Front. Comput. Sci., 2014, 8(6): 996–1011
DOI 10.1007/s11704-014-3430-1

RESEARCH ARTICLE

Detection of semantically similar code

Tiantian WANG (✉)[1], Kechao WANG[1,2], Xiaohong SU[1], Peijun MA[1]

1 School of Computer Science and Technology, Harbin Institute of Technology, Harbin 150001, China
2 School of Software, Harbin University, Harbin 150086, China

Abstract The traditional similar code detection approaches are limited in detecting semantically similar codes, impeding their applications in practice. In this paper, we have improved the traditional metrics-based approach as well as the graph-based approach and presented a metrics-based and graph-based combined approach. First, source codes are represented as augmented system dependence graphs. Then, metrics-based candidate similar code extraction is performed to filter out most of the dissimilar code pairs so as to lower the computational complexity. After that, code normalization is performed on the candidate similar codes to remove code variations so as to detect similar code at the semantic level. Finally, program matching is performed on the normalized control dependence trees to output semantically similar codes. Experiment results show that our approach can detect similar codes with code variations, and it can be applied to large software.

Keywords similar code detection, system dependence graph, code normalization, semantically equivalent

1 Introduction

Similar code (code clone/duplicated code) detection is an important task in software development and maintenance. It can be applied to many fields such as software evolution [1–5], refactoring [6], and bug detection [7–9]. Therefore, many similar code detection approaches have been developed [10]. Based on the level of analysis applied to the source code, similar code detection can be mainly divided into five categories: text-based [11–14], token-based [15–20], tree-based [21–25], metrics-based [26–30], and graph-based [31–35] approach.

The first three categories can only deal with identical duplicates or duplicates with a few modifications (such as identifier renaming and comments modification). However, often, duplicated codes are not textually similar. Because of the various syntax of a programming language, there are various ways that a given programming task can be implemented. This phenomenon is called code variation. Code variations widely exist in real world software systems. For example, a programmer may not realize that a function has already been implemented before so as to reimplement it. Semantic preserving transformations, such as expression moving, variable renaming, function extraction, and function inling, are often performed in software restruction. Plagiarized codes are often modified to disguise plagiarism. A good similar code detection approach will be robust enough in identifying such hidden semantically similar relationships.

The metrics-based approach is easy and fast, and it can detect near-miss similar codes. However, it usually has a low precision. Two code fragments with same metrics may not be same.

Nowadays, the graph-based approach is widely studied [31–35]. This approach represents source codes as program dependence graphs (PDG), and detects similar code based on subgraph isomorphism testing. For example, Liu et al. developed a plagiarism detection tool called GPlag, which detects plagiarism by mining PDGs [33]. GPlag takes as input an original program and a plagiarism suspect, and outputs a set of PDG pairs that are regarded as involving plagiarism. Firstly, PDGs of the two programs are collected. Then PDGs smaller than the given threshold are excluded by a lossless filter. Finally, GPlag searches for plagiarism PDG pairs.

Received October 28, 2013; accepted June 24, 2014

E-mail: sweetwtt@126.com

For each PDG that belongs to the original program, GPlag performs the isomorphism testing and obtains all PDGs that survive both the lossless and the lossy filters. Qu et al. [34] proposed a software reuse detection framework using an integrated space-logic domain model. The first step is joint space-logic analysis. Source codes are firstly represented as PDGs. Instead of directly searching the identical subgraphs between two PDGs, the model reorder the graphs according to the source code location. Finally, reordered PDGs are hashed line by line. The second step is space-domain matching between hash lists. Similar line pairs are detected as "seed matches" with the Winnowing algorithm. The third step is logic-domain matching. For each "seed" line pair, the model retrieves the associated PDGs. Then graph matching algorithm is performed on each PDG pair to further detect if there is any isomorphic subgraph inside the PDG pair.

The graph-based approach considers not only the syntactic structure of programs but also the data flow (as abstraction of the semantics), so it lays a good foundation for similar code detection at the semantic level. However, two key problems as follows have not been excellently solved.

1) Code variations are not well handled. Komondoor's [31] and Krinke's [32] approaches can deal with format alteration, identifier renaming, statement reording, insertion and deletion. Liu's [33] and Qu's [34] approaches can also recognize simple control replacement. However, semantically equivalent programs can be varied in many aspects. In particular, programs that implement the same task may be written in various module structures. Both the number of sub-functions and the sites of invocation statements may be different. This phenomenon is called function call variations. As the above graph-based approaches do not consider the relationship between functions, they cannot recognize such variation.

2) Most of the existing graph-based approaches have a high computational complexity. Due to the complexity of data flow analysis, the PDGs can only be constructed to a limited size. Besides, the problem of subgraph isomorphism testing is NP-hard. For these reasons, Komondoor's [31] and Krinke's [32] approaches cannot be applied to large software. In order to make their approaches scalable to large programs, Liu et al. [33] proposed a lossless filter and a lossy filter to prune the plagiarism search space, Qu et al. [34] introduced a space-domain matching approach, and Gabel et al. [35] proposed to reduce the difficult graph similarity problem to a simpler tree similarity problem. However, the data flow of the PDGs still needs to be analyzed for the whole program in these approaches.

In order to solve the above problems, we improved the traditional metrics-based approach as well as the graph-based approach, and integrated these two improved approaches to form the combined metrics-based and graph-based approach (CMGA). In CMGA, source codes are represented as augmented system dependence graphs (SDG). In order to detect similar codes at the semantic level, code normalization is performed on the SDGs to eliminate code variations. In order to make this approach scalable to large programs, a metrics-based candidate similar code extraction approach is proposed to prune the search space, and the similarity computation is based on tree matching, which has much lower computational complexity than that of graph matching.

2 Background

2.1 Dependence graph

Definition (program dependence graph) A program dependence graph [36] $G = (V, E)$ is a directed graph for a single procedure of a program. V is the set of nodes representing statements and predicates. E is the set of edges representing data or control dependence relationship between nodes.

Types of nodes in PDG include entry, declaration, assignment, if/if-else, switch, do-while, and so on. Types of edges in PDG include control dependence and flow dependence. Suppose v_1 and v_2 are two nodes, if the execution of v_2 is dependent on the predicate of v_1, v_2 is control dependent on v_1. If v_2 refers a variable defined by v_1, v_2 is data dependent on v_1.

Definition (system dependence graph) A system dependence graph [37,38] G={*GPDGs*, *EInterpro*} is a directed graph for a program with multiple procedures. *GPDGs* is the set of PDGs, one PDG per procedure. *EInterpro* is the set of interprocedural edges. Interprocedural control dependence edges connect procedure call sites to the entry point of called procedures. Interprodedural data dependence edges represent the data flow between actual parameters and formal parameters.

A SDG can be divided into a control dependence subgraph and a data dependence subgraph.

Definition (control dependence sub-graph) The control dependence sub-graph (CDS) represents the control flow of the program. It consists of nodes and control dependence edges.

Definition (data dependence sub-graph) The data dependence sub-graph (DDS) represents the data flow of the program. It consists of nodes and data dependence edges.

2.2 Code variations

Common code variations can be mainly divided into the following types.

1) Code formats variation: semantically equivalent codes may be different in code format. For example, more or fewer comments, blank lines, etc. are used. These variations can be removed when source codes are parsed, by ignoring the comments and blank lines.

2) Compound statement variation: a certain program element can be either written in the form of a statement sequence or in the form of a single statement.

3) Expression variation: semantically equivalent expressions can be in various forms, because of differences in laws or properties (e.g., priority and associative property) among operators.

4) Redundancies: programs may contain some useless codes or use different number of temporary variables.

5) Control structure variation: a program usually has three kinds of control structure (i.e., sequence, selection and iteration). Sometimes it also has control transfer statements. Each control structure can be written with more than one kinds of syntax.

6) Variable name variation: variable names may be quite different between two semantically equivalent code fragments.

7) Statements order variation: statements in two semantically equivalent code fragments may be placed in different orders.

