Hindawi Publishing Corporation
Mathematical Problems in Engineering
Volume 2015, Article ID 793697, 15 pages
http://dx.doi.org/10.1155/2015/793697

Research Article

Detecting Copy Directions among Programs Using Extreme Learning Machines

Bin Wang, Xiaochun Yang, and Guoren Wang

College of Information Science and Engineering, Northeastern University, Liaoning 110819, China

Correspondence should be addressed to Xiaochun Yang; yangxc@mail.neu.edu.cn

Received 21 August 2014; Revised 10 November 2014; Accepted 10 November 2014

Academic Editor: Tao Chen

Copyright © 2015 Bin Wang et al. This is an open access article distributed under the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

Because of the complexity of software development, some software developers may plagiarize source code from other projects or open source software in order to shorten development cycle. Many methods have been proposed to detect plagiarism among programs based on the program dependence graph, a graph representation of a program. However, to our best knowledge, existing works only detect similarity between programs without detecting copy direction among them. By employing extreme learning machine (ELM), we construct feature space for describing features of every two programs with possible plagiarism relationship. Such feature space could be large and time consuming, so we propose approaches to construct a small feature space by pruning isolated control statements and removable statements from each program to accelerate both training and classification time. We also analyze the features of data dependencies between any original program and its copy program, and based on it we propose a feedback framework to find a good feature space that can achieve both accuracy and efficiency. We conducted a thorough experimental study of this technique on real C programs collected from the Internet. The experimental results show the high accuracy and efficiency of our ELM-based approaches.

1. Introduction

The Internet and open source software are developing rapidly nowadays, providing developers easier accesses to get various open source software code. Meanwhile with the increase of users' need, software developers have to develop more complicated software product, leading to longer development cycle which directly determines development cost as well as enterprise profit. To save development cost, some illegal developers inevitably utilize all possible methods to shorten development cycle. One of the fastest methods is to develop their project on existing projects, open source software, or prototype products. Such behavior can be infringement to original authors. In order to prevent such infringement, a copy detection tool is needed to detect software source code plagiarism.

Many methods have been proposed to detect plagiarism among programs based on the program dependence graph (PDG for short) [1], a graph representation of a program. However, they only focus on detecting similarity but copy direction between any two similar programs (or PDGs). To our best knowledge, this is the first work to detect both plagiarism and copy direction among programs.

In this paper, we propose a framework based on ELM [2] to detect copy directions among programs. We first classify programs into a set of programs with possible plagiaristic relationship by adopting ELMs. Based on it, we detect copy direction by considering dependencies in programs.

The challenges and contributions of detecting copy directions are as follows.

(i) A source program and its plagiaristic program could be only partially similar since the plagiaristic program might contain some useless statements to make it different from the source program. As we know, finding common statements between two programs or common subgraphs between their corresponding PDGs is very time consuming. In this paper, we propose an extreme learning machine (ELM) based framework to learn this potential similarity and classify PDGs accordingly since ELM is well-known and very efficient for classification with high accuracy.

```
(1) int count(Istream * input, input c, int f1, int f2)
(2) {
(3)         int count, revise, valLeft, valRight, temp, findVal, count;
(4)         valLeft = getCrest(f1);
(5)         valRight = getCrest(f2);
(6)         revise = GlobalVal;
(7)         tmp = c*(valLeft - revise) + valRight;
(8)         findVal = temp/2;
(9)         count = 0;
(10)        char * inputStr;
(11)        while(inputStr = getNext(input))
(12)        {
(13)                    int inputVal = paserInt(inputStr);
(14)                    if(inputVal == 0)
(15)                    {
(16)                                int any = findVal + inputVal;
(17)                                int any2 = any + findVal;
(18)                    }
(19)                    if(inputVal == findVal)
(20)                                count++;
(21)        }
(22)        return count;
(23) }
```

Figure 1: A program.

(ii) A plagiaristic program might contain useless control statements like loops, selective structures, which makes PDGs of a source program and its plagiarism dissimilar and results in low detection accuracy. In order to enhance the accuracy of classification, we further utilize control dependencies in PDGs to remove subgraphs corresponding to those useless control statements and shrink the size of PDGs in Section 4. We show using ELM to classify PDGs with small sizes not only increases the classification accuracy but also decreases classification time, since using smaller PDGs as training set means a smaller feature space, which accelerates both training and classification time. This ELM-based approach greatly exceeds the detection ability of existing algorithms.

(iii) Among a set of programs with plagiaristic relationship, how can we determine which one is the source program and which one copies it? To our best knowledge, this is the first work to solve the problem. In Section 5 we propose a scoring scheme by combining both control dependencies and data dependencies in PDGs to detect copy directions.

In Section 6 we present experimental results on real data sets to demonstrate the accuracy and time efficiency of the proposed technique.

2. Preliminary and Background

2.1. Computer Program. A computer program is a sequence of statements, written to perform a specific task with a computer. In a program, we mainly focus on two kinds of statements, control statement and other statement. A control statement always generates a boolean value, whereas other statements have no such requirement. Two kinds of dependencies are defined as follows.

(i) Control Dependency. A statement S is control-dependent on a control statement C, if execution of S depends on the boolean value of C.

(ii) Data Dependency. A statement S_1 is data-dependent on another statement S_2, if S_1 reads the value of a variable v that has been changed by its preceding statement S_2.

For instance, Figure 1 shows an example program, where statement `inputVal = paserInt (inputStr)` (line 13) is control-dependent on statement `inputStr = getNext (input)` (line 11), because the execution of the former depends on whether the boolean value of the latter is 1. Statement `findVal = temp/2` (line 8) is data-dependent on statement `temp = c* (valLeft − revise) + valRight` (line 7), since the former reads the value of variable `temp`, and thelatter does an assignment on `temp`.

2.2. Program Dependence Graph. A program dependence graph (PDG) [1] is a graph structure consisting of a program unit, such as statements and relations among variables in a C language function. Given a program P, let $G(P)$ be the program dependence graph of P. Each node in a PDG represents a program statement, and edges between nodes represent dependencies among statements.

Definition 1. A PDG is a graph $G(V, E)$ and it satisfies the following.

(i) The node set V in $G(P)$ represents the set of statements in program P.

(ii) The edge set E in $G(P)$ represents the set of data dependencies and control dependencies between statements in program P.

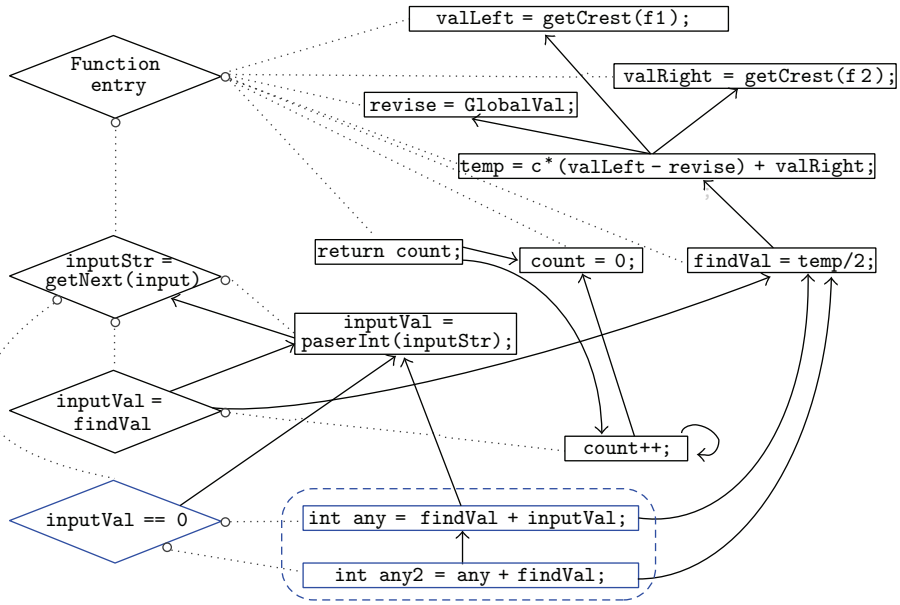


Figure 2: The PDG for the program shown in Figure 1.

For example, Figure 2 shows the PDG for the program in Figure 1. The PDG of every function starts from conditional statement "function entry." Each rhombic node in Figure 2 represents a conditional control statement, and the dotted line with a circle at the node represents a control dependency. Each edge represents a data dependence.

Problem Description. Given a set of PDGs $\{G(P_1), \ldots, G(P_n)\}$ corresponding to the set of programs $\{P_1, \ldots, P_n\}$, find all pairs of PDGs $\langle G(P_i), G(P_j)\rangle$ such that the program P_j is a plagiarism of the program P_i, denoted by $P_i \rightsquigarrow P_j$ $(1 \le i, j \le n, i \ne j)$.

2.3. Extreme Learning Machine (ELM). ELM [2, 3] has been widely applied in many classification applications [4]. It was originally developed for single hidden layer feedforward neural networks (SLFNs) and then extended to the "generalized" SLFNs where the hidden layer need not be neuron alike [5, 6]. ELM first randomly assigns the input weights and the hidden layer biases and then analytically determines the output weights of SLFNs. It can achieve better generalization performance than other conventional learning algorithms at an extremely fast learning speed. Besides, ELM is less sensitive to user-specified parameters and can be deployed faster and more conveniently [7–9].

