Hindawi
Scientific Programming
Volume 2020, Article ID 8835310, 15 pages
https://doi.org/10.1155/2020/8835310

Research Article

Flowchart-Based Cross-Language Source Code Similarity Detection

Feng Zhang,[1,2] **Guofan Li**,[1] **Cong Liu**,[3] **and Qian Song**[1]

[1]*College of Computer Science and Engineering, Shandong University of Science and Technology, Qingdao 266590, China*
[2]*Shandong Key Laboratory of Wisdom Mine Information Technology, Qingdao 266590, China*
[3]*School of Computer Science and Technology, Shandong University of Technology, Zibo 255000, China*

Correspondence should be addressed to Cong Liu; liucongchina@sdut.edu.cn

Received 18 July 2020; Revised 11 November 2020; Accepted 1 December 2020; Published 18 December 2020

Academic Editor: Jianping Gou

Source code similarity detection has various applications in code plagiarism detection and software intellectual property protection. In computer programming teaching, students may convert the source code written in one programming language into another language for their code assignment submission. Existing similarity measures of source code written in the same language are not applicable for the cross-language code similarity detection because of syntactic differences among different programming languages. Meanwhile, existing cross-language source similarity detection approaches are susceptible to complex code obfuscation techniques, such as replacing equivalent control structure and adding redundant statements. To solve this problem, we propose a cross-language code similarity detection (CLCSD) approach based on code flowcharts. In general, two source code fragments written in different programming languages are transformed into standardized code flowcharts (SCFC), and their similarity is obtained by measuring their corresponding SCFC. More specifically, we first introduce the standardized code flowchart (SCFC) model to be the uniform flowcharts representation of source code written in different languages. SCFC is language-independent, and therefore, it can be used as the intermediate structure for source code similarity detection. Meanwhile, transformation techniques are given to transform source code written in a specific programming language into an SCFC. Second, we propose the SCFC-SPGK algorithm based on the shortest path graph kernel to measure the similarity between two SCFCs. Thus, the similarity between two pieces of source code in different programming languages is given by the similarity between SCFCs. Experimental results show that compared with existing approaches, CLCSD has higher accuracy in cross-language source code similarity detection. Furthermore, CLCSD cannot only handle common source code obfuscation techniques used by students in computer programming teaching but also obtain nearly 90% accuracy in dealing with some complex obfuscation techniques.

1. Introduction

Since the 1970s, the source code similarity detection technique has attracted the attention of global researchers, and it has been widely used in the source code plagiarism detection in computer programming teaching and code intellectual property protection [1]. At present, there are mainly two kinds of code similarity detection approaches: attribute counting [2, 3] and structure metrics. Among them, structure metrics is the most commonly used approach that mainly includes string-based, tree-based, and graph-based code similarity detection approaches. In recent years, with the emergence of automatic code conversion tools (https://www.tangiblesoftwaresolutions.com), the similarity detection of cross-language source code poses a new research challenge. With the extensive applications of OJ (Online Judge) [4] in computer programming teaching, these tools are often used to copy programming assignments by students. Most of the existing cross-language code similarity detection approaches are based on token or tree-based intermediate representation. However, some complex code obfuscation techniques can decrease the similarity of source code. For example, changing the loop structure and adding redundant statements may affect the detection effectiveness of existing approaches [5].

Programming assignments in computer programming teaching are generally simple and short. As a result, for the given original code and plagiarized code written in different

programming languages, no matter what kind of code transformation or obfuscation techniques is adopted [6], their core processes are highly similar if their programming ideas are the same. This circumstance is close to type IV clones [7]. Therefore, aiming at the cross-language source code similarity detection in the teaching of computer programming, we propose a cross-language source code similarity detection approach named CLCSD (cross-language code similarity detection) based on code flowcharts. In this approach, source code written in different programming languages is transformed into corresponding flowcharts, and then the similarity of code is obtained by measuring the similarity between their flowcharts. Specifically, first of all, for two source code fragments written in different programming languages, there may be some differences between their flowcharts that are directly transformed by current code conversion tools even though they have the same processes. This is because the flowcharts obtained by the existing code flowchart conversion approaches and tools are strongly correlated with the syntax of the programming language. Therefore, we propose a standardized code flowchart (SCFC) model based on the code flowchart (CFC) and the program dependency graph (PDG) [8]. SCFC standardizes the code flowcharts of different languages. In addition, it is suitable for dealing with the most common code obfuscation techniques in programming assignments. Next, the approach of transforming a source code fragment of a specific programming language into a SCFC is given. Finally, the SCFC-SPGK algorithm based on the shortest path graph kernel (SPGK) [9] for measuring the similarity between two SCFCs is proposed to calculate the similarity between two code fragments written in different languages.

The main contributions of this paper are as follows: (1) a cross-language source code similarity detection (CLSCSD) approach is proposed based on standardized code flowcharts; (2) a standardized code flowchart model SCFC that is independent of programming language and suitable for code similarity detection is proposed; (3) a cross-language source code similarity detection algorithm SCFC-SPGK based on SPGK is proposed. In addition, the effectiveness of CLCSD in cross-language source code similarity detection is verified from the perspective of the accuracy and the ability of defeating the code obfuscation techniques through real datasets. Meanwhile, it is verified that this approach is also suitable for the similarity detection of source code in the same programming language.

The rest of this paper is arranged as follows. First, the related work of similarity detection of the source code written in the same language and different languages is introduced in Section 2. Section 3 introduces the basic idea and the core framework of the CLCSD. Next, the SCFC model and the way of converting a flowchart transformed from a specific programming language into a SCFC are given in Section 4. In Section 5, the code similarity calculation based on SCFCs is introduced in detail. In Section 6, the effectiveness of the proposed approach is evaluated through experiments. Finally, we conclude this paper in Section 7.

2. Related Work

Most existing source code similarity detection approaches measure the similarity between two source code fragments written in the same programming language. Meanwhile, there is some work in terms of cross-language source code similarity detection. Therefore, we first introduce the approaches of source code similarity detection in the same programming language in this section, and then we present the main work of existing cross-language code similarity detection.

2.1. Source Code Similarity Detection in the Same Language. There are mainly two kinds of source code similarity detection approaches: attribute counting and structure metrics. In the early days, the approaches of attribute counting mainly focus on how to get the measurable attributes of code, such as the number of distinct operators and distinct operands. However, the detection effectiveness of these approaches is poor because they ignore too much structure information of the code. At present, the approaches based on structure metrics are most commonly used in code similarity detection, which mainly includes the approaches based on strings [10–13], trees [14–19], and graphs [8, 20–23].

2.1.1. Source Code Similarity Detection Based on Strings. String-based detection approaches measure code similarity from the perspective of text structure and the lexical features of source code. The most widely used approaches are based on text strings [10, 24] and tokens, such as CPDP [11], SIM [12], and JPlag [13]. The former converts the source code into a string sequence and then measures the similarity based on the sequence. The latter converts the word symbols in the source code into hexadecimal tokens and measures the similarity based on the token sequences.

2.1.2. Source Code Similarity Detection Based on Trees. Tree-based detection approaches construct a parse tree [14, 15] or an abstract syntax tree (AST) [16–18] of the source code by lexical and syntax analysis. The parse tree focuses on syntax, while the abstract syntax tree focuses on logic. This kind of approach measures the similarity by matching subtrees or vectors that are transformed from the tree structures [19].

2.1.3. Source Code Similarity Detection Based on Graphs. The code similarity detection approaches based on graphs mainly use the program dependency graph (PDG) [8, 21–23] and the control flow graph (CFG) [20, 22]. A PDG reflects the logical structures of code, including the control dependency and data dependency between statements. The approaches based on PDGs measure the code similarity by matching the isomorphic subgraphs. A CFG reflects the control structures of source code. The approaches based on CFGs measure the code similarity by matching the paths in the CFGs. In addition, some approaches [8, 19] combine AST and PDG to detect the code similarity.

The above three kinds of approaches are used to measure the similarity of source code written in the same programming language. However, these approaches are not suitable for cross-language code similarity detection because of the syntax differences between different programming languages.

2.2. Source Code Similarity Detection in Different Languages. At present, there are mainly three kinds of approaches for cross-language source code similarity detection.

2.2.1. Cross-Language Source Code Similarity Detection Based on Intermediate Language. The main idea of these approaches is to convert the code written in different languages into common intermediate language code, such as RTL (Register Transfer Language) [25] and CIL (Common Intermediate Language) [26]. Then, these approaches measure the similarity of code written in different languages by converting the intermediate language code into tokens or directly comparing the intermediate language code. This kind of approach ignores too much structure information of the source code. In addition, some approaches have limitation for using. For example, the approaches using CIL only works with the Microsoft.Net languages.

