Hindawi
Scientific Programming
Volume 2020, Article ID 8865413, 15 pages
https://doi.org/10.1155/2020/8865413

Research Article

Flow Chart Generation-Based Source Code Similarity Detection Using Process Mining

Feng Zhang,[1,2] Lulu Li,[1] Cong Liu,[3] and Qingtian Zeng[1,2]

[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@163.com and Qingtian Zeng; qtzeng@163.com

Received 13 April 2020; Revised 2 May 2020; Accepted 13 May 2020; Published 7 July 2020

Academic Editor: Chenxi Huang

Source code similarity detection has extensive applications in computer programming teaching and software intellectual property protection. In the teaching of computer programming courses, students may utilize some complex source code obfuscation techniques, e.g., opaque predicates, loop unrolling, and function inlining and outlining, to reduce the similarity between code fragments and avoid the plagiarism detection. Existing source code similarity detection approaches only consider static features of source code, making it difficult to cope with more complex code obfuscation techniques. In this paper, we propose a novel source code similarity detection approach by considering the dynamic features at runtime of source code using process mining. More specifically, given two pieces of source code, their running logs are obtained by source code instrumentation and execution. Next, process mining is used to obtain the flow charts of the two pieces of source code by analyzing their collected running logs. Finally, similarity of the two pieces of source code is measured by computing the similarity of these two flow charts. Experimental results show that the proposed approach can deal with more complex obfuscation techniques including opaque predicates and loop unrolling as well as function inlining and outlining, which cannot be handled by existing work properly. Therefore, we argue that our approach can defeat commonly used code obfuscation techniques more effectively for source code similarity detection than the existing state-of-the-art approaches.

1. Introduction

Research studies on source code similarity detection can be tracked back to the 1970s, and such techniques have a wide range of applications in the source code plagiarism detection of computer programming teaching and software intellectual property protection. Existing source code similarity detection techniques mainly involve attribute counting [1] and structure metrics [2–13]. Structure metrics are most commonly used approaches that mainly contain string-based, tree-based, and graph-based code similarity measure approaches. In the current teaching of computer programming courses, the online judge (OJ) system [14] that implements online submission and automatic assessment of the programming assignments has been widely used. Meanwhile, OJ uses the code similarity detection tool to find the plagiarism in programming assignments. To support code similarity detection, most existing OJ systems use string-based and tree-based detection approaches. However, the antiobfuscation effectiveness of these approaches is weak because they can only deal with some simple code obfuscation techniques [15–18]. In computer programming teaching, students sometimes use some complex code obfuscation techniques, e.g., opaque predicates, loop unrolling, and function inlining and outlining, to reduce the similarity between code fragments. However, existing approaches cannot handle these complex obfuscation techniques. The main reason for this problem is that existing approaches measure code similarity only by the static features of the source code, such as the text or structure, and they do not consider the runtime dynamic features of the source code. Thus, existing approaches cannot cope with the above complex obfuscation techniques.

To solve this problem, we propose a novel source code similarity detection approach for computer programming teaching using process mining. The dynamic features of source code are obtained through the running of the code, and they are used as the basis for measuring the similarity between two code fragments. Specifically, given two pieces of source code, we first obtain their running logs by source code instrumentation and running. Next, process mining is used to obtain their flow charts that reflect their dynamic features at runtime. Finally, the similarity between two flow charts can be measured by a graph similarity algorithm, and the similarity value is taken as the final similarity of these two pieces of code.

The rest of this paper is structured as follows. Section 2 introduces the classification of code obfuscation techniques and existing code similarity detection approaches. Section 3 proposes the basic idea and overall framework of the source code similarity detection approach based on process mining. Next, Section 4 introduces the approach in detail, and the effectiveness of our approach is verified by experiments in Section 5. Finally, the whole work is summarized in Section 6.

2. Related Work

The performance of antiobfuscation is an important metric to evaluate source code similarity detection [19]. Therefore, we first summarize most commonly used code obfuscation techniques in this section. Then, we introduce the existing source code similarity detection approaches and their ability to fight against code obfuscation techniques, based on which we summarize the problems of existing approaches.

2.1. Code Obfuscation. Jones summarized ten commonly used approaches to plagiarizing programming assignment of students [20]. On this basis, another two code obfuscation techniques are proposed in the literature [15, 21]. To sum up, there are mainly fourteen kinds of obfuscation techniques, which are shown in Figure 1 from easy to difficult: (1) verbatim copying; (2) changing comments; (3) changing white space and formatting; (4) renaming identifiers; (5) reordering code blocks; (6) reordering statements within code blocks; (7) replacing constants; (8) changing the order of operators or operands in expressions; (9) changing data types; (10) adding redundant statements; (11) splitting expressions; (12) replacing control structures with equivalent structures; (13) loop unrolling; (14) function inlining and outlining.

We further classify adding redundant statements and loop unrolling in detail in this paper. First, adding redundant statements can be divided into adding sequential statements, adding reachable branches, and opaque predicates. Specifically, adding sequential statement refers to adding sequentially executed code that can be run without affecting the results; adding reachable branch refers to adding redundant executable branches that are added without changing the result; opaque predicate [21] is a control flow obfuscation technique, and it adds unreachable paths or branches to the source code without changing the final result. Second, loop unrolling is divided into partial unrolling and full unrolling, and it reduces the number of cycles and increases the source code by copying the code in a loop body [15].

2.2. Source Code Similarity Detection. The first approach for source code similarity measure is attribute counting [22], which mainly measures the similarity by counting various metrics of the source code. Because too much structure information of the source code is discarded, the accuracy of these methods is low. Therefore, researchers propose the approaches based on structure metrics that are mainly based on strings, trees, and graphs.

The string-based source code similarity detection approaches compare the text of the code. Now this kind of approaches is relatively mature, and there are some commonly used software systems, such as JPlag [2], MOSS [3], and Sim [4]. Similarly, in terms of cloning detection, a string-based cloning detection system that can be used for large code is proposed [23]. Although string-based source code similarity detection approaches have higher space-time efficiency, they rarely reflect the semantic and syntactic information of the code. As a result, it is difficult for these approaches to fight against some complex code obfuscation techniques.

The tree-based similarity detection approaches transform the source code into a tree structure and measure the similarity between trees. For example, The AST-CC algorithm [5] transforms source code into an abstract syntax tree and improves the efficiency of syntax tree comparison by transforming storage format. This approach has stronger antiobfuscation ability than string-based code similarity detection approaches. In addition, in terms of code clone detection, a tree-based code clone detection approach is also proposed in [24, 25]. However, the tree-based approaches mainly measure the similarity through the subtrees. As a result, they cannot fight against structural obfuscation techniques [20], such as adding redundant statements and loop unrolling. Consequently, this kind of approaches can only solve simpler code obfuscation techniques.

The graph-based source code similarity detection approaches transform the code into a graph structure and compares the similarity between graphs. At present, existing approaches use two kinds of graphs: program dependency graph (PDG) and control flow graph (CFG). First, GPLAG proposed by Liu et al. constructs a PDG [6] according to the source code and calculates the similarity between two PDGs. Although a PDG contains the semantic information of the source code, GPLAG can only detect the similarity between two single functions but not multiple functions [7]. Meanwhile, it is still susceptible to the code obfuscations techniques that retain semantics, such as opaque predicates [8] as well as function inlining and outlining [9]. In addition, in terms of code cloning detection, PDGs and program slicing are used to find isomorphic PDG subgraphs that represent clones [26]. Second, Lim et al. propose to use CFG to indicate the control structure of the source code [10] and

