Available online at www.ijpe-online.com

International Journal of Performability Engineering

vol. 15, no. 10, October 2019, pp. 2683-2691
DOI: 10.23940/ijpe.19.10.p14.26832691

Code Similarity Detection using AST and Textual Information

Wu Wen[a], Xiaobo Xue[b], Ya Li[a,*], Peng Gu[a], and Jianfeng Xu[b]

[a]Guangzhou University, Guangzhou, 510006, China
[b]Nanjing Mooctest Information and Technology Co. Ltd, Nanjing, 210000, China

Abstract

In the teaching process of computer language courses, a large amount of programming experimental content needs to be supplemented. Students sometimes copy codes from each other, which seriously reduces the teaching quality of computer language courses and makes it difficult to improve students' programming abilities. To solve this problem, this paper proposes a novel code similarity detection algorithm based on code text and AST. By removing comments, blank characters, and other "cleaning" processes from the code text, the normalized code text is obtained. Then, word segmentation, word frequency statistics, weight calculation, and other operations are carried out. The code fingerprint is obtained by using the Simhash algorithm. According to the specification of computer language grammar, lexical analysis and syntax analysis are conducted to extract the AST (abstract syntax tree), and redundant information is eliminated. According to the Zhang-Shasha algorithm, the AST edit distance is calculated and then compared to the AST. Finally, the similar degree between the text similarity and AST similarity is calculated. In order to verify the effectiveness of this method, taking Python code as an example, the code on the open source programming platform and LeetCode is used to build the test data set according to the common code plagiarism method. Experimental results show that this method is capable at detecting several common means of plagiarism, and low similarity can be obtained for the experimental detection of unrelated codes and non-plagiarized codes. Therefore, we believe that this algorithm can effectively be used for the code similarity detection of experimental code in computer language courses.

Keywords: code plagiarism; code similarity; text similarity; abstract syntax tree

(Submitted on August 7, 2019; Revised on September 20, 2019; Accepted on October 21, 2019)

1. Overview

Computer language courses involve gaining computer programming knowledge and learning programming languages to cultivate students' programming abilities. This type of course is not only a required professional course for computer- and software-related majors, but also a basic course for all non-professional students in higher education to master computers. At present, nearly all universities and colleges in China have computer language and programming courses for students to study. In this kind of course, experimentation is one of the essential teaching methods. Through the actual programming code, students can better understand the curriculum knowledge and improve their programming ability. In the actual teaching process, it was found that due to the different learning abilities and learning attitudes of students, plagiarism among students is very serious. A study of plagiarism in programming classes at Monash University in Australia found that 85.4% of students admitted to copying other people's codes [1-2]. In order to ensure the sound development and teaching quality of computer language courses, it is necessary to study an effective method to screen and detect codes with high similarity.

Researchers Faidhi and Robinson analyzed how and to what extent code was copied, defining seven levels of difficulty [3]:

- Level-1: Do not modify.
- Level-2: Modify comment statement.
- Level-3: Modify the program identifier.
- Level-4: Modify locations of declared variables.

* Corresponding author.
E-mail address: liya@gzhu.edu.cn

- Level-5: Program restructuring.
- Level-6: Modify control statement.
- Level-7: Modified control logic.

According to the statistical observation in the daily teaching process, most students who plagiarize others' codes are short of practical ability, fail to master the course knowledge well, or lack correct learning attitudes. When copying others' codes, no deep changes are made to the code (usually between level-1 and level-4). In other words, if we clone other people's codes and carry out deep modification and reconstruction while ensuring the integrity of the function, we believe that this behavior should not be considered plagiarism, and the similarity detection results should show that the similarity with the copied source is low. Therefore, we classify common plagiarism in the experiment of computer language courses into the following situations:

1) Copy and paste it completely.
2) Modify code comments.
3) Modify identifiers (class name, method name, variable name).
4) Modify the code order.
5) Code refactoring (function extraction/splitting, equivalent loop structure replacement).

According to the above content, we believe that the similarity comparison algorithm discussed in this paper should be able to detect the above five plagiarism methods well, which shows high similarity.

