ATLANTIS PRESS

Advances in Computer Science Research, (ACSR), volume 52

2016 International Conference on Computer Engineering and Information Systems (CEIS-16)

Code Similarity Detection by Program Dependence Graph

Zhen Zhang, Hai-Hua Yan, Xiao-Wei Zhang

Dept. of Computer Science, Beihang University, Beijing, China

E-mail: tater.zhangzhen@gmail.com, yhh@buaa.edu.cn, eipei@126.com

***Abstract*-Existing tools are rarely able to check semantic similarity code. Although some tools consider the programs' semantics, but their detection efficiency is not high. At the same time, function splitting problem confused graph-based tools. In this paper, We propose a approach which combines the software static analysis with the program dependence graph to detect similar code and solve function splitting problem. First, in order to reduce the computational complexity of the program dependence graph's building and comparison, we used software static analysis method to extract testing functions fair. Then, we create the program dependence graph for testing functions fair. Finally, graph matching is performed on isomorphism testing to output similar function and the set of functions which may be split or merged. Experiment results show that our approach can detect all kinds of variations in the code, and adding software static analysis can improve the the detection efficiency based on program dependence graph approach.**

Keywords-program dependence graph; software metrics; code similarity detection; isomorphism testing

I. INTRODUCTION

With the rich of open source project, copying code becomes more and more easy, which brought a lot of troubles to the protection of intellectual property. When checking whether a code similar to another, the similarity detection technology plays an important role. At the same time, the similarity detection also play a significant role on code clone detection, reconstruction and maintenance of software, bug search and so on.

A lot of detection techniques have been proposed for code similarity detection. Generally those techniques can be divided into text-based[1][2], token-based[3], tree-based[4], graph-based[5][6][7] and metrics-based[8]. Recent research shows that in order to improve the accuracy of detection, semantics-based [9][10[11] get a closer study, and a number of mixed use of technology[10][12][13] to improve the efficiency of detection. The method based on program dependence graph has a very good detection effect in semantic level, but there are two problems need to be solved. First, the method based on program dependence graph involves the establishment of graph and isomorphism testing, which often has a large computational complexity. Komondoor's[5] and krinke's[6] approaches can't be applied to large-scale program, Liu[7] proposed lossless filter and lossy filter to prune the plagiarism search space. However, program dependence graph still needs to be built for a process. Second, the method based on program dependence graph is concerned with the single function and the relationship between the functions is not considered. So splitting and merging of functions may confuse it.

In order to solve the problem mentioned above, we propose the use of software static analysis and program dependence graph combined approach to detect code similarity. For the first question, we use comparison of the function of metrics information and function parameters to get rid of the obvious is not related to the function to reduce the number of program dependence graph which need to generate and compare, in order to achieve the goal of reducing the amount of computation. For the second problem, we use function splitting set and function call relationship to find the splitting function.

Based on the above ideas, we implemented a program dependence graph based code similarity detection tool CSD. Experiment results show that CSD can effectively and efficiently detect all kinds of variations in the code, and give the results of simple function splitting and merging.

The main work we have done in this paper is summarized as follows:

- We designed and implemented a tool CSD based on program dependence graph and software static analysis, which can effectively detect the code similarity.
- We propose a method which combines the software static analysis with the program dependence graph to reduce the computation of graph-based detection approach. At the same time, we solved simple function splitting problem with call relationship between functions.
- The testing function pair extraction algorithm and program dependence graph generation algorithm are proposed, which can efficiently get testing function pair and semantic representation of a program.

The rest of the paper is organized as follows. Section II describes code variations and the related concepts. Section III introduces our tools and methods. Experiment verification is presented in section IV. Section V discusses the related work. Finally, sections VI concludes this study.

II. BACKGROUND

In this section we discuss code variations and related concepts of our research, for easy reading the following sections.

A. Code Variations

Joy [14] and Jones [15] summarized code variations, which can be mainly divided into the following type.