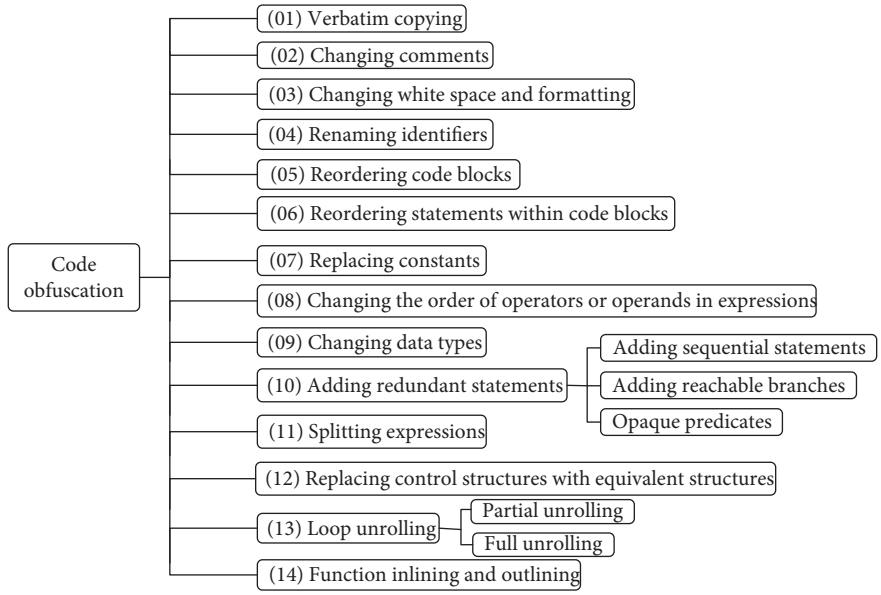


Figure 1: Code obfuscation techniques.

calculate the similarity between code fragments by analyzing the k-length flow paths of CFG. Similarly, Qiu et al. [27] propose a fault-tolerant graph matching approach. The matching subpaths of two CFGs are obtained by matching the basic blocks (a sequence of statements executed sequentially with only one entry and one exit) of two codes, and then the code similarity is calculated by weighted similarity of each path. In these approaches, the control logic of the source code that reflects the possible execution paths of the code is considered. However, because such kind of approaches cannot get the actual running path of the source code, they are still unable to resist some complex code obfuscation techniques, such as opaque predicates and reordering code blocks. In addition, an approach to measuring cross programming languages code similarity is proposes based on the static flow chart of source code [11]. Similarly, this approach only considers the static flow char of the code, making it difficult to fight against opaque predicates and function inlining and outlining. To sum up, graph-based code similarity detection approaches cannot deal with some code obfuscation techniques including opaque predicates, loop unrolling, and some other complex code obfuscation techniques.

3. An Approach Overview

3.1. Basic Idea. Existing work measures source code similarity only by the static features, such as the text or structure of source code, while it does not consider the runtime dynamic features of the source code. Figure 2 shows an example of source code to print the sum of $1 + 2 + 3 + \ldots + 100$, as well as the plagiarized code that uses two obfuscation techniques: loop unrolling and adding redundant statements. Two code fragments are shown in Figure 2(a) and Figure 2(c), respectively. After adding redundant statements, the text and structure of these two code fragments have some differences. Take their flow charts as an example. The flow chart of the source code is shown in Figure 2(b), and the flow chart of the obfuscated code is shown in Figure 2(d). The use of the two obfuscations reduces the similarity between the flow charts of the source code and obfuscated code. Therefore, the traditional source similarity detection approaches based on static features of code cannot deal with these obfuscation techniques. To solve this problem, we propose to obtain dynamic features of source code through the running of code and use the dynamic features to measure the similarity between two code fragments.

The dynamic features of source code are the key to the proposed approach. For two code fragments, we first obtain their running logs by source code instrumentation and running. Then, we use the process mining to obtain their flow charts that indicate their dynamic features at runtime. Process mining extracts process-related information from the event logs to discover, monitor, and improve the actual processes [28]. By process mining, the execution sequences of the activities in the execution logs based on existing process instances can be mined to obtain the dependencies and execution orders between the activities. Thus, the process model that expresses the logical relationship between the activities can be obtained. Borrowing the idea of process mining, if we regard the output logs of a piece of code by running once as the activity sequence in a process instance and obtain a group of output logs after running the code multiple times, these logs can be used as the input of a process mining algorithm to get a flow chart that reflects the actual running process of the code. Based on the above analysis, we propose to use process mining to get the flow chart by mining the output logs that are produced through running the source code and take the flow chart as the dynamic features of the code. Thus, the similarity between flow charts can be used to measure the similarity of the source code. In this way, the more accurate similarity of two pieces of code can be obtained.

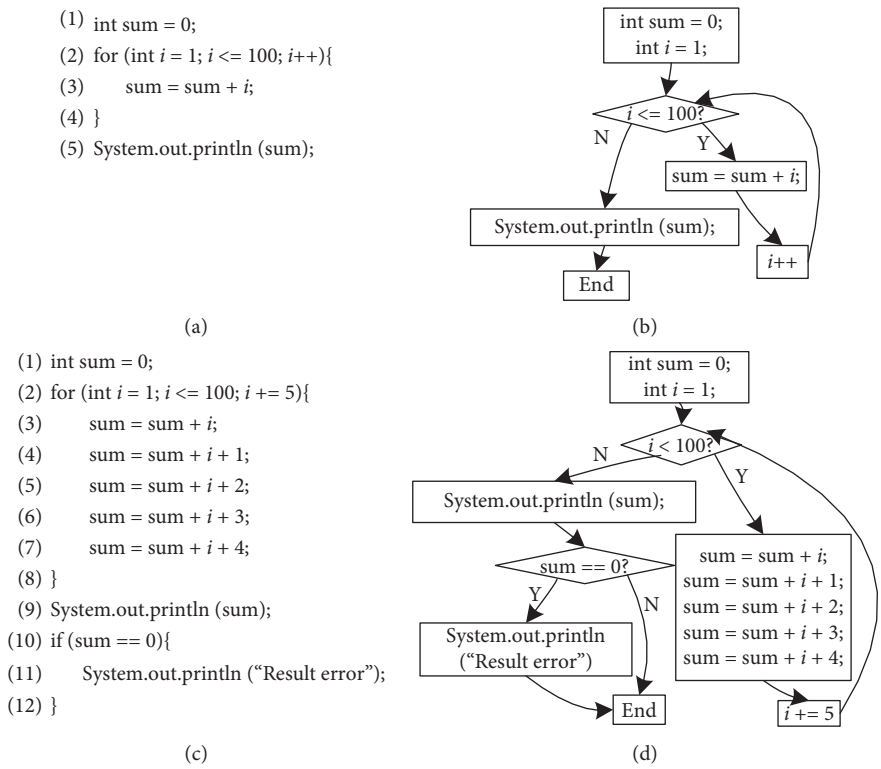


Figure 2: An example of code obfuscation.(a) Source code, (b) source code flow chart, (c) obfuscated code, and (d) obfuscated code flow chart.

3.2. Framework Design. A general framework for source code similarity detection using process mining is shown in Figure 3. First, the two pieces of source code are preprocessed to remove redundant statements. Then, some output statements are automatically inserted into these two preprocessed code fragments (this process is called "code instrumentation"). By running two code fragments, their running logs can be obtained. Next, the flow charts of the two code fragments are obtained by mining the logs of these two code fragments by a process mining algorithm. Finally, a graph similarity measure algorithm is used to calculate the similarity between these two flow charts, and the value is regarded as the final similarity between these two code fragments.

4. Proposed Approach

The source code similarity detection approach based on process mining is introduced in detail in this section according to the above framework. The approach mainly contains the following four steps: (1) preprocessing redundant statements based on PDGs; (2) automatic source code instrumentation; (3) the code flow charts generation based on process mining; (4) similarity calculation of code flow charts. In addition, three examples are given to show the effectiveness of the approach in fighting against the above three complex code obfuscation techniques.

4.1. Preprocessing Redundant Statements Based on PDGs. Adding redundant statements is a common code obfuscation technique. For example, redundant variable declaration statements can reduce the similarity of two code fragments. Therefore, it is necessary to remove redundant statements. For the code assignments submitted by students in OJ system, the last statement is usually a *return* or an *output* statement. Therefore, if the source code is converted into a PDG, the nodes in the PDG can be traversed in depth-first order from the final node of the PDG that corresponds to the final statement, and a new graph can be obtained. The statements that correspond to the nodes not in the graph can be regarded as redundant statements and can be deleted. Therefore, if there is no data dependence and control dependence between the redundant statements and the core code, we can convert the code into a PDG and find redundant statements. Algorithm 1 presents the PDG-based preprocessing algorithm for removing redundant statements, which converts the source code *oc* into a PDG $G(V, E, \mu, \delta)$. In this algorithm, V is the set of statement nodes, E is the set of dependency edges between nodes in V, μ is the type mapping function for statement nodes, and δ is the type mapping function (data dependency or control dependency) for edges [6].

