An Approach to Source Code Plagiarism Detection Based on Abstract Implementation Structure Diagram

Shuang Guo[1, 2, b], JianBin Liu[1, 2, a]

[1]School of Computer, Science Beijing Information Science & Technology University, 100101, Beijing, China
[2]Software Engineering Research Center，Beijing Information Science & Technology University, 100101, Beijing, China

Abstract. Source-code plagiarism detection in programming, concerns the identification of source-code files that contain similar and/or identical source-code fragments. Based on the analysis of the characteristics and defects of the existing program code similarity detection system, a method of source code similarity detection based on Abstract Implementation Structure Diagram (AISD) is proposed. The source code modelling and format into an abstract implementation structure diagram, and forming structural feature strings and variable reference relationship sequences by extracting structural features and variable position features. We calculate the overall similarity by calculating structural similarity and variable similarity. The results demonstrate that the performance of the proposed AISD-based approach overcomes other approaches on the same source code datasets, and reveals promising results as an efficient and reliable approach to source-code plagiarism detection.

1 Introduction

Plagiarism of source-code is a growing problem due to the growth of source-code repositories, and digital documents found on the Internet. In the field of computer science education, the phenomenon of students copying each other is widespread, which seriously affects the cultivation of students' abilities. About 33% of students in recent foreign studies admitted to having plagiarism[1]. Plagiarism has seriously affected the quality of computer science education. In order to curb bad academic style, research scholars have become increasingly necessary to study code plagiarism detection methods.

The Abstract Implementation Structure Diagram (AISD)[2] , as an external representation of the implementation layer of the process blueprint[2], uses the logic control constructs and operational expressions of the programming language to accurately represent the process control flow and data flow, including the generation process. Liang [3] has done a series of research work on repeated code detection based on process blueprint, pointing out the repeated code detection method based on process blueprint, avoiding the complicated process of transforming source code into suffix tree and reducing its complexity. The static information of the structure statement and the variable position in the program source code can be directly obtained by analyzing the node type of the abstract implementation structure diagram and the operation expression with the data stream.

2 Related work

There has been lot of work on plagiarism detection done by several researchers working in this domain. In this section, we summarize different types of approaches and tools that exist within the literature for plagiarism detection. There are three main categories of plagiarism detection approaches: attribute-based, structure-based.

As early as 1976, Ottenstein [4] used the basic Halstead metric, identified only four key properties $H(n_1,n_2,N_1,N_2)$. Faidhi et al. [5]introduced a minimum set of 23 different metrics. The indicators used include the average identifier length, the number of comment lines, the number of code blocks, the proportion of conditional statements, and more complex structural indicators such as the complexity of the McCabe circle. The attribute counting based measurement method does not consider the program structure information in the abstract process of the program code, and the false positive rate and the false negative rate of the detection result cannot be reduced by increasing the vector dimension [6].The structure-based approaches adds the internal structure of the program to the analysis and comparison. The common methods are the token-based method and the abstract syntax tree-based method. Plagiarism detection systems are JPlag[7], Sherlock[8], MOSS[9], Plaggie[10], XPlag[11], PGDT[12],. Because the structure tag string extracted by this method is too simplified and does not consider the statement information of the program, it cannot flexibly adapt to advanced code obfuscation methods such as expression splitting. Guo[13] used the

* Corresponding author: [a] 13126707629@163.com,[b] guoshuang1008@163.com

AST of lex and yacc constructors to calculate the hash value for each node by bottom-up cumulative operation and traversing the AST node, and the similarity by the node hash matching and matching node proportion. Resmi[14] used a modified grammar to construct an AST. The preamble traversed the AST to generate a sequence of nodes, and then used the Needleman-Wunsch algorithm and the LCS algorithm to measure the similarity.

To summarize, there are different tools and approaches in the literature to detect plagiarism in source code. Therefore, a more robust approach is necessary to handle these code transformations during plagiarism detection.

3 Detection process and algorithms

This section introduces an innovative computational intelligence framework for the purpose of analyzing source-code in the context of source-code plagiarism detection.