8) Function call variation: programs that implement the same task may be written in various module structures. Both the number of sub-functions and the sites of invocation statements may be different. Even if two sub-functions do the same thing, they can be different in parameter names and orders.

Our approach eliminates these code variations so as to recognize semantically similar codes.

2.3 Table of notations

To ease the reading of this paper, the frequently used abbreviations are summarized in Table 1.

3 An overview

3.1 Problem formulation

The process of similar code detection can be divided into two steps: first, source codes are transformed to an intermediate representation. Then comparison units represented by the intermediate format are extracted according to a certain granularity, and a similarity measure is applied to the comparison units. Our approach aims at detecting similar code at the semantic level, so the intermediate representation should be able to sufficiently represent the semantics of programs. SDG is a semantic representation for a program. It incorporates control dependence, data dependence and function call information into a single structure explicitly, which makes it a good foundation for semantic-preserving transformation and program analysis. As a result, SDG is adopted as the intermediate representation in our approach. In practice, a programming task is usually implemented by a module composed by a set of functions with invocation relationship between them, so that comparison units are extracted at a module granularity in our approach.

Table 1 Notations

Symbol	Explanation	Section
CMGA	Metrics-based and graph-based combined approach	1
PDG	Program dependence graph	2.1
SDG	System dependence graph	2.1
CDS	Control dependence sub-graph	2.1
DDS	Data dependence sub-graph	2.1
CDT	Control dependence tree	3.3

The similar code detection problem is formulated as follows: given an original program P and a target program P', suppose G and G' are the SDG of P and P' respectively, and each has n and m PDGs. Then the problem of detecting similar codes in P and P' consists of the following three sub-problems.

Problem 1 Due to the complexity of data flow analysis, the SDGs can only be constructed to a limited size. How to lower the complexity of SDG representation so as to make it scalable to large programs?

Problem 2 G and G' may contain many PDGs, so the computational complexity may be quite high if we adopt an exhaustive approach of comparing every PDG set. In this case, how to detect similar PDG sets both accurately and efficiently?

Problem 3 Given PDG sets $SG \subset G$, $SG' \subset G'$, how to judge SG and SG' are semantically similar, that is how to recognize similar SG and SG' which contain code variations?

3.2 Theoretical foundation

The theoretical foundation of CMGA is based on the follow-

ing facts.

Fact 1 (metrical similar codes) If two code fragments P and P' are similar under the set of features measured by M, then the values of $M(P)$ and $M(P')$ should be proximate [27].

Fact 2 (candidate similar codes) Based on Fact 1, P and P' are candidate similar codes if the metric similarity between them is no less than the predefined threshold of metric similarity T_m.

Fact 3 (SDG) SDG is a semantic representation for a program. If two programs have the same SDG, they are semantically equivalent. However, two semantically equivalent programs may have different SDGs [37,39].

Fact 4 (semantic-preserving transformation) A program transformation is semantic-preserving if it preserves the executing result for the same input. A sufficient sequence of semantic-preserving transformations can normalize semantically equivalent programs that use the same algorithm into those having the same SDG [40].

Fact 5 (code normalization) Code normalization is a process of performing semantic-preserving transformations on the SDG of a program according to a set of predefined rules, so as to transform the semantically equivalent codes to the same representation.

Fact 6 (semantic similarity) Based on Fact 3, Fact 4, and Fact 5, the semantic similarity of P and P' can be computed by matching the normalized SDGs of P and P'.

Fact 7 (semantically similar codes) Based on Fact 2 and Fact 6, P and P' are semantically similar codes if they are candidate similar codes and the semantic similarity between them is no less than the predefined threshold of semantic similarity T_s.

3.3 Model of CMGA

In order to solve the three problems proposed in Section 3.1, we develop a model of CMGA as shown in Fig. 1 based on the facts proposed in Section 3.2.

To solve Problem 1, we improve the traditional SDG. The structure theorem [41,42] puts forward that any proper program, of any complexity, can be represented only by three basic constructs: sequence, selection and iteration. Based on this theorem, some control transfer statements such as *goto* and *break* can be eliminated. Therefore, the CDS is represented as an ordered tree, which is called control dependence tree (CDT) by us. The data flow analysis is only performed on the candidate similar codes in the advanced code normalization stage. As the candidate similar codes only account a small part of the source code, the complexity of representing SDG is greatly decreased, so the augmented SDG is scalable to large programs.

To solve Problem 2, we propose a metrics-based candidate similar codes extracting approach based on Fact 1 and Fact 2 to filter out most of the dissimilar code pairs, so as to prune the search space. The metrics-based approach has an advantage that the computational complexity is much lower than that of graph matching. In order to make CMGA efficient enough for practical applications, we use the metrics-base approach for a fast selection of potential similar codes at a pre-processing stage to prune the search space, when using the more accurate but more computationally expensive method of graph matching.

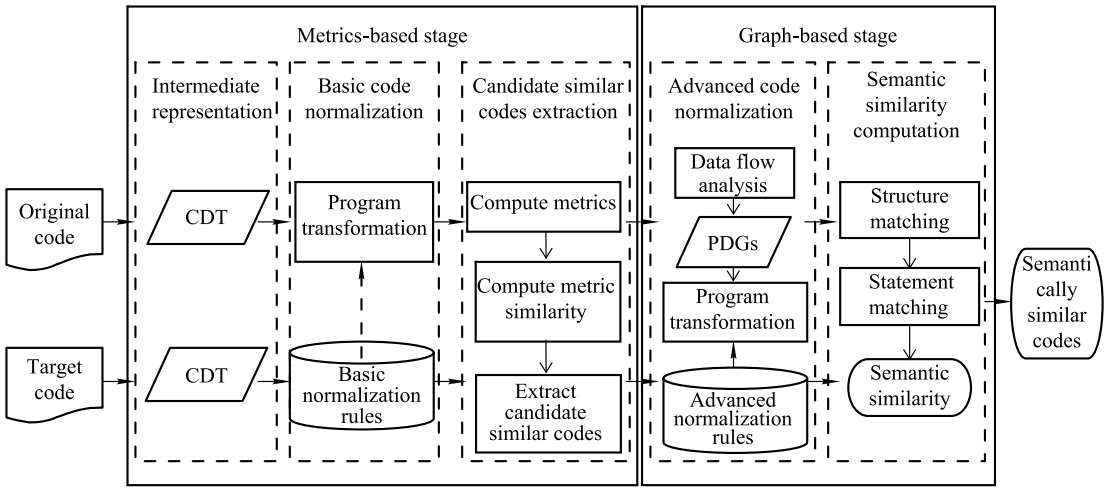

Fig. 1 Model of metric-based and graph-based combined similar codes detection

Due to code variations, both the metrics-based stage and graph-based stage may produce false negatives. To solve this problem, we propose code normalization based on Fact 4 and Fact 5. Semantic-preserving transformations are performed on the augmented SDGs according to a set of predefined transformation rules. We divide the code normalization into basic code normalization and advanced code normalization. The basic code normalization does not need to analyze the data flow information, whereas the advanced code normalization does. Therefore, the space and time cost of the basic code normalization is much lower than that of the advanced code normalization. The advanced code normalization has little influence on the metric similarity, but it affects the accuracy of computing semantic similarity. For these reasons, only the basic code normalization is performed before candidate similar codes extraction. After most of the dissimilar code pairs being filtered out, the advanced code normalization is performed on the candidate similar codes. Based on Fact 6 and Fact 7, semantic similarity can be computed by matching the normalized CDT. Thus syntactically different but semantically similar codes can be recognized, Problem 3 is solved.

4 Augmented SDG

We adapt the traditional SDG in the following aspects to make it suitable for analyzing large programs and detecting semantically similar codes.

1) Due to the complexity of data flow analysis, SDG is not scalable to large software, so we divided SDG into CDS and DDS, data flow analysis is only performed on the small sized candidate similar codes, and program matching is performed on CDS. This not only takes the advantage of SDG in representing program syntactic and semantic information as well as facilitating program matching, but also makes it scalable to large programs by decreasing the complexity of analyzing the data flow.