We take the code fragment in Figure 4(a) as an example. Given the code fragment to get the maximum value of two variables: *a* and *b*, the statements on lines 2, 8, and 9 are redundant statements and they have no data dependence with the last *return* statement in line 10. Figure 4(b) is the PDG generated by the code fragment, and Figure 4(c) is the PDG after deleting the redundant statements. Finally, the redundant statements on lines 2, 8, and 9 are removed.

4.2. Automatic Source Code Instrumentation. A certain amount of running logs is needed to obtain the flow chart of

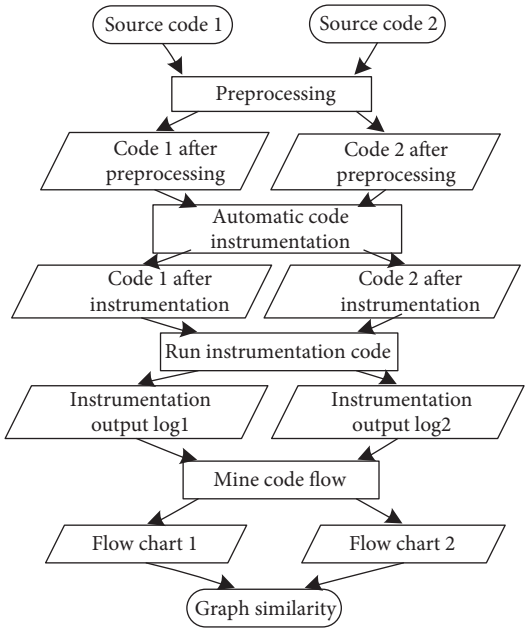


Figure 3: A general framework of source code similarity detection using process mining.

Input: The original code *oc*
Output: The target code *tc*
(1) Generate the PDG of *oc: G(V, E, μ, δ)*;//convert *oc* to a PDG
(2) Denote the end node as $G.v_e$, the target node set as *V'*; //*V'* is the node set that are dependent on the last node
(3) **for** each $v \in V$, $t = \mu(v)$ **do**
(4) **if** $(t == return || t == output)$ **then**//determine whether the current node *t* is a *return* statement or an *output* statement
(5) $v_e = v$;
(6) break;
(7) **end if**
(8) **end for**
(9) $V' = \boldsymbol{DFS}(V)$//get node set that are dependent on the last node
(10) **for** each $v \in G.V$ **do**
(11) **if**$(v \notin V')$ **then**//delete nodes that do not depend on the last node
(12) $tc = oc$.delete(v)
(13) **end if**
(14) **end for**

Algorithm 1: Redundant statements preprocessing based on PDGs.

a code fragment by process mining. However, there are not enough print statements that can indicate the flow of the code assignments. Therefore, it is necessary to insert some print statements into the code fragment to output some running logs of the code. Meanwhile, the output logs need to indicate the execution path of the code fragment, so that they can be used as the input of a process mining algorithm. By existing instrumentation software as well as the definition of nodes in PDG [6], we define two basic types of source code instrumentation statements: *Assign* and *Output*.

(1) **Assign ($\mathbf{Type_1}$,..., $\mathbf{Type_n}$)** represents variable declaration or variable assignment. $\mathbf{Type_1}$,..., $\mathbf{Type_n}$ are the variable types of *Assign*, where $Type_i \in$ {the variable types provided by a program language}.

(2) **Output** represents the print statement in a code fragment.

Based on two instrumentation statements, we give the following rules for source code instrumentation.

(1) Instrumentation statements *Assign* ($Type_i$) are inserted after assignment and variable declaration statements. If there is no data dependency between multiple consecutive assignment statements in a basic block [27], an instrumentation statement is inserted after the last assignment statement, and meanwhile the variable types of all assignment statements are merged; i.e., *Assign* ($Type_1$,..., $Type_n$) is inserted at the end of these statements.

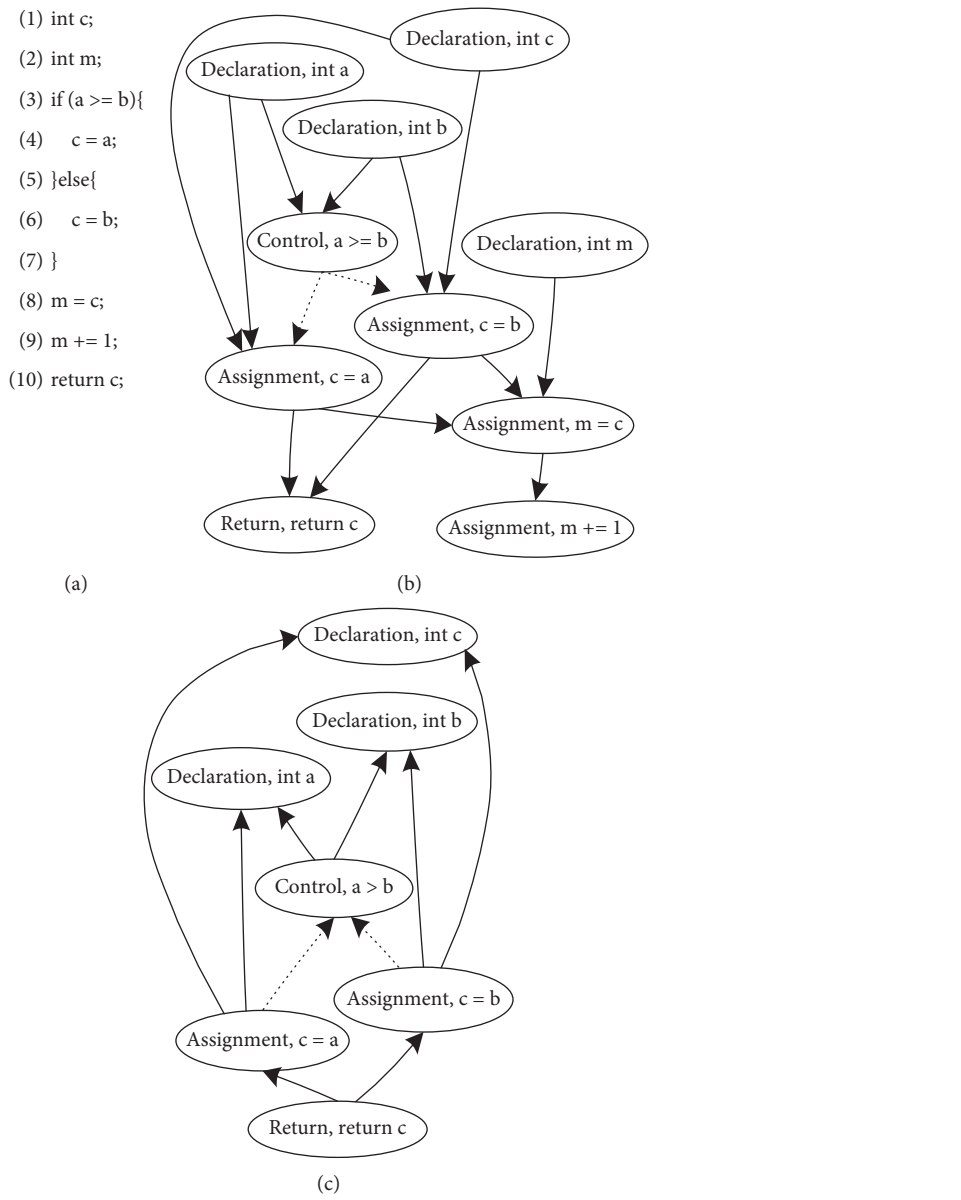


Figure 4: An example of PDG-based redundant statements preprocessing. (a) Source code. (b) PDG. (c) Graph after preprocessing.

(2) For the output statement, *Output* instrumentation is inserted after it.

(3) Instrumentation statements of the same type are numbered sequentially.