- Code format alteration: Semantically equivalent codes may be different in code format, such as

more or fewer code indentation, blank lines and comments.

- Identifier renaming: Replace variable name using another variable name.
- Statement reordering: Change the order of execution of program statements.
- Control replacement: The replacement control statement into an equivalent control statements, does not change the semantics of the program. For example, replace the while loop with a do-while loop or for loop.
- Code insertion: Insert some useless codes and some temporary variables in program.
- Data type replacement: Replace data type with a compatible data type.
- Generate a new function: A function is split into two or more different functions.

B. *Software Static Analysis*

1) Selection of metrics: Code variations will change the value of a lot of code metrics, like comment code number, variable name length, the number of system function call and the number of blank lines and so on. we select the metric is as follows under the consistent semantic code is not susceptible to modify the code.

2) Related concepts

- DEFINITION 1 (Cyclomatic Complexity). McCabe metric method [16] includes a number of metrics, we select the cyclomatic complexity as one of our metrics, it is used to measure the complexity of a module.
- DEFINITION 2 (Myers Extended Complexity). Myers proposed expanded the cyclomatic complexity metric [17], using the interval metric to reflect the data flow and the complexity of the decision statement.
- DEFINITION 3 (Maximum Control Depth). Represents the maximum depth of the nested control structure in a process.
- DEFINITION 4 (Numbers of Execution Statements). Number of statements that can be executed in a program.
- DEFINITION 5 (Numbers of Declaration Statements). Number of declaration statements in the program, including the function declaration and variable declaration .
- DEFINITION 6 (Interface Prototype). The interface prototype includes the return value type, the number of parameters and the type of parameters.
- DEFINITION 7 (Function Call Relationship). Represents the call between the functions, the function α call function β can be expressed C=(α→β).
- DEFINITION 8 (Testing Function Pair). The function pair need to generate program dependence graph and have a isomorphism testing, can be expressed TF=(f,f').

C. *Program Dependence Graph*

- DEFINITION 1 (Abstract Syntax Tree). Abstract syntax tree (AST) is a representation of the abstract syntax tree structure of the source code.
- DEFINITION 2 (Program Dependence Graph). Program dependence graph (PDG) is a directed graph for a single procedure of a program, can be expressed G = (V, E), V is the set of nodes representing statements and predicates. E is the set of edges representing data or control dependence relationship between nodes[10].
- DEFINITION 3 (Control Dependence Graph). Control dependence graph (CDG) represents the control flow of program. I consists of statement nodes and control dependence edges.
- DEFINITION 4 (Data Dependence Graph). Data dependence graph (DDG) represents the data flow of program. It consists of statement nodes and data dependence edges.
- DEFINITION 5 (γ -isomorphism). A graph G is γ -isomorphic to G if there exists a subgraph $S \subseteq G$ such that S is subgraph isomorphic to G, and $|S| \geq \gamma |G|$, $\gamma \in (0, 1]$[7].
- DEFINITION 6 (Program Dependence Graph Isomorphism Ratio). Program dependence graph isomorphism ratio IsomorphismRatio= Σ |g''|/|g'|, g'' is the data dependence sub-graph of g' and g'' is isomorphic with g, |g''| represent the number of nodes of g''. This ratio is used to find the splitting function.
- DEFINITION 7 (Similar Function Pair). Two functions may be similar if their PDGs is isomorphic, they can be expressed as SF = (f, f ').
- DEFINITION 8 (Function Splitting Set). One function may be split into several functions, expressed as f = (f1, f2, f3 ...).

III. CODE SIMILARITY DETECTION

In this section we introduce our tools and methods. Section A introduces the model of CSD, section B discusses the extraction of testing function pair, section C describes the generation of PDG and section D presents similarity computation and function splitting problem.