For N arbitrary distinct samples (x_j, t_j), where $\vec{x}_j = [x_{j1}, x_{j2}, \ldots, x_{jn}]^T \in \mathbb{R}^n$ and $\vec{t}_j = [t_{j1}, t_{j2}, \ldots, t_{jm}]^T \in \mathbb{R}^m$, standard SLFNs with L hidden nodes and activation function $g(x)$ are mathematically modeled as

$$\sum_{i=1}^{L} \beta_i g_i\left(\vec{x}_j\right) = \sum_{i=1}^{L} \beta_i g\left(\vec{w}_i \cdot \vec{x}_j + b_i\right) = \vec{o}_j \quad (1 \le j \le N), \tag{1}$$

where L is the number of hidden layer nodes, $\vec{w}_i = [w_{i1}, w_{i2}, \ldots, w_{in}]^T$ is the weight vector between the ith hidden node and the input nodes, $\beta_i = [\beta_{i1}, \beta_{i2}, \ldots, \beta_{im}]^T$ is the weight vector between the ith hidden node and the output nodes, b_i is the threshold of the ith hidden node, and $\vec{o}_j = [o_{j1}, o_{j2}, \ldots, o_{jm}]^T$ is the jth output vector of the SLFNs [5].

The standard SLFNs with L hidden nodes and activation function $g(x)$ can approximate these N samples with zero error. It means $\sum_{j=1}^{L} \|\vec{o}_j - \vec{t}_j\| = 0$ and there exist β_i, $\vec{w}_i$, and b_i such that

$$\sum_{i=1}^{L} \beta_i g\left(\vec{w}_i \cdot \vec{x}_j + b_i\right) = \vec{t}_j \quad (1 \le j \le N). \tag{2}$$

Equation (2) can be expressed compactly as follows:

$$\vec{H}\beta = \vec{T}, \tag{3}$$

where $\vec{H}$ is the hidden layer output matrix of the neural network, $\vec{H}(\vec{w}_1, \vec{w}_2, \ldots, \vec{w}_L, b_1, b_2, \ldots, b_L, \vec{x}_1, \vec{x}_2, \ldots, \vec{x}_L) =$

$$\left[h_{ij}\right] = \begin{bmatrix} g(\vec{w}_1 \cdot \vec{x}_1 + b_1) & g(\vec{w}_2 \cdot \vec{x}_1 + b_2) & \cdots & g(\vec{w}_L \cdot \vec{x}_1 + b_L) \\ g(\vec{w}_1 \cdot \vec{x}_2 + b_1) & g(\vec{w}_2 \cdot \vec{x}_2 + b_2) & \cdots & g(\vec{w}_L \cdot \vec{x}_2 + b_L) \\ \cdots & \cdots & \cdots & \cdots \\ g(\vec{w}_1 \cdot \vec{x}_N + b_1) & g(\vec{w}_2 \cdot \vec{x}_N + b_2) & \cdots & g(\vec{w}_L \cdot \vec{x}_N + b_L) \end{bmatrix}_{N \times L}, \tag{4}$$

$\beta = [\beta_1^T, \ldots, \beta_L^T]_{m \times L}^T$, and $\vec{T} = [\vec{t}_1^T, \ldots, \vec{t}_L^T]_{m \times N}^T$. Algorithm 1 shows the pseudocode of ELM.

Input: Training set $\mathcal{N} = \{(\vec{x}_j, \vec{t}_j) \mid \vec{x}_j \in \mathbb{R}^n (1 \le j \le N)\}$;
Hidden node output function $g(\vec{w}_i, b_i, \vec{x}_j)$;
Number of hidden nodes L;
Output: Weight vector β;
(1) **for** $i = 1$ *to* L **do**
(2) Randomly generate hidden node parameters $(\vec{w}_i, b_i)$;
(3) Calculate the hidden layer output matrix $\vec{H}$;
(4) return the calculated output weight vector $\beta = \vec{H}^{\dagger}\vec{T}$, where $\vec{H}^{\dagger}$ is the Moore-Penrose generalized by the inverse of matrix H;

Algorithm 1: ELM training.

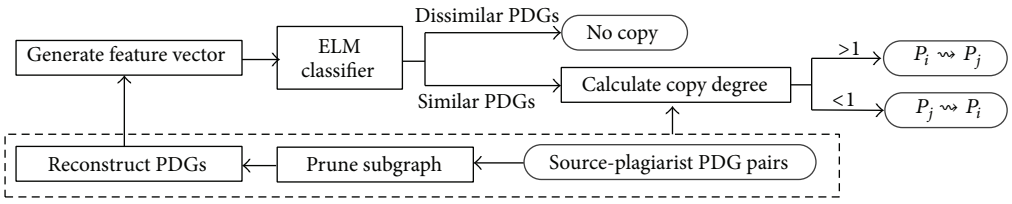


Figure 3: An ELM-based framework for determining copy direction.

3. An ELM-Based Framework for Determining Copy Direction

To our best knowledge, there is no detection algorithm that can directly compute plagiarism direction correctly. In this section, we propose an ELM-based framework for determining copy directions between any two PDGs.

Figure 3 shows the framework. Generally, a source PDG and its plagiarist PDG should be similar, so given a set of PDGs, we first use an ELM classifier to find all similar PDGs. Then for those similar PDGs, we check copy direction by calculating data dependencies scores as *copy degrees*. According to the calculated copy degree, we determine the copy direction between P_i and P_j.

In this section, we first show how to generate feature vectors for the ELM.

For a set of PDGs $\{G(P_1), \ldots, G(P_n)\}$, using ELM, the task is to learn a prediction rule from the training examples $\{D_j, y_j\}_{j=1}^n$, where D_j is the PDG in the training PDGs and $y_j \in \{+1, -1\}$ is the associated class label. Any similar PDGs have the same class label. Let $\mathcal{T}$ be the set of all patterns, that is, the set of all subgraphs included in at least one training PDG. Then each training PDG $G(P_i)$ is encoded as a $|\mathcal{T}|$-dimensional vector $\vec{x}_j = (x_{j,1}, \ldots, x_{j,|\mathcal{T}|})$,

$$x_{j,t} = I\left(t \subseteq G\left(P_i\right)\right), \tag{5}$$

where $I(\cdot)$ is 1 if the condition inside is true and 0 otherwise. For example, Figure 4(i) shows an example of this feature space. Figures 4(a) and 4(b) show an original code and its plagiaristic code, respectively. $G(P_1)$ and $G(P_2)$ are their corresponding PDGs shown in Figures 4(e) and 4(f). For simplicity, we only use a line number in each node to express its corresponding statement. A $|\mathcal{T}|$-dimensional vector $\vec{x}_j$ is shown in Figure 4(i), where every value corresponds to a subgraph pattern in the training set. We then use ELM to train these training examples $\{D_j, y_j\}_{j=1}^n$ since ELM is very efficient for large vector space.

Now the question is how to choose patterns to construct $|\mathcal{T}|$-dimensional vector as the training set. We choose *frequent subgraph* as patterns, which can be efficiently generated by the existing frequent graph mining algorithm, gIndex [10]. By setting a support threshold ρ, the algorithm in [10] chooses subgraphs whose supports are larger than or equal to ρ as frequent patterns. The support threshold is progressively increased when the subgraphs grow large. That is, we use low support for small subgraphs and high support for large subgraphs to avoid the exponential growth of $|\mathcal{T}|$ (our experimental results are reported in Section 6).

4. Achieving High Accuracy with Small Feature Space

Generally choosing large subgraphs as patterns achieves high accuracy of detecting copy relationship, since features of each PDG can be kept. However, it requires a small ρ, which results in a large number of patterns in $\mathcal{T}$ for constructing the feature space. Therefore, the detection efficiency is sacrificed as a consequence. Now the challenge is how to shrink feature space to make detection efficient and highly accurate.

We observe that for any two source and plagiarist programs $P_i \rightsquigarrow P_j$, P_j might contain useless control statements like loops, selective structures, which correspond to useless subgraphs for detection. If P_j contains a large number of such useless subgraphs, $G(P_i)$ is dissimilar with $G(P_j)$ and might not be detected by using ELM classifier.

In this section, we show we can still get a small feature space even when using a large support threshold, since removing such useless control statements only decreases the number of (useless) subgraphs. It is interesting to see that

```
(4) valLeft = getCrest(f1);
(5) valRight = getCrest(f2);
(6) revise = GlobalVal;
(7) tmp = c*(valLeft - revise) + valRight;
(8) findVal = temp/2;
```

(a) Original program P_1

```
(4) valLeft = getCrest(f1);
(5) valRight = getCrest(f2);
(6) revise = GlobalVal;
(7) tmp = c*(valLeft - revise) + valRight;
(8) mixVal = tmp;
(9) findVal = mixVal/2;
```

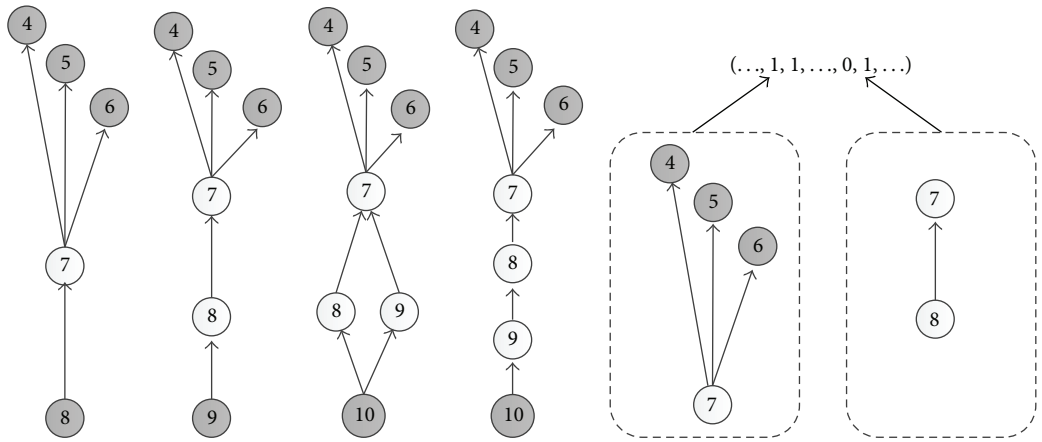
(b) Plagiaristic program P_2

```
(4)  valLeft = getCrest(f1);
(5)  valRight = getCrest(f2);
(6)  revise = GlobalVal;
(7) tmp = c*(valLeft - revise) + valRight;
(8)  mixValA = tmp - 100;
(9)  mixValB = tmp + 100;
(10) findVal = (mixValA + mixValB)/2;
```

(c) Plagiaristic program P_3

```
(4)  valLeft = getCrest(f1);
(5)  valRight = getCrest(f2);
(6)  revise = GlobalVal;
(7) tmp = c*(valLeft - revise) + valRight;
(8)  mixValA = tmp - 100;
(9)  mixValB = mixValA + 100;
(10) findVal = mixValB/2;
```

(d) Plagiaristic program P_4



(e) $G(P_1)$ (f) $G(P_2)$ (g) $G(P_3)$ (h) $G(P_4)$ (i) Feature space and feature vector

FIGURE 4: An example of changing the order of statements.

decreasing those useless subgraphs will increase the classification accuracy as well as accelerate the training efficiency, since such removal decreases not only noises for detection accuracy but also the number of patterns in $\mathcal{T}$.

