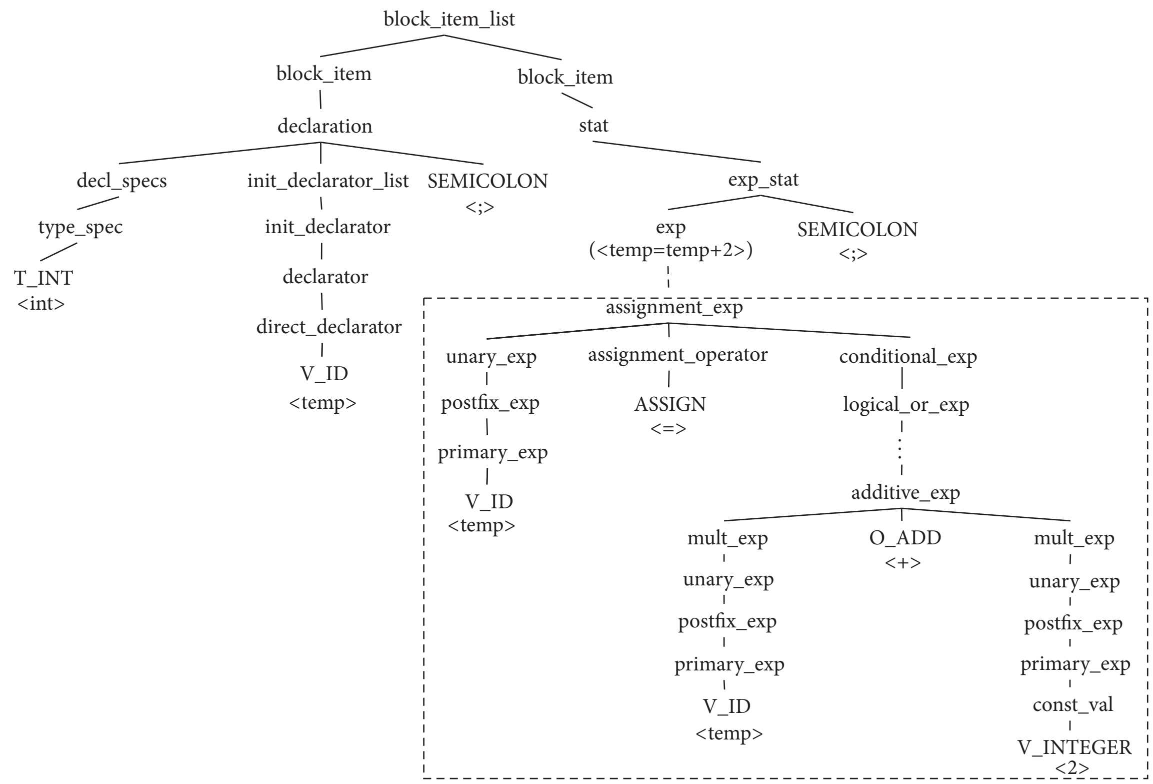


Figure 3: A new abstract syntax tree after adjustments.

Table 1: A list of stop words.

Type	Function	Example(s)
Round brackets		()
Square brackets		[]
Curly brackets		{}
Comma		,
Semicolon		;
Equality sign		=
Pointer star	The star denoting the pointer	*
Variable type	The type of a variable	int
Variable name	The name of a variable	a b temp
Pointer name	The name of a pointer	*a *b *temp
Size of array	The declaration of an array's size	100 N N+10
Definition of array	The definition of an array	a[100] a[N]

meaning and can be treated as stop words. A list of stop words is shown in Table 1.