2. Related Work

Since the 1970s, some scholars have studied the similarity measurement of program code. At present, there are some excellent methods and mature systems in this field, which can be roughly divided into attribute counting (AC) measurement methods and structural measurement (SM) methods.

The traditional text similarity detection method involves scanning the frequency of all words in the document, expressing them as vectors, and measuring their similarity by calculating the cosine distance, Euclidean distance, and other methods. The attribute counting [4] method proposed by Halstead based on the software science metric is one of the earliest algorithms for program code similarity detection. The program code contains a large amount of attribute information that can be used for statistics and analysis, such as the number of code lines and the number of identifiers. After the statistics of various attribute information of the program code, the method of correlation comparative analysis is the attribute counting method. The first similarity detection system was implemented by Ottenstein on the basis of Halstead thought. Due to the limitations of programming languages at that time, the system could only detect programs written in FORTRAN [5].

In order to improve the accuracy of the existing similarity detection system, many researchers continue to expand the number and types of attribute count statistics, such as methods in the code and the number of control structures. These increasing attributes make the system gradually become bloated and complex, but the detection efficiency is not significantly improved. Wise and Verco reported after testing the Accuse system in 1996 that attribute counting ignored the semantic structure of the program, and that simply increasing the number of statistical properties of the program would not significantly improve the measurement results. Therefore, Wise and Verco put forward a new measurement concept: only by combining the semantic and structural information of programs can the accuracy of similarity measurement be effectively improved.

At present, almost all program code similarity detection systems adopt SM technology or use AC and SM technology together. Donaldson's similarity measurement system, for example, combines AC and SM techniques, while other systems, such as Sire, Plague, and YAP3 [6], use structural measurement techniques to capture program semantics. Clough and Parker [7] respectively described the above procedure similarity measurement method in detail. In addition to AC and SM technologies, neural networks have also been proposed by relevant scholars for the measurement and detection of program code.

3. Algorithm Design

Computer language codes, like natural language texts, are written and read in the form of texts. Therefore, considering the common text similarity calculation methods in natural language processing, text similarity analysis can be carried out for program codes. Different from natural languages, computer languages have clear and rich structural features, which contain abundant information about program code. Simply changing variable names, class names, and other contents in the code cannot change its structural features.

Therefore, this paper designs a similarity detection algorithm based on Simhash and abstract syntax tree to measure the similarity degree of program code from the aspects of text similarity and code syntax structure similarity. The algorithm flow chart is shown in Figure 1. The algorithm first makes unified formatting processing for the program code pair, then takes it as ordinary text, selects the appropriate Hash algorithm to generate a code fingerprint, and measures its text similarity S1 by calculating the editing distance between the code pair generated by the code fingerprint. In order to improve the accuracy, considering the particularity of the program code text, we can ignore the text features to extract its syntax structure. According to the compilation process of computer programming language, it is an appropriate method to generate an abstract syntax tree to represent the code structure. The program code can generate the corresponding abstract syntax tree through lexical analysis and syntax analysis, and the structural similarity S2 of the program code can be obtained by calculating the similarity of the two abstract syntax trees. S1 and S2 are weighted reasonably to obtain a comprehensive similarity, which is used to measure the degree of similarity of the program pair. Specifically, as shown in Figure 1, this method includes the following steps:

1) Format the source code and delete comments, redundant blank lines, spaces, and other information in the source code to improve the efficiency and accuracy of similarity detection.

2) The locally sensitive Simhash algorithm is used to hash the formatted program code, calculate the Hamming edit distance between the two Simhash signatures, and measure the similarity between the two code texts according to the Hamming distance.

3) According to the grammar specification of the programming language, the lexical analyzer and the grammar analyzer are used to analyze and process the program code and extract the abstract grammar tree. Normalize the abstract syntax tree, remove redundant information, and calculate the editing distance between the two abstract syntax trees based on the Zhang-Shasha algorithm, so as to measure the structural similarity of the program code.

4) Seek an appropriate strategy, give different weights to text similarity and structural similarity, and carry out weighted calculation to obtain the final comprehensive similarity, indicating the similarity between program codes.

