International Journal of Hybrid Information Technology
Vol.8, No.11 (2015), pp.139-150
http://dx.doi.org/10.14257/ijhit.2015.8.11.11

Efficient Metric Vector-Based Code Clone Detection Using Function-calling Tree

Wei Li[1], Dongmei Li[1*], Chengjing Qiu[2] and Jiajia Hou[3]

[1]*School of Information Science and Technology, Beijing Forestry University, Beijing 100083, China*
[2]*School of Computer Science and Information Technology, Northeast Normal University, Changchun 130000, China*
[3]*School of Information, Renmin University of China, Beijing100872, China*
lidongmei@bjfu.edu.cn

Abstract

Most traditional code clone detections have less accurate results because they ignore the structure of the program itself, and some of them really think about it by creating a complex syntax tree but leading to a high time complexity. Confronting such situation, this paper proposes an efficient metric vector-based code clone detection method using function-calling tree. Considering the two program code to be detected, feature vectors in all defined functions of the two different code are extracted first. Then, two function-calling trees are created according to the function-calling process and node matches each other between two trees, at the same time, the matching similarities are calculated. Finally, by using the bottom-up approach and combining similarity values of all child nodes, the detection can get the similarity of the two program code to be detected. Our experiment selects a set of typical code sample to measure and the results demonstrate that, compared the famous JPlag system, it shows better detection effect.

Keywords: *code clone, similarity calculating, feature vector, function-calling Tree*

1. Introduction

Code clone is a serious phenomenon in higher education at home and abroad at present, especially in programming courses of computer specialty, whose homework is mostly handed in the form of electronic version. Because of that, clone costs little, and people who copy often just slightly modify other people's code, instead of thorough understanding. As a result, it influences the normal teaching to some extent. Therefore, code clone detections have important significance to improving teaching effect, the protection for intellectual property rights and so on. At present, the authoritative definition of code clone has not been recognized. An idea is proposed in [1] that code clone is a code portion of source files that are identical or similar to another. Another is proposed in [2] that two programs are considered similar to each other if they satisfy some clone transform approaches. In general, code clone approaches [3] can be classified from three aspects, which are the equivalent transformation of code layout, syntax and the semantic. Jones [4] and Zhao[5] summarizes eleven kinds of clone approaches that often appear in similar code. According to the efforts people pay, they are listed as follows from easy to difficult: ① complete copy. ② recomposition. ③ modify annotations. ④ constant replacement. ⑤ identifier renaming. ⑥ redefine data type. ⑦ statement reordering in program block. ⑧ change the sequence of operator or operand in the expression. ⑨ increase redundancy variables. ⑩ replace control structure for equivalent control structure. ⑪ reorder program blocks.

These clone approaches are simple and easy to achieve, making no change in the structure and function of program, thus providing an important basis for the detection method of this paper.

Using C language as an example, this paper proposes an efficient code clone detection method based on feature vector and function-calling tree. The method takes function as a unit, and computes the similarity of functions from three aspects. Then according to function-calling relations and the similarity between functions, the similarity between programs can be calculated. On this basis, this paper develops a code clone detection tool based on feature vector and function-calling tree aiming at C language. By testing a certain number of typical clone sample set, the results demonstrate that the method proposed in this paper can effectively identify the eleven kinds of clone approaches, and comparing to the famous JPlag system, it shows better detection effect and has practical value in use.

2. Related Work

Code clone detection originates from foreign countries, mainly experiencing the development of two phases: detection method based on attribute counting and detection method based on structure measurement.

Detection method based on attribute counting [6-8] mainly extracts various attributes in the program, regardless of program's internal structure. In 1976, Ottenstein [9] proposes a method that detects code clone using attribute counting for the first time, as well as uses Halstead metrics to detect the code clone of Fortran program. Later, Grier [10] and Faidhi [11] increase the number of statistical attributes, and design accuse system. The technique of attribute counting has the advantage of high efficiency, but its drawback is obvious, that is, it can cause the loss of program's structure information, and it can't be able to accurately detect the changes in the structure of program. Besides, detection accuracy can't be improved simply by increasing the number of attributes.

Detection method based on structure measurement [12] measures similarity between two programs according to the structure of programs, which requires the analysis about internal structure of the program, such as control flow, nesting depth and data dependence relations, etc. Donaldson [13] early considers the sequence of statements in clone detection system, and characterizes the source program according to the appearance sequence of statements. In 1998, D. Baxter [14] proposes a code clone detection method based on abstract syntax tree, which conducts syntax parsing on codes using C language to generate complete syntax tree, and then compares the similarity degree of two syntax trees. Structure measurement technique has the advantage of high detection accuracy, but its drawback is obvious, that is, it is inefficient. For a syntax tree with N nodes, it needs a computation time of $O(N^3)$ to compare its subtrees one by one. In general, a line of program statements will averagely generate around ten nodes of abstract syntax tree, in that way the time complexity of subtree comparison is $O(L^4)$ (L represents the statement number of codes).