Figure 5 shows an example of automatic source code instrumentation of a piece of Java code based on the above instrumentation rules. The first four lines of the code are consecutive assignment statements in a basic block. The assignment statements in lines 1 and 2 have no data dependency, so they are combined, and *Assign1(int, int)* is inserted according to the first rule. The statements in lines 3 and 4 have data dependency and two instrumentation statements, i.e., *Assign2(Scanner)* and *Assign3(int)*, are inserted after them, respectively. Since there are two basic blocks in the *if* branch (line 5–9), two *Assign* instrumentation statements are inserted for the assignment statement in each basic block. For the output statement in the last sentence, the *Output* instrumentation is inserted after it.

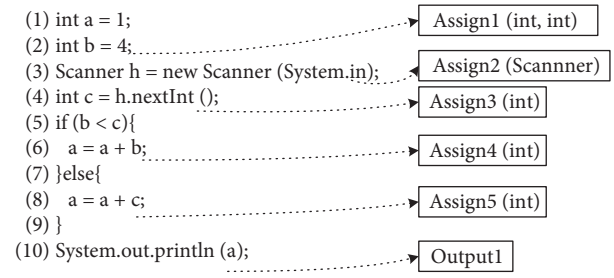


Figure 5: An example of source code instrumentation.

Input: instrumentation output sequences by running the source code, *Seq*
Output: CDFC = (*V*, *E*)
(1) Denote the following directly threshold as T_f, the threshold of dependence as T_d, the number of following directly as *num*[][] = 0, the dependence as *d*[][], the instrumentation output set as *IO*, the node set of CDFC *as V*, and the edge set of CDFC as *E*.
(2) **for** each *trace* in *Seq* **do**//traverse the instrumentation output sequences of each running of the code
(3) **for** $i = 0$; $i <$ *trace.size*-1; i++ **do**//traverse the adjacent instrumentation outputs in the instrumentation output sequences
(4) *num*[*trace*[*i*]*trace*[*i*+1]]++//record the following directly number of every two instrumentation outputs
(5) **if** *trace*[*i*] not exist in *V* **then**
(6) add *trace*[i] to *V*;
(7) **end if**
(8) **end for**
(9) **end for**
(10) **for** each io_1 in *IO* **do**
(11) **for** each io_2 in *IO* **do**
(12) **if** $io_1 == io_2$ **then**//the output of two instrumentation outputs is the same
(13) $d[io_1][io_2] = num[io_1][io_2]/(num[io_1][io_2] + 1)$//calculate the dependence of every two instrumentation outputs
(14) **if** $num[io_1][io_2] \geq T_f$ and $d[io_1][io_2] \geq T_d$ **then**
(15) *add* (io_1, io_2) to *E*
(16) **end if**
(17) **end if**
(18) **if** $io_1 != io_2$ **then**
(19) $d[io_1][io_2] = (num[io_1][io_2]\text{-}num[io_2][io_1])/(num[io_1][io_2] + num[io_2][io_1] + 1)$//calculate the dependence of every two instrumentation outputs
(20) **if** $num[io_1][io_2] \geq T_f$ and $d[io_1][io_2] \geq T_d$ **then**
(21) *add* (io_1, io_2) to *E*
(22) **end if**
(23) **end if**
(24) **end for**
(25) **end for**
(26) return CDFC = (*V*, *E*)

Algorithm 2: CDFC mining algorithm based on the heuristic process mining.

4.3. Source Code Flow Chart Generation Using Process Mining. The output logs of the source code can be obtained by running the code with instrumentation (instrumentation code for short). Then, a process mining algorithm can be used to mine the output logs that are made up of *Assign* and *Output* instrumentation statements to get the actual running process of the code that is expressed by a process model.

Existing process mining algorithms mainly include the α algorithm and the heuristic algorithm [28]. Among them, the α algorithm cannot deal with short loops with length of one or two. As a result, the algorithm cannot deal with the cases where the loops are executed only once or twice in the flow chart of the code. Heuristic mining algorithms [28] consider the frequency of events and sequences when building a process model, by which the actual running processes such as loops and branches in the source code can be mined. Therefore, we use the heuristic mining algorithm to mine the output logs and generate the code flow chart based on a causal network [28]. Since the flow chart is mined from the output logs produced by running the source code, it can indicate the dynamic features of the code. Therefore, we call such kind of flow charts *Code Dynamic Flow Chart*, which is defined as follows.

Definition 1. CDFC (Code Dynamic Flow Chart) = (*V*, *E*), where *V* is the set of *Assign* or *Output* nodes and $E \subseteq V \times V$ is the set of edges that represent the sequential relationship of nodes in *V*.

The output logs obtained by each running of the instrumentation code can be regarded as the running logs of a process instance that is required by a process mining algorithm. Meanwhile, each *Assign* and *Output* in the running logs of the instrumentation code (instrumentation output for short) can be regarded as an activity that is executed in the process instance. The sequence composed of *Assign* and *Output* is called the instrumentation output sequence, and it is regarded as the input the process mining algorithm. Based on this idea, we propose the CDFC mining algorithm that takes the logs obtained by running the instrumentation code as input, and outputs the corresponding CDFC based on a causal network. The instrumentation code needs to be run multiple times, and thus a set of output logs that are required by the process mining algorithm can be obtained. Then, the CDFC is obtained using the heuristic mining algorithm according to the occurrence times and dependence of *Assign* and *Output* in the instrumentation output sequence. Algorithm 2 shows the CDFC mining process based on the heuristic process mining algorithm.

Algorithm 2 first counts the directly following number (lines 2–6) of each two adjacent *Assign* or *Output* in the instrumentation output sequence. Then, the dependency (lines 10 and 16) between two instrumentation outputs is calculated. Finally, the connection between the instrumentation outputs is built. Thus, the code flow chart is obtained according to the directly following number and the dependency values (lines 11–13, lines 17–19).

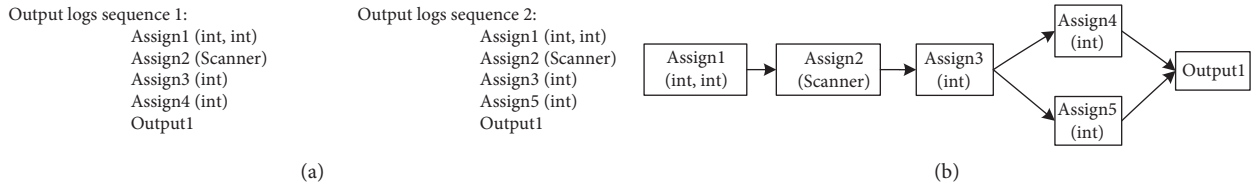


Figure 6: An example of CDFC mining. (a) Code instrumentation output logs. (b) CDFC.

Input: $\mathrm{CDFC}_1 = (V_1, E_1)$, $\mathrm{CDFC}_2 = (V_2, E_2)$, where, $V_1 = \{v_{11},..., v_{1n}\}$ and $V_2 = \{v_{21},..., v_{2m}\}$
Output: Minimum cost path $\lambda_{\min}$ that CDFC_1 is transformed to CDFC_2
(1) $OPEN \longleftarrow \{\emptyset\}$, $\lambda_{\min} \longleftarrow \emptyset$;
(2) **if** v_{1i} is a subset of v_{2j} **then**
(3) $v_{1i} = v_{2j}$;
(4) **end if**
(5) **for** each node $w \in V_2$ **do**
(6) $OPEN \longleftarrow OPEN \cup \{v_{11} \longrightarrow w\}$;
(7) $OPEN \longleftarrow OPEN \cup \{v_{11} \longrightarrow \varepsilon\}$; //the operation and cost value of v_{11} that replaces and deletes each vertex w in CDFC_2 are stored into *OPEN*
(8) **end for**
(9) **while** true **do**
(10) $OPEN \longleftarrow \lambda_{\min} = \mathrm{argmin}_{\lambda \in \mathrm{OPEN}}\{g(\lambda) + h(\lambda)\}$; //g($\lambda$) is the accumulated distance value from the starting point to the current search point, $h(\lambda)$ is the estimated distance value from the current point to the end point, and the point with the smallest function value is the next search point
(11) **if** $\lambda_{\min}$ is a complete edit path **then**
(12) return $\lambda_{\min}$;
(13) **else**
(14) $\lambda_{\min} = \{v_{11} \longrightarrow v_{2i},..., v_{1k} \longrightarrow v_{2k}\}$
(15) **if** $k < |V_1|$ **then**
(16) **for** each $w \in V_2 \backslash \{v_{2i},...,v_{2k}\}$**do**//$V_2 \backslash \{v_{2i},...,v_{2k}\}$ represents unmatched nodes in V_2
(17) $OPEN \longleftarrow \lambda_{\min} \cup \{v_{1k+1} \longrightarrow w\}$;
(18) $OPEN \longleftarrow \lambda_{\min} \cup \{v_{1k+1} \longrightarrow \varepsilon\}$;
(19) **end for**
(20) **else**
(21) **for** each $w \in V_2 \backslash \{v_{21},...,v_{2k}\}$ **do**
(22) $OPEN \longleftarrow \lambda_{\min} \cup \{\varepsilon \longrightarrow w\}$;
(23) **end for**
(24) **end if**
(25) **end if**
(26) **end while**

Algorithm 3: Graph edit distance of two CDFCs.