2.2.2. Cross-Language Source Code Similarity Detection through Tree-Based Intermediate Representation. The main idea of these approaches is to convert the source code written in different languages into common tree structures, such as eCST (enriched concrete syntax tree) [5], AST [27, 28], and CodeDOM (Code Document Object Model) [29]. Then, the tree structures are converted into token sequences or vectors to improve the efficiency of similarity measure. In addition, Nafi et al. [30] combine the approaches of AST and attribute counting to detect the similarity of cross-language source code. However, the intermediate representation based on trees cannot represent the logical structure of the source code completely, such as the loop structure. Meanwhile, these approaches cannot defeat the complex obfuscation techniques, such as adjusting the order of statements and adding redundant statements [5].

2.2.3. Cross-Language Source Code Similarity Detection Based on NLP (Natural Language Processing). Some approaches utilize NLP to detect the similarity of cross-language source code. These techniques mainly include n-gram model [31], LSA (Latent Semantic Analysis) [32], BOW (Bag of Words) [33], component analysis, and multiple logistic regression models [34]. These approaches also ignore the structural features of the source code. Although the approach proposed in [28] combines the AST and LSTM to detect the similarity between Java and Python code, they are greatly affected by some complex obfuscation technologies, e.g., the commonly used adding redundant statements. Meanwhile, this kind of approach needs to train their models with a lot of code rather than detecting the code similarity directly.

3. Framework of CLCSD

3.1. Basic Idea. A code flowchart can express the execution flow of an algorithm clearly and intuitively, and therefore, it is an essential tool for analyzing and designing an algorithm. For the code assignments submitted by students in the teaching of computer programming, existing common code obfuscation techniques commonly cannot modify the core process of the code. Therefore, the similarity between two pieces of code can be measured by comparing their corresponding processes that are expressed by flowcharts. Some existing tools can directly convert a source code into its corresponding flowchart. However, a flowchart generated by these tools is usually closely related to the syntax features of the programming language that the code is written in. As a result, the flowcharts generated from source code written in different languages by existing conversion tools are different due to the differences in the basic syntax of different programming languages. Figures 1(a) and 1(b) show a piece of Java code and Python code for finding all prime numbers between 100 and 200. These two code fragments are written based on the same idea, and their corresponding flowcharts directly generated by the Visustin (https://www.aivosto.com/visustin.html) tool are shown in Figures 1(c) and 1(d). We can see that there are some differences between two flowcharts. Thus, the code similarity calculated directly based on these two flowcharts cannot reflect the real similarity between these two pieces of source code.

Therefore, we give a standardized flowchart model that is independent of the specific programming language to solve this problem. Then, we take the standardized flowchart as the basis of similarity measure for the source code written in different languages. Finally, we present the standardized flowchart similarity measure approach to measure cross-language source code similarity.

3.2. Overall Framework. The overall framework of the proposed approach is shown in Figure 2. For two code fragments written in different programming languages, the whole process of calculating their similarity is divided into two steps.

First, two SCFCs are generated according to these two code fragments. In this step, there are three substeps. The code is preprocessed based on a PDG first to remove redundant statements. Then, the flowcharts of code (code flowchart (CFC)) are generated using existing conversion approaches according to the source code. Finally, two CFCs are standardized and converted into two SCFCs according to the definition of the SCFC in this study.

Second, the similarity between two SCFCs is calculated using the similarity measure algorithm based on graphs. Finally, the value is taken as the similarity between these two code fragments.

4. SCFC and Standardization

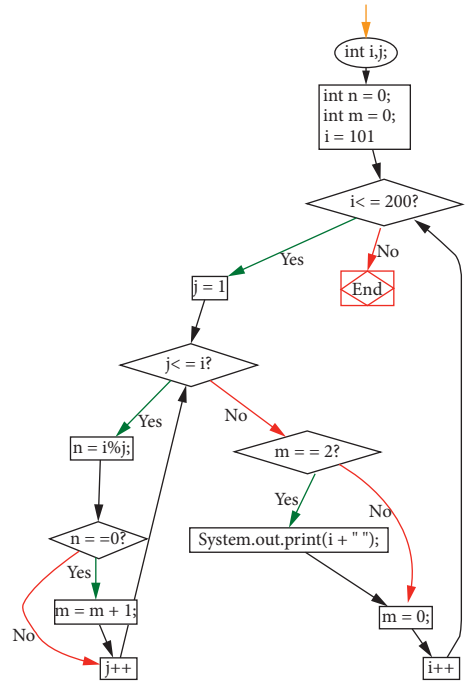
There are a few kinds of flowcharts, such as algorithm flowcharts, data flowcharts, code flowcharts, and system flowcharts. The proposed SCFC belongs to the code

```
int i,j;
int n = 0;
int m = 0;
for(i = 101;i< = 200;i++){
   for(j = 1;j< = i;j++){
      n = i%j;
      if(n = = 0) {
        m = m+1;
      }
   }
   if(m = = 2){
      System.out.print(i+" ");
   }
   m = 0;
}
```

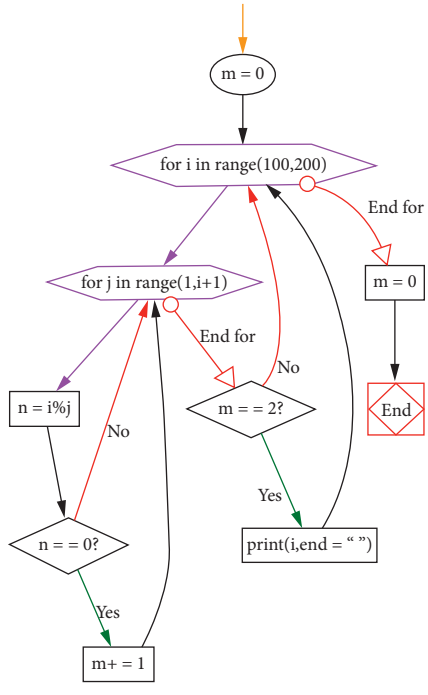
(a)

```
m = 0
for i in range(100,200):
   for j in range(1,i + 1):
      n = i%j
      if n = = 0:
         m + = 1
   if m = = 2:
      print(i,end = " ")
   m = 0
```

(b)



(c)



(d)

Figure 1: Flowcharts converted from Java and Python code using Visustin; (a) Java code example; (b) Python code example; (c) code flowchart of (a); (d) code flowchart of (b).

flowchart. We first introduce the SCFC model based on our previous work [35]. Then, we give the way of converting a piece of source code into its corresponding SCFC.

4.1. SCFC. In this section, we first introduce the node, edge, and structure of SCFC. Then, we give the definition of SCFC.

4.1.1. SCFC Node. A node is a basic element of a flowchart. According to the statements in the source code, we give nine types of SCFC nodes:

(1) declare: a statement that declares a variable

(2) assign: a variable assignment statement, such as = , + = , + +, and − −

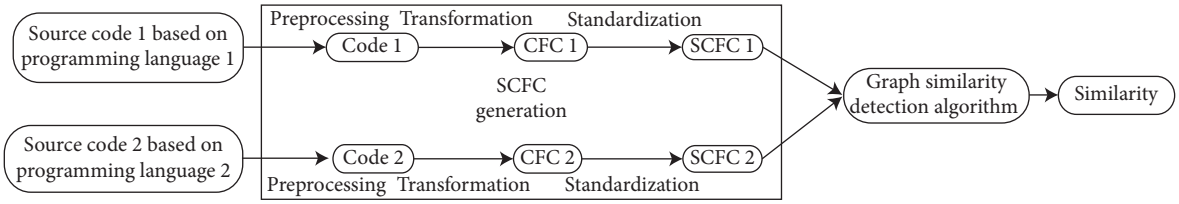


Figure 2: Overall framework of CLCSD.

(3) loop: a repeated statement, such as for, while, and do while

(4) jump: a process jump statement, such as goto, break, continue, and other statements used to implement a process jump in a process of circular execution

(5) call site: a function call statement

(6) return: a statement that returns the value of a function

(7) control: a branch statement, such as if, else, and switch

(8) output: an output statement

(9) combine: merging of adjacent declare or assign nodes in a flowchart that have no data dependency

As mentioned above, SCFC is a model that expresses the standardized flowchart of a piece of source code. It is also the basis for calculating the code similarity. Therefore, the definition of SCFC should consider the requirements of defeating code obfuscation techniques used in code similarity detection. Exchanging statement orders is a commonly used code obfuscation technique. In this study, we extend SCFC with a combine node to defeat this obfuscation technique. Specifically, adjacent declare or assign nodes that have no data dependency in the same basic block [20] are merged into a combine node. Figure 3 gives an example. The nodes int p and q = 0 correspond to the adjacent declare and assign nodes, which are independent and can be combined into a combine node.

4.1.2. SCFC Edge. There are two kinds of SCFC edges in a SCFC model.

Definition 1. (sequential execution edge (SEE)). For SCFC nodes v_1 and v_2, if the statement corresponding to v_2 is executed immediately after the statement corresponding to v_1, there is a sequential execution edge from v_1 to v_2.