A. *Model of CSD*

In order to solve the problems proposed in section I, we develop a model of CSD as shown in Fig. 1. First, CSD uses software static analysis (including software metrics and interface prototypes) to obtain the testing function pair. Then, the abstract syntax tree file is gained by compiling source file and program dependence graph is generated. Finally, graph matching is performed on isomorphism testing to output the similar function pairs and the splitting functions are analysed.

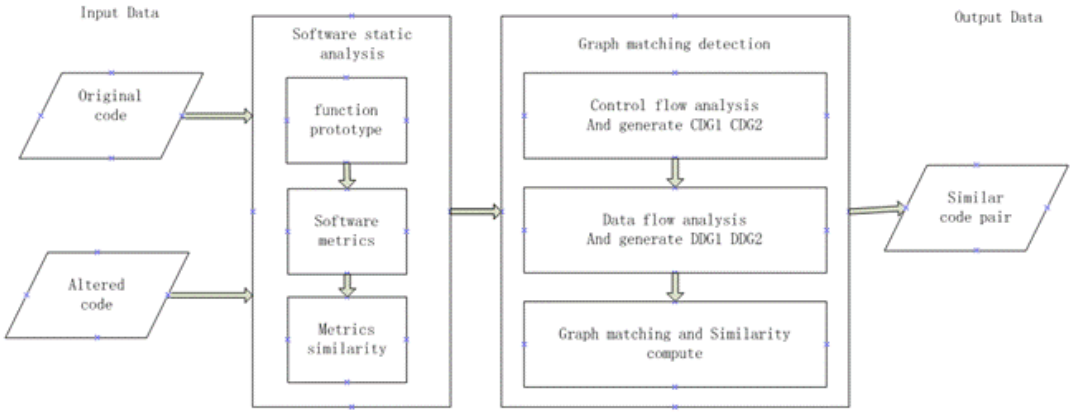

Figure 1. Model of software static analysis and graph-based combined code similarity detection.

B. *Software Static Analysis to Extract Testing Function Pair*

Generally If two functions are similar, their metric values and parameters are approximate. The metrics we selected include cyclomatic complexity, myers extended complexity, maximum control depth, numbers of execution statements, numbers of declaration statements. If the function is split, the parameters of the function are approximate. The number of their parameters is reduced or increased, and the metrics of the function call them are similar. Based on this fact, we select software metrics and interface prototype to improve the traditional metrics-based approach to extract testing function pair. If a function pair satisfies one of the following conditions, it is added to the set of testing function pair.

- Metric values of the function pair are approximate.
- The interface prototypes of the function pair are similar

The software static analysis approach is mainly divided into two parts, the function prototypes and the function metrics. We can obtain functions' prototype by AST files and function metrics by tool metre.

After collecting function prototypes and function metrics, we can obtain the testing function pair. If one function parameters set is subset of another's, we think the two functions prototypes are approximate. The threshold set of function metrics is defined as T(1,1,1,0.9,0.9) by our experiments and observations.

Algorithm Get Testing Function Pair

Input: Function prototypes set P, Function metrics set M M', Threshold of function metrics
Output: Testing Function Pair
Begin
1: T(t1,t2,t3,t4,t5)=The threshold of function metrics
2: F=The set of program1's functions
3: F'=The set of program2's functions
4: TF=The set of testing functions pair
5: for each f∈ F
6: for each f'∈F'
7: if P(f)⊂P(f') or P(f')⊂P(f)
8: TF=TF∪(f,f')
9: End if
10: If min(M,M')/max(M,M') ≥ T
11: TF=TF∪(f,f')
12: End if
13: return TF
End

C. *Generate Program Dependence Graph*

Mary proposed the algorithm constructs a program dependence graph when the program is being parsed[18]. Based this idea, we proposed our PDG generation algorithm considered our syntax tree structure information.

1) Generation of control dependence graph

a) Related concepts: The statements structure in programming language can be divided into sequence structure, selection structure and iteration structure. We create node for every type of statements, and create dependency edges for the nodes. The type of statements (Table I) and control dependency relations are defined as follows.

