A Hybrid Method for Detecting Source-code Plagiarism in Computer Programming Courses

Weijun Chen, Chenling Duan, Li Zheng, Youjian Zhao

Tsinghua University, China

0402

The European Conference on Education 2013

Official Conference Proceedings 2013

Abstract

The paper presents a hybrid method for detecting source-code plagiarism in computer programming courses. For many programming courses, the students' assignments are in the form of electronic source files and it is difficult for the teacher to manually detect the plagiarisms among the assignments. Our system can compare two source files automatically and help to solve this problem. The principle of the system is summarized: Firstly, the source files are processed with intension of filtering the noise elements such as header file include statements, comments, input/output statements and string literals. Secondly, a feature-based detection component is proposed. For each source file, a feature vector is generated which include physic metrics such as the number of source lines and the number of total words, Halstead metrics such as the statistics of source code operators, execution flows and operands. Then the distance between two feature vectors is computed which is considered as the similarity between the corresponding source files. Thirdly, a structure-based detection component is proposed. For each source file, the source code is transformed into a sequence of well-defined tokens. Then to improve the computational efficiency each token is transformed into a single character using a mapping table. The LCS (Longest Common Subsequence) is computed for each two strings, which is considered as the similarity between the corresponding source files. Lastly, an integration component is proposed which uses a two-stage strategy to combine the above two separate components into one complete system. Experimental results show that our system can effectively spot the suspect program copies, even when they have some kind of minor modifications.

1 Introduction

When teaching an introductory computer programming course, it is commonly agreed that the emphasis of the course should be put on improving students' skills of problem analysis, algorithm design and coding. Therefore, in the practicing section, most courses will require the students to complete a sufficient amount of programming assignments and submit the corresponding source code files. However, this may bring a potential problem to the instructor: the source-code plagiarism. Plagiarism occurs when programming exercises are copied and transformed with very little effort from other students. Although finding plagiarisms manually is possible in theory, it is much too time-consuming in practice and very few instructors have the patience to thoroughly search for plagiarisms. Therefore, a powerful automated searching tool is being needed.

In this paper, we present a hybrid method for detecting source-code plagiarism automatically in computer programming courses. It combines a feature-based component and a structure-based component in one system.

The rest of the paper is organized as follows. In Section 2, we present the related works on the automatic source-code plagiarism detection systems. Then we give a detailed description of our system in Section 3. In Section 4, the evaluation results are shown and discussed. Finally we conclude our work in Section 5.

2 Related Work

The previous source-code plagiarism detection systems can be roughly divided into two types: feature-based and structure-based.

The first known plagiarism detection system was an attribute counting program developed by Ottenstein (1976). It uses the basic Halstead (1977) metrics (number of unique operators, number of unique operands, total number of operators and total number of operands) to compare two FORTRAN programs and if all the four values coincide, the programs can be considered to be plagiarisms.

Other feature-based systems (Berghel &Sallach, 1984; Donaldson & Lancaster, 1981; Faidhi & Robinson, 1987) employ a similar strategy. For each program, a set of different software metrics are extracted and they make up a feature vector that corresponds to a point in an n-dimensional Cartesian space. The distance between two feature vectors can be seen as the similarity degree of corresponding programs.

The feature-based systems are efficient because they only need to compute a limit number of feature values for each program. However, they can hardly have very good performance because they throw away too much structural information.

The structure-based systems compare the structures of two programs directly. The basic idea is to convert each program into a stream of tokens and then compare these token streams to find common segments. The comparing algorithm needs to be designed carefully because it will determine the efficiency of the whole system.

Typical structure-based systems include Michael Wise's YAP3 (1992), Alex Aiken's MOSS and Guido Malpohl's JPlag (2000). The differences lie in the detection performance, the run time efficiency and the user interface.