Let T denote the abstract syntax tree. For each subtree s of T, there is a weight $w_{s,T}$. The in-order traversal on s is denoted as word_s. If word_s is treated as a stop word, the weight of s is specially adjusted to 0, that is, $w_{s,T} = 0$. For example, the weight of the nodes in Figure 3 is 0 whose type is "T_INT," "V_ID," and "SEMICOLON." On the contrary, if word_s is not a stop word, the weight $w_{s,T}$ can be computed as follows:

$$w_{s,T} = \text{TF}(s,T) \cdot \text{IDF}(s,T). \tag{2}$$

By Definition 3, $\text{TF}(s,T)$ and $\text{IDF}(s,T)$ can be calculated as follows:

$$\begin{aligned} \text{TF}(s,T) &= \frac{\text{cnt}(s,T)}{n(T)}, \\ \text{IDF}(s,T) &= \log_2\left(1 + \frac{N}{c(s)}\right), \end{aligned} \tag{3}$$

where $\text{cnt}(s,T)$ is the frequency of the appearances of subtree $s \in S_T$. $n(T)$ is the number of subtrees in T and $c(s)$ is the number of abstract syntax trees which contains s. N is used to represent the number of generated abstract syntax trees from programs related to a specific assignment.

3.3. Calculate Tree Kernel and Similarity. By Definition 2, the similarity between two abstract syntax trees T_1 and T_2 can be measured by a tree kernel and denoted by $K(T_1,T_2)$:

$$\begin{aligned} K(T_1,T_2) &= \mathbf{h}(T_1) \odot \mathbf{h}(T_2) \\ &= \sum_{s_i \in (S_{T_1} \cup S_{T_2})} \text{cnt}(s_i,T_1) \cdot \text{cnt}(s_i,T_2), \end{aligned} \tag{4}$$

where S_T denotes the set of all subtrees in T. When calculating $K(T_1,T_2)$ directly, it needs to enumerate each subtree in both T_1 and T_2 and then calculate $\text{cnt}(s_i,T_1)$ and $\text{cnt}(s_i,T_2)$, respectively. This method appears inefficient and a recursive method is applied:

$$K(T_1,T_2) = \sum_{s_1 \in S_{T_1}} \sum_{s_2 \in S_{T_2}} C(s_1,s_2). \tag{5}$$

$C(s_1, s_2)$ is calculated as follows.

(1) If the roots of s_1 and s_2 are both leaf nodes of the "exp" type, then

$$C(s_1, s_2) = \lambda_{\text{tree}} \cdot \text{dist}\left(\text{word}_{s_1}, \text{word}_{s_2}\right) \cdot w_{s_1,T_1} \cdot w_{s_2,T_2}, \quad (6)$$

where λ_{tree} is a decay factor to avoid the changes near the root producing too much influence [6]. As the height increases, the kernel value of the subtree is penalized by $(\lambda_{\text{tree}})^{\text{size}}$, where size is the height of the subtree. word_s is a string denoted the in-order traversal on the subtree s as defined in Section 3.2. $\text{dist}(a, b)$ is defined as follows:

$$\text{dist}(a, b) = \frac{\text{lev}_{a,b}}{\max(|a|, |b|)}, \quad (7)$$

where $\text{lev}_{a,b}$ is the edit distance between strings a and b. $|a|$ is the length of string a. Different from the traditional tree kernel, the similarity between expressions is not equal to 0. It is denoted by the edit distance between two expressions. By using the above definition of $\text{dist}(a, b)$, expression-level modifications that are mentioned in Problem 2 will be discovered by ASTK and WASTK.

(2) If the root of s_1 is different from the root of s_2, then

$$C(s_1, s_2) = 0. \quad (8)$$

(3) If the roots of s_1 and s_2 are both leaf nodes, then

$$C(s_1, s_2) = \lambda_{\text{tree}} \cdot w_{s_1,T_1} \cdot w_{s_2,T_2}. \quad (9)$$

(4) Otherwise,

$$C(s_1, s_2) = \lambda_{\text{tree}} \cdot \prod_{i}^{\text{ns}(s_1)} \left(1 + \max_{j}^{\text{ns}(s_2)} C(\text{st}(s_1, i), \text{st}(s_2, j))\right) \cdot w_{s_1,T_1} \cdot w_{s_2,T_2}, \quad (10)$$

where $\text{ns}(s)$ is the number of subtrees rooted at children nodes of root_s and $\text{st}(s, i)$ is the subtree rooted at the ith children node of root_s. Due to the root of s_1 being the same as the root of s_2, $\text{ns}(s_1) = \text{ns}(s_2)$.