DEFINITION(Control Dependence). If the execution of node v' depends on the predicate decision of node v, there is a control dependence between v' and v.

TABLE I. STATEMENT TYPE

Type	Description
Decl ,Bind,Call	declarations , statements block and function call statements
Cond, Switch,Case_label	If Switch and other statements in the selection structure
Goto	While for do-while and goto statements in the iteration structure

b) Adding control dependence for nodes: We use depth first search of abstract syntax tree to generate dependency edges for nodes. First, we obtain the program's statements and flow chart and create the first node. Then, we will use different algorithms to deal with different statements. For the sequence structure, the statement node is connected with the first node or the entry node of selection structure and iteration structure. For the selection structure, the judgment statement needs to add two dependence edges, which indicates the direction of the execution flow when the judgment statement is true or false. At the same time, we will add dependence edges for the successor statement and the last node of the selection structure. For the iteration structure, the dependence of the loop execution and the successor dependency of the loop is established. Meanwhile, we create dependence edges for the successor statement and break and continue statements. Finally, the program first node is connected with the entry node of the sequence structure, the selection structure and the iteration structure.

c) Algorithm: The algorithm for generating a control dependency graph is as follows.

Algorithm Generate CDG

Input: AST file, Function name
Output: CDG of function
Begin
1: Create program enter node
2: Find function AST information in AST file
3: ST=The set of function statement
4: for each st in ST
5: Do
6: switch(st type)
7: call deal_sequence
8: or call deal_selection
9: or call deal_loop
10: Done
11: Format Enter Node
End

2) Generation of data dependence graph

a) Related concepts

- DEFINITION(Data Dependence). There is a data dependence edge from program vertex V1 to V2 if v2 uses the variable defined in V1 and the variable value is not changed in the path V1 to V2[7].
- DEFINITION(Reaching Definitions). We say a definition d reaches a point p if there is a path from the point immediately following d to p, such that d is not "killed" along that path. We kill a definition of a variable x if there is any other definition of x any where along the path[19].
- DEFINITION(GEN Set). The set of definitions generated by the statement.
- DEFINITION(KILL Set). The set of definitions killed by the statement.
- DEFINITION(IN Set). The set of definitions reaching the entry of each program block.
- DEFINITION(OUT Set). The set of definitions reaching the exit of each program block.
- DEFINITION(USE Set). The set of definitions used by the statement. We propose a USE set to directly establish the data dependence between statements nodes when IN and OUT set are calculated.

b) Adding data dependence for nodes: We add data dependence for each node based on the control dependence graph. Since we have normalized the first node and the entry node of selection structure and iteration structure, so we can use depth first search algorithm to generate IN and OUT and USE set for every node. Based on the definition of reaching definitions, we add data dependence for node A and node B if node A used the variable defined in node B from IN set of node A. For the iteration structure, it is required to process data dependence many times for every nodes until the OUT set of each node in the iteration structure no longer changes.

c) Algorithm : For generation of data dependence, we also use the depth first search to traverse the control dependence graph previously generated. We establish the IN/OUT set for every statement node using iterative algorithm to compute reaching definitions[19]. In the meantime, we create the data dependence for node using the relationship between IN and USE set.

Algorithm Generate DDG

Input: CDG
Output: DDG
Begin
1: OUT[Entery]=Ø
2: For each basic block b in CDG
3: OUT[b]=Ø
4: while OUT value changed
5: DO
6: P=The predecessor set of basic block b
7: For each p in P
8: IN[b]= ∪ OUT[p]
9: OUT[b]=GEN[b]-(IN[b]-KILL[b])
10: Generate USE[b]
11: Generate data dependence between basic block b and node in IN[b]
12: Done
End

D. Similarity Computation and Function Splitting

1) Function similarity computation: We will generate PDGs for every function pair from the testing function pair and have γ -isomorphism test for them. If they pass the test, we think they are similar. If they fail, we will calculate the

program dependence graph isomorphism ratio for them. This ratio can be used to guess the function splitting. If a testing function pair don't pass the γ-isomorphism test but their program dependence graph isomorphism ratio is greater than a certain threshold defined as 9.5 by our experiment, we will mark this function pair as the splitting function.