4.1. Isolated Control Dependence Subgraphs

Definition 2 (isolated control dependence subgraph). Given a PDG G, a subgraph G' of G is an isolated control dependence subgraph, if the following two conditions hold.

(1) A variable v in a statement in G' is not used by its successive statements.

(2) For any subgraph G'' of G', v does not appear in G''.

For example, in Figure 2, the subgraph marked by dotted line is an isolated control dependence subgraph, whose corresponding statements are useless which could not affect the program's semantics but would change the PDG structure.

For the source-plagiarist PDG pair $\langle G(P_i), G(P_j)\rangle$, where $P_i \rightsquigarrow P_j$, in order to keep the correctness of execution semantics of P_i in P_j, copyists need to insert or disturb some nodes in $G(P_i)$ to generate $G(P_j)$ as follows.

Feature 1. A plagiarist PDG $G(P_j)$ and its source PDG $G(P_i)$ could be similar locally but dissimilar globally.

For a source PDG $G(P_i)$ and its copy $G(P_j)$, $G(P_i)$ and $G(P_j)$ could share a common subgraph, but globally dissimilar since a copyist may insert some useless control dependence edges in $G(P_j)$.

4.2. Pruning Isolated Control Dependence Subgraphs. Pruning isolated control dependence subgraphs might affect the execution of original program, because it can affect the function callings. As a consequence, such pruning might also remove subgraphs in an original PDG. Fortunately, according to our observation, this situation has no significant influence on detection result since if a control subgraph in the original program is removed, its corresponding control subgraph in the plagiaristic program would also be removed, and these two programs are still considered to be similar according to graph isomorphism algorithm.

Let λ be the percentage of total removed subgraph nodes in $V(G)$. Generally, we are not allowed to remove too many

```
    Input: PDG G, pruning threshold λ;
    Output: PDG after pruning isolated control dependence subgraphs;
(1) Generate control dependence subgraph set CG ← {g_1, ..., g_k} in G;
(2) foreach control dependence subgraph g in CG do
(3)     g.count ← 0;
(4)     foreach statement S in g do
(5)         if S has dependence relations with statements outside g then
(6)             g.count ← g.count + 1;
(7) Remove subgraphs in CG whose number of dependencies are greater than 0;
(8) maxNumDelNodes ← |V(G)| × λ;
(9) deletCount ← 0;
(10) G.removeNode ← 0;
(11) foreach subgraph g in CG do
(12)     if deletCount + |V(g)| ≤ maxNumDelNode then
(13)         delete g from G;
(14)         deletCount ← deletCount + |V(g)|;
(15) G.removeNode ← deleteCount;
(16) return G;
```

Algorithm 2: Pruning isolated control dependence subgraphs.

isolated control dependence subgraphs so that most nodes can be kept in a PDG. A small λ removes a small number of isolated control dependence subgraphs but leaves large number of useless control statements, which results in low detection recall. Whereas a larger λ could help to detect more plagiarisms but may remove some useless control statements in an original code by mistake. Removing such subgraphs makes detection fast but sacrifices precision (we report the effects of different λ values on real programs in Section 6).

Algorithm 2 describes the process of pruning isolated control dependence subgraphs. It first generates a set of control dependence subgraphs. For each subgraph g, it records the number of dependencies with the statements outside g (lines 2–6). It then only keeps isolated control dependence subgraphs in CG (line 7). The algorithm keeps removing remaining subgraphs in CG until the number of removed nodes is greater than a given threshold $|V(G)| \times \lambda$ (lines 8–14). It finally returns the new graph as well as the number of removed nodes.

4.3. Reconstructing Dependence Subgraphs. We can see from the three plagiarist programs P_2, P_3, and P_4 in Figures 4(b)–4(d) that the data dependencies are hidden by the confusing variables and statements inserted by copyists. To retain the execution of the source program, the inserted statements have to deliver such data dependencies. As their corresponding PDGs $G(P_2)$, $G(P_3)$, and $G(P_4)$ in Figures 4(f)–4(h) show, they all include a common path from the lowest node to the highest. Compared with the PDG $G(P_1)$ in Figure 4(e), these four PDGs are unlikely to be judged as isomorphic, and therefore it is difficult for previous detection tools or algorithms to detect such plagiarism method.

In order to deal with this antidetection method, we propose a control subgraph reconstruction algorithm, since reconstruction of control subgraph can eliminate the influences on PDG brought by most inserted variables and statements and thereby retain the PDGs of source program and plagiaristic program, largely increasing the algorithm's detection ability. As Figure 4 shows, to break original dependencies, copyists insert confusing variables and related operation statements, but on the other hand, they have to indirectly express those dependencies to make sure the execution results remain the same. Therefore the goal of reconstructing control subgraph is to restore those dependencies as much as possible or to express them with the same pattern. In this way, the PDGs of original program and plagiaristic program are the same. To achieve this goal, we have to firstly define some critical statements for expressing dependence. The basic idea of control subgraph reconstruction algorithm is uniformly expressing dependencies using these critical statements.

Since dependencies are to be transformed to dependencies of critical statements, how to select critical statements is thusly important. Critical statements commonly contain much semantics. Usually copyists conservatively add confusing variables and statements to ensure that the execution semantics are not changed. Obviously these added statements should not be critical statements, which means the program's execution semantics should not be expressed by these statements.

Definition 3 (critical statement). A program statement is a critical statement, if it satisfies one of the following conditions:

(i) containing subprogram calls,

(ii) reading or modifying nonlocal variables,

(iii) containing return statements.

For example, statements in lines 4, 5, 11, 13, and 19 in Figure 1 are critical statements, which are more difficult to imitate compared with other statements, and their mutual relations are difficult for copyists to conceal.

```
    Input: PDG G whose partial control subgraph has been deleted;
    Output: PDG after reconstruction of control subgraph;
(1) Generate G's control dependence subgraph set CG = {g_1, ..., g_k};
(2) foreach control dependence subgraph g in CG do
(3)     Mark irremovable nodes in g;
(4)     foreach irremovable node S in g do
(5)         Build an empty irremovable node Set;
(6)         Build a queue Q and put nodes that S is data-dependent on into Q;
(7)         while Q is not empty do
(8)             P = Q.dequeue();
(9)             if P is irremovable node then
(10)                Put P into Set;
(11)            else
(12)                Put nodes that P is data-dependent on into Q;
(13)        Add S into each node's data dependence in Set;
(14)    Delete all removable nodes and their dependencies in g;
(15) return G;
```

Algorithm 3: Reconstruction of control dependence subgraph.

Critical statement evaluates the importance of statement only from the point of program, while, analyzed from the point of PDG, another kind of statement is also very important, which is external-dependent statement. Every noncontrol statement belongs to some certain control dependence graph, and there exists such statement S that comes from statements in other PDGs or data dependencies in some control statements. Such statement S is called external-dependent statement, which is important because the values of variables it modifies are used by some loops or select statements. In this case, the execution path can only be determined when program is running, and therefore copyists have to retain this external dependence to ensure the correctness of program. Therefore we also use external-dependent statements as well as critical statements to express dependence in this paper.

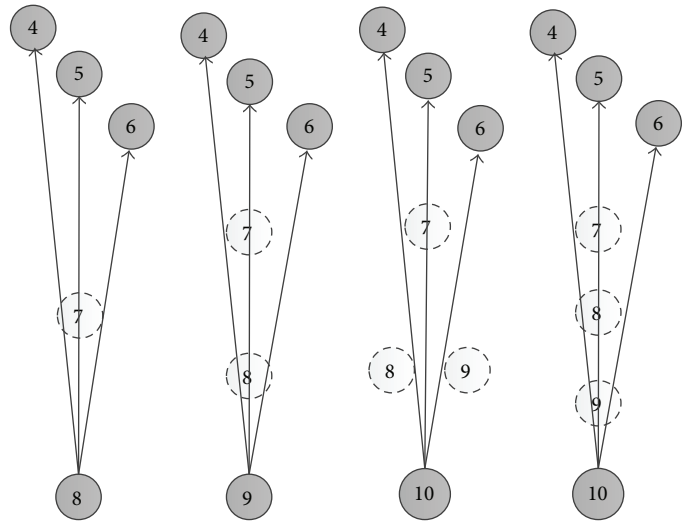
The goal of control subgraph reconstruction algorithm is reexpressing the dependencies within statements in control subgraphs, expressing them with uniform statements. Therefore this algorithm can eliminate most disguised dependencies. To elaborate this algorithm, irremovable node is defined in Definition 4.