After computing tree kernel $K(T_1, T_2)$ between T_1 and T_2, a normalization is needed. The method of normalization is as follows:

$$K'(T_1, T_2) = \frac{K(T_1, T_2)}{\sqrt{K(T_1, T_1) \cdot K(T_2, T_2)}}, \quad (11)$$

where $K'(T_1, T_2)$ is the cosine similarity of $\mathbf{h}(T_1)$ and $\mathbf{h}(T_2)$. In ASTK and WASTK, $K'(T_1, T_2)$ is the similarity of two pieces of source code.

Table 2: The sizes of datasets.

Fold #	Type of set	Number of code pairs
1	Training set	14810
1	Testing set	7404
2	Training set	14809
2	Testing set	7405
3	Training set	14809
3	Testing set	7405

4. Experimental Results

4.1. Datasets. There are two groups of data. The programs in each group are generated from 10 independent submissions of programs for the same assignment by applying to 40 different generators. Table 2 shows the sizes of datasets.

Threefold cross-validation has been adopted for the experiment. The dataset has been randomly split into 3 parts. Each time, one part is used as a testing set and the other two parts are used as a training set.

The 10 independent submissions of programs for each assignment are randomly picked from the submitted solution of "Problem I" and "Problem J" by students who attend the final exam of C programming course in Harbin University of Science and Technology. These pieces of code are very short as the average number of lines is no more than 20. Particularly, a programming framework is provided in "Problem J."

The 40 generators are designed by 40 players who attend the Jisuan-zhidao programming contest (https://nanti.jisuanke.com/contest). The generator reads original programs as the input and returns the plagiarized programs as results. Only the generated programs functionally equivalent to the original ones are used.

All valid generated programs are used in the datasets and labeled. The programs generated from the same program are labeled as the same origin. Any pair of programs that are labeled as the same origin will be treated as plagiarism.

4.2. Evaluation of Proposed Methods. After determining the similarity score of a pair of programs, it still needs to evaluate whether these two programs are plagiarized or not. Therefore, a threshold θ of the similarity score is suggested. The threshold θ is determined by setting it to different values in $T = \{0.01t \mid 0 < t < 100\}$ and using a training set to find which value generates the best outcome. The pair with the similarity score higher than θ will be treated as a plagiarism.

In this study, precision is a measurement method described as the number of true plagiarized programs over the number of all programs that are marked as plagiarism. Recall is another measurement method described as the number of all found plagiarized programs over the number of all plagiarized programs.

Since we have the assumption that the positive case of plagiarism is rare and the negative case of plagiarism is common, in addition to precision and recall, Jaccard score is picked as a reasonable method for evaluation.

Table 3: The thresholds θ of WASTK, ASTK, Sim, and JPlag.

Tool	1st-fold	2nd-fold	3rd-fold	Average
WASTK	0.77	0.77	0.77	0.77
ASTK	0.96	0.96	0.96	0.96
Sim	0.12	0.15	0.15	0.14
JPlag	0.11	0.10	0.08	0.10

Jaccard score is calculated as follows:

$$\text{Jaccard score} = \frac{\text{TP}}{\text{TP} + \text{FP} + \text{FN}}, \qquad (12)$$

where TP is the number of plagiarized programs that are marked as plagiarism. FP is the number of nonplagiarized programs that are marked as plagiarism. FN is the number of plagiarized programs that are not marked as plagiarism.

4.3. Results. Our experiments select Sim and JPlag to compare with ASTK and WASTK. In a survey, Deokate and Hanchate mentioned four popular plagiarism detection tools: JPlag, Sim, MOSS, and Plaggie in 2016 [19]. However, MOSS is running on the web server, and the results contain no more than 250 pairs of code in high similarity. Plaggie is similar to JPlag but it only supports Java [20]. As a result, we select Sim and JPlag as comparison tools while Sim represents the parse tree algorithm and JPlag represents the string-matching algorithm.

In all experiments, the decay factor λ_{tree} is set to 0.1 empirically. Table 3 shows the thresholds θ for WASTK, ASTK, Sim and JPlag, which are determined by using the training data. The performance results are judged according to Jaccard score. The average value is used as the threshold for each model in the experiments.

The results by applying different models on the testing data of each fold are shown in Table 4, and the corresponding comparison of these four tools is illustrated in Figure 4.