After that, many methods and tools appeared, such as methods based on case reasoning [15], neural network [16], optimizing compiler and disassembly [5], IDE [17] and so on. Currently, more popular online detection tools are MOSS from Stanford university [18], JPlag from University of the State of Baden-Wuerttemberg [19] and YAP from university of Sydney [20-21], *etc*.

Compared with previous methods [22], the detection method proposed in this paper is based on feature vector and function-calling tree. It focuses on various inside attributes of functions and function calling relations, which not only detects code clone from aspects of attribute counting and structure measurement, but also combines the information contained in both methods and generates two corresponding smaller function-calling trees aimed at two pieces of program code to be detected. Finally, we traverse the two trees using the bottom-up approach to map nodes, and then calculate the similarity of two pieces of code.

3. Method Overview

Currently, many measurement methods divide detection process into two stages: program format conversion and similarity determination [23]. This paper divides the whole code clone detection process into four parts: code standardization, generation of comparison units, node mapping and similarity calculation. First of all, input program segment should be standardized, which can slit code's word segmentation results according to the definition of function. Secondly, attribute information of functions and function-calling relations are extracted from segmentation results of function definition to generate function-calling tree. Then, taking the node of function-calling tree as comparison unit, this paper can establish corresponding relations between the nodes of two trees through node mapping, which means for each function calling in a program, the method will find the most similar function calling to it from programs to be compared and record its similarity. Finally, by using the bottom-up approach and combining similarity values of all child nodes, the detection can get the similarity of two pieces of programs. Specific code clone detection process is shown in Figure 1.

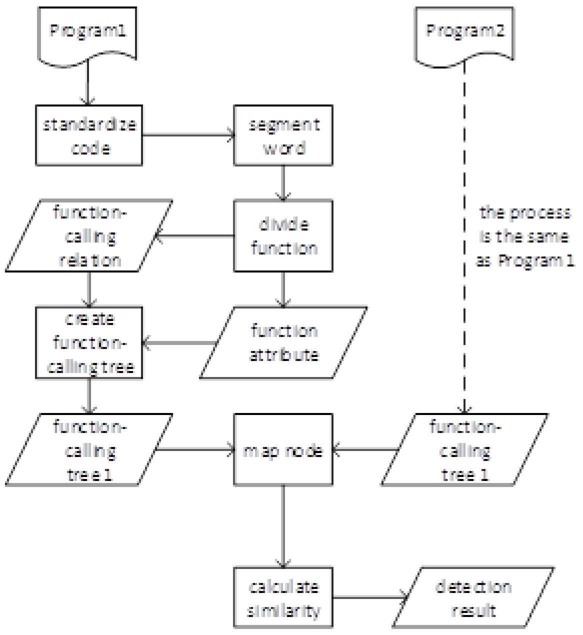


Figure 1. The Process of Code Clone Detection

4. Metric Vector-Based Code Clone Detection Using Function-calling Tree

4.1. Code Standardization

Code standardization can remove useless and interference information, improve the efficiency of the follow-up process and make code processing more accurate. Four rules are defined here for code standardization:

Rule 1: Remove the comments and white spaces not belonging to the output.

Rule2: Restore macro definition and redefine, such as that types defined by typedef will be restored to the original type.

Rule3: Replace constants types, such as that the const type constants will be replaced by constant values.

Rule4: Convert all the control structures like:

for (sentence1; sentence2; sentence3) sentence4

to control structures like:

{sentence1; while (sentence2){{ sentence4} sentence3;}}

Some clone based on text level clone could be filtered out via code standardization, while some common clone are not only based on text level, but also deep into the inner structure of codes.

4.2. Generation of Comparison Units

Different from those code clone detection based on syntax tree [14, 24], we use a kind of special function-calling tree as the middle program of code clone detection, in which the nodes of the tree are treated as the comparison units. In order to describe the generation of function-calling trees, related definitions are proposed firstly.

Definition 1(Function sets) Suppose F represents a function set. F is the sets of all the functions f defined in code P .

Definition 2(Function-calling sets) Suppose f_{call} represents function calling sets. f_{call} is the function sets that are called by function f and satisfying $f_{call} \in F$.

Definition 3(Feature vector) Suppose feature vector is represented by $\bar{v}$ and every dimension in it is d . $\bar{v}$ is a multidimensional vector, extracted from the definition of function f , consisting of some features of f . Information in $\bar{v}$ includes key words, operators, identifier, constants, arrays, structs, selection structure and loop structure. Each function's corresponding dimension of $\bar{v}$ is the same.

Definition 4(Tree node) Suppose tree nodes is N . N represents each function-calling in the program, including corresponding function f 's feature vector $\bar{v}$ and function-calling sets f_{call} .