Definition 4 (irremovable node). The nodes for all critical statements and external-dependent statements in PDG G are irremovable nodes, and the others are removable nodes.

The main processes of control subgraph reconstruction algorithm are dealing with every control subgraph, deleting all removable nodes in subgraphs and transforming the dependencies of irremovable nodes on removable nodes into dependence on irremovable nodes dependent on these removable nodes. Take the programs in Figures 4(a)–4(d) as examples, statement `findVal = temp/2` is an external-dependent statement (because control statement `inputVal == findVal` is data-dependent on this statement), and therefore it is an irremovable node. However, this statement is dependent on removable nodes in all these four programs. For example, node 7 in Figure 4(e) is a removable node. By using such nodes, copyists disguise the dependence in original program, and our algorithm will remove these nodes and meanwhile transform the dependence of `findVal = temp/2` on removable nodes into dependence on irremovable nodes that the removable nodes depend on. All the gray nodes in $G(P_1), G(P_2), G(P_3)$, and $G(P_4)$ are irremovable nodes and white nodes are removable nodes. Our algorithm will remove removable nodes and keep the dependencies among irremovable nodes. We show that the graphs for plagiaristic programs and original programs are the same after reconstruction. One principle for control subgraph reconstruction is retaining the PDG structure of original program as much as possible and eliminating the influences of copyists on them.

Algorithm 3 describes this algorithm. The loop from line 2 to line 14 deals with one control dependence subgraph each time. The algorithm marks the critical nodes and external-dependent statements in every control subgraph (line 3). The loop from line 4 to line 13 deals with all irremovable nodes in a control subgraph, while lines 7–12 are the breadth first search on subgraph g, where the search stops when encountering irremovable nodes (lines 9-10). If encountering removable nodes, the nodes they depend on are added to queue (line 12). This loop continues until the queue Q is empty which means statement S no longer depends on any removable nodes. It adds S to the irremovable nodes set (line 13). In line 14, all removable nodes and their corresponding dependence are deleted. The four PDGs $G'(P_1)$, $G'(P_2)$, $G'(P_3)$, and $G'(P_4)$ are the reconstructed subgraphs of $G(P_1)$, $G(P_2)$, $G(P_3)$, and $G(P_4)$, respectively.

After pruning isolated control dependence subgraphs from a PDG, and reconstructing PDGs, some unique subgraph patters could be removed from the PDG, therefore shrinking the number of dimensions in the feature space $\mathcal{T}$.



(a) $G'(P_1)$ (b) $G'(P_2)$ (c) $G'(P_3)$ (d) $G'(P_4)$

Figure 5: The PDGs after reconstruction.

```
void packageCmd(char * cmdStr, size_t blockNum, size_t blockSize)
{
(3)  struct Cmd * cmdStruct;
(4)  cmdStruct = (struct Cmd*) malloc (sizeof(struct Cmd));
(5)  cmdStruct->length = strlen(cmdStr);
(6)  cmdStruct->cmd = (char *)malloc(cmdStruct->length + 1);
(7)  strcopy(cmdStruct->cmd,cmdStr);
(8)  cmdStruct->cmdID = getCmdId(cmdStruct->cmd);
(9)  struct Buffer * buffer;
(10) buffer = (struct Buffer * ) malloc (sizeof(struct Buffer));
(11) size_t bufferSize;
(12) bufferSize = blockNum * blockSize;
(13) buffer->size = bufferSize;
(14) buffer->buf = (void*)malloc(buffer->size);
(15) initZero(buffer->buf, buffer->size);
(16) executeCmd(cmdStruct, buffer, blockNum, blockSize);
}
```

(a) Original program

```
void packageCmd(char * cmdStr, size_t blockNum, size_t blockSize)
{
(3)  struct Cmd * cmdStruct;
(4)  cmdStruct = (struct Cmd*) malloc (sizeof(struct Cmd));
(5)  struct Buffer * buffer;
(6)  size_t bufferSize;
(7)  cmdStruct->length = strlen(cmdStr);
(8)  buffer = (struct Buffer * ) malloc (sizeof(struct Buffer));
(9)  cmdStruct->cmd = (char*)malloc(cmdStruct->length + 1);
(10) bufferSize = blockNum * blockSize;
(11) strcopy(cmdStruct->cmd,cmdStr);
(12) buffer->size = bufferSize;
(13) cmdStruct->cmdID = getCmdId(cmdStruct->cmd);
(14) buffer->buf = (void*)malloc(buffer->size);
(15) initZero(buffer->buf, buffer->size);
(16) executeCmd(cmdStruct, buffer, blockNum, blockSize);
}
```

(b) Plagiaristic program

Figure 6: An example of changing the order of statements.

5. Determining Copy Direction

Besides inserting variables or statements in a program P_i, copyists would also modify the sequence of statements (on the premise that the execution semantics remain the same) to escape the detection of some algorithms based on statements or token sequence. In this section we propose a PDG reconstruction approach by considering the order of statements (Figure 5).

5.1. Effect of Data Dependencies. Generally, the styles and features of code have some fixed regulations. As for the example in Figure 6 two programs have exactly the same execution semantics. The only difference is the order of their statements. For example, the statements in lines 3–8 are related to the variable `cmdStruct` and they are consecutive. While in Figure 6(b), these statements are disperse, with other irrelevant statements mixed in. Obviously, relevant statements are always put together when programming, and therefore the code in Figure 6(a) conforms to programmers' logic better because consecutive statements have stronger relevance.

We summarize the following two features of statements in a source program.

Feature 2. In a source program P_i, variables in the same structure should be modified by successive statements.

For instance, statements in lines 4–6 in Figure 6(a) are all assignments of a structure variable `cmdStruct`, and therefore they are highly relevant. While in Figure 6(b), they are scattered in lines 4, 7, and 9, respectively. As a result, the program in Figure 6(b) is more likely to be a plagiaristic program.

Feature 3. In a source program P_i, on average, every two successive statements might be more data-dependent than they are in a plagiaristic program.

For example, in Figure 6(a), programmers are likely to consecutively write down statements related to `bufferSize` in lines 12–14, and these statements are expected to be in

consecutive order. However, they are dispersed in lines 10, 12, and 14 in Figure 6(b).

Given any two statements S_i and S_{i+1}, we define two scores $I_1(S_i+S_{i+1})$ and $I_2(S_i+S_{i+1})$ to measure their level of relevance. The higher the scores are, the more likely their serial numbers are consecutive:

$$I_1(S_i + S_{i+1}) = \begin{cases} 1 & \text{if } S_i \text{ and } S_{i+1} \text{ assign the same data structure,} \\ 0 & \text{otherwise.} \end{cases}$$

$$I_2(S_i + S_{i+1}) = \begin{cases} 1 & \text{if } S_{i+1} \text{ is data-dependent on } S_i, \\ 0 & \text{otherwise.} \end{cases} \quad (6)$$

Therefore, given a program, if most consecutive statements in a program P are not data-dependent on each other, we will give a relatively high score to this program P. We accumulate these two scores for every two statements and get the total score

$$I(P) = \frac{n-1}{\sum_{i=1}^{n-1} \left(I_1(S_i + S_{i+1}) + I_2(S_i + S_{i+1})\right)}, \quad (7)$$

where $S_i, S_{i+1} \in P$ and n is the number of statements in P.

5.2. Calculating Copy Degree by Combining Data Dependence and Control Dependence. We determine copy direction based on the following three factors by considering data dependencies and control dependencies in a PDG.

(i) Number of isolated control subgraphs in $G(P)$, denoted by $N_g(P)$: the larger it is, the more "bloated" the program is. This is probably caused by code fragment inserted by copyists to avoid being detected, and therefore a greater $N_g(P)$ indicates a greater possibility for a program to be plagiaristic.

(ii) Number of removable nodes in $G(P)$, denoted by $N_n(P)$: removable nodes represent noncritical statements and statements without any external dependence. The analysis in Section 4.3 shows that most statements inserted by copyists are removable nodes, and therefore the bigger N_n is, the more likely the program is plagiaristic.

(iii) Data dependence score $I(P)$; a greater $I(P)$ indicates a greater possibility for a program to be plagiaristic.

Let P_i and P_j be two programs, and let $G'(P_i)$ and $G'(P_j)$ be two similar reconstructed PDGs determined by the ELM. Equation (8) shows the copy degree of $P_i \rightsquigarrow P_j$. We say $P_i \rightsquigarrow P_j$ if score($P_i \rightsquigarrow P_j$) < 1, otherwise P_i copies P_j if score($P_i \rightsquigarrow P_j$) > 1:

$$\text{score}\left(P_i \rightsquigarrow P_j\right) = \frac{N_g(P_i) + 1}{N_g(P_j) + 1} \times \frac{N_n(P_i) + 1}{N_n(P_j) + 1} \times \frac{I(P_i)}{I(P_j)}. \quad (8)$$

6. Experiments

All the algorithms were implemented using GNU C++. The experiments were run on a PC with an Intel(R) Core(TM)2 Duo CPU T6600@2.20 GHz, 2 GB RAM, running a Ubuntu (Linux) 32-bit operating system. We adopted Frama-c as a source code analysis tool to generate PDGs.

We ran our experiments on C program sets collected from the Internet. To ensure the validity of algorithm, we copied part of the functions in these programs and adopted the following plagiarism methods [11] to disguise them as follows.

(i) Whenever m (usually 2 to 4) consecutive statements are not bounded by dependencies, reorder them.