Figure 6 gives an example, in which Figure 6(a) shows two instrumentation output sequences of the instrumentation code in Figure 5. Through two test cases with the input 5 and 3, respectively, two different instrumentation output sequences are obtained. As is shown in Figure 6(a), either *Assign*4 or *Assign*5 is executed in each instrumentation output sequence. Therefore, a mutually exclusive branch is included in the CDFC obtained based on Algorithm 2, which is shown in Figure 6(b).

4.4. CDFC Similarity Measure Based on Graph Edit Distance. The similarity between two code fragments can be measured by the similarity between two CDFCs. Because a CDFC is a directed graph, the similarity measure approach for directed graphs can be used. In recent years, researchers have proposed many approaches for graph similarity measure, among which graph edit distance (GED) [29] is a commonly used algorithm. GED calculates the transformation intensity that is needed to transform one graph into another, and therefore it measures the dissimilarity between two graphs. We use GED-based algorithm to calculate the similarity of two CDFCs.

Let $\mathrm{CDFC}_1 = (V_1, E_1)$ and $\mathrm{CDFC} = (V_2, E_2)$. The definition of graph edit distance between CDFC_1 and CDFC_2 is shown as follows:

$$\mathrm{d}_{\lambda\,\min}\left(\mathrm{CDFC}_1, \mathrm{CDFC}_2\right) = \min_{\lambda \in \gamma\left(\mathrm{CDFC}_1, \mathrm{CDFC}_2\right)} \left\{\sum_{e_j \in \lambda} c\left(e_j\right)\right\} \tag{1}$$

Among them, $\lambda_{\min}$ denotes the minimum cost and path among all the complete editing paths; γ (CDFC_1, CDFC_2)

denotes all the editing paths from $CDFC_1$ to $CDFC_2$; and $c(e_j)$ denotes the cost of editing e_j. In the proposed approach, for the *Assign* nodes in a CDFC, if the variable types in an *Assign* node are the subset of the variable types in another one, these two *Assign* nodes represent the same node. Because each edge in a CDFC points from one node to another, the cost of $replacement(v \longrightarrow w)$, $deletion(v \longrightarrow \varepsilon)$, and $addition(\varepsilon \longrightarrow v)$ of nodes is the same as that of $deleting(v \longrightarrow \varepsilon)$ and $adding(\varepsilon \longrightarrow v)$ of edges. In this paper, the operation cost function of the node and that of the edge are set to one.

The graph edit distance of two CDFCs based on GED is calculated in Algorithm 3. Its basic idea is to make every possible editing operation be considered by processing the nodes in $CDFC_1$ one by one. Specifically, a search tree is built dynamically and the process of traversing the search tree is equivalent to the process of solving an editing path.

In Algorithm 3, the first vertex v_{11} of $CDFC_1$ is selected by traversing the search tree. The operation and cost value of v_{11} that replaces and deletes each vertex w in $CDFC_2$ are stored in an initial empty set (lines 5–8). Then, the nodes with the lowest operation cost are selected to be traversed according to the heuristic search algorithm, and the *replacement* and *deletion* operations of the remaining nodes are inserted into the *OPEN* set (lines 17 and 18). After all the node operations of *replacement* and *deletion* in $CDFC_1$ are completed, the remaining nodes after processing in $CDFC_2$ are inserted (lines 21–23). Finally, the edit distance path λ_{min} of $CDFC_1$ and $CDFC_2$ is obtained.

Given the flow chart of the source code $CDFC_1 = (V_1, E_1)$ and the flow chart of the obfuscated code $CDFC_2 = (V_2, E_2)$, λ_{min} is the edit distance between $CDFC_1$ and $CDFC_2$. $|CDFC| = |CDFC.V| + |CDFC.E|$ denotes the module value of CDFC, where $|CDFC.V|$ denotes the number of nodes in CDFC, and $|CDFC.E|$ denotes the sum of the length of all edges in CDFC. The similarity of the source code and the obfuscated code can be calculated as

$$sim(CDFC_1, CDFC_2) = 1 - \frac{\lambda_{min}}{\max(|CDFC_1|, |CDFC_2|)}. \quad (2)$$

4.5. Effectiveness in Fighting Against Code Obfuscation. In this section, we analyze the effectiveness of the proposed approach in fighting against three relatively complex code obfuscation techniques, opaque predicates, loop unrolling, and function inlining and outlining, by three examples.

Firstly, in terms of opaque predicates, unreachable paths can be added to reduce the similarity of two code fragments. Taking the code fragment in Figure 2 as an example, an unreachable *if* branch is added (lines 10–12) in Figure 2(c). In addition, the statements in lines 2–4 of Figure 2(a) represent a loop structure, and the statements in lines 2–8 of Figure 2(c) constitute the code that is obfuscated by copying the loop body while the result from running the code does not change. The source code and obfuscated code in Figure 2 with inserted instrumentation code by the proposed approach are as shown in Figures 7(a) and 7(d). Because the *if* branch of the obfuscated code cannot be executed, the code in the *if* branch will not be executed. In addition, because the loop structure is changed, the code instrumentation output sequences are different after running the two instrumentation code fragments. There are one hundred and two lines in the source code instrumentation output sequences, while there are only 22 lines in the obfuscated code instrumentation output sequences, which are shown in Figures 7(b) and 7(e), respectively. The CDFCs of the two code fragments obtained by the heuristic process mining algorithm are shown in Figures 7(c) and 7(f), respectively. Because the variable types in *Assign2* node in the CDFC of the source code are a subset of that in *Assign2* nodes in the CDFC of the obfuscated code, the similarity between the two CDFCs is 100% according to Algorithm 3. Therefore, the proposed approach can fight against the opaque predicates and loop unrolling used in the example of Figure 2.

Secondly, we analyze the effectiveness of the proposed approach in fighting against function inlining and outlining. Figure 8 shows an example of the source code to get the maximum value between two input integers. The obfuscated code uses the *if* branch structure in the source code as a method invocation, and a new function is added. Therefore, the textual similarity between these two code fragments is reduced. However, the CDFCs obtained by the proposed approach are still the same, as shown in Figure 8(c). Therefore, the proposed approach can fight against function inlining and outlining used in this example.

5. Experiment and Evaluation

We conduct a group of experiments to evaluate the effectiveness and efficiency of the proposed approach for source code similarity detection in computer programming teaching. First, the experimental setup is introduced in 5.1. The data set used in the experiments is introduced in 5.2. Next, in 5.3, the effectiveness in fighting against code obfuscation techniques of the proposed approach is compared with that of existing source code similarity measure approaches. In addition, the efficiency of the proposed approach is verified compared with Sim and GPLAG in 5.4. Finally, the conclusions of the experiments are given.

5.1. Experimental Setup. For a pair of source code and obfuscated code, their similarity is calculated by the proposed approach and existing state-of-the-art approaches. Because Sim and GPLAG are representatives of the string-based and graph-based similarity measure approaches, respectively, we choose these two approaches to be compared with our approach. Thus, we can verify the effectiveness of the proposed approach in source code similarity detection and the ability to fight against code obfuscation techniques.