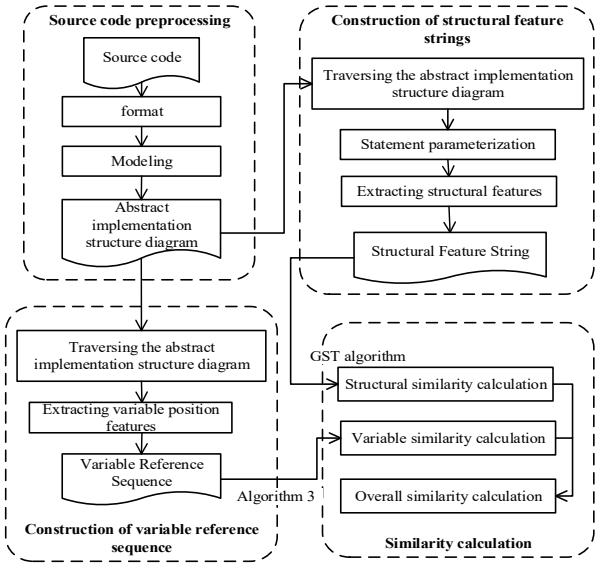

Figure 1. detection process framework

We implement our approach in four steps: In the first step we preprocess the source program. In the second step, we construct structural feature strings by extracting structural features from AISD. In the third step, we perform variable identification and extracting structural variable position features, where for each program statement, the algorithm 2 is used to identify and analyze variables in the program statement, constructing a sequence of variable references. Finally, the structural similarity is calculated by the GST algorithm, and the variable similarity is calculated by the algorithm 3. The overall similarity is calculated by structural similarity and variable similarity. Figure 1 details this process: In the following sections we describe our approach in detail. we describe our approach in detail.

3.1 Source Code Preprocessing

The operation object extracted by the program structure feature is an abstract implementation structure diagram of the process blueprint, which decomposes the program code into a process blueprint and takes its abstract implementation structure diagram view. Since the program statement feature contains the hierarchical relationship of the program action, it represents the positional relationship between the nodes in the abstract implementation structure diagram. The depth-first traversal can express the hierarchical relationship of the program statement and ensure the completeness of the program meaning, while the breadth-first traversal the nested structure of the program code will be lost. Therefore, the abstract implementation structure diagram is depth-first traversed, and the sequence of nodes is represented by parentheses.

3.2 Construction of structural feature strings

By transforming the program code formatting into an abstract implementation structure diagram, deep traversal the nodes and extracts structural features. The specific steps of the program conversion to the AISD are detailed in the literature [2] , and will not be elaborated in this paper. In an abstract implementation structure diagram with a statement expression that shows the implementation node of the data stream, statement expressions with the same variable type may have different variable names. This paper formalizes the structural features and introduces the parameter element parameterization process in the structural features of the extraction program. This paper considers the characteristics of the statement, the information of the statement element and the nesting relationship between the statements, to avoid the influence of the identifier on the grammatical structure of the statement. The resulting structural feature string is the basis of the comparison of the program structure.

Definition 1 (Structural Feature Strings**)** structural feature strings, including the hierarchical relationship between program statements and the sequence of statement features.

Algorithm 1 describes the construction process of a structure statement feature string based on an abstract implementation structure diagram. The sequence contains hierarchical relationships, and each node is operated as follows: Extract the control structure type, attribute, and serial number to get the control structure of the program (3-6 lines). The statement expression of the data flow of each node is processed, traverses each element of the statement expression, retrieves the variable dictionary, performs element-to-one mapping, and obtains the statement pattern (7-9 lines). The sequence of the type of the statement element in the control structure type and the statement mode attribute is represented as the node statement content, and the sequence formed by the parenthesis representation of the implementation node and the content of the node statement is the program structure feature string (11 line).

Algorithm 1 Construction of structural feature strings
Input: Abstract implementation map implementation node collection treeNode, symbol list.
Output: structural feature string strSeq [ ]

```
1:  Function GetStructString (treeNode,SymbolList)
2:      for (ele in treeNode)
3:          consType=getconsType (treeNode)//get the control structure type attribute through the symbol table and control structure type
4:          sValue=getsv (consType,SymbolList) //get the node code value of the node
5:          nNum=getnnumber (treeNode)
6:          codeop=GetAllcode (treeNode); //get the node's statement expression with data stream
7:          for code in codeop
              //Get statement mode
8:              codeF=codeizes (code);
9:          end for
10:     end for
11:
    strSeq[ ]=getstring(consType,sValue,nNum,codeF)
12:     return strSeq [ ]
13:  end function
```

3.3 Variable reference sequence construction

This paper uses the depth-first search algorithm to recursively traverse each node of the AISD to obtain some information we are interested in, such as type, number of sequence types, tags, number of positions, etc., and use the obtained information to construct a variable position set of variable position features. In this set, a variable pattern that satisfies the constraint is obtained, and a sequence of variable reference is constructed.