(ii) Replace a `while` loop with an equivalent `for` loop and vice versa. Occasionally, a `for` loop is replaced by an infinite `while` loop with a break statement.

(iii) Scatter variables in the same structure and keep their dependencies.

(iv) Replace `if (a) A` with `if (! (!a)) A` and `if (a) A else B` with `if (!a) B else A`; recurse `if` nested if block is encountered. Finally, apply DcMorgan's rule if the boolean expression is complex.

(v) Run JPlag. For any places that they are not confused, insert a statement or a label.

(vi) Run test scripts, and ensure that correctness is preserved during plagiarism.

We then put plagiarist codes together with original program sets to get test data set, including 250 functions and nearly 4000 lines of program. We labeled source and plagiaristic functions for evaluating detecting algorithms.

We compare our algorithms with the following three state-of-the-art tools:

(i) GPLAG: a classic source code plagiarism detection software [11],

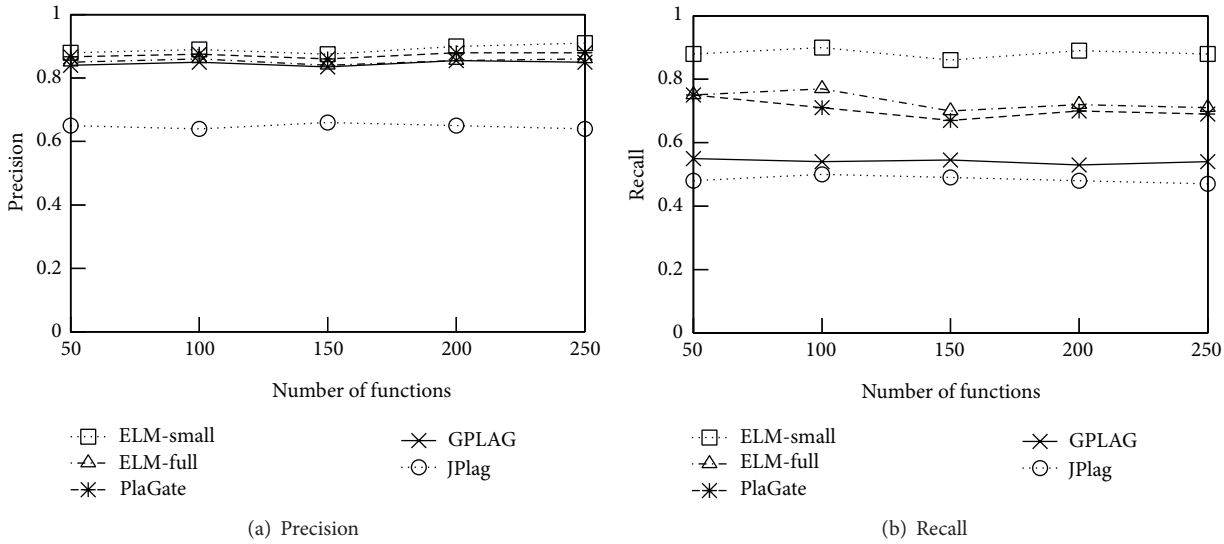
(ii) JPlag: a tool of finding plagiarisms among a set of programs [12],

(iii) PlaGate: a tool of integrating with existing plagiarism detection tools to improve plagiarism detection performance [13].

Notice that, to our best knowledge, there is no report on detecting copy directions among programs. So we compared our approaches with GPLAG, JPlag, and PlaGate to evaluate the precision and recall of finding plagiarist programs without detecting copy directions. We set support threshold $\rho = 3$ and the percentage of removed isolated control dependence subgraphs $\lambda = 20\%$.

6.1. Accuracy of Detecting Plagiarisms. In order to compare with other tools, we used precision and recall for copy relationship to evaluate accuracy of detecting plagiarisms.

Precision for copy relationship is the percentage of correctly detected source-plagiarism pairs in the list of pairs detected using detection algorithm. Precision is 1 when every pair detected is a source-plagiarism pair.



(a) Precision (b) Recall

Figure 7: Comparison of algorithms.

Recall for copy relationship is the ratio percentage source-plagiarism pairs that are detected to all labeled source-plagiarism pairs. Recall is 1 when all source-plagiarism pairs are detected.

Figure 7(a) shows precision of different detection approaches. ELM-full represents the algorithm using full feature space and ELM-small represents the algorithm using pruned feature space. Both ELM-based approaches had a higher accuracy than GPLAG. Since GPLAG is also based on PDG, the results show that PDG-based approaches can reflect programs' semantical features well. JPlag, which is based on sequence, was influenced by some small functions, leading to the decrease of accuracy. PlaGate, which is based on latent semantic analysis, was very close to our ELM-full.

As for recall, Figure 7(b) shows that ELM-small has a recall of nearly 90%, ELM-full has a recall of 77%, while GPLAG, which is also based on PDG, has a recall of only 55%. This is because many plagiarisms adopt methods to disguise codes, which greatly changed the structure of PDG, making GPLAG unable to detect such plagiarism. JPlag could only detect plagiarism without any modification, having an accuracy of around 50%, coinciding with the fact that half of the plagiarism was not disguised. PlaGate integrated existing plagiarism detection tools to improve plagiarism detection performance, which was higher than the other state-of-the-art tools, but less than ELM-small since the existing approaches did not consider removing and reconstruction PDGs.

6.2. Accuracy of Determining Copy Direction. We further evaluated accuracy of determining copy directions and defined *precision for copy directions* as the percentage of correctly detected plagiarism program in the list of all detected plagiarisms using detection algorithm. Precision is 1 when every function detected is a plagiarism.

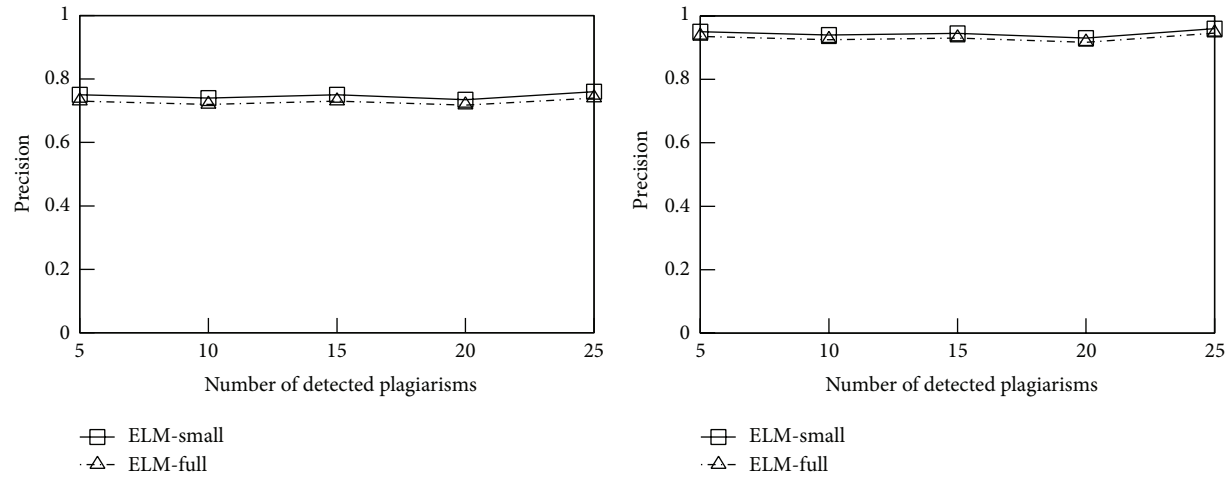
Figure 8 shows the accuracy of determining copy direction. Because no previous work can figure out plagiarism direction between programs, we only report detection results of using our ELM based approaches. Among 250 functions, 65 plagiarist functions were exactly the same with source functions.

Figure 8(a) shows the precision when the data set contains 65 unchanged plagiaristic programs, and Figure 8(b) shows the precision when the data set does not contain any unchanged plagiaristic functions. The precision shown in Figure 8(a) is only around 75%, since our approaches could not figure out the copy direction when plagiaristic programs were exactly the same with source programs. The precision in Figure 8(b) is around 95%, which shows that our approaches can distinguish the copy direction and further detect most plagiarisms.

The recall ratio was the same with the results shown in Figure 7(b) since among all labeled plagiarisms, the number of detected plagiarisms was the same with the number of detected copy-plagiarism pairs and the number of labeled plagiarisms was the same with the number of labeled source-plagiarism pairs.