2) Function splitting: There are many forms of function splitting, for example, the function prototype doesn't change, but the whole function is divided into two parts or many parts, and by calling these functions instead of the call of original function. A different situation is a part of function is divided into another function, and the function is called in the original function.

In our study, we only solve the first situation. CSD obtain the call functions of the function marked splitting function with the function call relationship. If one of them is main function, the function splitting is determined by program dependence graph isomorphism ratio. Otherwise, we compare the function metrics of the call function. If their metrics are approximate, it can be considered that the functions marked splitting function are split.

IV. EXPERIMENT EVALUATION

In this section, we evaluate the effectiveness and efficiency of CSD through experiments. Section A describes the experiment design and setup, and the following subsections discuss the details of the experiment results.

A Experiment Design and Setup

We compare CSD with APDG[20] and MOSS for effectiveness evaluation, using APDG's public data set. APDG is also a code similarity detection tool based on PDG, MOSS is a token-based code similarity detection tool. The classification of the data set is shown in Table II. We collect more than one hundred functions from open source flex-2.5.39 and less-406 to evaluate the efficiency of CSD with software static analysis.

TABLE II. PROGRAM CLASSIFICATION

Type	Number of functions
Format Alteration (FA)	6
Identifier Renaming(IR)	20
Declaration Reordering(DR)	14
Statement Reordering(SR)	15
Code Insertion(CI)	19
Control Replacement(CR)	11
Other Modification(OM)	8

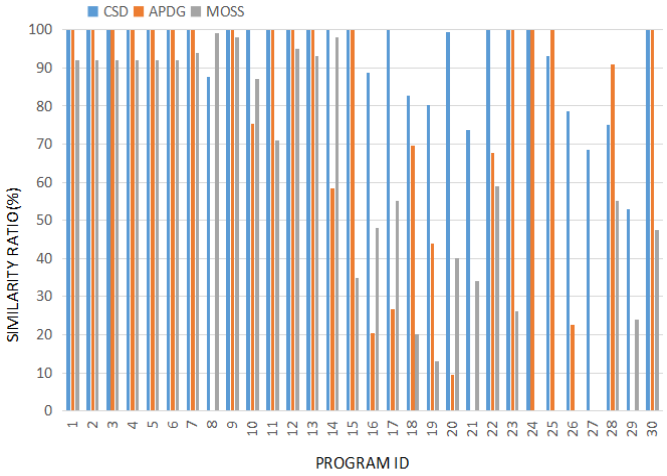

Figure 2. Similarity ratio of each program.

All experiments were run in a Core Duo (TM) 2 PC with 4G physical memory, running CENTOS7. The compiler is gcc4.8.3.

B. Effectiveness of CSD

When evaluating the effectiveness of CSD, we choose the APDG data set, which can be obtained on its web site. The code variations in this data set includes the format alteration, the identifier renaming, the statement reordering, the control structure replacement, inserting the redundant code, changing the data type, generating a new function. We get the metric files and the function calls files which corresponding to 30 programs in data set through a shell script, and then CSD detect every program pair. The experiment results is shown in Fig. 2.

From Fig. 2 we can know that CSD are more effective than APDG and MOSS. CSD have obvious advantages for the detection of program 16 to 22, at the same time, CSD can detect program 8,program 21,program 27 and program 29 while APDG can't.

Although similarity ratio APDG detected is higher than CSD's, but program 28 changed the semantics of the program, the data flow is no longer relevant. So we believe that the program pair is no longer similar.

Fig. 3 shows that CSD's detection effect is better than the APDG's and MOSS's for all kinds of code variations. Meanwhile the result indicates that the graph-based approach has a good effect on the first four classes of code variations.