2) The problem of subgraph isomorphism testing is NP-hard. However, the computational complexity of tree isomorphism is much lower than that of the subgraph isomorphism testing. Based on the structural theorem, any program can be transformed to an equivalent one without *goto* statements [41,42]. A CDS that does not contain *goto* statements can be represented by a tree structure. For these reasons, we transform the graph representation of CDS into tree representation (i.e., CDT).

3) The traditional SDG represents different selection statements as well as different iteration statements by different kinds of nodes. For example, if and switch statements are represented differently. This is not suitable for removing code variations, because it fails to represent syntactically different but semantically equivalent control statements in a unified way. The selection structure node and the iteration structure node are introduced by us to solve this problem. The selection structure node shown in Fig. 2 unifies the representation of all kinds of selection statements. The selection node represents a selection structure, and the selector node and its sub nodes represent a branch of the selection structure. The iteration structure node shown in Fig. 3 unifies the representation of all kinds of iteration statements. The iteration node represents an iteration structure, and its sub nodes represent the body of the iteration. The init_sub node and its sub nodes are only used in do-while statements.

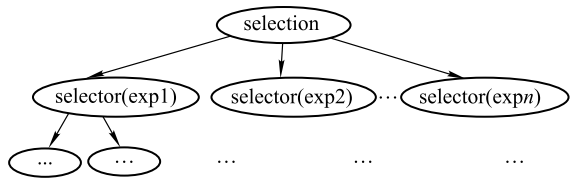

Fig. 2 The selection structure

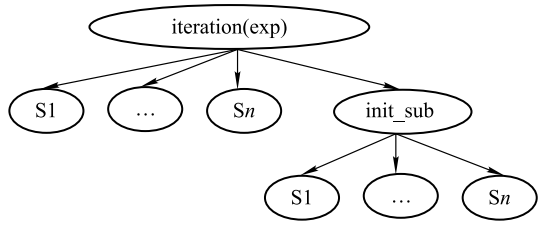

Fig. 3 The iteration structure

4) Expressions are augmented with abstract syntax trees linked to the statement nodes. The advantage of representing expressions by tree structure is that the augmented SDG can represent not only control and data flow, but also syntactic information. It is easy to perform transformations and matching on trees, which can facilitate expression normalization, renaming variables, and similarity computation.

5 Basic code normalization

Code variations, such as code formats variation, compound statement variation, expression variation, and control structure variation, affect the computational accuracy of metric similarity. Therefore, basic code normalization is performed on the CDTs of programs to eliminate such code variations

before the calculation of metrics.

1) Compound statement separation separates compound statements into statement sequences to remove compound statement variations. It includes side effects removal, initializing variable declarations separation, concatenated assignments separation, bringing function calls out of expressions etc. For example, if(sqrt(y) >0){...} is transformed to t=sqrt(y); if(t >0){...}.

2) Expression normalization eliminates expression variations. The rules of expression normalization are built on the basis of computer arithmetic and boolean laws or properties such as identity, commutative laws, distributive laws, and associative laws. They are applied to expression trees of the augmented SDG repeatedly until no further transformations can be applied. For example, ($A1||A2$)&&$A3$ is transformed to ($A1$&&$A3$)||($A2$&&$A3$), where $A1$, $A2$, $A3$ are variables, constants or sub-expressions.

3) Basic control structure normalization transforms the selection and iteration structures to remove control structure variations. It includes unifying the representation of all kinds of selection statements, unifying the representation of all kinds of iteration statements, removing empty or unreachable selection branches, removing iteration structures which conditional expression is constantly false, transforming nested selection to multi-branch selection, merging some special adjoining selection structure, etc. For example, if (exp)A if (!exp)B is transformed to if(exp)A else B.

6 Metrics-based candidate similar code extraction

The metrics-based approach is fast and easy to use. However, it has a drawback that low precision values are obtained. The traditional metrics-based approach usually computes metrics for a single function, rather than considering the interior implementation of the functions called by this function, so it cannot accurately recognize similar code fragments with different module structures. For these reasons, we improve the traditional metrics-based approach, and use it for a fast selection of candidate similar codes at a pre-processing stage to prune the search space.

In our approach, the granularity for comparing code fragments is at the level of module.

Definition (module) A module P is a code fragment composed by the function P and the functions called by P.

Our improved metrics-based approach is based on the assumption that if two code fragments P and P' are similar, the produced code fragments by performing function inlining on P and P' should be similar and have proximate metrics.

Definition (metrics vector) The metrics vector for a module is denoted as $v(f)$: (v_1, v_2, v_3, v_4, v_5, v_6, v_7, v_8, v_9), where v_1 to v_8 represents the number of nodes, operators, selection structures, iteration structures, assignments, special data structures (such as pointer, array, struct etc.), system function calls, control dependence edges respectively, and v_9 represents the depth of the CDT of the module.

The algorithm for extracting candidate similar codes from source code P and target code P' is composed by two steps. Firstly, metrics vector for each module is calculated by traversing the CDTs (the algorithm for computing metrics vectors can be seen in Fig. 4). The metrics of the functions called by a function are added up to the metrics of this function. This is equivalent to calculating the metrics of the function after it being inlining expanded, though the function inlining is not actually performed.

```
Algorithm Compute metrics vectors
Input: function f, CDT
Output: metrics vector v(f) for module f
Process
Calculate the metrics vector v of function f by traversing the
CDT of f
v(f)=v;
if f calls the other functions then
  foreach function g called by f
    if v(g) has already been calculated then
        v(f)=v(f)+v(g);
    else
        v(g)=CMVM(g,CDT);
        v(f)=v(f)+v(g);
    end if
  end for
end if
mark v(f) as calculated;
return v(f);
end process
```

Fig. 4 Algorithm for computing metrics vectors

Then, metric similarities are computed based on metrics vectors, and candidate similar codes are extracted according to the predefined metric similarity threshold T_m.

Suppose the metrics vectors of modules $f \subset P$ and $f' \subset P'$ are $v(f)$: (v_1, v_2, v_3, v_4, v_5, v_6, v_7, v_8, v_9), and $v(f')$: (v'_1, v'_2, v'_3, v'_4, v'_5, v'_6, v'_7, v'_8, v'_9) respectively, the metric similarity of f and f' can be computed as follows.

$$sim(v(f), v(f')) = 1 - \sqrt{\frac{1}{n}\sum_{i=1}^{n}((v_i - v'_i)/\max(v_i, v'_i))^2}. \quad (1)$$

If $sim(v(f), v(f')) > T_m$, the set SG={the CDT subtree of f

and those of the functions called by f} and the set SG'={the CDT subtree of f' and those of the functions called by f'} are inserted into the list of candidate similar codes L, i.e., $L = L \cup \{(SG, SG')\}$.

7 Advanced code normalization

Some code variations have already been eliminated in the basic code normalization stage, but additional code variations need to be eliminated to improve the computational accuracy of semantic similarity. The advanced code normalization stage removes such code variations by transforming the CDTs of the candidate similar codes. In order to make the source codes preserve the same semantic before and after the application of program transformations, data flow analysis is performed before the transformations.

1) Advanced control structure normalization transforms the selection and iteration structures to remove control structure variations. It includes normalizing some special logical expressions in iteration statements, loop fusion, loop invariant motion, eliminating *if-continue*, *if-break*, and *if-return* structures, etc.

2) Redundancies removal eliminates useless statements. It includes constant propagation, identity expression removal, common subexpression elimination, dead code elimination, etc.

3) Variables renaming renames variables according to types and occurrence frequencies. First, variables of the same type are ranked by occurrence frequencies. Then, variables are renamed in the form of "##_#". The first "#" represents the type, such as "*i*" for integer, and "*f*" for float. The second "#" represents the special kind, such as "*A*" for array, and "*P*" for pointer. The third "#" represents the rank of the occurrence frequency. Loop control variables are specially named. Their first character is "*x*". Sub-functions of recursive programs are renamed in the form of *Sub_func_i*. *i* is the order of the sub-function being called for the first time. By variable renaming, semantically equivalent fragments with different variable names can be recognized.