Definition 2. (control dependency edge (CDE)). For SCFC node v_1 that belongs to a *loop* or *control* node type, if the value of condition in v_1 controls whether a basic block executes while v_2 is the first node of the block, then there is a control-dependent edge from v_1 to node v_2. The control dependency edge has two properties: CDE-Y indicating that the edge meets the control condition and CDE-N indicating that the edge does not meet the control condition.

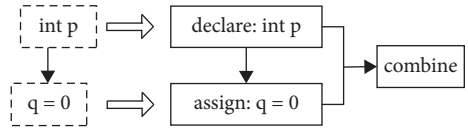


Figure 3: A combine node example.

4.1.3. SCFC Structure. In Figure 1, the control structure and loop structure of flowcharts, which are transformed from source code written in different languages, may also be different due to syntax differences of programming languages. Therefore, we propose the standardized structure of SCFC.

Based on the three basic structures sequence, branch, and loop in the flowchart, we give three standardized structures of SCFC: (1) SS: sequential structure; (2) BS: branch structure; (3) LS: loop structure. These three structures are shown in Table 1.

4.1.4. Definition of SCFC. Based on the definition of the SCFC node and edge, the definition of SCFC is as follows.

Definition 3. standardized code flowchart (SCFC)). $\mathrm{SCFC_p} = (V, E, T_V, T_E, \mu, \delta)$ is the standardized code flowchart of a piece of code p, where,

(i) V is the set of nodes and $E \subseteq V \times V$ is the set of edges

(ii) T_V = {assign, declare, control, loop, jump, call site, return, output, combine} is the set of node types

(iii) T_E = {SEE, CDE} is the set of edge types

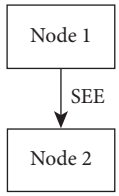
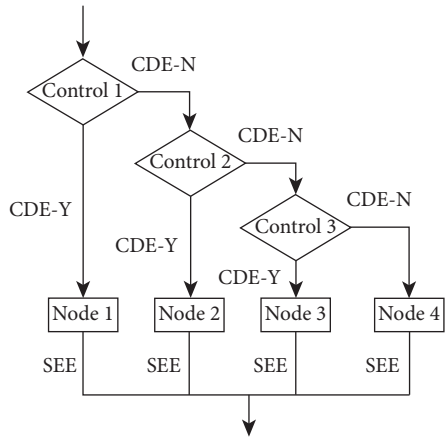
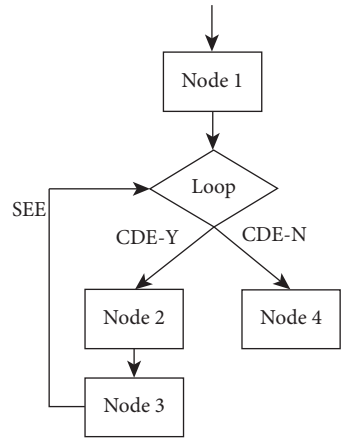
(iv) μ: $V \longrightarrow T_V$ is the function of assigning a node type $t_v \in T_V$ to a node $v \in V$

(v) δ: $E \longrightarrow T_E$ is the function of assigning an edge type $t_e \in T_E$ to an edge $e \in E$

4.2. SCFC Conversion for Language-Specific Flow Charts. There are two steps in converting a piece of code in a specific programming language into a SCFC. First, the code is preprocessed to deal with redundant statements. Then, the preprocessed code is converted into a flowchart by existing tools, which is closely correlated with the syntax of the programming language. Finally, the flowchart is converted into the corresponding SCFC.

4.2.1. Source Code Preprocess Based on PDGs. Redundant statements may exist in a piece of code. In addition, adding redundant statements is also a common obfuscation

Table 1: The structures of SCFC.

Structure types	SS	BS	LS
Structure charts			

technique because redundant statements increase the number of nodes and edges in the generated flowchart. As a result, the code similarity precision based on the flowchart is affected. Take the code in Figure 4(a) for example. Given a code fragment that calculates the largest value of two numbers, the code contains redundant statements in the third and the sixth line. Figure 4(b) shows the corresponding PDG of the code. We can see that the declared variables and assignments in the third and sixth lines have no data dependencies on the *return* statements. That is, the code in the third and sixth lines is redundant. We preprocess the source code by the traditional PDG to deal with redundant statements and obtain a SCFC that can reflect the real process of the code. Given a piece of code, the preprocessing steps are as follows:

(1) The source code is converted into a PDG.

(2) The origin and end points of all edges in the PDG are exchanged.

(3) Deep traversal in the PDG from the nodes corresponding to the output statement and return statement (the traversed node will not repeat the traversal) is executed. Then, a new directed graph G_{new} is obtained.

(4) The code corresponding to graph nodes that are not in G_{new} is removed from the source code

Figure 5 shows the preprocessing result of the code in Figure 4 according to the above steps.

Next, the preprocessed code needs to be transformed into its corresponding SCFC. The details are described as follows.

4.2.2. Transformation of SCFC. Learning from the node definition in PDGs, a node in a SCFC needs not to embody specific code statements for source code similarity measure. For the flowchart obtained by the existing approaches and tools (we call it the original flow chart), the nodes in it can be mapped to the nodes of a SCFC one by one according to their types. Similarly, edges in the original flowchart can be mapped to SEE, CDE-Y, and CDE-N according to their types.

For two pieces of source code, if they use different types of loop structure or different forms of the same loop structure, the loop structure in the corresponding flow chart may also be different [35]. Therefore, it is necessary to transform both the loop structure of different types and different forms of the same loop structure into a standardized LS. An example of the transformation of the loop structure is shown in Figure 6.

Similarly, the branching structures, including if, if...else, if...else if, and switch case, are all converted to the BS structure of SCFC.

Finally, the corresponding SCFC of a code fragment can be obtained based on the mapped nodes, edges, and standardized structures. Take the Java and Python code in Figure 1 for example; Figure 7 shows their SCFCs. We can see that the code written in two different languages with the same idea has a high similarity in terms of the SCFC.

5. Similarity Measure of SCFCs

A SCFC is a directed graph. Therefore, we can apply graph similarity calculation algorithms to measure the similarity of two SCFCs. The similarity measure based on the graph kernel is one of the most important approaches in the research of graph similarity measure, which include graph edit distance-based, tree-based, and path-based approaches. First, the approach based on graph editing distance [36] is inefficient because its time complexity increases exponentially with the number of vertices. Second, the tree-based similarity measure approaches are mainly used to calculate the similarity between directed acyclic graphs. Meanwhile, the similarity measurement complexity of trees is lower than that of graphs [37, 38]. However, the circle is one of the important structures in the structures of the source code. These approaches are suitable for directed acyclic graphs, which limits its application in the graph-based code similarity detection. Finally, the similarity measure approaches based on paths determine the similarity of two graphs by

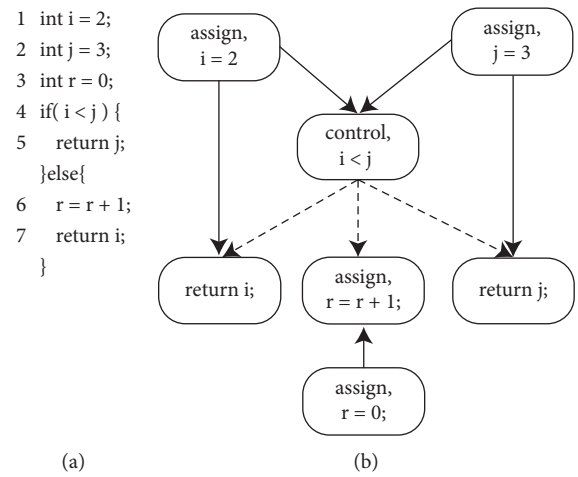


Figure 4: (a) The example code fragment and (b) the PDG of the example code fragment.

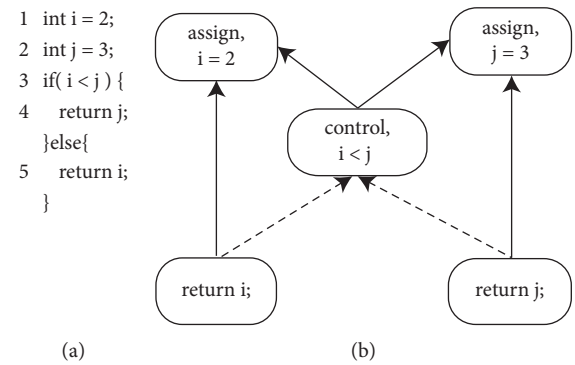


Figure 5: The code fragment after removing redundant statements. (a) The example code fragment after preprocessing; (b) the PDG of code fragment in (a).