There is a problem of function splitting in program 29, the result CSD detected is listed in table III. Since the similarity ratio which testing functions pair was compared is less than threshold γ, we will add those function pair into function splitting set. According to the algorithm described in section III can be known that the main function may be split into function charToint and function factorial.

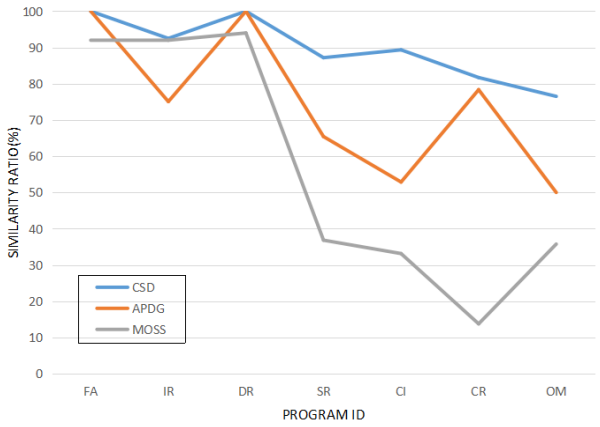

Figure 3. Similarity ratio of each class of program.

E. Efficiency of CSD

In this section, we evaluate the efficiency of CSD with 173 functions from open source flex and less. In the previous sections, since we evaluated the effectiveness of CSD, we just detect two identical program with CSD and CSD without software static analysis. Table IV shows the detection result.

Table IV reveals that adding static analysis approach reduces the number of comparison functions, thereby reducing the cost of comparison PDG, while the effectiveness is consistent with the traditional graph-based approach .

V. RELATED WORK

Dup[21] was developed in 1992, and since then a lot of code similarity detection approaches have been proposed. Based on the level of source code, similar code detection approaches can be mainly divided into five categories: text-based, token-based, tree-based, graph-based and metrics-based. In these methods, the graph-based approach has a better detection effect, but it also has the problem of large time and space cost and splitting function.

APDG[20] is also a code similarity detection tool based on PDG. Niklas divided the work into three parts: textual matching, the generation and analysis of abstract syntax trees and the generation and analysis of program dependence graphs. Then the tool prunes PDG match pairs by looking at the number of nodes and frequencies of the node-types in the graph. In the end, sub-graph isomorphism matching is performed. The full output by the APDG is the match-matrix, similarity ratio and time information.

TABLE III. DETECTION OF PROGRAM 29

Testing function pair	**similarity ratio**	**Program dependence graph isomorphism ratio**
(charToint,main')	0.5	1.0
(factorial,main')	0.09	1.0
(main,main')	1.0	×

TABLE IV. EXPERIMENT RESULT OF CSD'S EFFICIENCY

Tool	**Number of lines of code**	**Tested function pairs**	**Matches**	**Time(s)**
CSD (without static analysis)	7848	16179	173	180
CSD	7848	5531	173	69

GPLAG[7] is a plagiarism detection tool which is based on program dependence graph. GPLAG takes as input an original program and a plagiarism suspect, and output a set of similar PDG pairs. First, PDGs of two programs are generated. Then PDGs smaller than the given threshold are excluded by lossless filter and lossy filter. Finally, each PDG belongs to the plagiarism suspect has γ -isomorphism testing with PDG belongs to the original program and obtain the set of similar PDG pairs. However, the splitting function may confuse GPLAG.

CMGA[10] is also a graph-based tool that detects semantic similar code by combining software metrics and program dependence graphs. First, CMGA generates the augmented system dependence graph and transforms the control dependence graph into the control dependence tree. At the same time, the basic code normalization and the advanced code normalization are performed on the augmented system dependence graph to reduce false negatives. Then the comparison units are extracted by control dependence tree and similarity measure. Finally CMGA find the similarity of the modules through the similarity of the control dependence tree. When the function call depth is large, the control dependence tree may be relatively large. That may cause a serious problem of efficiency.