Generally speaking, the feature-based systems are more efficient and have relatively lower performance, while the structure-based systems have better performance and are less efficient, especially when dealing with a large data set (Verco &Wise, 1996). Therefore a natural idea is to produce a hybrid system that combines both the structure and the feature comparison. For example, a hybrid method was proposed by Donaldson & Lancaster (1981). In the first phase, it uses eight features and proposes three different methods determine a similarity or difference factor. In the second phase, each assignment is transformed into a statement order sequence and the sequences of each pair of assignments are compared to determine if the structures of the assignments are similar enough to indicate plagiarism.

For the first author of this paper, he is teaching a computer programming course to non-computer science students at Tsinghua University, China. In every semester, there will be about 150 students enrolled in this class and in each week, the students have to complete five or six programming assignments. Therefore, in order to detect the plagiarisms in such a real situation, we have to consider the following aspects:

- We actually don't need a "perfect" plagiarism detection system. As a teacher, the more important thing is to educate the students not to cheat and let them know the consequences of cheating. The plagiarism detection system is just an auxiliary technical tool that helps us to find as many plagiarisms as possible in limited time.
- There needs to be a balance between effectiveness and efficiency. In every week, we will have five or six programming assignments and for each assignment we will receive 150 source code files. This is a large data set and the time efficiency is more important to us.
- Although there are several free plagiarism detection tools are available, it is still time consuming to use these tools. We have to prepare the data files in a local machine and submit them to the servers. In fact, we already have a powerful E-Learning platform for computer programming courses (Shi, Chen, Zhang and Luo, 2012). If we can integrate the plagiarism detection system into the platform, it will be convenient to use and no extra time is required because all the data are already there.

3. The System

The source-code plagiarism detection system consists of five main components: the pre-processing component, the feature-based component, the structure-based component, the integration component and the main process component. Figure 1 shows the framework of the system.

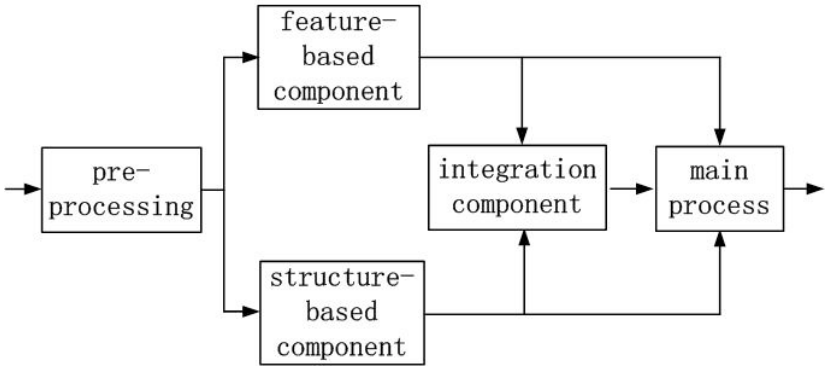

Figure 1. Framework of the system

3.1 Pre-processing Component

For each programming assignment, the students will submit the corresponding source code files. These files can't be compared directly because they contain many noise elements that have negative effects on measuring the similarity between source files. Therefore, we need a pre-processing component to filter out these noise elements.

Firstly, we need to filter out all the "header file include" statements. The following are some of the examples:

```
#include <stdio.h>
#include <string.h>
```

For a valid C source file, the "header file include" statements are obviously necessary because they will tell the compiler the correct prototypes of specified functions and make the compiling go smoothly. However, these statements are meaningless when computing the similarity value of two source files. Instead, they may have negative effect. For example, a student may insert functionally useless "header file include" statements in a source file on purpose to make it look different with others.

Secondly, we need to remove all the comments and blank lines from the source files. Otherwise, one can easily fool the automatic plagiarism detection system by writing fake statements in the comment area. For example,

`sum = 0;`
`i = 1;`
`while(i <= 10)`
`{`
`  sum += i;`
`  i++;`
`}`
	`/* fake code`
	`r = 5;`
	`PI = 3.1415;`
	`area = PI*r*r;`
	`*/`
	`sum = 0;`
	`i = 1;`
	`for(; i <= 10; i++)`
	`{`
	`  sum += i;`

	}

From a programmer's point of view, the above two source segments are completely the same. However, an automatic detection system may think they are different because the second source segment has three more "statements" (the first three statements). Actually these statements are just comments and they have no effect on the running of the program.