3.1. Algorithm for Detecting Text Similarity of Code

In order to ensure the efficiency and accuracy of identification, this algorithm selects an appropriate Hash algorithm to generate the fingerprint of code text, which is composed of a series of 01 strings. Therefore, the Hamming distance can be used to calculate the similarity between two code fingerprints efficiently and accurately. In the code of computer language, modifying the code order, variable name, and other operations often does not change the semantics and correctness of the code, so the Hash algorithm is required to have locally sensitive characteristics:

1) The same text generates the same fingerprint.
2) Different texts generate different fingerprints.
3) Similar texts generate similar fingerprints.

The traditional Hash algorithm cannot guarantee local sensitivity and does not have the above characteristics. Therefore, Simhash is adopted in this paper as the construction algorithm of code fingerprints. It is shown in Figure 2. This algorithm is a Hash algorithm with low collision rate and local sensitivity [8], which is widely used by Google for massive web pages.

The pseudocode of text similarity detection algorithm based on Simhash is shown in Algorithm 1.

This algorithm first calls the Simhash algorithm to process the input codes. First, code text segmentation processing, form code text feature words, and calculate the weights of each word according to the word frequency. Obtain a sequence consisting of 0 and 1 to 0 and 1 sequence weighted, and convert from 0 to 1. The sequence on each number is multiplied by the weight of the word, and all words are calculated number sequences accumulated into a sequence. Reduce the dimensionality of the sequence to 0-1 sequence, convert each digit greater than 0 to 1, and convert to 0 if it is less than or equal to 0 to obtain the final code fingerprint sequence. The Hamming distance between two code fingerprints is calculated, and the number 1 in the result obtained by xor operation on two sequence strings is the distance between the two code fingerprints. The maximum digit of the distance and the fingerprint sequence is normalized, and finally the similarity represented by 0~1 can be obtained.