According to the results, the precision values of ASTK and WASTK are much higher than Sim and JPlag. The recall values of ASTK and WASTK are much higher than JPlag but slightly less than Sim. The Jaccard score, as the overall evaluation measure, shows the advantage of ASTK and WASTK. The Jaccard scores of ASTK and WASTK are much greater than Sim and JPlag.

The added weights in WASTK damage precision but increase recall. The Jaccard score of WASTK is higher than ASTK, showing that the TF-IDF weighting is worthy.

4.4. An Online Example. WASTK has been applied to an online exam held on Jisuanke (https://www.jisuanke.com/) for code plagiarism detecting, and 290 accepted pieces of source code are collected about one problem in this exam. This problem inputs only two numbers a and b and outputs the result whether number a can be divided by number b. The average number of lines of these pieces of code is 11.

The following illustrates two pieces of code selected to be analyzed.

```
#include"stdio.h"
int main()
{
int M,N;
scanf("%d",&M);
scanf("%d",&N);
if(M%N==0)
printf("YES\n");
else
printf("NO\n");
return 0;
}
```

```
#include<stdio.h>
int main()
{int a,b;
scanf("%d",&a);
scanf("%d",&b);
if(a%b==0)
        printf("YES");
else
        printf("NO");





}
```

Their similarity is 0.694 detected by WASTK while the similarity is 0.93 detected by Sim. It is easy to find that the structures of these two pieces of code are almost the same. The variable names in them are different. Also, the methods of the head file reference are different. There is one line of "return 0;" at the end of the left one but it is not found in the right one. Moreover, these two pieces of code contain different linefeed at the beginning and the end of the function "main."

For these short pieces of code of simple problems, a higher similarity is obtained by the ordinary detection tool like Sim, but it may lead to a wrong judgment. However, WASTK acquires a more accurate result by catching the unique features of each piece of code.

5. Conclusions

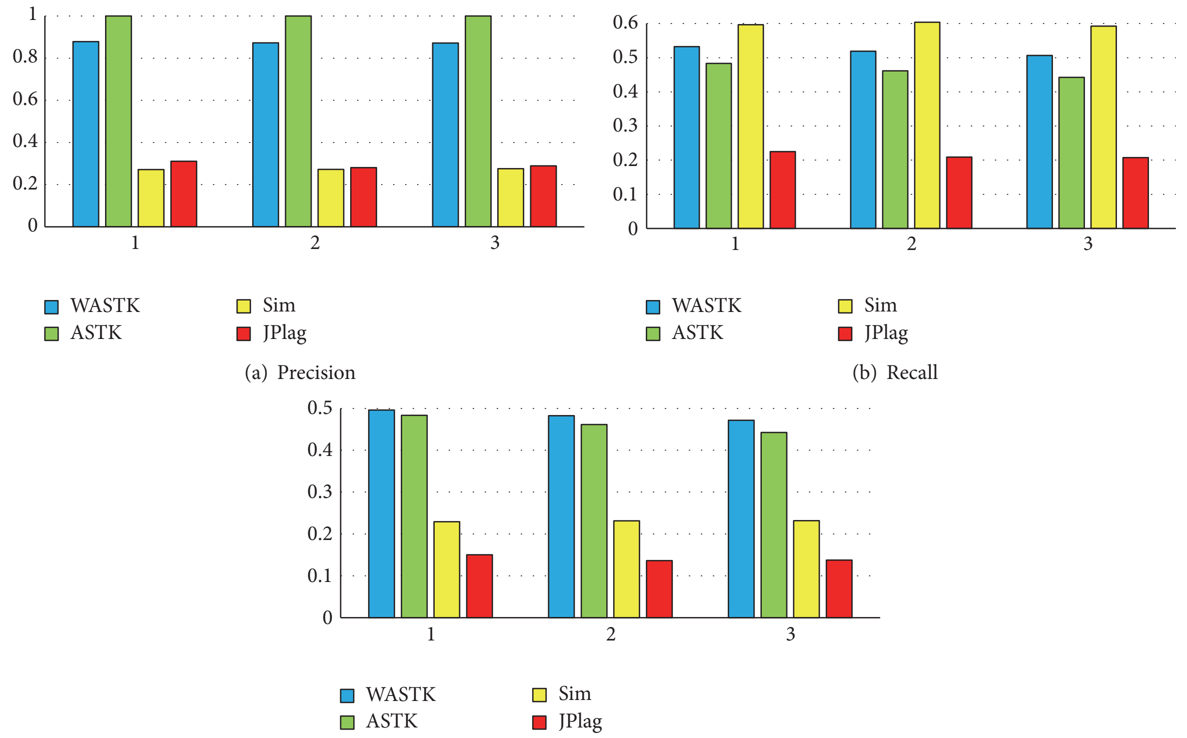
In this paper, a method and its enhancement for detecting source code plagiarism are proposed. This method not only considers the similarity between two pieces of code but also takes the context of programming into consideration. WASTK transforms string based programs to abstract syntax trees. The nodes on trees are weighted according to both tree structural similarity between a pair of programs and common structures of all programs in the dataset.