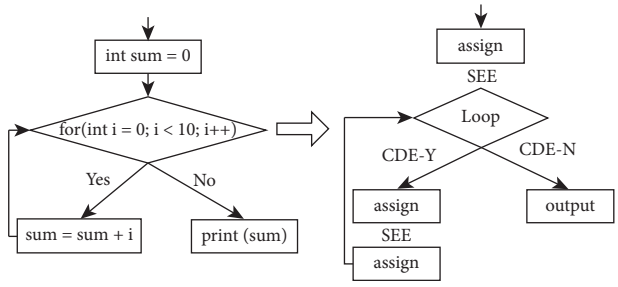


Figure 6: The standardization of LS.

comparing the number of common paths in these two graphs. These approaches mainly include the random walk graph kernel (RWGK) [39, 40], the common path graph kernel (CPGK) [41], and the shortest path graph kernel (SPGK) [9, 42]. RWGK considers the same vertex in the path, which leads to the tottering phenomenon [43] and affects the similarity of graphs. Moreover, this method does not take the similarity of the node tags into account for the graph similarity measure [44]. CPGK mainly solves the similarity measure of directed acyclic graph [41], which is unsuitable for measuring the similarity of SCFCs. SPGK does not consider repeated traversal of the same edges in similarity measure. As a result, it can avoid the tottering phenomenon. However, its time complexity is slightly higher, which is O (n^4).

A SCFC can be represented as a directed cyclic graph with a root node. Considering the validity and time complexity of SCFC similarity measure, we choose the SPGK as the basis of SCFC similarity measure. Combining with the features of nodes and edges in the SCFC, we propose the SCFC-SPGK algorithm for the similarity calculation of SCFCs.

5.1. SCFC-SPGK Algorithm. The SPGK algorithm is first introduced in this section, and then, the SCFC-SPGK algorithm based on SPGK is presented in detail.

SPGK first uses the Floyd–Warshall algorithm [41, 44] to find the distance between any two vertices in the graph based on the adjacency matrix of the graph. Assume that A_1 and A_2 are the weighted adjacency matrices of the graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$, respectively. The shortest distance between any two vertices in the graphs is obtained by the Floyd–Warshall algorithm, and the shortest path matrices A'_1 and A'_2 and the transformed graphs $R_1 = (V'_1, E'_1)$ and $R_2 = (V'_2, E'_2)$ are obtained by combining A_1 and A_2. The

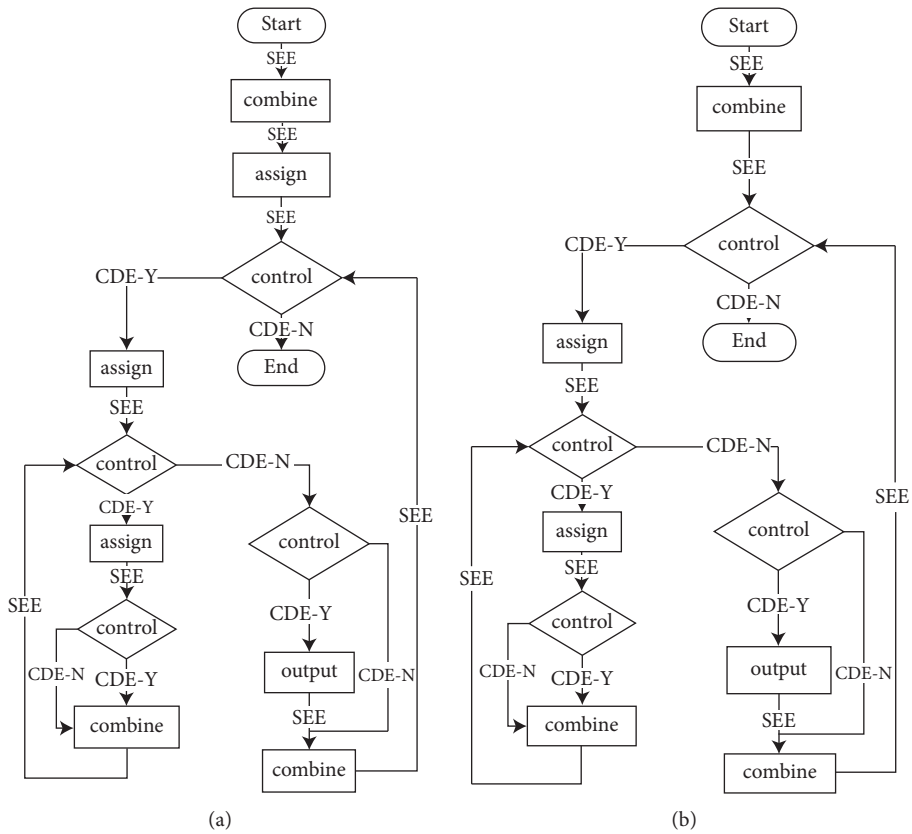


Figure 7: An example of SCFC. (a) Java-SCFC and (b) Python-SCFC.

definition of the shortest path kernel function to compare the similarity of two graphs is shown in the following equation:

$$\mathbf{k}(\mathbf{G}_1, \mathbf{G}_2) = \sum_{\mathbf{e}_1 \in \mathbf{E}_1} \sum_{\mathbf{e}_2 \in \mathbf{E}_2} \mathbf{k}^*_{\mathbf{path}=1}(\mathbf{e}_1, \mathbf{e}_2), \tag{1}$$

where $k_{path} = 1 * (e_1, e_2)$ represents a subkernel function of length 1, which is defined as follows:

$$\mathbf{k}^*_{\mathbf{path}=1}(\mathbf{e}_1, \mathbf{e}_2) = \exp\left(-\frac{\left|\mathbf{A}_1'(\mathbf{i}, \mathbf{j}) - \mathbf{A}_2'(\mathbf{k}, \mathbf{l})\right|}{\mathbf{m}_1 * \mathbf{m}_2}\right), \tag{2}$$

where
$\mathbf{e}_1 = (\mathbf{V}'_{1_i}, \mathbf{V}'_{1_j}), \mathbf{e}_2 = (\mathbf{V}'_{2_k}, \mathbf{V}'_{2_l}), \mathbf{m}_1 = |\mathbf{V}'_1|$, and $\mathbf{m}_2 = |\mathbf{V}'_2|$.

In the SPGK algorithm, the time complexity of the shortest path between all vertices in the graph is $O(n^3)$ when using the classical Floyd–Warshall algorithm, and the time complexity that compares all paths in the two graphs is $O(n^4)$. Therefore, the time complexity of the shortest path graph kernel algorithm is $O(n^4)$.

We propose the SCFC-SPGK algorithm based on SPGK. Since the SCFC is directed graph with the root node, the shortest path from the root node to other nodes (single source path) is enough to reflect the features of a SCFC for obtaining the shortest path set. Thus, the time complexity is reduced. The main idea of the algorithm is as follows. First, the shortest path sets $S_1 = \{p_1, p_2, ..., p_m\}$ and $S_2 = \{p'_1, p'_2, ..., p'_n\}$ are obtained. In terms of optimal matching, the path length L is obtained for each path p_i in S_1, then the path is matched with the path length in S_2 within $[L-1, L+1]$, and then the set $S = \{(p_i, p'_j) | \ 0 < j < n\}$ is obtained. Second, for each path pair, the edit distance D_v of the two path node attribute sequences (V_i, V_j) and the edit distance D_e of the edge attribute sequence (E_i, E_j) are calculated. Finally, the path p'_j with the minimum sum of D_v and D_e is selected to pair with p_i. At the same time, the paired paths are no longer included in the pairing between the remaining paths. The resulting path matching set $S_{final} = \{(p_i, p'_j) \mid 0 < i < t, 0 < j < t, t = \text{Min}(m, n)\}$. The kernel function definition of SCFC-SPGK is shown in the following equation:

$$\mathbf{k}(\mathbf{G}_1, \mathbf{G}_2) = \sum_{i=1,j=1}^{t} \sum_{(p_i, p_j) \in S_{\text{final}}} \exp\left(-\frac{\left(\mathbf{D_v}(\mathbf{p_i}, \mathbf{p_j}) + \mathbf{D_e}(\mathbf{p_i}, \mathbf{p_j})\right)}{\mathbf{m_i} * \mathbf{m_j}}\right), \tag{3}$$

where $\mathbf{m_i} = \mathbf{len}(\mathbf{p_i}), \mathbf{m_j} = \mathbf{len}(\mathbf{p_j})$. After obtaining the kernel of the two graphs, the ratio of the kernel and the number of matched path pairs in the matching set is taken as the similarity of the two graphs, which is shown as follows:

$$\mathbf{sim}(\mathbf{G}_1, \mathbf{G}_2) = \frac{\mathbf{k}(\mathbf{G}_1, \mathbf{G}_2)}{\mathbf{len}(\mathbf{S_{final}})}. \tag{4}$$

The whole SCFC-SPGK algorithm is shown as follows. The function ShortestPath_Floyd obtains the shortest path set between the root node and other nodes in the graphs G

and G' through the Floyd–Warshall algorithm. D_v and D_e are the functions that return the edit distance of the two paths, in which parameters of D_v are two node sequences and parameters of D_e are two edge sequences.