In a variable feature set, if the element type is a custom type in all elements of the set, the element belongs to the variable pattern set, otherwise it is not a variable pattern set. In the variable pattern set, if the types of the set elements are the same, the sequence formed by the position numbers of the set elements is a sequence of variable reference.

Definition 2 (Variable Reference Sequence) Let a sequence composed of position numbers of variable position features in the variable pattern, recorded as: $varSeq\ \{value_s, s \in [0, simvlen-1], s \in Z\}$ .The number of types of variable position features in the variable pattern set.

Algorithm 2 describes the construction of a sequence of variable reference based on an abstract implementation diagram. Algorithm 2 is mainly divided into three stages. The first stage is the stage of custom variable position recording. The variable pattern is constructed by obtaining the elements of the variable identifier in the variable positional feature (2-6 lines). The second stage is the *Scanmark* stage, which scans all the elements of the variable position feature set. If the type of the *j* element in the variable pattern *Str* is equal to the type of the variable position feature set element and the element is not marked. Put all of them above into the *varloc* and the current element is marked (8-20 lines). The third stage is the build phase, building the variable reference sequence *varSeq* (17-18 lines).

Algorithm 2: Construction of a variable reference position relation sequence
Input: abstract data stream token string, symbol list corresponding to similar nodes
Output: variable reference sequence varSeq [ ]

```
1:  Function getvarLoc (tkStr,SymbolList)// Get the identifier variables in the SymbolList for tkStr
2:    for (int i=0; i < tkStr.length; i++)
3:        if (Identifier (SymbolList_i))
4:        Str [ ].add(<i>);
5:        end if
6:    end for
7:    var←0;
8:    for (int j = 0; j < Str.length; j++)
9:        if (unmarked (Str_j)) //if not be marked
10:           marklist←Str_j;
11:           for (int k = 0; k < tkStr.length-Str_k-1; k++)
12:               if (Str_j in tkStr equals k+marklist in tkStr)
13:                   marked (k+marklist);
14:                   varloc←addvarloc (k+marklist);
15:               end if
16:           end for
17:           varSeq [ ].add (<var,varloc>);
18:           var←var+1;
19:       end if
20:   end for
21: return varSeq [ ]
22: end function
```

3.4 Similarity calculation

The overall similarity is calculated by the structural similarity and variable similarity of the program code, and the calculation formula is as shown in (1).The number of w_1 and w_2 values represent the weights in the comparison, and the overall similarity calculation formula is:

$$sim(X,Y) = w_1 \cdot structsim(X,Y) + w_2 \cdot \mathrm{var}\, sim(X,Y) \quad (1)$$

Where $structsim(X,Y)$ is the structural similarity of the program files X and Y, and $\mathrm{var}\, sim(X,Y)$ is the variable similarity of program files X and Y.

The GST algorithm[15] is a greedy string matching algorithm that finds the largest common substring of two strings by greedy search. The calculation formula of structural similarity is shown in (2), where |X|, |Y| is the length of the structural feature string in the program files X and Y, x, y is the starting position of the structural feature string. $Match(x, y, length)$ represents the same substring with x and y lengths of length.

$$\begin{cases} structsim(X,Y) = \dfrac{2 \times Coverage(tiles)}{|X| + |Y|} \\ Coverage(tiles) = \displaystyle\sum_{match(x,y,length) \in tiles} length \end{cases} \quad (2)$$

A sequence consists of an array of elements arranged in a regular order, and a sequence of strings is also an array of individual characters. The string matching calculation process is that a single character of a string sequence is equivalent to a single character of another string sequence, and the variable reference sequence also

has the same properties and operation methods.

Definition 3 (Variable Similarity) variable similarity refers to the ratio of the matching length of a variable reference sequence to the product of 2 and the sum of the lengths of the two sequences. The calculation formula of variables similarity is shown in (3), where |X|, |Y| is the length of the Variable reference sequence in the program files X and Y and *maxslen* is the method for solving the maximum number of matching elements in two sets of sequences.

$$\text{var}\,sim(X,Y)=\frac{2\times\max slen(X,Y)}{|X|+|Y|} \tag{3}$$

Algorithm 3 is a calculation process of variable similarity. It is mainly divided into three phases. The first phase is the construction phase. The variable position sequence is constructed by the node information of the data flow, and the same variable reference relationship sequence set *varSeq* of the custom identifier in the program statement is obtained (2-9 lines). The second phase is the comparison phase, comparing whether each element of the sequence set is equal (12-16 lines). The third stage is the calculation phase, which calculates the variable similarity by comparing the number of identical elements sim (17 line).