4) Statements reordering rearranges statements which are not dependent on each other. Each node of the CDT is represented by a string symbol in the form of "*X*:*Y*". "*X*" is the type of the node, such as "#declare", "assignment", and "entry". "*Y*" is the expression or statement of the node. For two nodes *S*1 and *S*2, if *S*1 and *S*2 have the same control dependent parent node, do not data depend on each other, and the string symbol of *S*2 is alphabetically less than that of *S*1, *S*2 is put in front of *S*1.

5) Function call normalization inlining expands non-recursive function call statements of user-defined functions based on CDT transformation. As a result, the original code fragment is transformed into a semantically equivalent form that is free of function calls and sub-functions, and the function call variation is removed.

8 Semantic similarity computation

The code normalization stage has transformed semantically equivalent code fragments into the same CDT, so the semantic similarity can be calculated by matching CDTs at the structure level and statement level. The semantic similarity is calculated as follows:

$$SemanticSim = \lambda_{stru} * StructureSim + \lambda_{stat} * StatementSim, \quad \lambda_{stru} + \lambda_{stat} = 1. \tag{2}$$

StructureSim and *StatementSim* are structure similarity and statement similarity respectively. λ_{stru} and λ_{stat} are weights of *StructureSim* and *StatementSim* respectively. They are defined as 0.5 by our experience. Not only the control structure but also the concrete expressions of the program are considered, so the semantic similarity can be accurately evaluated.

Definition (structure tree) A structure tree is a tree extracted from the CDT of a code fragment. It keeps the nodes and control dependence edges of the CDT, however, the nodes in it only have types but no expressions.

Definition (matching node pair) (v_1, v_2) is a matching node pair of structure tree T_1 and T_2, if the following conditions are all satisfied:

1) v_1 is a node of T_1, and v_2 is a node of T_2. Each node can appear in at most one matching node pair.

2) v_1 and v_2 belong to the same type.

3) Their parent nodes constitute a matching node pair.

4) Suppose v_1 matches v_2, w_1 matches w_2, v_1 and w_1 are siblings, and v_2 and w_2 are siblings. Then v_1 comes before w_1 if and only if v_2 comes before w_2.

The structure tree represents the control structure of a program sufficiently. Therefore, the structure similarity can be measured by matching structure trees. The computation of maximum matching between two structure trees is equivalent to computing the number of maximum matching node pairs. We apply the dynamic programming algorithm [43] as shown in Fig. 5 to compute the maximum matching between two structure trees. Finally, the structure similarity is computed

as follows:

$$StrcutureSim = \text{StructureMatching}(S, T)/\max(|S|, |T|). \quad (3)$$

Algortihm: StructureMatching
Input: structure trees *S* and *T*
Output: The maximum matching between *S* and *T*, and the number of maximum matching node pair
Process
 if the roots of *S* and *T* are not matching **then**
  return 0;
 end if
 m=the number of first level subtrees of *S*;
 n= the number of first level subtrees of *T*;
M[*i*][0]=0 **for** *i*=0, …, *m*
M[0][*j*]=0 **for** *j*=0, …, *n*
for(*i*=1; *i*<=*m*; *i*++)
 for(*j*=1; *j*<=*n*; *j*++)
  W[*i*][*j*]= StructureMatching (*Si*, *Tj*);
  Si and *Tj* are the *ith* and *jth* first level subtrees of *S* and *T*.
  M[*i*][*j*]=max(*M*[*i*][*j*-1], *M*[*i*-1][*j*], *M*[*i*-1][*j*-1]+*W*[*i*][*j*]);
 end for
end for
return (*M*[*m*][*n*]+1);
end process

Fig. 5 Algorithm for structure tree matching

The statement matching examines the statement and expression similarity of candidate similar codes. As the structure matching process has already found the type matching nodes, statement matching can be performed on the basis of structure matching. The algorithm for matching expression is similar to that of structure tree matching. It traverses the abstract syntax trees of expressions, computes the expression similarity of each node pair denoted as $value(v_i)$, and stores it in the corresponding node of CDT S. Finally, S is traversed, and statement similarity is computed as follows:

$$StatementSim = \sum value(v_i)/|S|. \quad (4)$$

9 Example

A sample program from Flex2.5.4A and its variation are shown in Fig. 6. Program A and Program B are semantically equivalent but syntactically different. Another sample program is shown in Fig. 7. The *dump_associated_rules* functions in Program C and Program A are identical (for space limit, the implementation of *dump_associated_rules* in Fig. 7 is omitted). But the bubble functions called by *dump_associated_rules* in Program A and Program C are different.

We use Program A as the original code, Program B and Program C as target codes. The processing steps of CMGA are as follows:

1) Each program is represented as a CDT.

2) Basic code normalization is performed on CDTs. For example, the transformations performed on the CDT of Program A are as follows. The macro replacement replaces *MAX_ASSOC_RULES* with 100; compound statement separation separates compound array subscript, variable declaration, and initializing variable declarations into statement sequences; basic control structure normalization transforms the nested selection structure into a multi-branches structure.

3) Candidate similar codes are extracted. Metric similarities of the modules are shown in Table 2. The metric similarities above the metric similarity threshold 0.6 are marked with an asterisk. Two modules are candidate similar codes if and only if the metric similarity between them is larger than the metric similarity threshold, or else they are filtered out. In this example, two thirds of code pairs are filtered out, so the search space is greatly pruned.

Table 2 Metric similarities between the modules of the sample programs

Target codes	Metric similarity between modules of Program A and target modules	
	bubble	*dump_associated_rules*
Program B's *dump_associated_rules*	0.33	*0.90
Program C's bubble	0.58	0.22
Program C's *dump_associated_rules*	0.22	*0.88

4) Data flow analysis is performed on the CDTs of candidate similar codes, and advanced code normalization is performed. For example, the transformations performed on the *dump_associated_rules* module of Program A are as follows. Advanced control structure normalization eliminates the if-break structure; function call normalization inlines bubble into *dump_associated_rules*; variable renaming transforms the corresponding variables names into the same form; statement reordering makes the corresponding statements have the same order. Finally, the *dump_associated_rules* modules of Program A and Program B are transformed to the same CDT as shown in Fig. 8. Because the bubble function of Progam A is different from that of Program C, the *dump_associated_rules* module of Program A is different from that of Program B after function inling being performed.

5) Semantic similarities are computed based on the standardized CDTs. The semantic similarities of the candidate similar codes are shown in Table 3. After advanced code normalization, the *dump_associated_rules* modules of Program A and Program B have the same CDT, so the semantic similarity between them is 1. This means that they are semantically equivalent. The CDT of the *dump_associated_rules*

```
extern int *assoc_rule;
extern int *rule_linenum, *rule_useful;
extern int    **dss, *dfasiz;
#define MAX_ASSOC_RULES 100
void dump_associated_rules( file, ds )
FILE *file;
int ds;
{
      register int i, j;
      register int num_associated_rules = 0;
      int rule_set[MAX_ASSOC_RULES + 1];
      int *dset = dss[ds];
      int size = dfasiz[ds];
      for ( i = 1; i <= size; ++i )
      {
            register int rule_num = rule_linenum[assoc_rule[dset[i]]];
            for ( j = 1; j <= num_associated_rules; ++j )
                  if ( rule_num == rule_set[j] )
                        break;
            if ( j > num_associated_rules )
            {
                  if ( num_associated_rules < MAX_ASSOC_RULES )
                        rule_set[++num_associated_rules] = rule_num;
            }
      }
      bubble( rule_set, num_associated_rules );
      fprintf( file, _( " associated rule line numbers:" ) );
      for ( i = 1; i <= num_associated_rules; ++i )
      {
            if ( i % 8 == 1 )
                  putc( '\n', file );
            fprintf( file, "\t%d", rule_set[i] );
      }
      putc( '\n', file );
}
void bubble( v, n )
int v[], n;
{
      register int i, j, k;

      for ( i = n; i > 1; --i )
            for ( j = 1; j < i; ++j )
                  if ( v[j] > v[j + 1] )
                  {
                        k = v[j];
                        v[j] = v[j + 1];
                        v[j + 1] = k;
                  }
}
```