We conduct three experiments. First, for a data set of source code, we use three obfuscation techniques in the first experiment to modify the source code: opaque predicate, loop unrolling, and function inlining and outlining. In this way, we can construct the source code and obfuscated code pairs. Then, the similarity of each pair of code is calculated

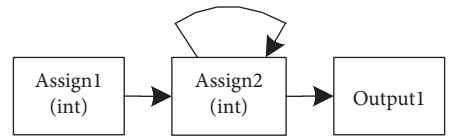
```
int sum = 0;
System.out.println ("Assign1(int)");
for (inti = 1;i < = 100;i++){
    sum = sum + i;

System.out.println ("Assign2(int)");
}
System.out.println (sum);
System.out.println ("Output1");
```

(a)

```
Assign1 (int)
Assign2 (int)
Assign2 (int)
...
...
...
Assign2 (int)
Output1
```

(b)



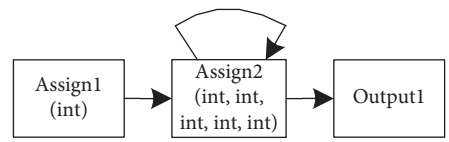
(c)

```
int sum = 0;
System.out.println ("Assign1(int)");
for (inti = 1;i < = 100;i + = 5){
        sum = sum + i;
        sum = sum + i + 1;
        sum = sum + i + 2;
        sum = sum + i + 3;
        sum = sum + i + 4;
        System.out.println ("Assign2(int,int,
        int,int,int)");
}
System.out.println (sum);
System.out.println ("Output1");
if (sum == 0){
        System.out.println (”Resulterror”);
        System.out.println (”Output2”);
}
```

(d)

```
Assign1 (int)
Assign2 (int, int, int, int, int)
Assign2 (int, int, int, int, int)
...
...
...
Assign2 (int, int, int, int, int)
Output1
```

(e)



(f)

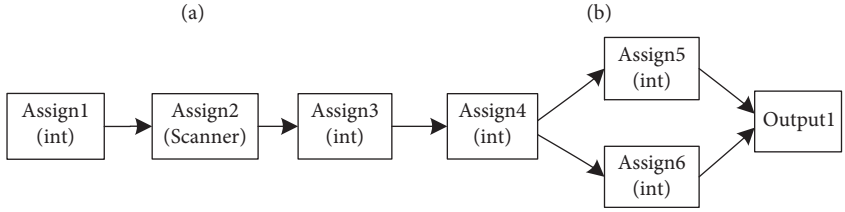
Figure 7: Examples of opaque predicates and loop unrolling. (a) Source code instrumentation. (b) Source code instrumentation output sequence. (c) Source code flow chart. (d) Obfuscated code instrumentation. (e) Obfuscated code instrumentation output sequence. (f) Obfuscated code flow chart.

```
(1)  int a = 2;
(2)  Scanner m = new Scanner (System.in);
(3)  int b = m.nextInt();
(4)  int c = m.nextInt();
(5)  if (b < c){
(6)      a = a + b;
(7)  }else{
(8)      a = a + c;
(9)  }
(10) System.out.println (a);
```

(a)

```
(1)   int a = 2;
(2)   Scanner m = new Scanner(System.in);
(3)   int b = m.nextInt();
(4)   int c = m.nextInt();
(5)   a = Min(b,c);
(6)   System.out.println(a);
(7)  private static int Min(int b, int c){
(8)   if (b < c){
(9)       b = b + 2;
(10)      return b;
(11)   }else{
(12)      c = c + 2;
(13)      return c;
(14)  }
(15)  }
```

(b)



(c)

Figure 8: An example of function inlining and outlining. (a) Source code. (b) Obfuscated code. (c) Code flow chart.

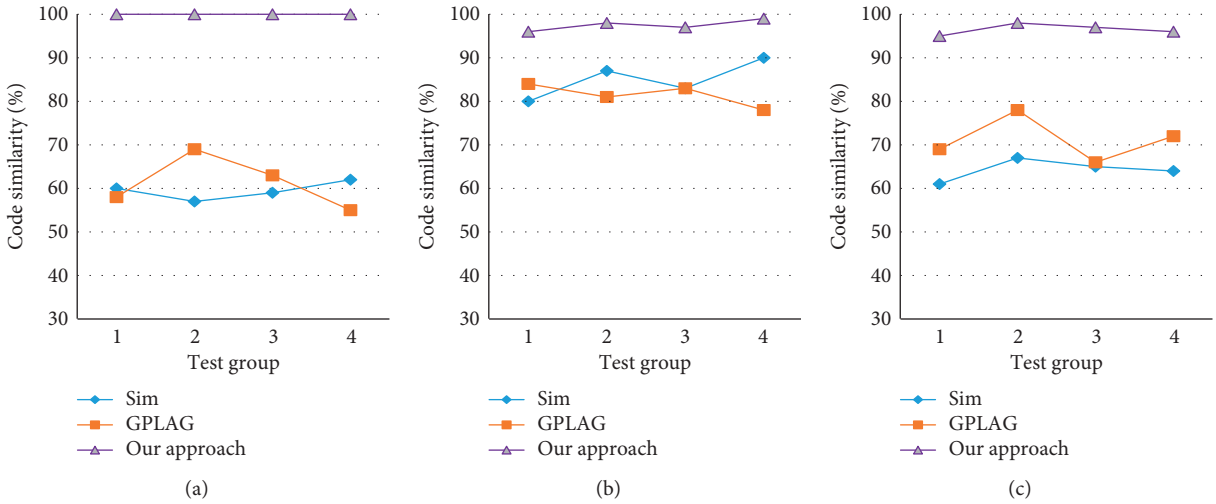


Figure 9: Experimental results of three approaches in fighting against opaque predicates, loop unrolling, and function inlining and outlining. (a) Opaque predicate. (b) Loop unrolling. (c) Function inlining and outlining.

by the proposed approach, Sim, and GPLAG to verify the effectiveness of the proposed approach in fighting against the above three obfuscation techniques.

Second, to verify the practicability of the proposed approach in source code similarity detection, experiment 2 evaluates the antiobfuscation ability of the proposed approach by some commonly used obfuscation techniques that existing approaches can fight against. Specifically, Sim, GPLAG, and the proposed approach are used to compare the average similarity between the source code and the obfuscated code that is processed by seven commonly used code obfuscation techniques.

The first two experiments demonstrate the ability of the proposed approach to fight against a single code obfuscation technique. To further verify the effectiveness of the proposed approach against multiple code obfuscation techniques, we select more complex source code and use a few obfuscation techniques to process the source code in the third experiment. Then, we use Sim, GPLAG, and our approach to calculate code similarity and compare their effectiveness in antiobfuscation.

In addition, the efficiency of Sim, GPLAG, and our approach is compared in the fourth experiment to verify the efficiency of the proposed approach.

5.2. Data Sets. We take the code assignments submitted by students in the OJ system as data sets. Because code obfuscation techniques used by students in the data set are not complete, we obfuscate the code manually and construct three source code sets. It should be noted that each exercise problem in OJ system contains a set of test cases, in which the input data of each test case can be used as the input of the proposed approach when running the source code. Therefore, through running all test cases, the instrumentation output sequences of the code can be obtained.

The first code set is used for the first experiment. We randomly select twenty Java programming problems from our OJ system and divide them into four groups, and therefore there are five problems in each group. Then, the code assignment of a student that is selected randomly is modified using three obfuscation techniques: opaque predicate, loop unrolling, and function inlining and outlining. As a result, for each obfuscation technique, an experimental data set consisting of twenty pairs of source code and obfuscated code in four groups is obtained.

The second code set is used for the second experiment. We randomly select twenty Java problems with medium length from our OJ system. Meanwhile, the code assignment of a student which is randomly selected is obfuscated using the following seven code obfuscation techniques: (1) changing comments; (2) renaming identifiers; (3) reordering statements within code blocks; (4) replacing control structures with equivalent structures; (5) replacing constants; (6) changing data types; (7) splitting expressions. As a result, there are seven different obfuscated code fragments for each problem. For each obfuscation technique, twenty pairs of source code and obfuscated code are obtained in this experiment.

The third code set is used for the third experiment. We choose three problems with high complexity from our OJ system. Meanwhile, we select a code assignment of a student randomly for each problem and use the following five code obfuscation techniques to modify the code: renaming identifiers, reordering statements within code blocks, opaque predicates, replacing control structures with equivalent structures, and function inlining and outlining. Thus, three pairs of source code and obfuscated code are obtained in this experiment.