Thirdly, we need to remove all the input/output statements. These statements are mainly used for user interface purpose and have few relationships with the core functions of the program.

Lastly, all the string literals are removed from the source files. Strictly speaking, a string literal is actually a type of data, not a statement. It is mainly used for program debugging and user interface purpose. Just like the comments, the string literals in a program can be easily used to fool the automatic plagiarism detection system. For example, one can define a carefully designed long string literal that contains all types of operators, operands and keywords in his program, and then this string literal will cause big troubles to any type of detection systems.

3.2 Feature-based Component

The first step of designing a feature-based detection system is to propose a set of feature values. In our system, we employ two types of features: physical features and Halstead's software metrics.

Physical features are quite straightforward. They include file size, number of source lines, number of words, number of characters, etc. In our system, we use two physical features, i.e. number of source lines and number of words. These two features seem to be more valuable than others.

Halstead features refer to the statistics of source code attributes, specifically, the usage of different identifiers. The identifiers in the C language consist of operators, keywords, predefined identifiers and user defined identifiers. In our system, we will consider six attributes of source code: arithmetic operator metrics, relational operator metrics, logical operator metrics, execution flow metrics, operand metrics and the number of different operands.

The feature extraction component in our system can transform a pair of source files into two feature vectors. Then the next step is to compute their similarity, which is summarized as follows:

Suppose we have two vectors G and H, we need to transform them into another two vectors G' and H' using the following formulas:

$$g_k' = \frac{g_k}{g_k + h_k} \quad (1)$$

$$h_k' = \frac{h_k}{g_k + h_k} \quad (2)$$

g_k: the k-th element of G
h_k: the k-th element of H

After the transformation, each element value is limited in the range of [0, 1].

Then the vector difference is computed using the following formula:

$$D = G' - H' \quad (3)$$

The similarity of the two feature vectors is defined as:

$$\text{sim} = \begin{cases} 1 - \frac{|D|}{\sqrt{6 * sen}}, & |D| <= \sqrt{6 * sen} \\ 0, & |D| > \sqrt{6 * sen} \end{cases} \quad (4)$$

sen is the sensitivity coefficient whose value is in the range of [0, 1]. A smaller value indicates that the similarity detection is stricter, i.e., if the vector difference is bigger than a small value, the two source files are considered to be not similar to each other. In our system, *sen* is set to 0.2.

Suppose the physical similarity is $S1$ and the Halstead similarity is $S2$, then the final similarity of two source files is computed using the following formula:

$$S = S_1 * W_1 + S_2 * W_2 \quad (5)$$

W_1 and W_2 are the corresponding weights, in our system, $W_1 = 0.2$, $W_2 = 0.8$.

3.3 Structure-based Component

A structure-based component is proposed to detect the source-code plagiarism automatically in another way.

3.3.1 Identifier Tokens

The first step of the structure-based component is to transform the source code into a sequence of well-defined identifier tokens. There are different kind of identifiers in a C/C++ source file such as keywords, reserved words and user-defined identifiers. They should be transformed into a set of standard tokens such that the modifications of the variable names, type definitions and function names will have no effect on the measurement of program similarity.

The paper defined the following identifier tokens for a C/C++ program:

- CLASS: user defined *class* in the program
- STRUCT: user defined *struct* type
- TYPE: data type
- OBJ: *class* instance
- FUNC: functions defined in the program
- VAR: data variables
- CON: constants