Algorithm 3: Calculating Variable Similarity
Input: the list of data flow nodes corresponding to the abstract implementation diagram
Output: Variable similarity VarSim

```
1:  Function createnode (CodeNodeA,CodeNodeB)
2:    for (int i = 0; i < CodeNodeA.length; i++)
3:          tokenStringA←getNodeValue (CodeNodeAi);
4:          SymbolListA←getNodeType (CodeNodeAi);
5:          tokenStringB←getNodeValue (CodeNodeAi);
6:          SymbolListB←getNodeType (CodeNodeAi);
7:    end for
8:    varSeqA← getVarLoc (tokenStringA,SymbolListA);
9:    varSeqB←getVarLoc (tokenStringB, SymbolListB);
10:  sim←0;
11:  nosim←0;
12:  for (int var = 0; var < varSeqA.size; var++)
13:         if (varSeqAvar equals varSeqBvar)
14:         sim++;
15:         end if
16:  endfor
17:
varSim←sim*2/(CodeNodeA.length+CodeNodeB.length);
18:  return varSim
19:  end function
```

4 Experiments

4.1 Datasets and Metrics

The proposed AISD-based system was tested on two Java source-code datasets. These datasets are described in subsection 4.1.1. The performance of the proposed is evaluated against the JPlag by means of the evaluation measures described in subsection 4.1.2.

4.1.1 Datasets

The evaluation assembly consists of two Java source-code datasets A and B.Basic information about the data set is shown in Table 1.

Table 1. The Datasets

Name	A	B
size in number of programs	25	23
size in number of program pairs	300	253
number of pairs that are plagiarism pairs	19	19
fraction of plagiarism pairs	6.3	7.5
the average number of variables	480	430
Average number of functions	6	5
Average number of code lines	90	85

4.1.2 Performance evaluation measures for plagiarism detection

Due to the fact that the similarity values provided by these approaches are not directly comparable, Table 1 shows the characteristics of each dataset, where performance evaluation measures for plagiarism detection. This section describes the performance evaluation measures for comparing the performance of the proposed. In order to verify the validity of the test results, the measurement methods proposed in [7] were combined with manual methods for analysis. The evaluation will mostly be based on the measures “precision” (P), “recall” (R) and “F-measure” (F), defined as follows.

Assume we have a set of n programs. This set allows to form $p=n\cdot(n-1)/2$ pairs. Assume further that g of these pairs are plagiarism pairs, i.e., one program was plagiarized from the other or both were (directly or indirectly) plagiarized from some common ancestor that is also part of the program set.Now assume that we let our fully automatic plagiarism detector run and it returns f pairs of programs flagged as plagiarism pairs. If t of these pairs are really true plagiarism pairs and the other are not, then we define precision and recall and F-measure as $P=t/f$, $R=t/g$, $F=\frac{2\times P\times R}{P+R}$ that is, precision is the percentage of flagged pairs that are actual plagiarism pairs and recall is the percentage of all plagiarism pairs that are actually flagged. F-measure is a measure of the performance of the system, related to precision and recall.

4.2 Experiment result

This section describes the results from the experiments performed on two datasets. The test results are shown in Table 2 and Table 3.The retrieved pairs are obtained by the plagiarism system when the similarities of the pairs exceed a defined cut-off threshold (CT). The CT separates plagiarized program pairs from non-plagiarized ones. If the similarity value of two programs is larger than cut-off threshold value CT, then the program pair is marked as suspect. We used CT values from 10% to 90%.The CT value is an artificially set threshold value for determining whether or not to copy. The last behavior is the average of the P, R and F values of

different CT values.

For the P and F values, the detection result of the AISDS system is higher than that of the JPlag system.Especially for assembly B with simple structure, the average precision of JPLAG is 36.3%. To further explore the reasons for this, let the two students complete the output separately " Two Exchange " simple program, although the two have no plagiarism, but the similarity detected by JPLAG is 100%, which means that even if the CT value is set to 90, it will be misjudged as plagiarism. The root cause is this program. The structure and logic are very simple, and there are not many variable definitions. If only the structural similarity is analyzed, it will definitely lead to misjudgment. Using the system analysis of this paper, the structural similarity and variable similarity are 55.48% and 45%, respectively, and the overall similarity is 48.93%, so that an accurate CT result can be obtained by setting an appropriate CT value, such as CT = 50.