According to the results of the experiments, ASTK and WASTK perform much better than other popular methods on the same datasets. ASTK and WASTK are both based on the structure of programs instead of text like JPlag or token sequences like Sim. Besides, improved tree kernel considers the similarity between corresponding expressions for two subtrees, which is helpful for detecting plagiarism with minor changes. Additionally, WASTK adds weights to ASTK, which increases the recall and Jaccard score by applying TF-IDF. When the pieces of code have common frameworks or are based on similar solutions, TF-IDF sets lower weights to their nodes to avoid misjudgments.

WASTK can help instructors to detect whether plagiarism exists in the assignments of students in both online and offline computer science education. Also, it can be applied to online programming contests to detect plagiarism. When programs

Table 4: The results of WASTK, ASTK, Sim, and JPlag on the testing data of each fold.

Fold #	Tool	Precision	Recall	Jaccard	TP	FP	FN
1	WASTK	0.878981	0.532134	**0.495808**	414	57	364
1	ASTK	**1.0**	0.48329	0.48329	376	0	402
1	Sim	0.271504	**0.596401**	0.229362	464	1245	314
1	JPlag	0.311388	0.224936	0.150215	175	387	603
2	WASTK	0.872527	0.518954	**0.482382**	397	58	368
2	ASTK	**1.0**	0.461438	0.461438	353	0	412
2	Sim	0.272888	**0.603922**	0.231463	462	1231	303
2	JPlag	0.280702	0.20915	0.13617	160	410	605
3	WASTK	0.871965	0.50641	**0.47136**	395	58	385
3	ASTK	**1.0**	0.442308	0.442308	345	0	435
3	Sim	0.275656	**0.592308**	0.231695	462	1214	318
3	JPlag	0.289286	0.207692	0.137521	162	398	618



(a) Precision

(b) Recall

(c) Jaccard

Figure 4: Comparison of four tools.

contain a common framework or solutions for problems are simple, WASTK will show a better performance.

However, there is still space for improvement. The current method employs the tree kernel, a symmetric similarity measurement which may lead the judgment to a wrong direction. Moreover, efficiency is still a problem since the comparison has to be made between each pair of programs.

Disclosure

This work is performed when Deqiang Fu is visiting Jisuan Institute of Technology at Beijing Judao Youda Network Technology Co. Ltd. as a research intern.

Competing Interests

The authors declare that they have no competing interests.

Acknowledgments

This work is supported by the Fundamental Research Funds for the Central Universities (no. 2016JX06), the National Natural Science Foundation of China (no. 61472369), and the Computer Science Education Foundation of Jisuan Institute of Technology at Beijing Judao Youda Network Technology Co. Ltd.

References

[1] D. Kermek and M. Novak, “Process model improvement for source code plagiarism detection in student programming assignments,” *Informatics in Education*, vol. 15, no. 1, pp. 103–126, 2016.

[2] E. Eret and A. Ok, “Internet plagiarism in higher education: tendencies, triggering factors and reasons among teacher candidates,” *Assessment and Evaluation in Higher Education*, vol. 39, no. 8, pp. 1002–1016, 2014.

[3] D. Sraka and B. Kaučič, “Source code plagiarism,” in *Proceedings of the ITI 31st International Conference on Information Technology Interfaces (ITI ’09)*, pp. 461–466, Cavtat/Dubrovnik, Croatia, June 2009.