5.3. Experimental Results. Figure 9 shows the result of the first experiment, in which the ordinate axis stands for the average similarity between source code and obfuscated code for each group of five questions, and the abscissa axis represents four code groups. The source code is obfuscated by opaque predicate, loop unrolling, and function inlining

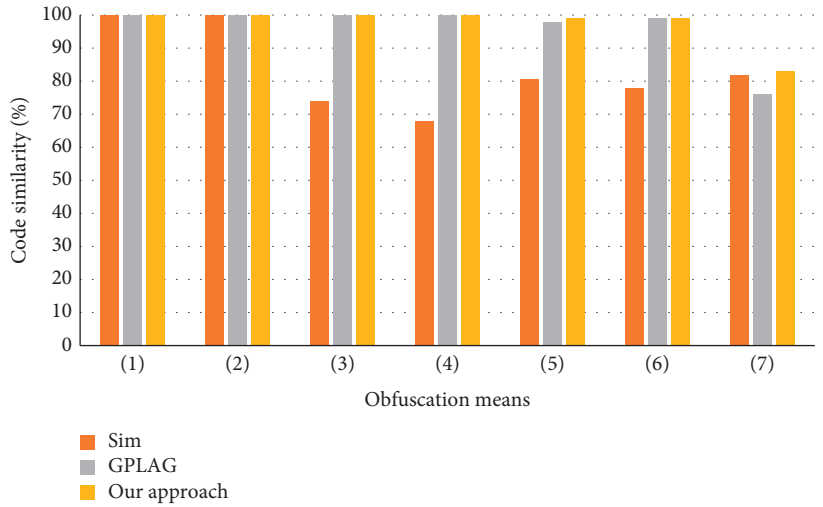


Figure 10: Experimental results of three approaches in fighting against seven obfuscation techniques.

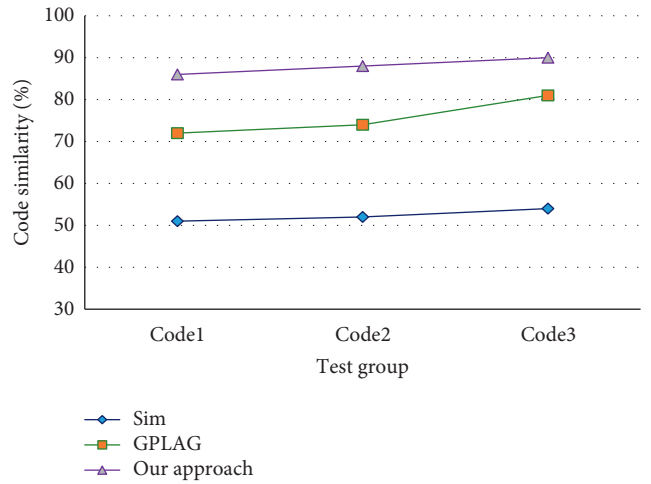


Figure 11: Experimental results of three approaches in fighting against multiple code obfuscation techniques.

Table 1: Efficiency of the three approaches.

Code	Code byte	Matches	Sim (s)	GPLAG (s)	Our approach (s)
Code 1	3147	58	1.2	1.87	2.32
Code 2	3871	72	1.95	2.79	3.14
Code 3	3655	69	1.72	2.71	2.91

and outlining. Opaque predicates and loop unrolling obfuscation techniques add redundant strings into the source code, while the function inlining and outlining disturbs the order of some strings in the source code. At the same time, the structures of PDGs are also changed. As a result, the antiobfuscation ability of Sim and GPLAG in fighting against these three kinds of code obfuscation techniques is poorer than the proposed approach. The approach in this paper can obtain the actual flow of the code by running the code, and therefore it can detect opaque predicates. For the source code obfuscated by loop unrolling as well as function inlining and outlining, some redundant code that has data dependence with the final *return* or *print* statements of the code may be added to the obfuscated code. As a result, this part of the redundant code cannot be deleted by the preprocessing, making the similarity result of the proposed approach slightly lower. However, the detection precision of the proposed approach is still better than that of Sim and GPLAG. To sum up, compared with Sim and GPLAG, the proposed approach has better effectiveness in fighting against opaque predicate and loop unrolling as well as function inlining and outlining.

Figure 10 shows the similarity between the source code and the obfuscated code given by three code similarity detection approaches in the second experiment. Among them, the ordinate axis represents the average similarity of twenty pairs of source code and obfuscated code that is obfuscated by one code obfuscation technique, and the abscissa axis represents the seven code obfuscation techniques. It can be seen that Sim, GPLAG, and the proposed approach can deal with the simple code obfuscation techniques including changing comments and renaming identifiers; Sim cannot fight against adjusting the order of statements in code blocks, replacing control structures with equivalent structures, replacing constants, and changing data types, while GPLAG and the proposed approach can defeat these four obfuscation techniques. Moreover, these three approaches cannot fully fight against splitting expressions because this obfuscation technique splits a complex expression into multiple simple expressions and add multiple lines of code with data dependency. To sum up, the proposed approach can defeat the code obfuscation techniques that can be handled by existing approaches.

Figure 11 shows the result of the third experiment, in which the ordinate axis stands for the similarity value between the source code and the code obfuscated by multiple obfuscation techniques. The code similarity values given by Sim are all less than 60%. GPLAG cannot be affected by reordering statements within code blocks and replacing control structures with equivalent structures, and therefore the code similarity detection effectiveness is better than that of Sim. However, GPLAG measures the code similarity by the static PDG of the source code, and it cannot obtain the actual running characteristics of source code. Therefore, GPLAG cannot fight against opaque predicates and function inlining and outlining, making the source code similarity of each group lower than that of our approach.

5.4. Efficiency Evaluation. We use three groups of source code and obfuscated code pairs in the third experiment to evaluate the efficiency of the proposed approach. Specifically, we count the average byte length of the code in each group and the number of nodes in each generated CDFC and compare the execution time of the proposed approach with that of Sim and GPLAG. The experiment is carried out on a computer with 3.0 GHz Intel(R) Core(TM) i5-7267U CPU, 8 G memory, and Win 10. Each group of experiments is carried out ten times, and the average running time is taken as the result. The experimental results are shown in Table 1. Sim is the fastest because it is a string-based approach to measuring the similarity. In the proposed approach, source code instrumentation, running code, and the process mining are needed, and therefore the execution time is more than that of Sim and GPLAG. However, the proposed approach can provide acceptable performance in detecting the similarity of code assignments in computer programming teaching.

6. Conclusion

In the teaching of computer programming courses, the code assignments submitted by students through online judge system are generally the code to solve a specific problem. This kind of code is usually short, and the overall complexity is not high. However, students may use some complex code obfuscation techniques, such as opaque predicates, loop unrolling, and function inlining and outlining, to reduce the similarity between source codes. Aiming at source code similarity detection in computer programming teaching, we propose an approach to measure source code similarity based on process mining. Through the running of source code and the mining of output logs, the dynamic features that indicate the actual flow of the source code are obtained and used to calculate the code similarity. The results show that the proposed approach can fight against not only the code obfuscation techniques that existing approaches can defeat, but also more complex opaque predicates and loop unrolling as well as function inlining and outlining, which existing approaches cannot defeat. Therefore, the proposed approach has stronger antiobfuscation ability compared with the existing approaches. From the perspective of the efficiency, the complexity of the proposed approach is higher than that of the existing approaches because it involves code instrumentation, code running, process mining, and graphs similarity measure. Therefore, the performance needs to be improved further. However, the approach can meet the actual requirement of code similarity detection in computer programming teaching. Meanwhile, the proposed approach can be combined with the existing approaches in the practical application of OJ and other systems. For example, some existing approach can be used first to measure the code similarity. Then, the proposed approach can be used further when the obtained similarity is lower than a given threshold. In addition, the dynamic flow chart of the source code obtained by process mining represents the dynamic feature of the source code by the graphical process, which can also provide the basis for code plagiarism.

