Program Plagiarism Detection Using Parse Tree Kernels

Jeong-Woo Son, Seong-Bae Park*, and Se-Young Park

Department of Computer Engineering
Kyungpook National University
702-701 Daegu, Korea
{jwson, sbpark, separk}@sejong.knu.ac.kr

Abstract. Many existing plagiarism detection systems fail in detecting plagiarism when there are an abundant garbage in the copied programs. This is because they do not use the structural information efficiently. In this paper, we propose a novel plagiarism detection system which uses parse tree kernels. By incorporating parse tree kernels into the system, it efficiently handles the structural information within source programs. A comparison with existing systems such as SID and JPlag shows that the proposed system can detect plagiarism more accurately due to its ability of handling structural information.

1 Introduction

Parker and Hamblem [1] defined a software plagiarism as *a program which has been produced from another program with a small number of routine changes.* By using another students' program, one can produce a program without understanding what the source code does. Consequently, the detection of plagiarism has been on essential task for a professor.

The program plagiarism detection systems are divided into two categories. The first one is an attribute-counting-metric system. This system represents a program as a vector of which elements are the number of operators, operands, unique operators, and unique operands. Halstead first proposed a plagiarism detection system using a software science metric [2]. However, these systems shows relatively poor performance compared with those in the second type.

The second type of plagiarism detection systems not only used attribute-counting metrics, but also compared structure of source codes. Prechelt et al. proposed the JPlag system [3] based on the greedy string tiling algorithm. Chen et al. suggested the SID system [4]. Even if they show good performance and use structural information partially, the structural information is not fully reflected within their metric. It results in poor performance when plagairists insert large number of useless codes into the source code.

To efficiently express the structural information in the program source code, the system should use the parse tree which has structural information of a source code. The effective way for this purpose is using a kernel method. Kernel methods

* Corresponding author.

Q. Yang and G. Webb (Eds.): PRICAI 2006, LNAI 4099, pp. 1000–1004, 2006.

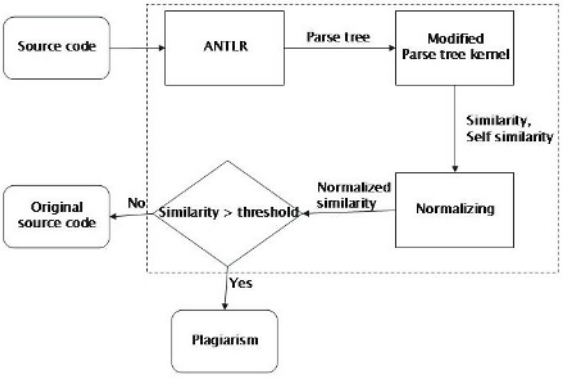

Fig. 1. The structure of the proposed system

can handle complex data in high dimensional space easily and represent the inner product of the vector representations as a similarity. Collins and Duffy proposed a parse tree kernel [5] derived from the convolution kernel.

In this paper, we modify a parse tree kernel to be applicable to the plagiarism detection. With the modified parse tree kernel, we propose a novel plagiarism detection method for Java language. We evaluated the proposed method with 4 variations of a Java source code. The experimental results prove that the proposed method can successfully detect plagiarism. Especially, the high performance in detecting the variation with a structural attack implies that the method can handle structural information efficiently.

2 Plagiarism Detection System

The proposed plagiarism detection system detects plagiarism using the parse tree of a source code. Figure 1 illustrates the proposed system which operates in three steps.

2.1 Extracting Parse Trees

To use structural information of source code, parse trees are extracted with ANTLR [6]. ANTLR is a language tool that provides a framework for constructing compiler, translator from grammatical descriptions. It also has tree parser to translate source code into a parse tree.

2.2 Parse Tree Kernel

Let $subtree_1, subtree_2, \ldots$ be all of subtrees in a parse tree. Then, the parse tree T can be represented as a vector

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

where $\#subtree_i(T)$ is a frequency of $subtree_i$ in parse tree T. The inner product of two trees, T_1 and T_2 can be defined:

$$\begin{aligned} < V_{T_1}, V_{T_2} > &= \sum_i \#subtree_i(T_1) \cdot \#subtree_i(T_2) \\ &= \sum_i (\sum_{n_1 \in N_{T_1}} I_{subtree_i}(n_1)) \cdot (\sum_{n_2 \in N_{T_2}} I_{subtree_i}(n_2)) \\ &= \sum_{n_1 \in N_{T_1}} \sum_{n_2 \in N_{T_2}} C(n_1, n_2) \end{aligned}$$

where N_{T_1} and N_{T_2} are all of nodes in tree T_1 and T_2. The indicator function $I_{subtree_i}(n_1)$ is to be 1 if $subtree_i$ is seen rooted at node n. $C(n_1, n_2)$ is a function which is defined as

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

This function can be calculated with the recursive definition [5].