Table 2. Datasets A results

CT	AISDs-A			JPLAG-A		
	P	R	F	P	R	F
90	100	31.58	48	60	31.58	41.38
80	100	68.42	81.25	76.47	68.42	72.22
70	100	78.95	88.24	78.95	78.95	78.95
60	100	84.21	91.43	62.07	94.74	75
50	82.61	100	90.48	61.29	100	76
40	59.38	100	74.51	59.38	100	74.51
30	42.22	100	59.38	59.38	100	74.51
20	42.22	100	59.38	54.29	100	70.37
10	42.22	100	59.38	43.18	100	60.32
Mean	74.29	84.80	72.45	61.67	85.96	69.25

Table 3. Datasets B results

CT	AISDs-B			JPLAG-B		
	P	R	F	P	R	F
90	100	53	69.3	53	95	68
80	100	100	100	54	100	70.1
70	100	100	100	54	100	70.1
60	100	100	100	28	100	43.8
50	52	100	68.4	28	100	43.8
40	28	100	43.8	28	100	43.8
30	28	100	43.8	28	100	43.8
20	28	100	43.8	28	100	43.8
10	28	100	43.8	27	100	42.5
Mean	62.7	94.8	75.5	36.3	99.5	53.2

5 Conclusion and future work

This paper proposes a method for detecting code similarity based on abstract implementation structure diagram. The experimental results show that the above method can effectively detect plagiarism between program pairs, avoiding the misjudgment caused by previous detection methods and improving the precision value. The method is transformed into an abstract implementation structure diagram and the feature quantization calculation amount is large, and the subsequent work can continue to study how to improve the efficiency of the method.

Acknowledgments

This work was supported by the Information+ Discipline Construction Project (5111823414) and the Science Research Level Improvement Project (5211823406) of Beijing Information Science & Technology University.

References

1. G. Cosma and M. S. Joy, “Towards a Definition on Source-Code Plagiarism,” IEEE Trans. Educ., (2008).
2. L. Jianbin, *Process blueprint design methodology*. Beijing：Science Press, (2005).
3. L. Jianzhong, L. Jianbin, and Y. Chuying, “Research on Parametric Repetitive Code Detection Technology Based on Process Blueprint,” J. Shantou Univ. Nat. Sci. Ed., **22**, 1, 54–59, (2007).
4. K. J. Ottenstein, An algorithmic approach to the detection and prevention of plagiarism. ACM, 1976.
5. J. A. W. Faidhi and S. K. Robinson, “An empirical approach for detecting program smilarity within a university programming environment,” Comput. Educ., **11**, 11–19, (1987).
6. K. L. Verco and M. Wise, “Software for Detecting Suspected Plagiarism: Comparing Structure and Attribute-Counting Systems,” 81-88, (1996).
7. L. Prechelt, G. Malpohl, and M. Philippsen, “Finding Plagiarisms among a Set of Programs with JPlag,” J. Univers. Comput. Sci., **8**, 11, 1016–1038, (2002).
8. M. Joy and M. Luck, “Plagiarism in programming assignments,” Educ. IEEE Trans., **42**, 2, 129–133, (1998).
9. S. Schleimer, D. S. Wilkerson, and A. Aiken, “Winnowing: Local Algorithms for Document Fingerprinting,” Proc. ACM SIGMOD Int. Conf. Manag. Data, **10**, 76–85, (2003).
10. A. Ahtiainen, S. Surakka, and M. Rahikainen, “Plaggie: GNU-licensed source code plagiarism detection engine for Java exercises,” in Proceedings of the 6th Baltic Sea conference on Computing education research Koli Calling 2006 - Baltic Sea ’06, 141, (2006).
11. C. Arwin and S. M. M. Tahaghoghi, “Plagiarism Detection across Programming Languages,” in Proceedings of the 29th Australasian Computer Science Conference, **48**, 277–286, (2006).

12. G. Cosma and M. Joy, “An Approach to Source-Code Plagiarism Detection and Investigation Using Latent Semantic Analysis,” IEEE Trans. Comput., **61**, 3, 379–394, (2012).
13. G. Tao, D. Guowei, Q. Hu, and C. Baojiang, “Improved Plagiarism Detection Algorithm Based on Abstract Syntax Tree,” in Proceedings of the 2013 Fourth International Conference on Emerging Intelligent Data and Web Technologies, 714–719, (2013).
14. Resmi NG and Soman KP, “Abstract Syntax Tree Generation using Modified Grammar for Source Code Plagiarism Detection,” International Journal of Computing and Technology, **1**, 6, (2014).
15. M. J. Wise, “String similarity via greedy string tiling and running Karp-Rabin matching,” Basser Dep. Comput. Sci. Tech. Report, Sydney Univ. (1993).