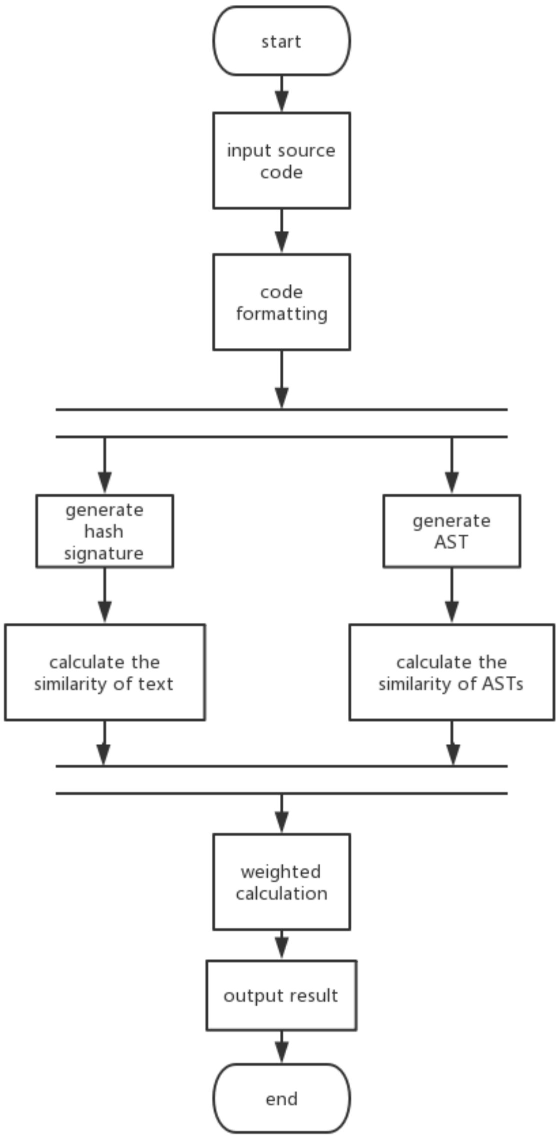

Figure 1. Algorithm flow chart

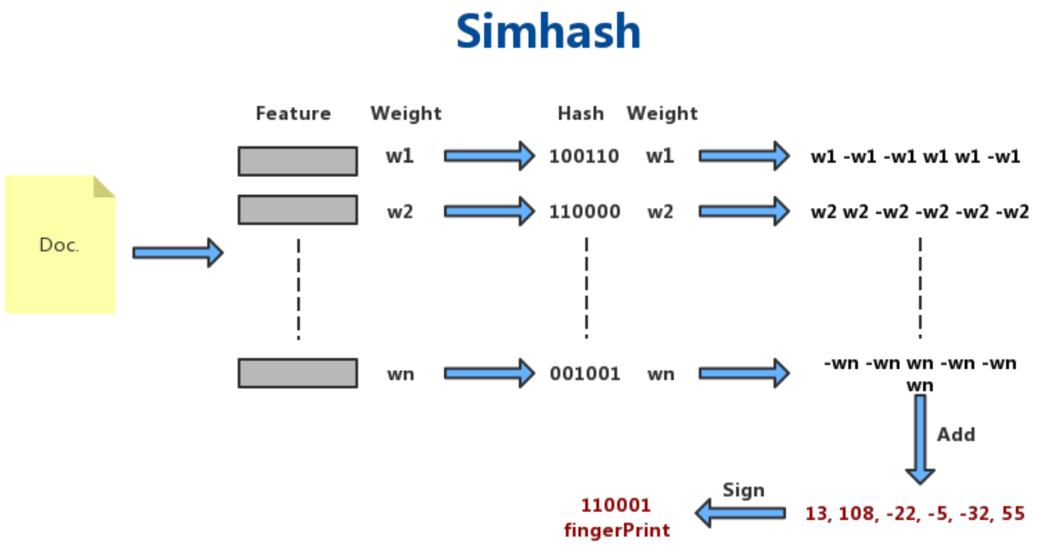

Figure 2. Simhash algorithm

Algorithm 1 Detecting text similarity of code

```
Input: formatted code text, code1, code2
Output: text similarity between code1 and code2, $S_t$
 1: Call the Simhash algorithm
 2: { // Simhash has 5 steps
 3: word segmentation
 4: statistical word frequency, calculation weight
 5: dimensionality reduction by hash, weighted calculation
 6: vertical accumulation
 7: dimension reduction, generate code fingerprint
 8: return $fingerPrint_1$
 9: }
10: Call the Simhash algorithm to get $fingerPrint_2$
11: $distance=hammingDistance\$(fingerPrint_1, fingerPrint_2)$
12: $S_t$=1-($distance$ / bitsOf (simhash))
13: return $S_t$
```

3.2. Algorithm for Detecting Text Similarity of Code

Computer programming language programming codes cannot be directly run. Commands can usually be understood and executed through lexical analysis, syntax analysis, semantic analysis, and other processes [9]. In this process, natural language-like features in the program code are gradually removed, leaving the structure and information needed for computer execution. An abstract syntax tree is an abstract representation of a program's source code. Abstract syntax trees hide the details of the actual syntax in the code, such as nesting and branching. The AST provides a more intuitive representation of the syntactic structure of a program's code. For example, in the Java language grammar, the "a+b=c" is parsed to obtain the AST as shown in Figure 3.