(a)

```
extern int *assoc_rule;
extern int *rule_linenum, *rule_useful, *dss;
extern int    **dss, *dfasiz;
void dump_associated_rules( file, ds )
FILE *file;
int ds;
{
      register int i, j;
      register int l_ i, l_ j, l_ k;
      register int num_associated_rules;
      int rule_set[101];
      int *dset;
      int size;
      register int rule_num;
      int t1,t2;
      num_associated_rules = 0;
      size = dfasiz[ds];
      dset = dss[ds];
      for ( i = 1; i <= size; ++i )
      {
            t1=dset[i];
            t2=assoc_rule[t1];
            rule_num =rule_linenum[t2];
            for ( j = 1; j <= num_associated_rules
                        && rule_num != rule_set[j]; ++j );
            if ( j > num_associated_rules
                        &&100>num_associated_rules)
            {
                  num_associated_rules=
                                    num_associated_rules+1;
                  rule_set[num_associated_rules]=rule_num;
            }
      }
      for ( l_i = num_associated_rules; l_i > 1; -- l_i )
            for (l_j = 1; l_j < l_i; ++ l_j )
                  if (rule_set [l_j] > rule_set [l_j + 1] )
                  {
                        l_k = rule_set [l_j];
                        rule_set [l_j] = rule_set [l_j + 1];
                        rule_set [l_j + 1] = l_k;
                  }
      fprintf( file, _( " associated rule line numbers:" ) );
      for ( i = 1; num_associated_rules>= i; ++i )
      {
            if ( i % 8 == 1 )
                  putc( '\n', file );
            fprintf( file, "\t%d", rule_set[i] );
      }
      putc( '\n', file );
}
```

(b)

Fig. 6 A sample program from Flex2.5.4A and its variation. (a) Program A; (b) Program B

```
void dump_associated_rules( file, ds )
{/*the body of the function is same to that of Program A,
so it is omitted*/}
void bubble( v, n )
int v[], n;
{
        int i, temp;
        for (i=0, j=len-1; i<j; i++, j--)
        {       temp = str[i];
                str[i] = str[j];
                str[j] = temp;
        }
}
```

Fig. 7 A sample program, Program C

module of Program C is different from that of Program B, so they are not semantically equivalent. If the semantic similarity threshold is 0.8, the result of similar code detection is that the *dump_associated_rules* modules of Program A and Program B are similar code.

Table 3 Semantic similarities of the candidate similar codes

Candidate similar modules in target codes	Semantic similarity between *dump_associated_rules* of Program A and target modules
Program B's *dump_associated_rules*	*1
Program C's *dump_associated_rules*	0.65

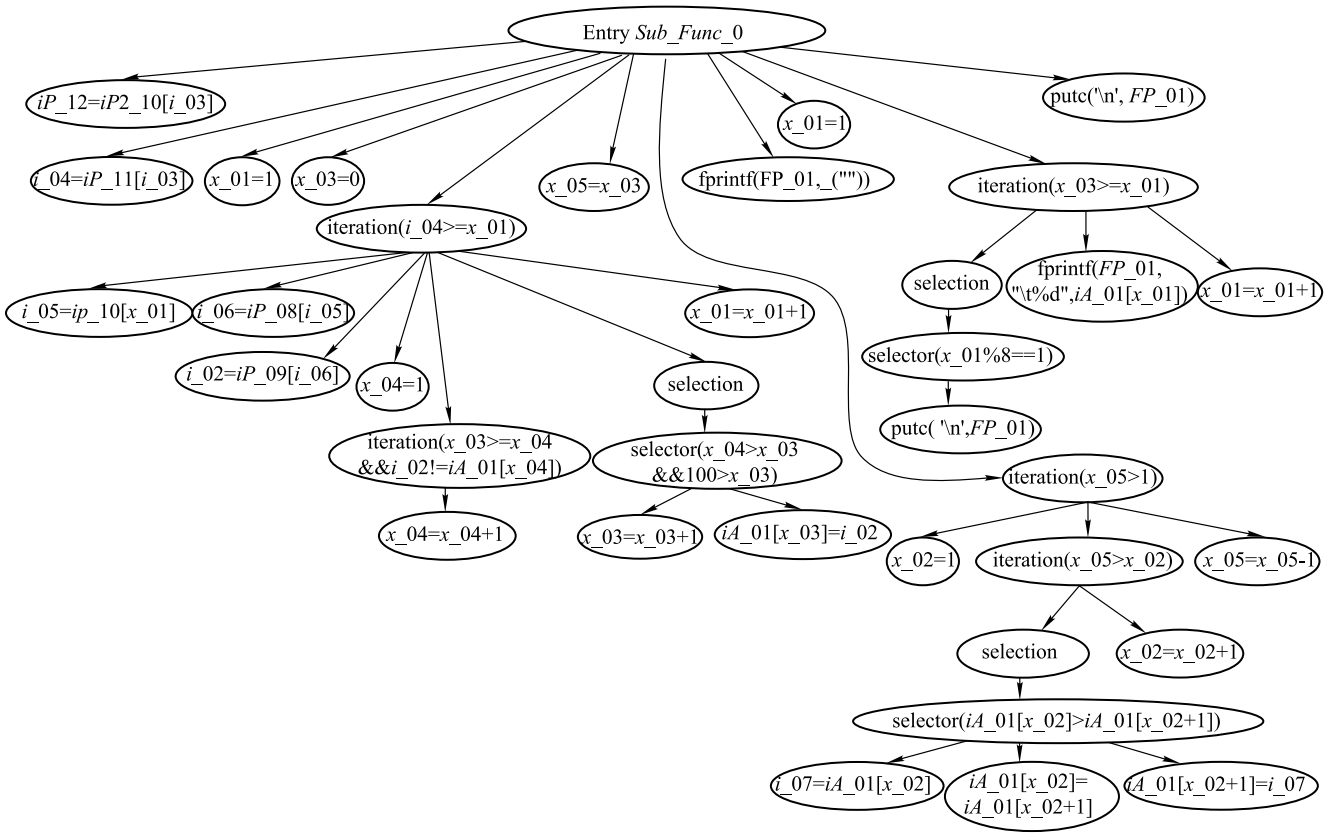

Fig. 8 Normalized CDT for the programs in Fig. 6

It is important to note that with other graph-based approaches, such as GPlag [33] and the integrated space-logic domain model [34], the *dump_associated_rules* module of Program C will be detected as semantically equivalent to the *dump_associated_rules* modules of Program A, as they do not consider the relationships between functions. This is a limitation of these approaches.

10 Computational complexity

The algorithm for constructing SDG is like that of the other graph-based approaches. So we focus on the analysis of the other steps of the CMGA. In all steps, G and G' refer the augmented SDG of the original code and the target code respectively, and each have n and m PDGs. T_1 and T_2 refer the CDT of the original code and the target code respectively.

1) The basic code normalization step: all transformations on the CDTs are stated as tree traversal, so the complexity is $O(|T_1| + |T_2|)$.

2) The candidate similar code extraction step: first, the metrics vectors for every module are computed by traversing CDTs, the time complexity is $O(|T_1| + |T_2|)$. Then, candidate similar codes are extracted, the complexity is $O(|T_1| + |T_2|)$.

3) The advanced code normalization step: all transformations on the PDGs of candidate similar codes are stated as graph walks. The complexity is

$$O(\sum_{i\leqslant k}(|V_i| + |E_i|) + \sum_{j\leqslant l}(|V_j| + |E_j|)), k \leqslant m, l \leqslant n,$$

where G_i=$\{V_i, E_i\}$ and G_j=$\{V_j, E_j\}$ are the PDGs of the original and target candidate similar codes, respectively.