As an improvement of existing static analysis-based approaches, the proposed approach explores source code similarity measure based on dynamic features. However, there are still some problems that need to be further investigated. First of all, the approach cannot defeat some more complex obfuscation techniques, such as adding sequential statements that have data dependence with the core process, adding reachable branches, and splitting expressions. Second, the efficiency of the proposed approach needs to be further improved because of its higher time complexity. Therefore, the following possible future works should be explored. First, combined with the existing static similarity detection approaches, the code similarity detection approaches combining static and dynamic features can be investigated. In addition, there is some work that uses machine learning techniques [30] to measure the similarity of the source code and obtain higher precision. Therefore, we can strengthen our proposed approach by machine learning techniques, especially the deep learning techniques [31]. Second, the source code instrumentation statements and rules of the proposed approach are relatively simple. As a result, for two code fragments, the obtained similarity may be higher than their real similarity in some cases. Therefore, the source code instrumentation statements and rules can be

further optimized to obtain more accurate similarity and deal with more complex code obfuscation techniques. Finally, the efficiency of the approach needs to be further improved to be applied to the similarity detection of larger-scale source code.

Data Availability

We took the code assignments submitted by students in the OJ system as data sets, and an encrypted version of the data sets is available from the corresponding author upon request.

Conflicts of Interest

The authors declare that there are no conflicts of interest regarding the publication of this paper.

Acknowledgments

This research was funded by the Education Ministry Humanities and Social Science Research Youth Fund Project of China ("User-Steering Multi-Source Education Data Integration Approach Research in Big Data Environment" with grant number 19YJCZH240 and "Research on the Dynamic Evolution Tracking and Evaluation Method of Government's Internet Word-of-Mouth in Dealing With Emergencies Based on Big Data" with grant number 18YJAZH017); Qingdao Social Science Planning Research Project (grant number QDSKL1901123); the NSFC (grant numbers U1931207, 61902222, and 31671588); Sci. & Tech. Development Fund of Shandong Province of China (grant numbers 2016ZDJS02A11 and ZR2017MF027); Taishan Scholars Program of Shandong Province (tsqn201909109 and ts20190936); SDUST Research Fund (grant number 2015TDJH102); the Education and Teaching Research "Constellation" Project of Shandong University of Science and Technology (grant number QX2018M22); and SDUST Excellent Teaching Team Construction Plan (grant number JXTD20180503).

References

[1] S. Engels, V. Lakshmanan, and M. Craig, "Plagiarism detection using feature-based neural networks," *ACM SIGCSE Bulletin*, vol. 39, no. 1, pp. 34–38, 2007.

[2] 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.

[3] K. W. Bowyer and L. O. Hall, "Experience using MOSS to detect cheating on programming assignments," in *Proceedings of the FIE'99 Frontiers in Education. 29th Annual Frontiers in Education Conference. Designing the Future of Science and Engineering Education. Conference Proceedings*, vol. 3, pp. 18–22, San Juan, PR, USA, November 1999.

[4] D. Gitchell and N. Tran, "Sim," *ACM SIGCSE Bulletin*, vol. 31, no. 1, pp. 266–270, 1999.

[5] J. Feng, B. Cui, and K. Xia, "A code comparison algorithm based on AST for plagiarism detection," *Emerging Intelligent Data and Web Technologies*, vol. 1, pp. 393–397, 2013.

[6] C. Liu, C. Chen, J. Han et al., "Detection of software plagiarism by program dependence graph analysis," in *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pp. 872–881, Philadelphia, PA, USA, August 2006.

[7] Z. Zhang, H. H. Yan, and X. W. Zhang, "Code similarity detection by program dependence graph," in *Proceedings of the International Conference on Computer Engineering and Information Systems*, pp. 255–261, Vienna, Austria, May 2016.

[8] Y. C. Jhi, X. Wang, X. Jia et al., "Value-based program characterization and its application to software plagiarism detection," in *Proceedings of the 33rd International Conference on Software Engineering*, pp. 756–765, Honolulu, HI, USA, May 2011.

[9] X. Wang, Y. C. Jhi, S. Zhu et al., "Detecting software theft via system call based birthmarks," in *Proceedings of the Twenty-Fifth Computer Security Applications Conference*, pp. 149–158, Tokyo, Japan, December 2009.

[10] 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.

[11] Q. Song, *Research on Cross-Language Code Similarity Detection Method Based on Program Flow Chart*, Shandong University of Science and Technology, Qingdao, China, 2019.

[12] J. Ming, F. Zhang, D. Wu, P. Liu, and S. Zhu, "Deviation-based obfuscation-resilient program equivalence checking with application to software plagiarism detection," *IEEE Transactions on Reliability*, vol. 65, no. 4, pp. 1647–1664, 2016.

[13] D. Fu, Y. Xu, H. Yu, and B. Yang, "Wastk: a weighted abstract syntax tree kernel method for source code plagiarism detection," *Scientific Programming*, vol. 2017, pp. 1–8, 2017.

[14] 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.

[15] C. Ragkhitwetsagul, J. Krinke, and D. Clark, "A comparison of code similarity analysers," *Empirical Software Engineering*, vol. 23, no. 4, pp. 2464–2519, 2018.

[16] V. J. Marin and C. R. Rivero, "Towards a framework for generating program dependence graphs from source code," in *Proceedings of the 4th ACM SIGSOFT International Workshop*, pp. 30–36, Lake Buena Vista, FL, USA, November 2018.

[17] M. Novak, D. Kermek, and M. Joy, "Calibration of source-code similarity detection tools for objective comparisons," in *Proceedings of the 41st International Convention on Information and Communication Technology*, Electronics and Microelectronics, Opatija, Croatia, pp. 0794–0799, May 2018.

[18] M. J. Mišić, D. V. Nikolov, J. Ž. Protić et al., "Parallelization of GST algorithm for source code similarity detection," in *Proceedings of the 24th Telecommunications Forum*, IEEE, Belgrade, Serbia, pp. 1–4, November 2016.

[19] M. Novak, M. Joy, and D. Kermek, "Source-code similarity detection and detection tools used in academia: a systematic review," *Acm Transactions on Computing Education*, vol. 19, no. 3, pp. 27–37, 2019.

[20] E. L. Jones, "Metrics based plagiarism monitoring," *Journal of Computing Sciences in Colleges*, vol. 16, no. 4, pp. 253–261, 2001.

[21] G. Myles and C. Collberg, "Detecting software theft via whole program path birthmarks," *Lecture Notes in Computer Science*, vol. 3225, pp. 404–415, 2004.

[22] G. Whale, "Software metrics and plagiarism detection," *Journal of Systems and Software*, vol. 13, no. 2, pp. 131–138, 1990.

[23] 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.

[24] M. Gabel, L. Jiang, and Z. Su, "Scalable detection of semantic clones," in *Proceedings of the 30th International Conference on Software Engineering (ICSE'08)*, pp. 321–330, New York, NY, USA, May 2008.

[25] L. Jiang, Z. Su, G. Misherghi et al., "DECKARD: Scalable and accurate tree-based detection of code clones," in *Proceedings of the 29th International Conference on Software Engineering*, Minneapolis, MN, USA, May 2007.

[26] R. Komondoor and S. Horwitz, "Using slicing to identify duplication in source code," in *Proceedings of the 8th International Symposium on Static Analysis*, pp. 40–56, Paris, France, July 2001.

[27] 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. 7, pp. 1171–1197, 2015.

[28] L. Maruster, A. J. M. M. Weijters, W. M. P. V. D. Aalst et al., "Process mining: discovering direct successors in process logs," in *Proceedings of the Discovery Science International Conference, DS 2002*, Lübeck, Germany, November 2002.

[29] Z. Abu-Aisheh, R. Raveaux, J. Y. Ramel et al., "An exact graph edit distance algorithm for solving pattern recognition problems," in *Proceedings of the International Conference on Pattern Recognition Applications and Methods*, vol. 1, pp. 271–278, Lisbon, Portugal, January 2015.

[30] Z. Huang, J. Tang, G. Shan, J. Ni, Y. Chen, and C. Wang, "An efficient passenger-hunting recommendation framework with multitask deep learning," *IEEE Internet of Things Journal*, vol. 6, no. 5, pp. 7713–7721, 2019.

[31] S. Zhao, D. M. Zhang, and H. W. Huang, "Deep learning–based image instance segmentation for moisture marks of shield tunnel lining," *Tunnelling and Underground Space Technology*, vol. 95, no. 1, pp. 1–11, 2020.