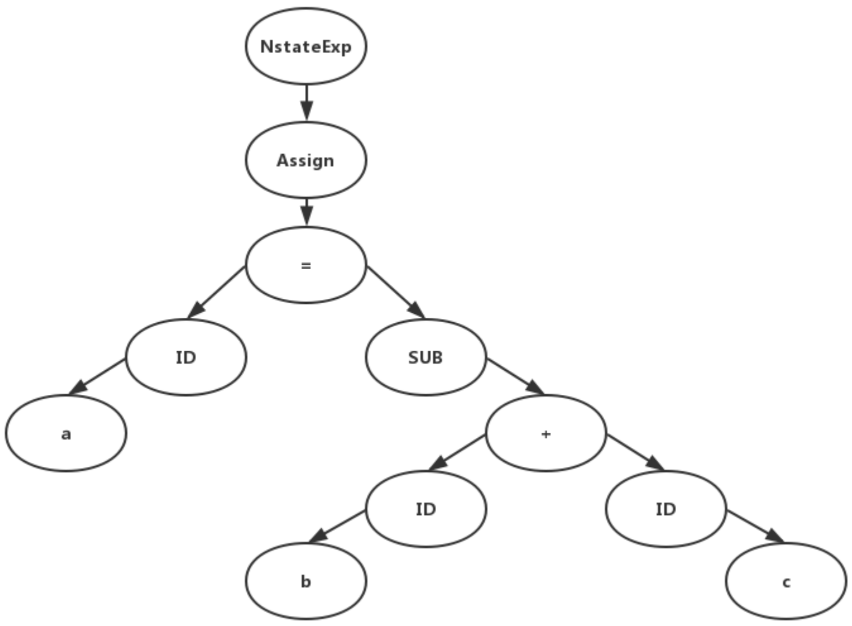

Figure 3. Example of AST

In this article, the AST of the program code is extracted. By comparing the AST similarity, the structural similarity of the code can be obtained. The Zhang-Shasha algorithm [10] is a classical tree editing distance algorithm that was proposed by Zhang and Shasha in 1989. It is often used to calculate the editing distance between two trees and solve the similarity measurement problem [11]. The pseudocode of the algorithm process is shown in Algorithm 2.

According to the grammar of the language that the input code belongs to, lexical analysis and syntax analysis are carried out on the code to generate an abstract grammar tree. At this time, AST often contains a large amount of redundant information, which will reduce the accuracy and efficiency of similarity detection. Taking the following python code as an example, the contents of the ast generated by calling the ast module for lexical and syntactic analysis are shown in Listing 1.

Algorithm 2 Detecting structural similarity of code

```
Input: tokens_of_code_1, tokens_of_code_2
Output: structural similarity, Sc
 1: tokens_of_code_1 = lexicaAlnalyzer(code1)
 2: tokens_of_code_2 = lexicaAlnalyzer(code2)
 3: ast1 = grammarAnalyzer(tokens_of_code_1)
 4: ast2 = grammarAnalyzer(tokens_of_code_2)
 5: normalization(ast1)
 6: normalization(ast2)
 7: distance = ZhangShasha(ast1, ast2)
 8: base = max(num_of_nodes_ast_1, num_of_nodes_ast_2)
 9: Sc = 1 – (distance/base)
10: return Sc
```

Listing 1. Test algorithm

```
# example code
b = 1
c = 2
a = b + c

 # print the AST
 Module(body=[
   Assign(
     targets=[Name(
       id='b',
       ctx=Store())],
     value=Num(n=1)),
   Assign(
     targets=[Name(
       id='c',
       ctx=Store())],
     value=Num(n=2)),
   Assign(
     targets=[Name(
       id='a',
       ctx=Store())],
     value=BinOp(
       left=Name(
         id='b',
         ctx=Load()),
       op=Add(),
       right=Name(
         id='c',
   ctx=Load())))])
```

It can be seen that the generated AST can not only display the structural characteristics of the program, but also contain a large number of attribute characteristics. For each node in the AST, only the structure, type, attribute, and other information of the node need to be clarified, such as the specific information of "id = 'b'". A different "id" only has a different name, which will not cause a change in node attributes and will not affect the function of the program, but it will make the editing distance of the tree larger and affect the accuracy of the algorithm. Therefore, this part of the algorithm only focuses on the structural information of the code, so it is necessary to normalize the generated AST tree, delete the redundant information, and retain only the structure and node type information of the tree, so as to improve the accuracy and efficiency of the algorithm. Normalize the AST in the above example as Listing 2.

After the normalized AST is obtained, the minimum cost distance required to convert from *ast1* to *ast2* is calculated according to the definition of the Zhang-Shasha algorithm, which is the edit distance between two AST trees. Divide the edit distance with the maximum number of AST nodes, and further calculation can obtain a value between 0 and 1 to represent the structural similarity of AST.

3.3. Calculate the Synthetic Similarity

According to the above two algorithms, the text similarity S_t and structural similarity S_c of the program code can be obtained. They should be given different weights W_t and W_c and weighted calculation to obtain a reasonable comprehensive similarity as the final output result. In order to ensure the accuracy of the results, and in combination with the actual situation of computer language courses, the following two criteria should be used to set the weight:

- In principle, it should be inclined to think that the code may be copied.
- Different means of plagiarism have different impacts on the text features and structural features of the code.