4) The semantic similarity computation step: the dynamic programming algorithm of matching two trees T_i and T_j is $|T_i|\times|T_j|$ [43], therefore the complexity of structure matching is $O(\sum_{i<m,j<n}(T_i|\times|T_j|))$. The complexity of statement matching is $O(\sum_{i\leqslant k,j\leqslant l}(|T_i| + |T_j|)), k \leqslant m, l \leqslant n$, where T_i and T_j are the CDT of the original and target candidate similar code, respectively.

In the worst case, k is equivalent to m, and l is equivalent to n in the advanced code normalization step and the semantic similarity computation step. This means, none of the dissimilar code pairs is filtered out in the candidate similar codes extraction step. However, this seldom happens in practice. Most of the dissimilar code pairs can be filtered out.

CMGA is superior to the other graph-based similar code detection approaches, in that it can recognize code variations. Therefore, it needs additional time for dealing with code vari-

ations. However, its program matching is the polynomial time complexity of the number of nodes in CDTs. Its performance is better than the traditional subgraph isomorphism testing approach, as the problem of subgraph isomorphism testing is NP-hard. Besides, CMGA does not need to analyze the data flow of the PDGs for the whole program, so that the space complexity is decreased.

11 Experiments and discussion

11.1 Research questions

Experiments were carried out to analyze the following research questions.

RQ1: how does the setup of CMGA's similarity thresholds affect its performance on clone detection?

RQ2: is CMGA robust in detecting code clones of various types? In particular, what kinds of code variations can be effectively detected by CMGA?

RQ3: is CMGA time efficient?

RQ4: what is the effectiveness of CMGA of recognizing real world code variations?

RQ5: is CMGA scalable to large source code?

11.2 Experiments design and setup

We implemented CMGA, GPlag [33] and the integrated logic-domain model [34]. GPlag and the integrated logic-domain model are existing graph-based similar code detection models. Experiments were performed on the three models to compare their performance.

All experiments were performed on a 3.2 GHz Pentium IV PC. The programs used in the experiments are shown in Table 4. Flex, Less, Gcc, and Linux are real-world source codes.

Table 4 Real world programs used in the experiments

Source code	LOC	Description
Flex 2.5.4a	22 720	A fast lexical analyzer generator version 2.5.4a
Flex 2.5.39	24 178	A fast lexical analyzer generator version 2.5.39
Less 406	25 678	A file viewer, version 406
Less 458	25 450	A file viewer, version 458
Gcc 1.4	98 791	The GUN C compiler gcc 1.4
Gcc 2.0	261 580	The GUN C compiler gcc 2.0
Linux-drivers	3 273 697	Drivers of linux-2.6.16.8 version
Original code	22 720	Identical to Flex
Target code 1	22 720	Identical to Flex
Target code 2	23 474	A variation of Flex by performing semantically equivalent modification to 52 functions
Target code 3	48 132	The mixture of Flex and Less

To answer RQ1 and RQ2, we firstly created an original code and three target codes based on Flex and Less. The clone detection results between the original code and each target were analyzed.

In order to test how well CMGA performs in detecting similar code with code variations, we created Target code 2 by manually performing semantically equivalent modification to 52 functions. Code variations demonstrated in Section 2.2 and the algorithm variation were injected. That is to say Target code 2 was created in a manually injected way rather than choosing another natural version of Flex. This is for two reasons. First, to compute the false negatives and recall, we need to know the exact number of clone pairs and the number of code variations in the original and target code. However, it is hard to get these numbers for the natural versions. Second, by manually injection, code variations that may not exist in the natural versions can be injected.

Then, the natural versions of Flex (v2.5.4a versus v2.5.39) and Less (v406 versus v458) were analyzed by CMGA, GPlag and the integrated logic-domain model, and the results were manually inspected and compared.

To answer RQ 3, the time of running the above experiments were recorded.

To answer RQ 4, we applied CMGA to analyze Gcc v2.0 and v1.4.

To answer RQ 5, we analyzed the source code of Linux drivers to test the scalability of CMGA.

11.3 Similarity thresholds setup and the effect of metrics-based filtering

CMGA has two similarity thresholds that is metric similarity threshold and semantic similarity threshold. The setup of the two thresholds directly affects the accuracy and efficiency of CMGA. Based on our experience, the metric similarity threshold is defined as 0.6, and the semantic similarity threshold is defined as 0.9.

Figure 9 plots the effect of the candidate similar code extracting step on the three target programs, when the metric similarity threshold varied from 0 to 1.

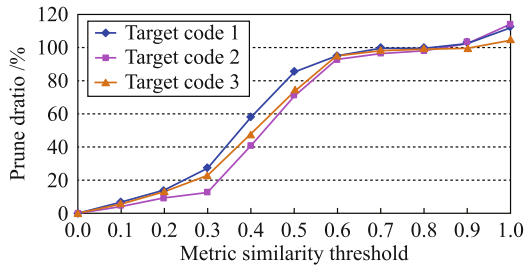

Fig. 9 Relationship between metrics similarity threshold and pruned ratio

The y-axis is the *prunedratio* calculated as follows, where m is the number of modules in the original code, n is the number of modules in the target code, *needsemanticmatch* is the number of candidate similar code pairs, *realmatch* is the number of manually verified semantically similar code pairs.

$$prunedratio = \frac{m \times n - needsemanticmatch}{m \times n - realmatch} \times 100\%. \quad (5)$$

The *prunedratio* should be no larger than 100%. Or else *needsemanticmatch* is less than *realmatch*, which means that there exist false negatives. In the experiment, when the metric similarity threshold is above 0.9, the *prunedratio* is larger than 100%, so false negatives appear. When the metric similarity threshold is between 0.6 and 0.7, the *prunedratio* is larger than 90% but smaller than 100%. This indicates that with proper metric similarity threshold, our metrics-based approach can filter out above 90% semantically dissimilar code pairs. It can greatly reduce the workload of the following semantic analysis.

Figure 10 plots the effect of the semantically similar code detecting step on the three target programs, when the semantic similarity threshold varied from 0 to 1. The y-axis is the F-measure calculated as follows, where P is the precision, and R is the recall of the semantically similar code detection.

$$\text{F-measure} = \frac{2PR}{P+R} \times 100\%. \quad (6)$$

F-measure is a widely used measure for retrieval performance. It is a combination of precision and recall. Only when the value of precision and recall are both high, the value of F-measure is high. In our experiment, when the semantic similarity threshold is 0.9, best result is produced.

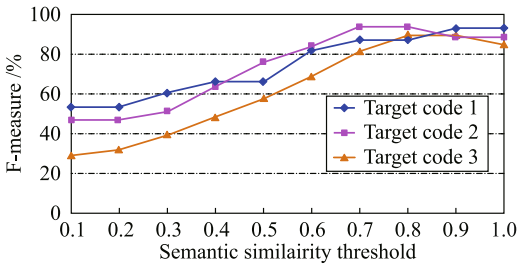

Fig. 10 Relationship between semantic similarity threshold and F-measure

11.4 Robustness analysis

Both injected versions and natural versions of Flex and Less were analyzed to test the robustness of CMGA.

Firstly, CMGA, GPlag and the integrated logic-domain model were used to detect semantically similar code in the original code and each target code. They produced similar result when dealing with Target code 1 and Target code 3. All the three models were efficient in detecting identical codes. However, GPlag, and the integrated logic-domain model did not behave so well as CMGA in detecting similar codes with code variations. We used the false negative rate, and number of false positives to evaluate the robustness of the three models. The result of detecting similar modules with code variations is shown in Table 5.

Table 5 Results of robustness experiment on injected versions

Measure	GPlag	Integrated logic-domain model	CMGA
False negative rate	31.9%	27.1%	4.3%
Number of false positives	10	10	2

As can be seen from Table 5, CMGA had the lowest false negative rate and produced the fewest false positives among the three models. CMGA achieved the best result, because code normalization was performed both in the metrics-based stage and in the graph-based stage. More semantically similar but syntactically different codes were detected by CMGA.