VI. CONCLUSION

In this paper, we develop a code similarity detection tool, called CSD, which uses software static analysis and program dependence graph combined approach to detect code similarity. Experiment results show that our approach can detect all kinds of variations in the code and has a high efficiency.

We expect to find more effective metrics and to solve the problem of more complex function splitting.

REFERENCES

[1] Ducasse S, Rieger M, Demeyer S. A Language Independent Approach for Detecting Duplicated Code[C].IEEE International Conference on Software Maintenance. IEEE Computer Society, 1999:109-118.

[2] Baker B S. On finding duplication and near-duplication in large software systems[C].Reverse Engineering, 1995. Proceedings of, Working Conference on. IEEE, 1995:86-95.

[3] Kamiya T, Kusumoto S, Inoue K. CCFinder: A Multilinguistic Token-Based Code Clone Detection System for Large Scale Source Code. IEEE Trans Softw Eng[J]. IEEE Transactions on Software Engineering, 2002, 28(7):654-670.

[4] Baxter I D, Yahin A, Moura L, et al. Clone detection using abstract syntax trees[C].Icsm. IEEE, 1998:368-377.

[5] Komondoor R, Horwitz S. Using Slicing to Identify Duplication in Source Code[M].Static Analysis. Springer Berlin Heidelberg, 2003:40-56.

[6] Jens Krinke. Identifying Similar Code with Program Dependence Graphs[C].Conference on Reverse Engineering. IEEE, 2001:301-309.

[7] Liu C, Chen C, Han J, et al. GPLAG: detection of software plagiarism by program dependence graph analysis[C].ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. ACM, 2006:872--881.

[8] Kontogiannis K A, Demori R, Merlo E, et al. Pattern matching for clone and concept detection[J]. Automated Software Engineering, 1996, 3(1-2):77-108.

[9] Gabel M, Jiang L, Su Z. Scalable detection of semantic clones[C].2008:321-330.

[10] Wang T, Wang K, Su X, et al. Detection of semantically similar code[J]. Frontiers of Computer Science Selected Publications from Chinese Universities, 2014, 8(6):996-1011.

[11] Higo Y, Yasushi U, Nishino M, et al. Incremental Code Clone Detection: A PDG-based Approach[C].Working Conference on Reverse Engineering. IEEE, 2011:3-12.

[12] Sarkar M, Chudamani S, Roy S, et al. A Hybrid Clone Detection Technique for Estimation of Resource Requirements of a Job[C]. Third International Conference on Advanced Computing & Communication Technologies. IEEE Computer Society, 2013:174-181.

[13] Higo Y, Kusumoto S. Code Clone Detection on Specialized PDGs with Heuristics[C].European Conference on Software Maintenance and Reengineering. IEEE Computer Society, 2011:75-84.

[14] Joy M, Luck M. Plagiarism in Programming Assignments[M]. University of Warwick, 1998.

[15] Jones B E L. Metrics based plagiarism monitoring[C].Ccsc Northeastern Conference. 2010:1--8.

[16] [16].Mccabe T J. A complexity measure[C].International Conference on Software Engineering. IEEE Computer Society Press, 1976:308-320.

[17] Myers G J. An extension to the cyclomatic measure of program complexity[J]. Acm Sigplan Notices, 1977, 12(10):61-64.

[18] Harrold M J, Malloy B, Rothermel G. Efficient Construction of Program Dependence Graphs[J]. Acm Sigsoft Software Engineering Notes, 2000, 18(3):160-170.

[19] Aho A V, Sethi R, Ullman J D. Compilers: Princiles, Techniques, and Tools.[M]. 1986.

[20] Holma N. Program Dependence Graph Generation and Analysis for Source Code Plagiarism Detection[J]. Institute of Technology, 2013.

[21] Baker B S. A Program for Identifying Duplicated Code[J]. Computing Science & Statistics, 1992.