[4] H. Kikuchi, T. Goto, M. Wakatsuki, and T. Nishino, “A source code plagiarism detecting method using alignment with abstract syntax tree elements,” in *Proceedings of the 15th IEEE/ACIS International Conference on Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing (SNPD ’14)*, 6, 1 pages, July 2014.

[5] Bradley Beth. A comparison of similarity techniques for detecting source codeplagiarism, Department of Computer Science, The University of Texas at Austin, 2014, https://www.cs.utexas.edu/~bbeth/files/AComparisonOfSimilarityTechniquesForDetectingSourceCodePlagiarism.pdf.

[6] H.-J. Song, S.-B. Park, and S. Y. Park, “Computation of program source code similarity by composition of parse tree and call graph,” *Mathematical Problems in Engineering*, vol. 2015, Article ID 429807, 12 pages, 2015.

[7] M. Collins and N. Duffy, “Convolution kernels for natural language,” in *Proceedings of the 15th Annual Neural Information Processing Systems Conference (NIPS ’01)*, pp. 625–632, Vancouver, Canada, December 2001.

[8] K. S. Jones, “A statistical interpretation of term specificity and its application in retrieval,” *Journal of Documentation*, vol. 28, no. 1, pp. 11–21, 1972.

[9] M. Mozgovoy, “Desktop tools for offline plagiarism detection in computer programs,” *Informatics in Education*, vol. 5, no. 1, pp. 97–112, 2006.

[10] S. Schleimer, D. S. Wilkerson, and A. Aiken, “Winnowing: local algorithms for document fingerprinting,” in *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pp. 76–85, San Diego, Calif, USA, June 2003.

[11] 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.

[12] G. Whale, *Plague: Plagiarism Detection Using Program Structure*, School of Electrical Engineering and Computer Science, University of New South Wales, 1988.

[13] M. J. Wise, “Yap3: improved detection of similarities in computer program and other texts,” *ACM SIGCSE Bulletin*, vol. 28, no. 1, pp. 130–134, 1996.

[14] G. Malpohl, Jplag: detecting software plagiarism, 2006, http://www.ipd.uka.de/.

[15] M. Mozgovoy, S. Karakovskiyz, and V. Klyuev, “Fast and reliable plagiarism detection system,” in *Proceedings of the 37th Annual Frontiers in Education Conference-Global Engineering: Knowledge Without Borders, Opportunities Without Passports*, pp. S4H-11–S4H-14, IEEE, Milwaukee, Wis, USA, October 2007.

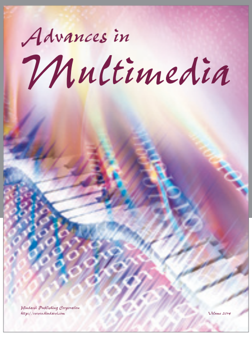
[16] B. S. Baker, “On finding duplication and near-duplication in large software systems,” in *Proceedings of the 2nd Working Conference on Reverse Engineering*, pp. 86–95, July 1995.

[17] D. Gitchell and N. Tran, “Sim: a utility for detecting similarity in computer programs,” *ACM SIGCSE Bulletin*, vol. 31, pp. 266–270, 1999.

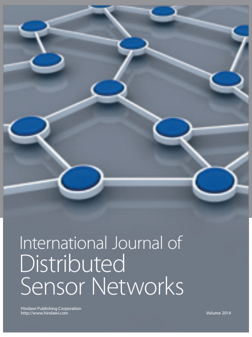
[18] B. Belkhouche, A. Nix, and J. Hassell, “Plagiarism detection in software designs,” in *Proceedings of the 42nd Annual Southeast Regional Conference (ACM-SE 42 ’04)*, pp. 207–211, Huntsville, Ala, USA, April 2004.

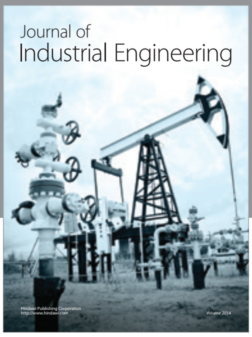
[19] B. V. Deokate and D. B. Hanchate, “Software source code plagiarism detection: a survey,” *Journal of Multidisciplinary Engineering Science and Technology*, vol. 3, no. 1, pp. 3747–3750, 2016.