5.2. Time Complexity Analysis. Assuming that S_1 and S_2 are the shortest path sets from each node to their root nodes in the graph G_1 and G_2, respectively. The time complexity that obtains the shortest distance between other nodes and the root node in the graph by the Floyd–Warshall algorithm is $O(n^2)$. Let p_i be a path with n nodes in S_1 and p_j be a path with m nodes in S_2. The time complexity of the shortest edit distance between the node sequence of p_i and the node sequence of p_j is $O(mn)$. In conclusion, the time complexity of Algorithm 1 is $O(n^2)$. The algorithm improves the matching accuracy and reduces the time complexity based on the features of SCFCs.

6. Experiment and Evaluation

In this section, we verify the effectiveness of the proposed approach by experiments. We perform a comparative experiment to compare CLCSD with related approaches in cross-language source code similarity detection through real code datasets. In addition, we also conduct an experiment on similarity detection of the source code written in the same language. In terms of the implementation of CLCSD, for the given code written in two different languages, their PDGs are generated automatically based on the PDG generation framework (https://github.com/victorjmarin/sourcedg) first, and the code is preprocessed using the approach proposed in Section 4.2. Next, the preprocessed code is transformed into flowcharts expressed by the dot script [45]. Thus, the two flowcharts are converted into corresponding SCFCs. Finally, the SCFC-SPGK algorithm is used to calculate the similarity between these two SCFCs, and the obtained similarity value is regarded as the similarity between two pieces of source code.

6.1. Effectiveness Evaluation for Cross-Language Source Code Similarity Detection. We construct four experimental code sets (https://github.com/langtaosha1/CodeSet.git). First, we construct the first code set to verify the accuracy of each approach. In this code set, there are ten groups of code, and each group contains six programming questions selected from OJ platform. We ask a volunteer to submit the Java, Python, and C# code for each question following the same solution idea. Any two of the three answers can be regarded as the source code and the plagiarized code because each question is solved in the same way by the same volunteer. Second, we construct the second code set to investigate the code obfuscation techniques that each approach can defeat in cross-language code similarity detection. We keep the Python code in the first code set unchanged and use ten commonly used code obfuscation techniques [46, 47] to modify the Java and C# code in each code group. The ten code obfuscation techniques are shown in Table 2 from easy to difficult. Specifically, we modify the Java code in the way that the N-th code obfuscation technique is used for the N-th group of code.

We construct the third and fourth code sets by the public dataset provided by Vislavski et al. [5] to further verify the effectiveness of our approach and avoid the contingency of special datasets. The third code set is constructed in the same way as the first one, while the fourth code set is constructed in the same way as the second code set. Moreover, they had five times as much data as the first and second code sets, respectively.

6.1.1. Code Similarity Detection Effectiveness Comparison

(1) Experiment Setup. Based on the first and the third code sets, we first compare our approach with three existing similarity detection tools in cross-language similarity detection effectiveness. The theory of these three existing tools is based on tree, attribute counting, and NLP, respectively. Among them, Vislavski et al. [5] propose LICCA, which mainly relies on the SSQSA platform and generates a common intermediate representation called eCST (enriched concrete syntax tree). Nafi et al. [29] propose CLCDSA, which selects nine measurement attributes and obtain feature measurement values by traversing the AST (abstract syntax tree). Flores et al. [30] propose DeSoCoRe to extract code features by tri-gram model and weights word frequency based on normalized term frequency. The similarity between codes is calculated by cosine similarity. In this experiment, the effectiveness of these four approaches is compared. The evaluation indicator is the average similarity of the source code pairs corresponding to all the questions in each group, and the calculation method is shown in formula (5). In this experiment, we set n to ten:

$$\overline{\text{sim}} = \frac{\sum_{i=1}^{n} \text{sim}_i}{n}. \tag{5}$$

(2) Experimental Result. The comparison results of the above approaches with CLCSD in the effectiveness of cross-language source code similarity detection are shown in Figure 8. We can see the average similarity values calculated by CLCSD for each group of questions are greater than that of DeSoCoRe, CLCDSA, and LICCA. Among them, DeSoCoRe is string-based approaches, and it strongly depends on the syntax of the programming language. As a result, the average similarity values obtained by DeSoCoRe are lower than those of the other three approaches. The similarity obtained by LICCA is lower than that of CLCDSA and CLCSD because LICCA requires two code fragments with the same block size, control flow, and sequences along with the same flow of statements. However, it is difficult to satisfy these preconditions in cross-language code [29] due to the syntax differences of different programming languages. Experimental results in Figure 8 show that the similarity detection value of CLCDSA is lower than CLCSD in cross-language code similarity detection. CLCDSA may be influenced by the syntax differences of different languages because the attribute values of two code fragments of different programming languages may be

Input: the graphs $G = (V, E, T_V, T_E, \mu, \delta)$ and $G' = (V', E', T_V, T_E, \mu, \delta)$;
Output: sim, the similarity value between G and G';
(1) Path set $S = \{\}$, path set $S' = \{\}$
(2) sim = 0, $k = 0$
(3) $V_0 = Get\_RootNode(G)$; $V'_0 = Get\_RootNode(G')$;
(4) Get the adjacency matrix A of G and adjacency matrix A' of G' by E and E' respectively.
(5) S = ShortestPath_Floyd (V_0, A)//get the shortest path set of G between V_0 and other nodes.
(6) S' = ShortestPath_Floyd (V'_0, A')//get the shortest path set of G' between V'_0 and other nodes.
(7) **for** each $p \in S$:
(8) assume match set $S_t = \{\}$
(9) **for** each $p' \in S'$:
(10) **if** $((\text{len}(p)-1) \le \text{len}(p') \le (\text{len}(p)+1))$ **then**:
(11) $D = D_v(p, p') + D_e(p, p')$
(12) add D to S_t
(13) **end if**
(14) **end for**
(15) $d = \min(S_t)$//the path with the highest degree of matching is the final match.
(16) $k += \exp(-(d/(|\text{pd}| * |\text{pd}'|)))$
(17) $S_t = \{\}$
(18) sim = k/len(S)
(19) **end for**
(20) **output** sim

Algorithm 1: SCFC-SPGK.

Table 2: The obfuscation techniques.

Number	Obfuscation technique
1	Copying the original code completely
2	Modifying the comments
3	Changing the code format and adding blank lines
4	Renaming identifiers
5	Adjusting code statements order
6	Replacing constants
7	Changing data types
8	Substituting equivalent operators
9	Adding redundant statements
10	Substituting equivalent control structures

different even if they implement the same function. For example, the Python language does not require variables to be declared in advance.

6.1.2. Anti-Obfuscation Effectiveness Comparison

(1) Experiment Setup. Based on the second and fourth code sets, we conduct a code antiobfuscation experiment for the above four cross-language similarity detection approaches. Similar to the first experiment, we choose the average similarity of the code corresponding to all the questions in each group as the evaluation indicator. At the same time, we multiply obfuscate N-th group of data corresponding to N-th obfuscation technique and take the average of the experimental results as the final result to ensure the effectiveness of the experiments.

(2) Experimental Result. The comparison results of the four approaches in defeating cross-language code obfuscation techniques are shown in Figure 9.

By comparing Figures 8 and 9, we can see that the four approaches can completely defeat the first three obfuscation techniques. Since DeSoCoRe directly extracts features from the source code, the obfuscation techniques that change the original code other than formatting and comments can affect the effectiveness of DeSoCoRe. LICCA uses the tree-based intermediate representation to detect cross-language code similarity. Therefore, the obfuscation techniques that can change the structure of the code may adversely affect the effectiveness of this approach, such as the fifth, seventh, ninth, and tenth obfuscation techniques. CLCDSA detects cross-language code similarity based on attribute counting, so the obfuscation technique that changes the attributes of the original code has a negative effect on the effectiveness of this approach, such as the seventh, eighth, ninth, and tenth obfuscation techniques. In particular, the ninth obfuscation technique has a great effect on CLCDSA, LICCA, and DeSoCoRe because the obfuscation technique adds redundant statements and changes the attribute values and structures of the code. The proposed approach is less affected by this obfuscation technique because the preprocessing based on PDGs can completely remove the redundant statements that have no data dependency on the original code. In addition, the proposed approach can defeat eight other obfuscation techniques except the fifth and ninth obfuscation techniques and partially defeat the fifth and ninth obfuscation techniques because the fifth obfuscation technique may change the position of *combine* nodes in SCFC (e.g., converting global variables to local variables),