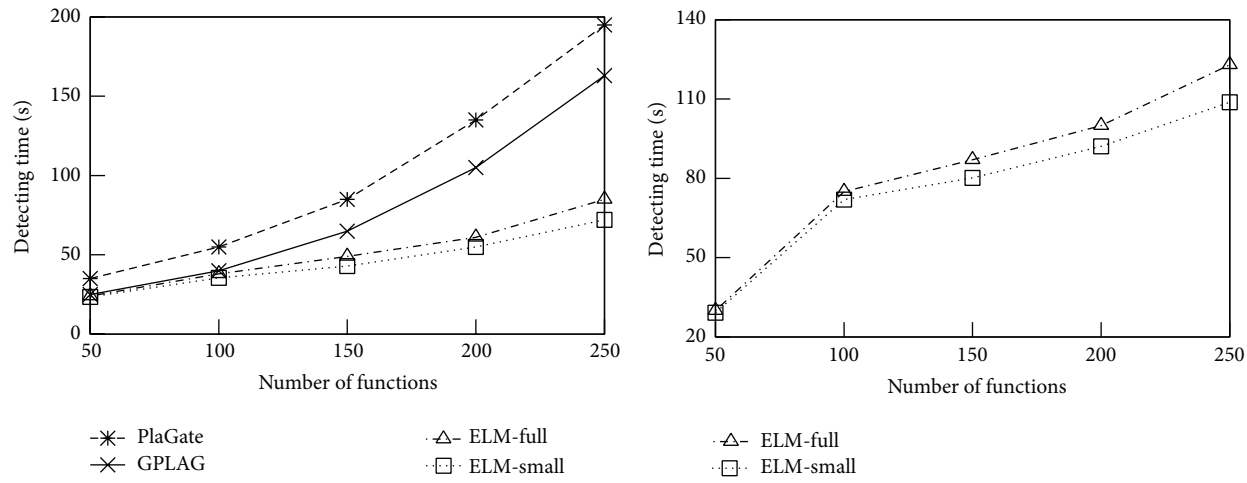
6.3. Detection Time. We compared time for detecting programs with plagiaristic relationship. Figure 9(a) shows the comparison results. The detecting time mainly depended on the time cost for static analysis tools and the efficiency of identifying programs with plagiaristic relationship. Our approaches and GPLAG costed the same time for static analysis, however, by using ELM, our ELM-based approach ran faster than GPLAG since GPLAG needed to find isomorphic subgraphs, which was time consuming, whereas we used ELM to classify PDGs, which was very efficient. PlaGate ran slowest since it required more detection time to integrate existing plagiarism detecting tools [13].

We also test the time for detecting copy directions. Since our work is the first one to detect copy directions, we only report the detecting time of our approaches using different feature spaces. Figure 9(b) shows the running time for



(a) With 65 unchanged plagiaristic functions (b) Without unchanged plagiaristic functions

Figure 8: Precision of determining copy direction.



(a) Detecting plagiarisms (b) Detecting copy directions

Figure 9: Detecting time.

detecting copy directions. Besides identifying programs with plagiaristic relationship, our approaches needed time to detect copy directions by considering data dependencies and control dependencies in PDGs.

Figure 10 shows number of generated feature spaces for different number of functions. We can see that pruning isolated control dependence subgraphs decreased relatively high number of features in each feature space.

6.4. Effect of Feature Space Size. Figure 12 shows the the effect of support threshold ρ on our algorithms. We used the whole data set with 250 functions, $\lambda = 20\%$, and varied ρ from 3 to 15.

Figure 11(a) shows that the size of feature space $\mathcal{T}$ shrinks when increasing ρ; particularly when ρ increases to 9, size $|\mathcal{T}|$ drops quickly. It is consistent with our analysis in Section 3. The detection time dropped correspondingly as shown in Figure 11(b). It is interesting to see that ELM-small improved the detection time when ρ was small but costed a lot when ρ increased to 12. This is because ELM-small required time to remove isolated control dependence subgraphs. When ρ was small, removing such subgraphs improved detection time a lot; however, when ρ was large, removing subgraphs from a small feature space would not bring benefit.

Figures 11(c) and 11(d) show that when increasing ρ, precision of the two algorithms drops slightly and recall drops significantly. Using a small ρ, our approaches chose large subgraphs in PDGs as patterns, which contain representative features in $\mathcal{T}$. So ELM-full provided relatively high precision and recall ratios. Based on the large feature space, ELM-small removed isolated control dependence subgraphs to further enhance the accuracy. When increasing ρ, the number of detected similar subgraphs in every PDGs pairs decreased, so recall dropped quickly. However, the precision in the detected

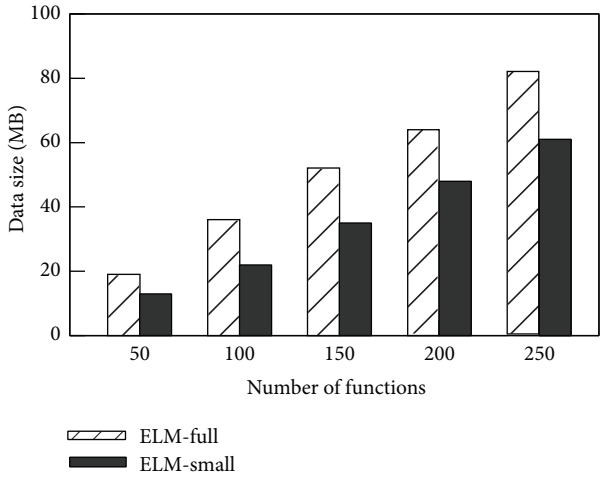


Figure 10: Feature space.

pairs did not change too much. These figures also show that the effect of removing isolated control dependence subgraphs disappears gradually when ρ becomes large since a smaller feature space corresponds to less isolated control dependence subgraphs.

6.5. Effect of Removed Isolated Control Dependence Subgraphs. Figure 12 shows the effect of parameter λ on ELM-small algorithm. We used the whole data set with 250 functions, $\rho = 3$, and varied λ from 5 to 30. Figure 12(a) shows that precision drops slightly but is not affected too much when increasing λ. Figure 12(a) shows that recall increases quickly when increasing λ from 5% to 20% and then increases slowly, which indicates that removing partial isolated control dependence subgraphs will help to detect more source-plagiarism pairs. And it is not surprising to see removing more isolated control dependence subgraphs makes detection faster as shown in Figure 12(c).

7. Related Work

The detection of source code plagiarism basically contains three steps: *process*, *transform*, and *match* [11, 14, 15]. The operations in *process* are dealing with original source code, extracting information that would be used in subsequent operations and removing unnecessary information for judging similarity. For example, most algorithms would remove blank characters and comments in *process*. The second step is *transform*. In *transform*, preprocessed programs will be transformed into certain intermediate form which varies according to algorithm. Programs are represented by intermediate form because it only contains information needed for detecting similarity and is beneficial to subsequent calculations in *match*, which only needs intermediate form to determine whether code fragments are similar. The intermediate form of an algorithm is important, because it determines the basis for judging similarity between source codes, and the effectiveness of an algorithm depends on whether its intermediate form can completely and correctly represent the similarity relation between original codes. Meanwhile intermediate form also determines the algorithm's efficiency. For instance, an algorithm adopting a tree-based intermediate form generally runs faster than a graph-based one but has lower correctness. The next step is *match*, which is conducting similar pair operations on intermediate form of the source program. This is the key step in the entire algorithm.

Based on different intermediate form, the existing work can generally be categorized into five groups.

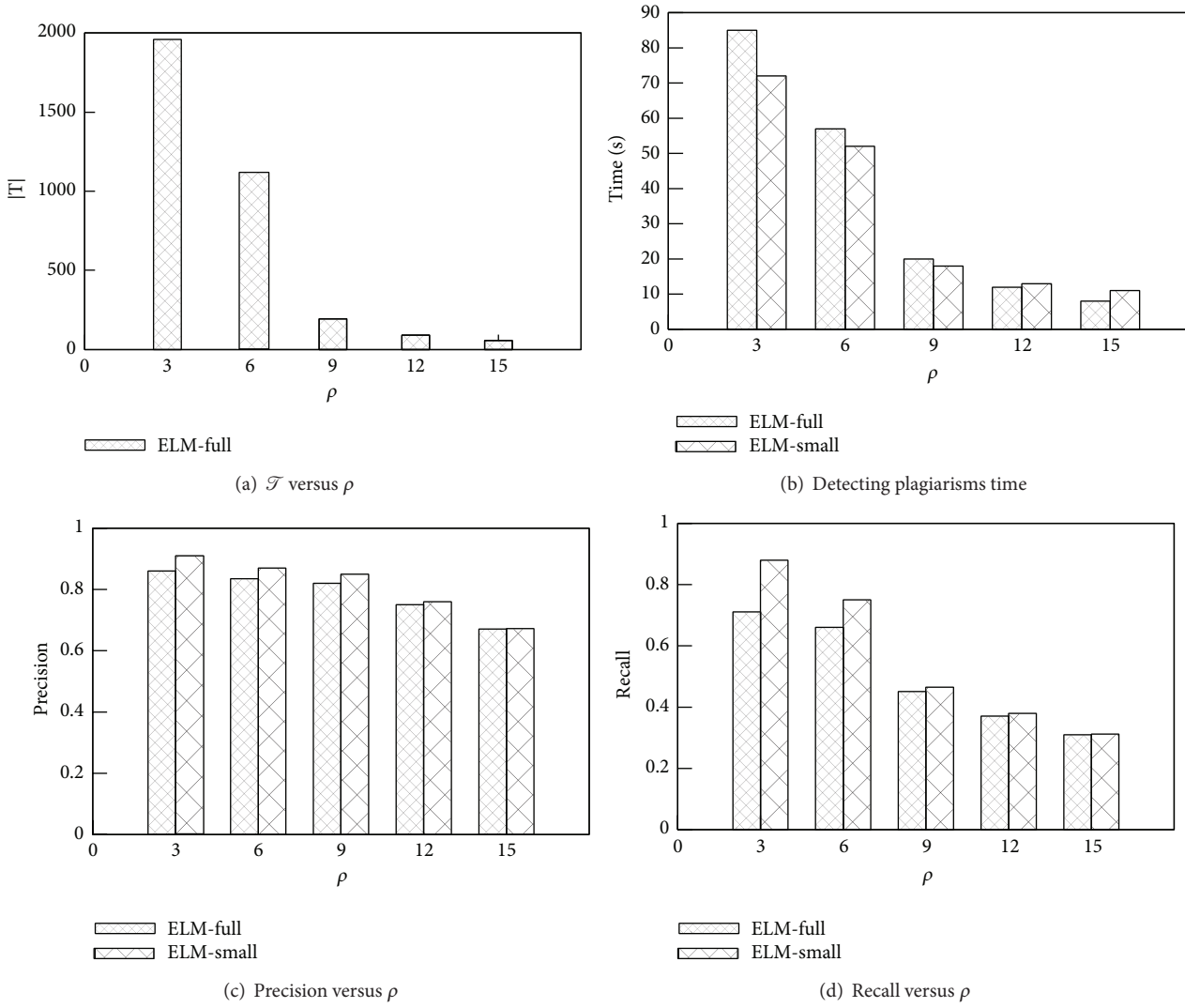
(i) Text-Based Method. Since source code is a text set, text-based methods directly use source code as intermediate form to compare without utilizing the grammar and syntax features of program. Johnson first proposed a text-based method. In [16, 17], he firstly separated the source code into several equi-length sub-programs, and then conducted hash calculations on separated programs. Each hash value is called "fingerprint." In *match*, a slide window was used to recognize program fragments with the same hash value. To recognize fragments of different lengths, the slide window algorithm needs running several times to find similar programs of different lengths.