Table 6 lists the number of failures to show what kind of variation can cheat the similar code detection models. As we can see, GPlag and the integrated logic-domain model could successfully handle code format variation, variable name variation, and statements order variation. However, they made more mistakes than CMGA in handling compound statement variation, expression variation, control structure variation, and function call variation. Test results show that CMGA can handle most of the code variations. It is superior to the other models in handling code variations.

Table 6 Number of code variation recognition failures in the injected versions

Kind of code variation	Number of false negatives		
	GPlag	Integrated logic-domain model	CMGA
Code format	0	0	0
Compound statement	44	44	0
Expression	32	32	6
Redundancies	5	3	0
Control structure	30	25	3
Variable name	0	0	0
Statements order	0	0	0
Function call	26	26	0
Algorithm	5	5	5

Secondly, a comparison on natural versions of Flex (v2.5.4a versus v2.5.39) and Less (v406 versus 458) was performed, The results are shown in Tables 7 and 8. The types of clone pair detected by these models includes identical, insert/delete/modify statements, different code format, and with code variations (such as control structure variation, statement order variation). CMGA detected more true clone pairs and

less false positives than the other two models.

Table 7 Results of robustness experiment on the natural versions

Measure	Subject	Clone detection model		
		GPlag	Integrated logic-domain model	CMGA
Number of correct clone pairs	Flex	88	97	104
	Less	338	421	445
Number of false positives	Flex	6	6	2
	Less	8	9	1

Table 8 Types of clone pairs detected by the three models in the natural versions

	Identical	Insert/delete/modify statements	Different code format	Other variations
Flex	25	35	40	4
Less	357	84	1	3

The limitation of code variation identification is the main reason for higher false negative rate of GPlag and the integrated logic-domain model. Besides, GPlag produced several false negatives because it excluded the PDGs smaller than a certain size. Moreover, it also used a lossy filter in order to improve the efficiency. The integrated logic-domain model also produced false negatives for the reason of the threshold setup of the winnowing algorithm.

Some false positives of GPlag and the integrated logic-domain model were caused by lacking of consideration of function calls as demonstrated in Section 9. On the other hand, GPlag adopted γ-isomorphic graph matching, which allowed certain degree of mismatching. The integrated logic-domain model used graph matching algorithm on each PDG pair to detect if there was any matched subgraph inside the PDG pair. In order to reduce computational cost, it only output the first match in case there were several. This sometimes produced false positives.

Although CMGA can correctly deal with most of the code variations, it still had false negatives and false positives. This is because detecting semantically equivalent source codes is an undecidable problem. The normalization rules and similarity thresholds are all induced based on our experience, so that they may not be complete to deal with all the cases. This is a limitation.

11.5 Speed analysis

We experimented on the three target codes to evaluate the speed of CMGA against GPlag and the integrated logic-domain model. The result is shown in Table 9.

As we can see, CMGA has better performance than GPlag and the integrated logic-domain model. This is because CMGA uses the metrics-based approach for a fast selection of potential similar codes at a pre-processing stage, and then performs the more accurate matching based on the CDT tree structure. The cost of tree-matching is much lower than that of graph-matching adopted by GPlag and the integrated logic-domain model.

Table 9 Speed performance comparison

Code	Execution time /s		
	GPlag	Integrated logic domain model	CMGA
Target code 1	680	560	416
Target code 2	693	577	439
Target code 3	2356	1235	1062
Flex natural versions	731	608	459
Less natural versions	962	887	509

11.6 Study of Gcc v.2.0 and v1.4

We applied CMGA to analyze the source codes of Gcc 2.0 and 1.4. The source code of Gcc 2.0 was used as the original code, and the source code of Gcc 1.4 was used as the target code. Nine hundred and seventeen code pairs were detected as similar codes. They were verified to be real similar codes by manual inspection. Not only syntactically similar but also semantically similar source codes were detected. 86 similar code pairs containing code variations were detected. Ten syntactically different but semantically equivalent code pairs were recognized by CMGA. The code variations correctly recognized by CMGA is shown in Table 10. As can be seen, code variations widely exist in real world software, the code normalization and detecting similar codes at the semantic level proposed by us is necessary and effective.

Table 10 Number of code variations of Gcc2.0 and 1.4 recognized by CMGA

Kind of code variation	Number of code variations
Compound statement	10
Expression	9
Redundancies	6
Control structure	10
Variable name	29
Statements order	3
Function call	25

11.7 Study of Linux-2.6.16.8\drivers

We applied CMGA to analyze the source codes of Linux-2.6.16.8\drivers. The execution time is shown in Table 11. As can be seen, the execution time of CMGA for analyzing larger scale source code is acceptable.

59 pairs of modules were detected as similar codes. They

were verified to be similar codes by manual inspection. Nine clone pairs containing code variations were recognized by CMGA. As shown in Table 12 similar codes containing code variations, such as variable name variation, statement order variation, control structure variation, function call variation were correctly recognized.

Table 11 Execution time of CMGA for analyzing Linux drivers

Steps of CMGA	Execution time
Construct CDT	47min18s
Basic code normalization	20min9s
Candidate similar code extraction	7min23s
Advanced code normalization	18min42s
Semantic similarity computation step	22min56s
Total	116min28s

Table 12 Number of code variations of Linux -2.6.16.8\drivers recognized by CMGA

Kind of code variation	Number of code variations
Control structure	1
Variable name	4
Statements order	3
Function call	1

12 Conclusions

The traditional similar code detection approaches can not detect similar code at the semantic level, limiting their applications in practice. In this paper, we have presented a metrics-based and graph-based combined approach. It can detect not only identical codes but also similar codes with code variations, thus it can detect similar codes at the semantic level. Test results show that it is superior to the existing graph-based similar code detecting models GPlag and the integrated logic-domain model in accuracy and efficiency, and it can be applied to large software.

The novel aspects of our approach are as follows:

1) The traditional SDG is augmented. The CDS of SDG is represented as an ordered tree called CDT, and data flow analyzing is only performed on the small sized candidate similar codes, so that the complexity of SDG representation is greatly lowered. The similar code detection is based on tree matching. This avoids the problem of high computational complexity of subgraph isomorphism testing.

2) Code normalization is proposed to eliminate code variations, and metric similarity is computed at the level of module. By doing this, the accuracy of computing metric similarity is improved and similar codes with code variations can be correctly detected. With the normalization rules, syntactically different but semantically equivalent code fragments can be transformed to a uniform representation, so that these rules can also be applied to facilitate program comprehension, program optimization, etc.

3) The improved metrics-based approach and the improved graph-based approach are well integrated to form the similar code detection model CMGA. First, source codes are represented as CDTs. Then the metric-based candidate similar code detection step filters out most of the dissimilar code pairs to lower the computational complexity. After that, advanced code normalization is performed on the candidate similar codes to remove code variations so as to detect similar code at the semantic level. Finally, program matching is performed on the standardized CDT to output more accurate semantic similarity.

The metrics-based and graph-based combined similar code detection approach is not limited to plagiarism code detection, software components library lookup, and duplicated codes detection, but can be applied to the other software analysis applications. For example, we plan to apply it to program comprehension for recognizing the semantics of codes.

Acknowledgements This work was supported by the National Natural Science Foundation of China (Grant Nos. 61202092 and 61173021), the Research Fund for the Doctoral Program of Higher Education of China (20112302120052), Research Fund for the Innovative Scholars of Harbin (RC2013QN010001), and Young Colleger Academic Backbone Project of Heilongjiang.