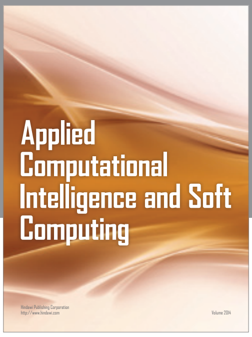
[20] J. Hage, P. Rademaker, and N. van Vugt, *A Comparison of Plagiarism Detection Tools*, Utrecht University, Utrecht, The Netherlands, 2010.

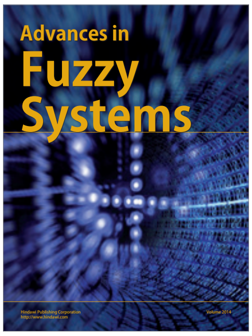

Advances in
Multimedia

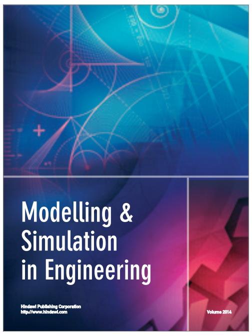
The Scientific
World Journal

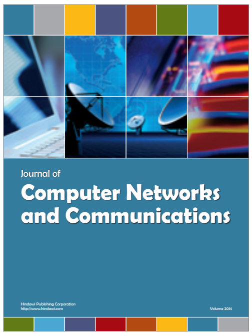

International Journal of
Distributed
Sensor Networks

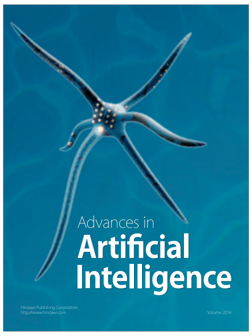

Journal of
Industrial Engineering

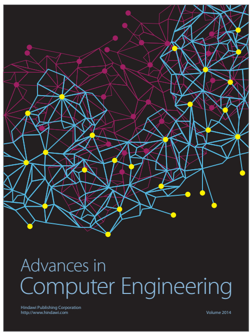

Applied
Computational
Intelligence and Soft
Computing


Advances in
Fuzzy
Systems

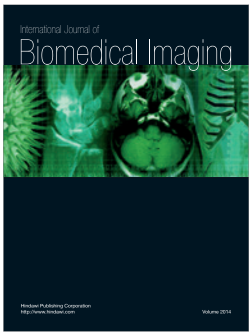

Modelling &
Simulation
in Engineering

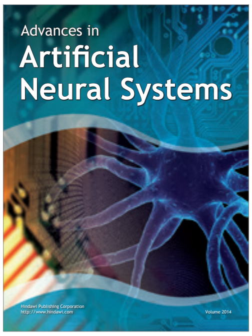

Journal of
Computer Networks
and Communications

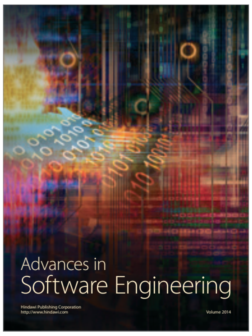

Advances in
Artificial
Intelligence


Advances in
Computer Engineering

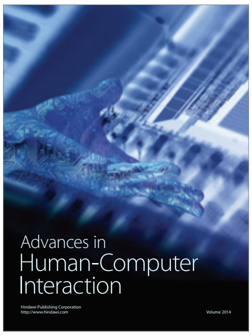
International Journal of
Computer Games
Technology

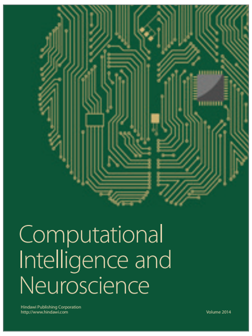

International Journal of
Biomedical Imaging


Advances in
Artificial
Neural Systems


Advances in
Software Engineering

Journal of
Robotics


Advances in
Human-Computer
Interaction


Computational
Intelligence and
Neuroscience

International Journal of
Reconfigurable
Computing

Journal of
Electrical and Computer
Engineering