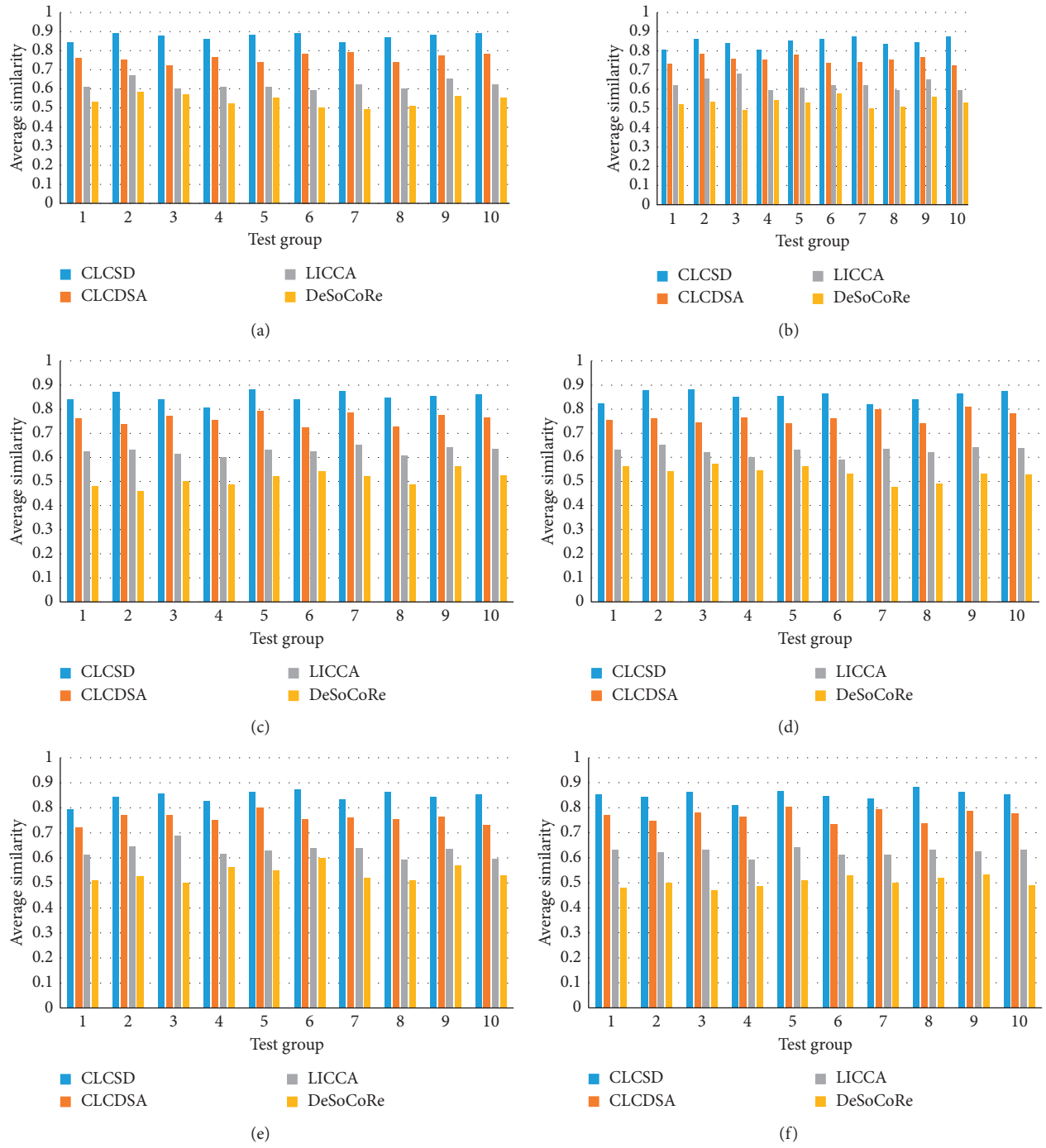


Figure 8: The similarity comparison of source code written in different languages. The effectiveness comparison (a) between Java and Python on the first code set, (b) between Java and C# on the first code set, (c) between Python and C# on the first code set, (d) between Java and Python on the third code set, (e) between Java and C# on the third code set, and (f) between Python and C# on the third code set.

while the other eight obfuscation techniques cannot change the structure of SCFC after the preprocessing.

6.2. *Effectiveness Evaluation for the Same Language Source Code Similarity Detection*

6.2.1. Experiment Setup. We construct the fifth code set to evaluate the effectiveness of CLCSD in the similarity detection of the code written in the same language. Meanwhile, we also evaluate its effectiveness in dealing with code obfuscation techniques. We regard the Java code in the third code set as original code, while the Java code for the same question in the fourth code set is plagiarized because the fourth code set is constructed by obfuscating the Java code and C# code in the third code set. To construct the fifth code set, we collect the answer with Java code in the third code set and the answer with Java code in the fourth code set for each selected question. The whole code set is still divided into ten

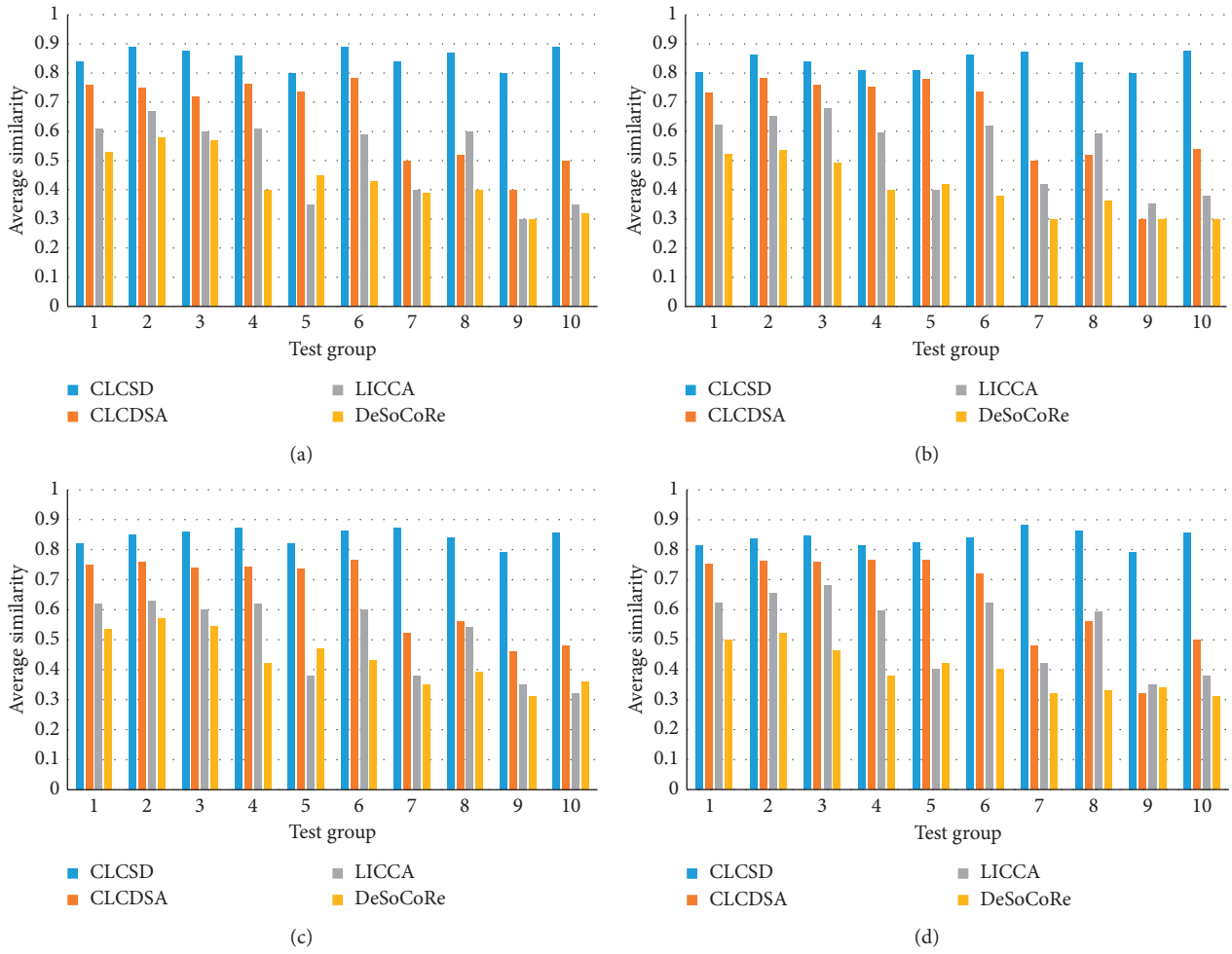


Figure 9: Resistance effectiveness for cross-language code obfuscation techniques. The resistance effectiveness comparison (a) between Java and Python on the second code set, (b) between Python and C# on the second code set, (c) between Java and Python on the fourth code set, and (d) between Python and C# on the fourth code set.