For an input source code file, all the identifiers are substituted with corresponding tokens using a set of rules. For example, all the keywords such as *short*, *int*, *long*, *float*, *char* and *bool* are substituted with the identifier token TYPE.

```
#include <stdio.h>
#include <stdlib.h>
struct student *creatLA();
struct student *creatLB();
void display(student *head);
struct student
{
    int id;
    int oper;
    student *pnext;
};
void display(student *head)
{
    student *p = head;
    while(1)
    {
        printf("%d\n", p->id);
        if(p->pnext == NULL)
        {
            break;
        }
        p = p->next;
    }
}
void main()
{
    student *headLA, *headLB;
```

```
struct STRUCT *FUNC( );
struct STRUCT *FUNC( );
TYPE FUNC ( STRUCT * OBJ );
struct STRUCT {
TYPE VAR ;
TYPE VAR ;
STRUCT * OBJ ; }
TYPE FUNC ( STRUCT * OBJ ) {
STRUCT * OBJ = OBJ ;
while( CON )
{
if( OBJ . OBJ == CON )
{
break;
}
OBJ = OBJ . OBJ ;
} }
TYPE FUNC ( ) {
STRUCT * OBJ , * OBJ ;
OBJ = FUNC ( ) ;
OBJ = FUNC ( ) ;
FUNC ( OBJ ) ; }
```

```
    headLA = creatLA();
    headLB = creatLB();
    display(headLA);
}
```

(a) A C program (b) The string of identifier tokens

Figure 2. An example of identifier token transformation

Figure 2(a) is an example C program. It will be transformed into a string of identifier tokens that is illustrated in Figure 2(b) after all the substitution rules are applied sequentially.

3.3.2 Digitization of the Token String

To improve the computational efficiency, each token is transformed into a single character using a mapping table. As we know, there are 10 digits, 26 lower case letters ('a' ~ 'z') and 26 upper case letters ('A' ~'Z'). Therefore, we totally have 62 single characters. Figure 3 is a part of the mapping table.

asm	0	CLASS	A
auto	1	STRUCT	B
break	2	TYPE	C
case	3	OBJ	D
catch	4	FUNC	E
const	5	VAR	F
const_cast	6	CON	G
continue	7	static	H
default	8	switch	I
delete	9	this	J

Figure 3. Part of the mapping table

For example, the token string illustrated in Figure 2(b) will be transformed into the following character string:

yB*E()yB*E()CE(B*D)CE(B*D,B*D)yB{CFCFB*D}CE(B*D){B*D=DR(G){m(D.D==G){2}D=D.D}}CE(){B*D,*DD=E()D=E()E(D)}

It is obvious that this string is far shorter than the original token string, which means it will need less processing time.

3.3.3 Similarity Measurement

For each two character strings, the LCS (Longest Common Subsequence) is used to compute their similarity degree. When detecting the plagiarisms among source code files, we should consider the fact that some students may make some minor modifications such as changing the order of function definitions. Therefore, the system divides

each character string into several blocks using the separator characters "{" and "}". The LCS is computed for each block in the string.

```
s1_set = Code_split(s1)
s2_set = Code_split(s2)
lcs_len = 0
for s1_block in s1_set:
    max_lcs_len = 0
    for s2_block in s2_set:
        lcs_len_tmp = Lcs_len(lis_func,s1_block,s2_block)
        if lcs_len_tmp>max_lcs_len:
            max_lcs_len = lcs_len_tmp
    lcs_len += max_lcs_len
```

Figure 4. The LCS length algorithm

Figure 4 is the algorithm used to compute the LCS length of two strings *s*1 and *s*2. For each block in *s*1, find the corresponding block in *s*2 that has the longest LCS length with it and add that length to the value *lcs_len*, which is the LCS length of *s*1 and *s*2. Finally, the similarity degree between s1 and s2 is computed using the following formula:

$$sim = \frac{2 * \text{lcs\_len}}{|\text{s1}| + |\text{s2}|}$$

3.4 Integration Component

Generally speaking, the feature-based methods are more efficient and have relatively lower performance, while the structure-based methods have better performance and are less efficient, especially when dealing with a large data set. In our system, we proposed a hybrid method that combines both the structure and the feature comparison. The algorithm is summarized in Figure 5.