2.3 Modified Parse Tree Kernel

Even though the parse tree kernel can handle tree data efficiently, it is not adequate for plagiarism detection. The first problem is that the similarity value increases too fast to handle. The scale of similarity should be reduced by using a logarithm.

Second, the value of the kernel between two different trees is typically much smaller than the value between same trees. To tackle this problem, the depth of subtrees is limited to a threshold value determined empirically. By restricting the depth, the modified kernel can calculate both the similarity between codes and the self similarity without influence of small changes.

2.4 Detecting Plagiarism

In the proposed system, a source code is decided as plagiarism when the similarity is larger than a threshold value. However, the similarity obtained via the kernel does not have any upper bound, so it needs to be normalized. Let S_1, S_2 be source codes to be compared and a function $K(S_1, S_2)$ returns the similarity using the modified parse tree kernel. Then, the normalizing function $N(S_1, S_2)$ is defined as

$$N(S_1, S_2) = \frac{K(S_1, S_2)}{E(S_1, S_2)} \times 100,$$

where a function $E(S_1, S_2)$ is normalizing factor defined as

$$E(S_1, S_2) = \frac{K(S_1, S_1) + K(S_2, S_2)}{2}.$$

Table 1. Experimental result of the proposed method on simple attacks

Source Code	Group1	Group2	Group3	Group4	Source Code	Group1	Group2	Group3	Group4
1	27	**100**	32	34	9	25	32	**100**	31
2	29	**55**	33	35	10	26	33	**47**	32
3	27	**45**	29	32	11	23	29	**70**	30
4	28	**45**	31	33	12	25	31	**58**	30
5	**100**	27	25	30	13	30	34	31	**100**
6	**47**	27	25	30	14	31	34	29	**49**
7	**60**	25	23	25	15	25	31	28	**62**
8	**51**	25	23	25	16	29	33	30	**64**

3 Experiments

3.1 Data Set

The data set for experiments are prepared by 4 programmers. Each programmer writes an original source code, and then modifies his code with following two types of attacks.

1. Simple attack type
   (a) Modify variable, function or class names.
   (b) Change expressions such as 'for', 'do∼while', and 'if' to the synonymous expressions.
   (c) Merge or Separate functions and classes.
2. Structural attack type
   (a) Add useless 80 lines into a source code.
   (b) Add useless 120 lines into a source code.
   (c) Add useless 660 lines into a source code.

3.2 Experimental Result

In all the experiments, the depth of subtrees is limited to 5 and set the similarity threshold to 40. Table 1 represents the experimental results of the proposed system with simple attack type data. In this table, the bold item indicates the source codes that are recognized as plagiary. Not only the proposed method but also all other plagiarism detection systems such as SID and JPlag also succeeded in finding the plagiarized programs. All the compared systems showed 100.00% of accuracy. This result implies that most plagiarism detection systems can find a plagiarism which is simply modified from an original source code.

Table 2 shows that the proposed method uses the structural information more efficiently than other systems. As more useless lines are added, the similarity gets down slowly but is still within the threshold. JPlag absolutely fails in detecting plagiarism when many useless lines are inserted. Even though SID is not affected by the added useless lines, the similarity values are too low, which makes it difficult to make a decision of plagiarism. In conclusion, the more useless codes are added, the better performance is shown by the proposed system.

Table 2. Performance on structural attacks

Number of added lines	Proposed Method	SID	JPlag
80	65.9	38	73.30
160	58.1	16	67.80
660	42.5	34	15.40

4 Conclusions

In this paper, we proposed a new system to detect program plagiarism by using the parse tree kernel to efficiently reflect structural information in a source code. The similarity between parse trees extracted by a ANTLR parser is measured by a parse tree kernel which is modified for the program plagiarism.

In the experiments of two attack types, the proposed method outperformed other systems. The method achieved 100% of accuracy for the simple attack, and it is not much affected by the structural attack while others are fatal to the structural attack.

An advantage of the proposed method is that it is independent of programming languages. The only requirement for other languages such as C, C++, and Pascal is the parsers for the languages.

Acknowledgements

This work was supported by grant No. (R01-2006-000-11196-0) from the Basic Research Program of the Korea Science & Engineering Foundation.

References

1. A. Parker and J. Hamblen, "Computer Algorithms for Plagiarism Detection," *IEEE Transactions on Education*, Vol. 32, No. 2, pp. 94–99, 1989.
2. M. Halstead, *Elements of Software Science*, Elsevier, 1977.
3. L. Prechelt, G. Malphol, 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.
4. X. Chen, B. Francia, M. Li, B. McKinnon, and A. Seker, "Shared Information and Program Plagiarism Detection," *IEEE Transactions on Information Theory*, Vol. 50, No. 7, pp. 1545–1551, 2004.
5. M. Collins and N. Duffy, "Convolution Kernels for Natural Language," In *Proceedings of the 14th Neural Information Processing Systems*, 2001.
6. T. Parr and R. Quong, "ANTLR: A Predicated-LL(k) Parser Generator," *Journal of Software Practice & Experience*, Vol. 25, No. 7, 1995.