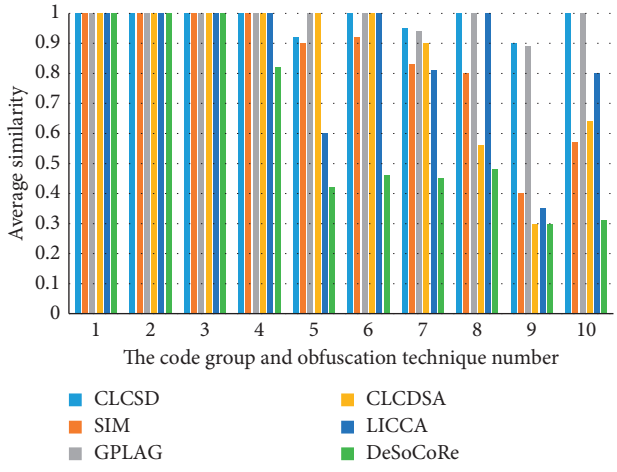


Figure 10: The similarity detection comparison in the same language.

groups, and each group contains thirty questions. Thus, for each question, there is the original Java code and the obfuscated Java code. We use SIM [12], GPLAG [21], DeSoCoRe [31], LICCA [5], CLCDSA [30], and CLCSD to calculate the similarity of each Java source code pair. For the six approaches, we compare their ability to defeat common code obfuscation techniques. The evaluation indicator of the experiment is the average similarity of each approach against the code obfuscation techniques in each group, as shown in formula (5).

6.2.2. Experimental Result. The comparison results of the above six approaches in the effectiveness of the same language source code similarity detection are shown in Figure 10. In terms of difficulty, we divide the ten obfuscation techniques into three categories. The first category is simple obfuscation techniques, including the first, second, and third code obfuscation techniques. The second category is relatively complex obfuscation techniques, including the fourth, fifth, sixth, seventh, and eighth code obfuscation techniques. The third category is the most complex obfuscation techniques, including the ninth and tenth techniques.

First, all the six approaches can completely defeat these code obfuscation techniques. The simple obfuscation

techniques have no effect to the detection effectiveness after applying simple preprocess to the original code, such as removing comments, whitespace, and blank lines.

Second, for relatively complex obfuscation techniques, CLCSD and GPLAG are fully resistant to the fourth, sixth, and eighth obfuscation techniques. This is because they consider not the content of the nodes, but the structure of the code. For the fifth obfuscation technique, it has no effect on GPLAG because GPLAG only analyzes the dependencies between statements. However, because the fifth obfuscation may change the location of combine nodes in a SCFC, CLCSD is slightly less resistant to this obfuscation technique than GPLAG. For the seventh obfuscation technique, CLCSD and GPLAG are fully resistant to it because it may add redundant statements that depend on the original code. CLCDSA is fully resistant to the fourth, fifth, and sixth obfuscation techniques because they cannot change the attribute values of the original code. SIM and DeSoCoRe are less resistant to obfuscating techniques other than the fourth. Because SIM and DeSoCoRe are string-based approaches, the obfuscation techniques except the fourth have a greater impact on the content and the order of a piece of code. SIM is implemented based on tokens. In the pretreatment, it converted all identifiers into token sequences. Thus, it can defeat the fourth obfuscation technique. However, DeSoCoRe does not have a uniform identifier, and it is greatly affected.

Third, for the most complex obfuscation techniques, SIM, CLCDSA, LICCA, and DeSoCoRe cannot defeat these techniques because they change the content or structure of the original code greatly. For the ninth obfuscation technique, CLCSD and GPLAG cannot fully defeat it because the added redundant statements may have dependencies on the original code. However, their antiobfuscation ability is better than other approaches. For the tenth obfuscation technique, GPLAG is resistant to it because it does not affect the control dependencies of the source code. Meanwhile, CLCSD can also defeat this obfuscation due to the SCFC unifies the control structures of the code.

6.3. Experimental Conclusion. Through the above experiments, we can draw the following conclusions. First, the proposed approach has a higher accuracy in terms of cross-language source code similarity detection compared with the existing approaches. At the same time, CLCSD is resistant to the common obfuscation techniques such as modifying the comments, copying completely, changing the code format and adding blank lines, renaming identifiers, replacing equivalent control structure, replacing constant, and adding nondependent redundant statements. Secondly, for the same language source code similarity detection, the effectiveness of CLCSD to defeat the fifth obfuscation techniques is slightly lower than that of GPLAG. However, the effectiveness to defeat other obfuscation techniques of the proposed approach is nearly the same with that of GPLAG. Meanwhile, CLCSD is more accurate in similarity detection compared with SIM, CLCDSA, LICCA, and DeSoCoRe.

7. Conclusion and Future Work

Existing code similarity detection approaches transform the source code into the structure that can express the features of the source code, such as strings, trees, and graphs. Then, the similarity between source codes is measured based on these structures. However, these methods are not suitable for cross-language code similarity detection because these structures are often related to the syntax features of the programming languages that the code is written. The code flowcharts describe the core process of the code; therefore, for a pair of plagiarized source code written in different programming languages, their corresponding core code flow charts are similar. Based on this idea, we propose the CLCSD approach for cross-language code similarity detection. The approach converts source code into standardized flowcharts based on SCFC and determines the source code similarity by the similarity of SCFCs using the proposed SCFC-SPGK algorithm. The SCFC-SPGK algorithm reduces the node search space in the SCFC and improves the detection efficiency. In addition, some approaches, including the code preprocessing based on a PDG and the introduction of the *combine* node, improve the ability of the proposed approach in fighting against the code obfuscation techniques, such as adding redundant statements and adjusting the sequence of statements.

The proposed approach is the preliminary exploration of cross-language source code similarity detection based on flowcharts. We can further improve the approach from the following three aspects.

First, we can further investigate the approach to fight against more complex obfuscation techniques, such as adding redundant statements with data dependencies. We can combine CLCSD with existing dynamic similarity detection approaches [48]. In this way, we can obtain the code similarity through the running results of the source code to defeat more complex code obfuscating techniques.

Second, we can combine machine learning techniques with the proposed approach to detect the similarity between large-scale source codes. If the experimental dataset is large enough, machine learning algorithms such as the neural network [1] can be used to cluster similar code sets. Thus, the accuracy and efficiency of code similarity detection can be further improved.

Third, the idea of similarity measure based on flowcharts can be generalized to the similarity measure in other fields. For example, in the field of educational process mining, students can be clustered by measuring their similarity of learning processes that can be discovered from the learning data in a MOOC platform using process mining techniques [49].

Data Availability

We constructed our datasets based on the submission of OJ system and the public dataset provided by Vislavski et al. [5].

Conflicts of Interest

The authors declare that they have no conflicts of interest.

Acknowledgments

This research was funded by Education Ministry Humanities and Social Science Research Youth Fund Project of China under grant 19YJCZH240 (User-steering multi-source education data integration approach research in big data environment), Qingdao Social Science Planning Research Project under grant QDSKL1901123, National Science Foundation of China under grants 61902222 and U1931207, Taishan Scholars Program of Shandong Province under grants tsqn201909109 and ts20190936, SDUST Research Fund under grant 2015TDJH102, and SDUST Excellent Teaching Team Construction Plan under grant JXTD20180503.

References

[1] X. Xu, C. Liu, Q. Feng, H. Yin, L. Song, and D. Song, "Neural network-based graph embedding for cross-platform binary code similarity detection," in *Proceedings of the CCS*, pp. 363–376, Dallas, Texas, USA, November 2017.

[2] S. Engels, V. Lakshmanan, and M. Craig, "Plagiarism detection using feature-based neural networks," in *Proceedings of the 38th SIGCSE Technical Symposium on Computer Science Education, SIGCSE 2007*, pp. 34–38, Covington, Kentucky, USA, March 2007.

[3] L. Moussiades and A. Vakali, "PDetect: PDetect: a clustering approach for detecting plagiarism in source code datasets," *The Computer Journal*, vol. 48, no. 6, pp. 651–661, 2005.

[4] S. Wasik, M. Antczak, J. Badura, A. Laskowski, and T. Sternal, "A survey on online Judge systems and their applications," *ACM Computing Surveys*, vol. 51, no. 1, pp. 1–34, 2018.

[5] T. Vislavski, G. Rakić, N. Cardozo et al., "LICCA: a tool for cross-language clone detection," in *Proceedings of the 25th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER 2018)*, pp. 512–516, Campobasso, Italy, March 2018.

[6] E. L. Jones, "Metrics based plagarism monitoring," *Journal of Computing Sciences in Colleges*, pp. 253–261, Vermont, USA, 2001.

[7] Q. Y. Chen, S. P. Li, M. Yan, and X. Xia, "Code clone detection: a literature review," *Journal of Software*, vol. 30, no. 4, pp. 962–980, 2019.

[8] J. Krinke, "Identifying similar code with program dependence graphs," in *Proceedings of the Eighth Working Conference on Reverse Engineering WCRE*, pp. 301–309, Stuttgart, Germany, February 2001.

[9] K. M. Borgwardt and H. P. Kriegel, "Shortest-path kernels on graphs," in *Proceedings of the International Conference on Data Mining ICDM*, pp. 74–81, Houston, Texas, USA, December 2005.

[10] B. S. Baker, "On finding duplication and near duplication in large software systems," in *Proceedings of the Working Conference on Reverse Engineering WCRE*, pp. 86–95, Toronto, Ontario, Canada, August 1995.

[11] B. Muddu, A. Asadullah, and V. Bhat, "CPDP: a robust technique for plagiarism detection in source code," in *Proceedings of the 2013 7th International Workshop on Software Clones ICSC*, pp. 39–45, San Francisco, CA, USA, May 2013.