Listing 2. Normalize the AST

```
<'_ast.Module'>
  <'_ast.Assign'>
    <'_ast.Name'>
      <'_ast.Store'>
    <'_ast.Num'>
  <'_ast.Assign'>
    <'_ast.Name'>
      <'_ast.Store'>
    <'_ast.Num'>
  <'_ast.Assign'>
    <'_ast.Name'>
      <'_ast.Store'>
    <'_ast.BinOp'>
      <'_ast.Name'>
      <'_ast.Load'>
      <'_ast.Add'>
      <'_ast.Name'>
      <'_ast.Load'>
```

In this paper, the weights are given according to the two similarity results. According to the above criteria, we believe that the dimension with a higher similarity value is less affected by plagiarism means and should be given a higher weight. Based on this, the weights are set dynamically, and the final results are obtained by calculating $S_t \times W_t + S_c \times W_c$.

4. Experiment

In this paper, three groups of experiments were designed. According to the common situation of plagiarism classification, the sample codes were processed with different plagiarism hand segments, and the actual results of the algorithm proposed in this paper were tested with the control codes as the experimental input. In experiment 1 (section A), the variable name, function name, class name, and other information in the sample code were replaced and modified by changing the name to obtain the reference code of the sample code. In experiment 2 (section B), on the premise of ensuring the correctness and functional integrity of the code, the code sequence was modified to obtain the reference code. In experiment 3 (section C), the sample code was reconstructed by means of modification cycle and function disassembly/extraction. The output results of three experiments verify that the method proposed in this paper has a better detection effect on common plagiarism.

The code in the experiment takes Python code as an example. The code mainly comes from the student practice code on the MoocCode programming platform (http://code.mooctest.net) and the reference code in the comments of Leetcode (http://www.leetcode.com). In the experiment, a dynamic weight allocation strategy was designed for text similarity and structural similarity. If $S_t > S_c$ and $S_t > 0.6$, $W_t = 0.8$ and $W_c = 0.2$; otherwise, $W_t = 0.2$ and $W_c = 0.8$.

4.1. Experiment 1: Modify Identifiers

The experimental data were all derived from the practice codes of students on the MoocCode programming platform. Five sample codes were randomly selected, and all variable names, custom function names, and class names were modified. The input algorithm was used for analysis, processing, and calculation. The experimental results include text similarity results, structural similarity results, and the final similarity calculated by combining the two (the result is reserved as four decimal places).

The results of experiment 1 are shown in Table 1. The identifier code name is modified and does not cause changes to the structure of the program; it affects only the text similarity. The structure similarity of the control output is 1, and there are some fluctuations. The text similarity shows that the results conform to the actual and expected. The AST generated according to the code is completely consistent and conforms to the algorithm of AST in the design of standard processes.

4.2. Experiment 2: Modify the Code Order

The experimental data were all derived from the practice codes of students on the MoocCode programming platform. Five codes were randomly selected as samples, and the statement order, function definition, and call order in the sample codes were modified without affecting the functional integrity and code correctness.

Table 1. Results of experiment 1

Group	Text sim similarity	Structural similarity	Results
1	0.6667	1.0000	0.9333
2	0.7576	1.0000	0.9515
3	0.6970	1.0000	0.9394
4	0.5606	1.0000	0.9121
5	0.6154	1.0000	0.9231

According to the experimental results shown in Table 2, the artificial check sample code that is affected by the code of the original structure can be found. Modifying the code sequence may greatly influence the grammatical structure of the code. The result reflected in the experiment is that the structural similarity fluctuates greatly. Because the locally sensitive Simhash algorithm is used in the text similarity measurement algorithm, it still demonstrates an optimal and stable detection effect for code order modification. Therefore, the final weighted synthesis result is also optimal.

Table 2. Results of experiment 2

Group	Text sim similarity	Structural similarity	Results
1	0.9394	0.3699	0.8255
2	0.9194	0.9348	0.9317
3	0.9688	0.9237	0.9597
4	0.9242	0.6851	0.8643
5	0.9692	0.9290	0.9612

4.3. Experiment 3: Code Refactoring

Because simple refactoring requires a certain complexity of the program code itself, the experimental data part selects the answer code Shared in the exercise review of the Leetcode website. Without affecting the functional integrity and code correctness, the functions in the sample code are decomposed, the code block is extracted as a function, and the for loop is modified as while and other operations.