References

1. Bettenburg N, Shang W Y, Ibrahim W, Adams B, Zou Y, Hassan A E. An empirical study on inconsistent changes to code clones at the release level. Science of Computer Programming, 2012, 77(6): 760–776
2. Duala-Ekoko E, Robillard M P. Clone region descriptors: representing and tracking duplication in source code. ACM Transactions on Software Engineering and Methodology, 2010, 20(1): Article No. 3
3. Krinke J. A study of consistent and inconsistent changes to code clones. In: Proceedings of the 14th Working Conference on Reverse Engineering. 2007, 170–178
4. Nguyen H A, Nguyen T T, Pham N H, Al-Kofahi J, Nguyen T N. Clone management for evolving software. IEEE Transactions on Software Engineering, 2012, 38(5): 1008–1026
5. Thummalapenta S, Cerulo L, Aversano L, Penta M D. An empirical study on the maintenance of source code clones. Empirical Software Engineering, 2010, 15(1): 1–34
6. Bruntink M, Van Deursen A, Van Engelen R, Tourwe T. On the use of clone detection for identifying crosscutting concern code. IEEE Transactions on Software Engineering, 2005, 31(10): 804–818
7. Li J, Ernst M D. CBCD: cloned buggy code detector. In: Proceedings of the 34th International Conference on Software Engineering. 2012,

310–320

8. Li Z, Lu S, Myagmar S, Zhou Y. CP-Miner: finding copy-paste and related bugs in large-scale software code. IEEE Transactions on Software Engineering, 2006, 32(3): 176–192
9. Rahman F, Bird C, Devanbu P. Clones: what is that smell? Empirical Software Engineering, 2012, 17(4–5): 503–530
10. Roy C K, Cordy J R, Koschke R. Comparison and evaluation of code clone detection techniques and tools: a qualitative approach. Science of Computer Programming, 2009, 74(7): 470–495
11. Church K W, Helfman J I. Dotplot: a program for exploring self-similarity in millions of lines of text and code. Journal of Computational and Graphical Statistics, 1993, 2(2): 153–174
12. Ducasse S, Rieger M, Demeyer S. A language independent approach for detecting duplicated code. In: Proceedings of the IEEE International Conference on Software Maintenance. 1999, 109–118
13. Manber U. Finding similar files in a large file system. In: Proceedings of the 1994 Usenix Winter Technical Conference. 1994, 1–10
14. Roy C K, Cordy J R. NICAD: accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization. In: Proceedings of the 16th IEEE International Conference on Program Comprehension. 2008, 172–181
15. Baker B S. On finding duplication and near-duplication in large software systems. In: Proceedings of the 2nd Working Conference on Reverse Engineering. 1995, 86–95
16. Baker B S. Finding clones with dup: analysis of an experiment. IEEE Transactions on Software Engineering, 2007, 33(9): 608–621
17. Kamiya T, Kusumoto S, Inoue K. CCFinder: a multilinguistic token-based code clone detection system for large scale source code. IEEE Transactions on Software Engineering, 2002, 28(7): 654–670
18. Livieri S, Higo Y, Matushita M, Inoue K. Very-large scale code clone analysis and visualization of open source programs using distributed CCFinder: D-CCFinder. In: Proceedings of the 29th International Conference on Software Engineering. 2007, 106–115
19. Ueda Y, Kamiya T, Kusumoto S, Inoue K. On detection of gapped code clones using gap locations. In: Proceedings of the 9th Asia-Pacific Software Engineering Conference. 2002, 327–336
20. Higo Y, Kamiya T, Kusumoto S, Inoue K. Method and implementation for investigating code clones in a software system. Information and Software Technology, 2007, 49(9): 985–998
21. Baxter I D, Yahin A, Moura L, Sant'Anna M, Bier L. Clone detection using abstract syntax trees. In: Proceedings of the International Conference on Software Maintenance. 1998, 368–377
22. Koschke R, Falke R, Frenzel P. Clone detection using abstract syntax suffix trees. In: Proceedings of the 13th Working Conference on Reverse Engineering. 2006, 253–262
23. Prechelt L, Malpohl G, Philippsen M. JPlag: finding plagiarisms among a set of programs. Technical Report, Department of Informatics, University of Karlsruhe. 2000
24. Wahler V, Seipel D, Wolff J, Fischer G. Clone detection in source code by frequent itemset techniques. In: Proceedings of the 4th IEEE International Workshop on Source Code Analysis and Manipulation. 2004, 128–135
25. Balazinska M, Merlo E, Dagenais M, Lague B, Kontogiannis K. Measuring clone based reengineering opportunities. In: Proceedings of the 6th International Software Metrics Symposium. 1999, 292–303
26. Davey N, Barson P, Field S, Frank R, Tansley D. The development of a software clone detector. International Journal of Applied Software Technology, 1995, 1(3–4), 219–236
27. Kontogiannis K A, DeMori R, Merlo E, Galler M, Bernstein M. Pattern matching for clone and concept detection. Automated Software Engineering, 1996, 3(1–2): 77–108
28. Mayrand J, Leblanc C, Merlo E M. Experiment on the automatic detection of function clones in a software system using metrics. In: Proceedings of the International Conference on Software Maintenance. 1996, 244–253
29. Patenaude J F, Merlo E, Dagenais M, Lague B. Extending software quality assessment techniques to java systems. In: Proceedings of the 7th International Workshop on Program Comprehension. 1999, 49–56
30. Schleimer S, Wilkerson D S, Aiken A. Winnowing: local algorithms for document fingerprinting. In: Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data. 2003, 76–85
31. Komondoor R, Horwitz S. Using slicing to identify duplication in source code. Lecture Notes in Computer Science, 2001, 2126: 40–56
32. Krinke J. Identifying similar code with program dependence graphs. In: Proceedings of the 8th Working Conference on Reverse Engineering. 2001, 301–309
33. Liu C, Chen C, Han J, Yu P S. 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. 2006, 872–881
34. Qu W, Jiang M, Jia Y. Software reuse detection using an integrated space-logic domain model. In: Proceeding of the IEEE International Conference on Information Reuse and Integration. 2007, 638–643
35. Gabel M, Jiang L, Su Z. Scalable detection of semantic clones. In: Proceedings of the 30th International Conference on Software Engineering. 2008, 321–330
36. Ferrante J, Ottenstein K J, Warren J D. The program dependence graph and its use in optimization. ACM Transactions on Programming Languages and Systems, 1987, 9(3): 319–349
37. Binkley, D, Horwitz, S, Reps, T. The Multi-Procedure Equivalence Theorem. CS Technical Reports, Computer Sciences Department, University of Wisconsin-Madison. 1989
38. Church K W, Helfman J I. Dotplot: a program for exploring self-similarity in millions of lines of text and code. Journal of Computational and Graphical Statistics, 1993, 2(2): 153–174
39. Horwitz S, Prins J, Reps T. On the adequacy of program dependence graphs for representing programs. In: Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. 1988, 146–157
40. Xu S, San Chee Y. Transformation-based diagnosis of student programs for programming tutoring systems. IEEE Transactions on Software Engineering, 2003, 29(4): 360–384
41. Ammarguellat Z. A control-flow normalization algorithm and its complexity. IEEE Transactions on Software Engineering, 1992, 18(3): 237–251
42. Williams M H, Ossher H L. Conversion of unstructured flow diagrams to structured form. The Computer Journal, 1978, 21(2): 161–167
43. Yang W. Identifying syntactic differences between two programs. Software: Practice and Experience, 1991, 21(7): 739–755

Tiantian Wang is an associate professor at Harbin Institute of Technology, China. She received the PhD degree from Harbin Institute of Technology in 2009. Her current research interests are program analysis, automatic software debugging, and computer aided education.

Kechao Wang received the MS degree from Huazhong Univeristy of Science and Technology, China in 2006. Since 2012, he has been a PhD candidate in Computer Science Department of Harbin Institute of Technology. His current research interests are software fault localization and program analysis.

Xiaohong Su is a professor of Harbin Institute of Technology. She is a senior membership of China Computer Federation. Her main research interests are software bug detection, graphics and image processing, information fusion, and intelligent computation.

Peijun Ma is a professor of Harbin Institute of Technology. His main research interests are software engineering, information fusion, color matching, image processing, and intelligent control.