Ducasse et al. in [18] proposed another text-based method, based on an idea of using a two-dimension axis, where each axis represents a set of source codes, and each point on the axis is a row of source code. The value of the point is the hash value of this source code. If a point A on X axis has the same value with a point B on Y axis, then the point (A, B) will be marked. Therefore the problem of detecting similar code fragments can be transformed into finding diagonal connected points. Ducasse used a pattern matching algorithm to find diagonal connected points, which allows a minor number of point misses.

(ii) Word-Based Method. Word-based methods lexically analyze source code at first and transform source code into a sequence consisting of token, on which the subsequent steps will be conducted. Obviously, word-based methods can better deal with modifications on code fragments like variable rename, pattern adjustment, and so on. Compared with text-based methods, they are more robust.

There are many word-based methods, one of which is CCFinder proposed by Kamiya et al. in [15]. It firstly formalizes the source program, which is aimed at removing incorrect influences of matching results caused by some languages' grammar. Take scope operator in C++ language as an example; it has no meaning in judging similarity, and as a result it can be removed during formalization. Moving on, CCFinder would conduct lexical analysis to programs to transform the program into a parameterized token sequence. Its matching operation is using this sequence to build a suffix tree, which is used for figuring out identical subsequences as similar program fragments.

CP-Miner is another representative algorithm, which divides program and then calculate every row's hash value. Afterward programs are transformed to a sequence of hash value. Before matching, CP-Miner firstly divides hash value sequence into several subsequences according to program's scope. Thus programs are expressed as a set of subsequences.



(a) $\mathcal{T}$ versus ρ

(b) Detecting plagiarisms time

(c) Precision versus ρ

(d) Recall versus ρ

Figure 11: Effect of parameter ρ.

CP-Miner adopts CloSpan [19], a frequent subsequence algorithm, to figure out subsequences which appear frequently in programs' subsequence set. These subsequences are considered as similar programs.

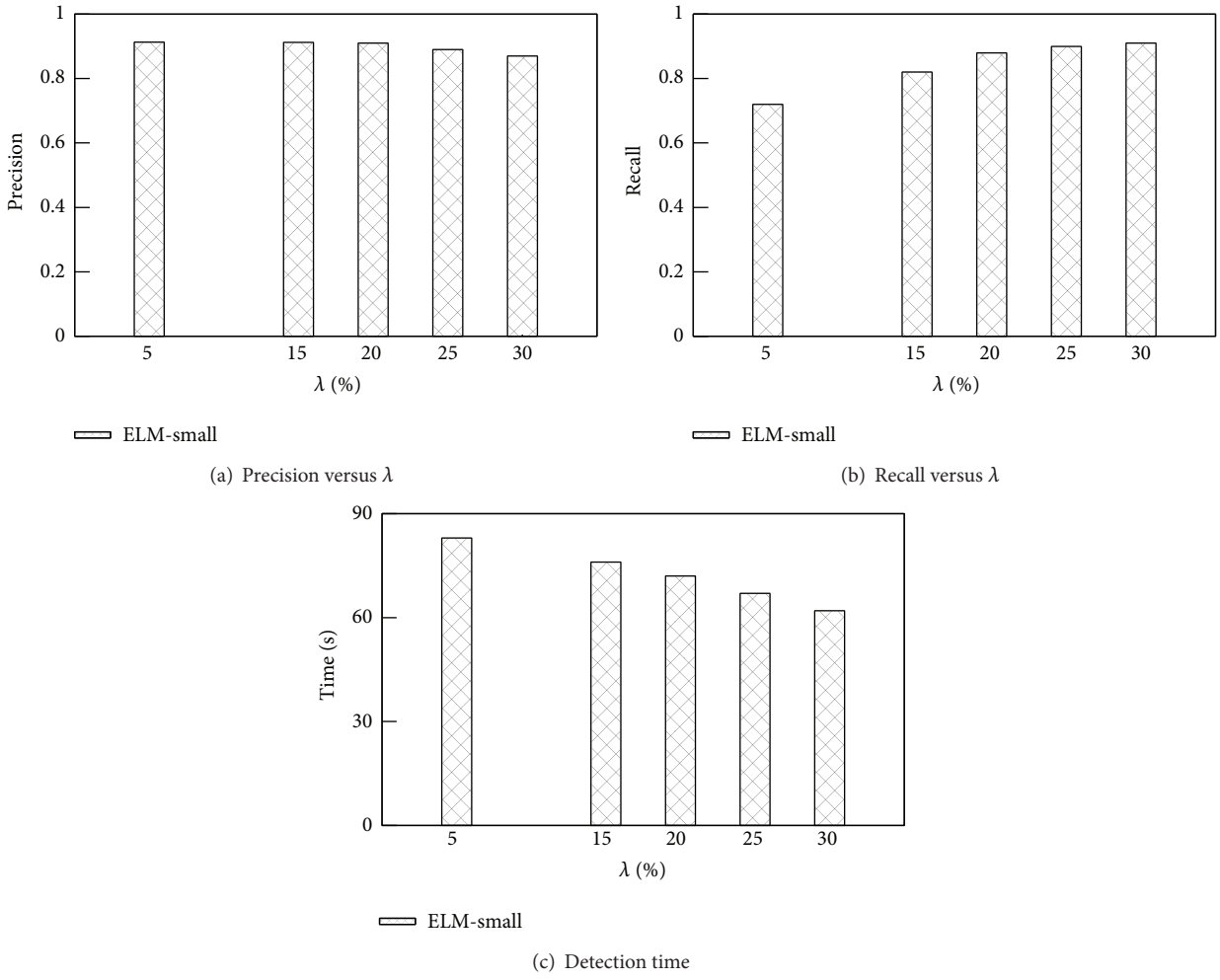
(iii) Tree-Based Method. Since program is a kind of text with precise meaning, detecting similar programs can be more accurately completed by further analyzing the programs. To begin with, tree-based methods conduct lexical and grammatical analysis on the program, transform it into an abstract syntax tree, and then use tree matching or other methods to conduct similarity detection.

Baxter et al. in [14] proposed a tree-based method CloneDr, which uses a compiler generator to generate a commented syntax tree generator. Then a tree matching algorithm is used to do one on one matching. To reduce running time, an optional procedure is to conduct hash operations on generated subtrees and then compare subtrees with the same hash value pairwise. By doing this, running time can be largely reduced.

The time complexity of subtree comparison is large, and to avoid subtree matching, Koschke et al. in [20, 21] built an abstract syntax tree for the program first of all and then sorted them, transforming the tree into a sequence of tree nodes to build a suffix tree for similarity matching.

Jiang et al. in [22] proposed a novel algorithm Deckard. Similarly, Deckard firstly computes abstract syntax tree in programs and calculates subtrees' eigenvector which is used to approximately represent trees. After that, LSH technology is used to cluster eigenvectors to find similar program fragments.

(iv) Metric-Based Method. The basic idea of metric-based methods is to extract some features or indicators from program fragments and thereby use them to compare instead of directly comparing code or abstract syntax trees. A prevalent method is to extract some parameters as fingerprints or features from every program's grammar unit, such as class, function, method, and statement. To begin with, most algorithms would grammatically analyze the programs to



(a) Precision versus λ

(b) Recall versus λ

(c) Detection time

FIGURE 12: Effect of λ.

compute its abstract syntax tree or PDG, based on which eigenvalues are computed. Huang et al. in [9] adopted some features to represent functions, and functions with the same eigenvalue are recognized as similar programs. The calculations of these eigenvalues come from functions' names, distributions, expressions, and their simple control flows.

(v) Semantic-Based Method. Semantic-based methods are used to find more veiled similarities among programs. The source code of plagiaristic programs may be quite different but must be semantically similar with copied programs. Therefore semantic-based methods conduct static analysis instead of simple grammatical analysis to provide more precise information. Many semantic-based methods adopt PDG to represent source programs. PDG is a graph structure that represents the inner data and control dependence of the program, where the nodes are used to represent programs' statements, such as statements and expressions. By using such representation, consecutive statements that have no semantic relation can be independent. Therefore the problem of finding similar programs is transformed into the problem of finding isomorphic subgraphs. Komondoor and Horwitz in [23] proposed a PDG-based method, using reverse program fragments to find isomorphic subgraphs. Liu et al. in [11] developed a PDG-based tool GPLAG to detect plagiarism.

There are other methods using semantic analysis to detect similar program fragments except PDG-based ones. Kim et al. in [24] proposed a method based on static memory comparison. Its basic idea is to use static semantic analyzer to build an abstract memory status for every subprogram and compute the similarity between two programs by comparing the similarity of memory status. This method is completely based on semantics that even if two programs are literally different, their abstract memory can be similar. And similar memory status reflects programs' similar functions.