As shown in Table 3, code refactoring has a direct impact on the syntactic structure of code, but text similarity detection works well without adding or reducing too many identifiers. As can be seen from the results, even if the code is refactored, the structural similarity based on AST is still not very low. This is mainly because of the need to ensure the integrity and correctness of the function as well as the limitation of the code quantity itself, which may lead to the limited reconfigurable code. Code refactoring has a great impact on code changes. Compared with other methods, it requires higher ability of students. Therefore, the output similarity result is lower than in experiments 1 and 2.

Table 3. Results of experiment 3

Group	Text sim similarity	Structural similarity	Results
1	0.7879	0.5616	0.7426
2	0.6032	0.6000	0.6025
3	0.7879	0.8759	0.8583
4	0.8939	0.5581	0.8267
5	0.9077	0.7674	0.9027

5. Conclusions

Aiming at the phenomenon of plagiarism widely existing in the experiments of computer language courses, this paper proposes a similarity comparison algorithm that measures the text similarity and structural similarity of codes. The method makes full use of the text characteristics and structure of program code, generates codes using the Simhash text of fingerprints, calculates the Hamming distance, and establishes the AST to represent the structure information of the code, so as to measure structural similarity, choose a suitable strategy weighted calculation to get the final result, and guarantee the stability and accuracy of the method.

Through experimental verification, the method in this paper can be used to identify the common means of code plagiarism in computer language courses. In this way, the phenomenon of plagiarism can be reduced, and the teaching quality and effect can be guaranteed. The next step is to analyze and optimize the temporal and spatial efficiency of the algorithm, study and improve the efficiency of the method, and achieve good effects in practice in the face of a large number of student codes.

Acknowledgements

This work was partially supported by the Ministry of Education - Nanjing Mooctest Industry-University Cooperation, Collaborative Education (No. 201702083003).

References

1. G. Cosma and M. Joy, "Source-Code Plagiarism: A UK Academic Perspective", in Research Report RR-422, Department of Computer Science, University of Warwick, 2006
2. J. Sheard, M. Dick, S. Markham, L. MacDonald, and M. Walsh. "Cheating and Plagiarism: Perception and Practices of First Year IT Students," in *Proceedings of the 7th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education*, pp. 183-187, ACM Press, New York, 2002
3. J. A. W. Faidhi and S. K. Robinson, "An Empirical Approach for Detecting Program Similarity and Plagiarism within a University Programming Environment," Computers and Education, pp. 11-19, 1987
4. M. H. Halstead, "Elements of Software Science (Operating and Programming Systems Series)," 1978
5. K. J. Ottenstein, "An Algorithmic Approach to the Detection and Prevention of Plagiarism," *ACM SIGCSE Bulletin*, Vol. 8, No. 4, pp. 30-41, 1976
6. M. J. Wise, "YAP3: Improved Detection of Similarities in Computer Program and Other Texts," *ACM SIGCSE Bulletin*, pp. 130-134, 1996
7. P. Clough, "Plagiarism in Natural and Programming Languages: An Overview of Current Tools and Technologies," Research Memoranda Cs, 2000
8. G. S. Manku, A. Jain, and A. D. Sarma, "Detecting Near-Duplicates for Web Crawling," in *Proceedings of the 16th International Conference on World Wide Web*, pp. 141-150, ACM, New York, 2007
9. D. C. Atkinson and W. G. Griswold, "Effective Pattern Matching of Source Code using Abstract Syntax Patterns," *Software Practice and Experience*, Vol. 36, No. 4, pp. 413-447, 2006
10. K. Z. Zhang and D. Shasha, "Simple Fast Algorithms for the Editing Distance Between Trees and Related Problems," in SIAM Journal on Computing, pp. 1245-1262, 1989
11. K. L. Verco and M. J. Wise, "Software for Detecting Suspected Plagiarism: Comparing Structure and Attribute-Counting Systems," in Computer Science, University of Sydney, pp. 3-5, 1996