S1	input a pair of source files
S2	compute their similarity *sim1* using the feature based component
S3	if *sim1* is lower than a threshold value, *sim* = *sim1*, go to S6
S4	otherwise, compute their similarity *sim2* using the structure based component
S5	*sim* = *sim1*w1* + *sim2*w2*
S6	return (*sim*)

Figure 5. The integration algorithm

4. Experimental Results

A series of experiments are made to verify our system and sound results are achieved.

Generally speaking, it is not likely for a student to submit a program that is exactly the same as another student's program. He may make some changes in the program on purpose to make it looks different with the original one. Therefore, a good plagiarism detection system should be able to deal with these changes.

In the first experiment, for a given program submitted by a student, we modified some functionally useless statements in it and made a "new" program. For example, we added some comments and "header file include" statements, re-edited the program in another style and used the blank spaces, carriage returns and indentation in another way. However, when we used our system to measure the similarity of these two programs, the result was 1.0 which means they are regarded to be completely the same.

In the second experiment, we changed the names of different type of identifiers, such as the variable names, function names and user-defined *struct* names. Again, the measurement result was still 1.0. This is because the feature based component only computes the numbers and distributions of operators and operands, it does not care about the names of different identifiers. And in the structure based component, all the user-defined identifiers are substituted with a set of standard tokens.

In the third experiment, we changed the structure of the code. For example, we changed the order of some functions and substituted a function call with the body of the function itself. The similarity degree was more than 0.9.

We also used a real data set to testify the system. The data set consists of a group of source files submitted by the students in our class. There are 145 students and the length of each source file is about 100~200 lines. Those pairs of programs that have a relatively high similarity degree are examined by us manually. It turns out that for any pair of programs whose similarity value is greater than 0.7, there are always exist many code blocks that seem to be similar enough.

5 Conclusion

The paper presents a software tool that can help teacher to find the potential plagiarisms among the vast number of programming assignments. It is a hybrid system that combines two types of detection methods. The system consists of four main components: the pre-processing component, the feature-based component, the structure-based component and the integration component. Experimental results show that it can effectively spot the suspect program copies, even when they have some kind of modifications.

Ongoing and future work includes the further improvements of the system. For example, the user interface is a big problem. Currently the output results are saved in an Excel file, this is not convenient for the users to understand the results, especially when the data set is big. We need to develop a graphic user interface to present the results.

References

1. Ottenstein, K. J. 1976, An algorithmic approach to the detection and prevention of plagiarism. ACM SIGSCE Bulletin, 8(4), 30-41.
2. Halstead, M. H. 1977, Elements of Software Science. Operating and Programming Systems Series. Elsevier North-Holland, New York.
3. Berghel, H. L. and Sallach, D. L. 1984, Measurements of program similarity in identical task environments. ACM SIGPLAN Notices, 19(8), 65-76.
4. Donaldson, J. L., Lancaster, A. M. and Sposato, P. H. 1981, A plagiarism detection system. ACM SIGSCE Bulletin, 13(1), 21-25.
5. Faidhi, J. A. W. and Robinson, S. K. 1987, An empirical approach for detecting program similarity within a university programming environment. Computers and Education, 11(1), 11-19.
6. Wise, M. J. 1992, Detection of similarities in student programs: YAP'ing may be preferable to Plague'ing. ACM SIGSCE Bulletin, 24(1), 268-271.
7. Aiken, A. MOSS (Measure Of Software Similarity) plagiarism detection system, http://www.cs.berkeley.edu/ ˜moss/, University of Berkeley, CA.
8. Prechelt, L., Malpohl, G. and Phlippsen M. 2000, JPlag: Finding plagiarisms among a set of programs. Technical Report, University of Karlsruhe, Germany.
9. Verco, K. K. and Wise, M. J. 1996, Software for detecting suspected plagiarism: Comparing structure and attribute-counting systems. In John Rosenberg, editor, Proc. of 1st Australian Conference on Computer Science Education, Sydney.
10. Shi, K., Chen, W., Zhang, L. and Luo, L. 2012, Kaleidia: A Practical E-Learning Platform for Computer Programming Courses. In Proceedings of the Canada International Conference on Education (CICE-2012), University of Guelph, Canada.

iafor