[12] S. Burrows, S. M. M. Tahaghoghi, and J. Zobel, "Efficient plagiarism detection for large code repositories," *Software Practice and Experience*, vol. 37, no. 2, pp. 151–175, 2006.

[13] L. Prechelt, G. Malpohl, 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.

[14] J. W. Son, S. B. Park, and S. Y. Park, "Program plagiarism detection using parse tree kernels," in *Proceedings of the 9th Pacific Rim International Conference on Artificial Intelligence PRICAI*, pp. 1000–1004, Guilin, China, August 2006.

[15] K. Oscar and Simon, "Syntax trees and information retrieval to improve code similarity detection," in *Proceedings of the Twenty-Second Australasian Computing Education Conference*, pp. 48–55, New York, NY, February 2020.

[16] Z. Liping, L. Dongsheng, L. Yanchen, and Z. Mei, "AST-based plagiarism detection method," *Computer Engineering and Design*, vol. 33, no. 4, pp. 1660–1664, 2012.

[17] H. Kikuchi, T. Goto, M. Wakatsuki, and T. A. Nishino, "Source code plagiarism detecting method using alignment with abstract syntax tree elements," in *Proceedings of the 2014 15th IEEE/ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing ACIS*, pp. 1–6, Las Vegas Nevada, USA, une 2014.

[18] T. Guo, G. Dong, H. Qin, and B. Cui, "Improved plagiarism detection algorithm based on abstract syntax tree," in *Proceedings of the 2013 Fourth International Conference on Emerging Intelligent Data and Web Technologies (EIDWT)*, pp. 714–719, Xi an, China, September 2013.

[19] M. Gabel, L. Jiang, and Z. Su, "Scalable detection of semantic clones," in *Proceedings of the 30th International Conference on Software Engineering ICSE*, pp. 321–330, Leipzig, Germany, May 2008.

[20] H.-i. Lim, H. Park, S. Choi, and T. Han, "A method for detecting the theft of Java programs through analysis of the control flow information," *Information and Software Technology*, vol. 51, no. 9, pp. 1338–1350, 2009.

[21] C. Liu, C. Chen, J. Han, and P. S. Yu, "GPLAG: detection of software plagiarism by program dependence graph analysis," in *Proceedings of the Conference: Proceedings of the Twelfth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining SIGKDD*, pp. 872–881, Philadelphia, USA, January 2006.

[22] D. Qiu, J. Sun, and H. Li, "Improving similarity measure for Java programs based on optimal matching of control flow graphs," *International Journal of Software Engineering and Knowledge Engineering*, vol. 25, no. 07, pp. 1171–1197, 2015.

[23] R. Komondoor and S. Horwitz, "Using slicing to identify duplication in source code," in *Proceedings of the International Static Analysis Symposium*, pp. 40–56, Springer, Berlin, Heidelberg, January 2001.

[24] T. Kamiya, S. Kusumoto, and K. Inoue, "CCFinder: a multilinguistic token-based code clone detection system for large scale source code," *IEEE Transactions on Software Engineering*, vol. 28, no. 7, pp. 654–670, 2002.

[25] C. Arwin and S. M. M. Tahaghoghi, "Plagiarism detection across programming languages," in *Proceedings of the Twenty-Nineth Australasian Computer Science Conference (ACSC2006)*, pp. 277–286, Hobart, Tas, Australia, January 2006.

[26] F. Al-Omari, I. Keivanloo, C. K. Roy, and J. Rilling, "Detecting clones across microsoft. net programming languages," in *Proceedings of the 19th Working Conference on Reverse*

Engineering, pp. 405–414, IEEE, Kingston, ON, Canada, October 2012.

[27] L. Nichols, M. Emre, and B. Hardekopf, "Structural and nominal cross-language clone detection," *Fundamental Approaches to Software Engineering, FASE*, pp. 247–263, Springer, Berlin, Germany, 2019.

[28] D. Perez and S. Chiba, "Cross-language clone detection by learning over abstract syntax trees," in *Proceedings of the IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, pp. 518–528, IEEE, Montreal, QC, Canada, May 2019.

[29] N. A. Kraft, B. W. Bonds, and R. K. Smith, "Cross-Language clone detection," in *Proceedings of the Twentieth International Conference on Software Engineering & Knowledge Engineering (SEKE'2008)*, pp. 54–59, San Francisco, CA, USA, January 2008.

[30] K. W. Nafi, T. S. Kar, B. Roy et al., "CLCDSA: cross Language code clone detection using syntactical features and API documentation," in *Proceedings of the IEEE/ACM 34th International Conference on Automated Software Engineering (ASE)*, pp. 1026–1037, San Diego, CA, USA, January 2019.

[31] E. Flores, A. Barrón-Cedeno, P. Rosso, and L. Moreno, "DeSoCoRe: detecting source code re-use across programming languages," in *Proceedings of the 2012 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pp. 1–4, Montr ´eal, Canada, June 2012.

[32] E. Flores, A. Barrón-Cedeno, L. Moreno, and P. Rosso, "cross-language source code Re-use detection using latent semantic analysis," *Journal of Universal Computer Science*, vol. 21, no. 13, pp. 1708–1725, 2015.

[33] X. Cheng, Z. Peng, L. Jiang, H. Zhong, H. Yu, and J. Zhao, "CLCMiner: detecting cross-language clones without intermediates," *IEICE Transactions on Information and Systems*, vol. 100, no. 2, pp. 273–284, 2017.

[34] F. Ullah, J. Wang, M. Farhan, M. Habib, and S. Khalid, "Software plagiarism detection in multiprogramming languages using machine learning approach," *Concurrency and Computation: Practice and Experience*, Article ID e5000, 2018.

[35] Q. Song, *Research on Cross-Language Code Similarity Detection Method Based on Program Flow Chart*, Shandong university of science and technology, Qingdao, China, 2019.

[36] M. Neuhaus, K. Riesen, and H. Bunke, *Fast Suboptimal Algorithms for the Computation of Graph Edit Distance*, pp. 163–172, SPR & SSPR. Springer, Berlin, Heidelberg, 2006.

[37] Z. Lin, H. Wang, S. Mcclean et al., "A multidimensional sequence approach to measuring tree similarity," *IEEE Transactions on Knowledge and Data Engineering*, vol. 24, no. 2, pp. 197–208, 2012.

[38] Z. Lin, H. Wang, and S. Mcclean, "Tree similarity measurement for classifying questions by syntactic structures," in *Proceedings of the International Conference on Intelligent Computing*, pp. 379–390, Lanzhou, China, August 2016.

[39] F. Den Hollander, H. Kesten, V. Sidoravicius et al., "Random walk in a high density dynamic random environment," *Indagationes Mathematicae*, vol. 25, no. 4, pp. 785–799, 2014.

[40] U. Kang, H. Tong, and J. Sun, "Fast random walk graph kernel," in *Proceedings of the Society of Indian Automobile Manufacturers*, pp. 828–838, Anaheim, CA, USA, April 2012.

[41] C. H. Elzinga and H. Wang, "Kernels for acyclic digraphs," *Pattern Recognition Letters*, vol. 33, no. 16, pp. 2239–2244, 2012.

[42] R. W. Floyd, "Algorithm 97: shortest path," *Communications of the ACM*, vol. 5, no. 6, p. 345, 1962.

[43] P. Mahe, N. Ueda, T. Akutsu, J. Perret, and J. Vert, "Extensions of marginalized graph kernels," in *Proceedings of the Twenty-first International Conference on Machine Learning*, p. 70, Alberta, Canada, January 2004.

[44] S. V. N. Vishwanathan, N. N. Schraudolph, R. Kondor, and K. M. Borgwardt, "Graph kernels," *Journal of Machine Learning Research*, vol. 9, no. 2, pp. 1201–1242, 2010.

[45] S. Warshall, "A theorem on boolean matrices," *Journal of the ACM*, vol. 9, no. 1, pp. 11-12, 1962.

[46] E. Flores, A. Barrón-Cedeño, L. Moreno, and P. Rosso, "Uncovering source code reuse in large-scale academic environments," *Computer Applications in Engineering Education*, vol. 23, no. 3, pp. 383–390, 2015.

[47] M. Novak, M. Joy, and D. Kermek, "Source-code similarity detection and detection tools used in academia," *ACM Transactions on Computing Education*, vol. 19, no. 3, pp. 1–37, 2019.

[48] Z. Tian, Q. Zheng, T. Liu, M. Fan, E. Zhuang, and Z. Yang, "Software plagiarism detection with birthmarks based on dynamic key instruction sequences," *IEEE Transactions on Software Engineering*, vol. 41, no. 12, pp. 1217–1235, 2015.

[49] C. Liu, H. Duan, Q. Zeng, M. Zhou, F. Lu, and J. Cheng, "Towards comprehensive support for privacy preservation cross-organization business process mining," *IEEE Transactions on Services Computing*, vol. 12, no. 4, pp. 639–653, 2019.