8. Conclusion and Future Work

Aimed at the disadvantages of current source code plagiarism detection tools and algorithms, we propose a new ELM-based detection approach in this paper. We summarize common methods of plagiarism, deeply analyze the effects of these methods on PDGs, and propose corresponding detection strategies. Our detection approach can detect plagiarism

between source codes that most classic detection algorithms cannot detect, and its correctness has been proved.

Our future work will focus on how to analyze program's semantics and thereby improve detection ability of algorithm.

Conflict of Interests

The authors declare that there is no conflict of interests regarding the publication of this paper.

Acknowledgments

The work is partially supported by the National Natural Science Foundation of China (no. 61272178), the National Basic Research Program of China (973 Program) (no. 2012CB316201), the National Natural Science Foundation of China (nos. 61322208, 61129002), the Doctoral Fund of Ministry of Education of China (no. 20110042110028), and the Fundamental Research Funds for the Central Universities (no. N110804002).

References

[1] J. Ferrante, K. J. Ottenstein, and J. D. Warren, "The program dependence graph and its use in optimization," *ACM Transactions on Programming Languages and Systems*, vol. 9, no. 3, pp. 319–349, 1987.

[2] G. B. Huang, Q. Y. Zhu, and C. K. Siew, "Extreme learning machine: theory and applications," *Neurocomputing*, vol. 70, no. 1–3, pp. 489–501, 2006.

[3] G.-B. Huang, C.-K. Siew, and L. Chen, "Universal approximation using incremental constructive feedforward networks with random hidden nodes," *IEEE Transactions on Neural Networks*, vol. 17, no. 4, pp. 879–892, 2006.

[4] J. W. Cao, T. Chen, and J. Fan, "Fast online learning algorithm for landmark recognition based on bow framework," in *Proceedings of the 9th IEEE Conference on Industrial Electronics and Applications*, pp. 1163–1168, June 2014.

[5] G.-B. Huang and L. Chen, "Convex incremental extreme learning machine," *Neurocomputing*, vol. 70, no. 16–18, pp. 3056–3062, 2007.

[6] G.-B. Huang and L. Chen, "Enhanced random search based incremental extreme learning machine," *Neurocomputing*, vol. 71, no. 16-18, pp. 3460–3468, 2008.

[7] J. Cao, Z. Lin, G.-B. Huang, and N. Liu, "Voting based extreme learning machine," *Information Sciences*, vol. 185, pp. 66–77, 2012.

[8] G.-B. Huang, X. Ding, and H. Zhou, "Optimization method based extreme learning machine for classification," *Neurocomputing*, vol. 74, no. 1–3, pp. 155–163, 2010.

[9] G.-B. Huang, H. Zhou, X. Ding, and R. Zhang, "Extreme learning machine for regression and multiclass classification," *IEEE Transactions on Systems, Man, and Cybernetics Part B: Cybernetics*, vol. 42, no. 2, pp. 513–529, 2012.

[10] X. Yan and J. Han, "CloseGraph: mining closed frequent graph patterns," in *Proceedings of the 9th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pp. 286–295, Washington, DC, USA, August 2003.

[11] C. Liu, C. Chen, J. Han, and P. S. Yu, "GPLAG: detection of software plagiarism by program dependence graph analysis," in *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '06)*, pp. 872–881, August 2006.

[12] 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.

[13] G. Cosma and M. Joy, "An approach to source-code plagiarism detection and investigation using latent semantic analysis," *IEEE Transactions on Computers*, vol. 61, no. 3, pp. 379–394, 2012.

[14] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier, "Clone detection using abstract syntax trees," in *Proceedings of the IEEE International Conference on Software Maintenance (ICSM '98)*, pp. 368–377, Bethesda, Md, USA, November 1998.

[15] 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.

[16] J. Johnson, "Identifying redundancy in source code using fingerprints," in *Proceedings of the 1993 Conference of the Centre for Advanced Studies on Collaborative Research (CASCON '93)*, pp. 171–183, 1993.

[17] J. Johnson, "Visualizing textual redundancy in legacy source," in *Proceedings of the Conference of the Centre for Advanced Studies on Collaborative Research (CASCON '94)*, p. 32, 1994.

[18] S. Ducasse, M. Rieger, and S. Demeyer, "A language independent approach for detecting duplicated code," in *Proceedings of the IEEE International Conference on Software Maintenance (ICSM '99)*, pp. 109–118, September 1999.

[19] X. Yan, J. Han, and R. Afshar, "Clospan, Mining closed sequential patterns in large databases," in *Proceedings of the 3rd SIAM International Conference on Data Mining*, pp. 166–177, San Francisco, Calif, USA, May 2003.

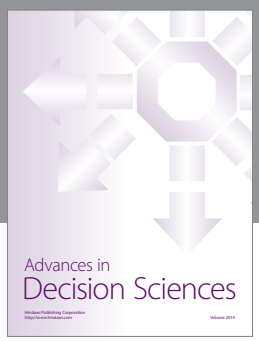
[20] R. Falke, P. Frenzel, and R. Koschke, "Empirical evaluation of clone detection using syntax suffix trees," *Empirical Software Engineering*, vol. 13, no. 6, pp. 601–643, 2008.

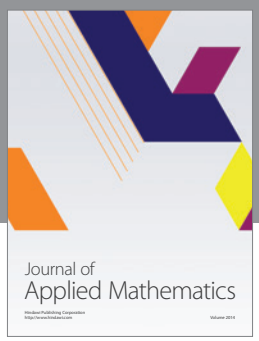
[21] R. Koschke, R. Falke, and P. Frenzel, "Clone detection using abstract syntax suffix trees," in *Proceedings of the 13th Working Conference on Reverse Engineering (WCRE '06)*, pp. 253–262, October 2006.

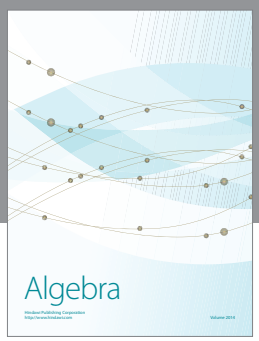
[22] L. Jiang, G. Misherghi, Z. Su, and S. Glondu, "DECKARD: scalable and accurate tree-based detection of code clones," in *Proceedings of the 29th International Conference on Software Engineering (ICSE '07)*, pp. 96–105, Minneapolis, Minn, USA, May 2007.

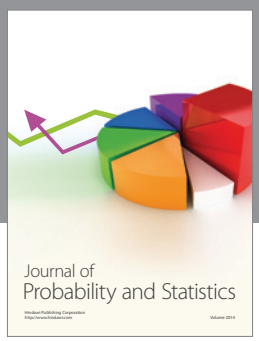
[23] R. Komondoor and S. Horwitz, "Using slicing to identify duplication in source code," in *Proceedings of the 8th International Symposium on Static Analysis (SAS '01)*, pp. 40–56, 2001.

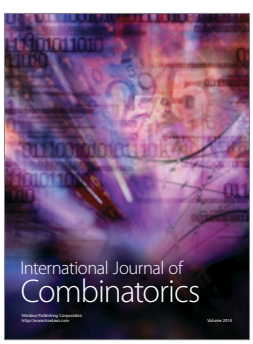
[24] H. Kim, Y. Jung, S. Kim, and K. Yi, "MeCC: memory comparison-based clone detector," in *Proceedings of the 33rd International Conference on Software Engineering (ICSE '11)*, pp. 301–310, May 2011.

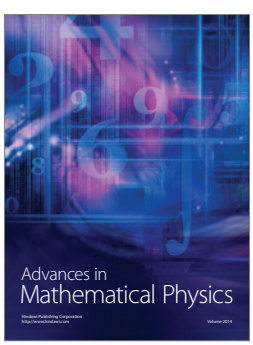


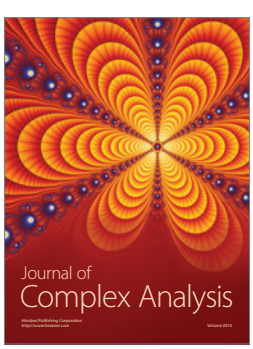


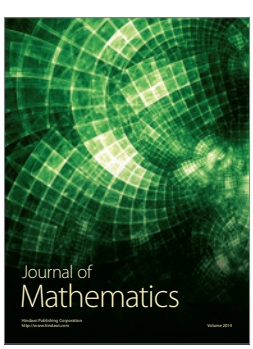


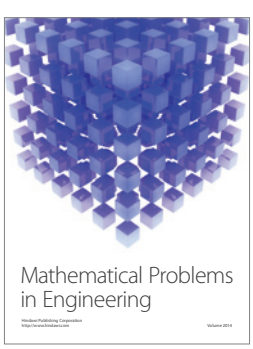


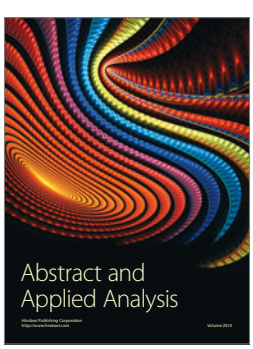


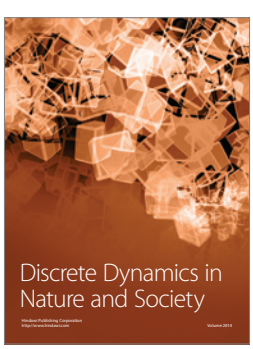


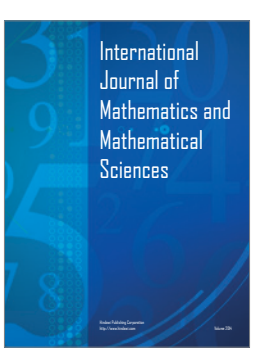












Hindawi

Submit your manuscripts at
http://www.hindawi.com