We present the generation algorithm of functional-calling tree by using the above definitions, as it showed in Algorithm 1.

Algorithm 1. Generation of function-calling tree

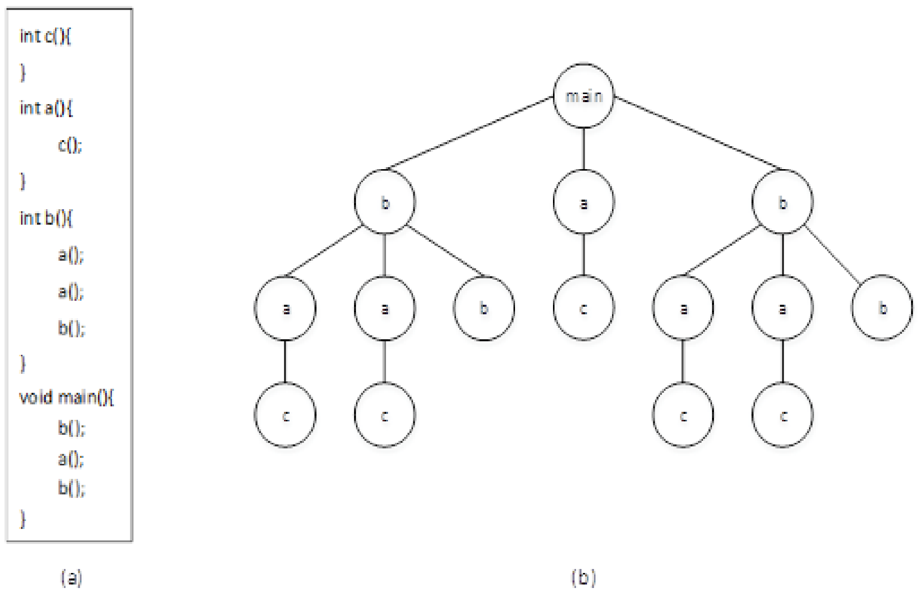
Step 1 Determine the root nodes. Corresponding functions of root nodes is the entry function of code, which is usually called main, or functions specialized by others, using root nodes as current nodes.

Step 2 Generate of child nodes. Check whether the function-calling tree of current nodes is void or not; if it is void, go to step 3, otherwise:

- If there are still unread functions in the function-calling sets, call the next function, and use it as the current node. Then go to step 2;(Recursive calls are handled specially).
- If all the function-calling sets have been read, go to step 3.

Step 3 Finish generating current nodes. All the current nodes and its child nodes have been generated; return the results.

For instance, function-calling tree's structure generated by the codes in Figure 2(a) is showed in Figure 2(b).



```
int c(){
}
int a(){
        c();
}
int b(){
        a();
        a();
        b();
}
void main(){
        b();
        a();
        b();
}
```

Figure 2. Samples of Function-calling Tree Generation

4.3. Node Mapping

Node mapping is the core of clone detection discussed in this paper. Two program code which is going to be detected generate two function-calling tree by the former two steps. Node mapping builds a corresponding relationship in nodes of two trees, namely that every function calling in a code finds out the most similar function calling in programs which are going to be compared.

Three factors, direction similarity, attribute similarity and depth similarity, are taken into consideration in calculation of node mapping. Suppose two nodes $N_1 \in T_1, N_2 \in T_2$, contain feature vector

$$\bar{v}_1 = (d_{11}, d_{12}, \cdots, d_{1n}), \bar{v}_2 = (d_{21}, d_{22}, \cdots, d_{2n})$$

The detail calculations of the three factors, direction similarity, attribute similarity and depth similarity, are decribed as followed.

4.3.1. Direction Similarity: Owing to the directional property of the vector, similarity can be judged according to the direction of multidimensional vectors in coordinate system. Suppose the similarity of feature vector $\bar{v}_1$, $\bar{v}_2$ is $s1$, then:

$$s1 = \cos\theta = \frac{\bar{v}_1 \times \bar{v}_2}{|\bar{v}_1| \times |\bar{v}_2|} \tag{1}$$

The smaller the angle between $\bar{v}_1$ and $\bar{v}_2$ is, the closer the direction is and the more similarity is; the lower the vice. Because that every value of dimension of $\bar{v}_1$ and $\bar{v}_2$ is larger than or equal to 0, so $s1 \in [0,1]$.

4.3.2. Attribute Similarity: Attribute similarity is improved by Manhattan Distance based on vector, recording the practical difference degree between two programs. Suppose the attribute similarity of feature vector $\bar{v}_1, \bar{v}_2$ is $s2$, and the definition of $s2$ is defined as followed:

$$s2 = 1 - \frac{\sum_{i=1}^{n} \left| d_{1i} - d_{2i} \right|}{\sum_{i=1}^{n} \max(d_{1i}, d_{2i})} \tag{2}$$

$d_{1i} - d_{2i}$ is the distance of $\bar{v}_1$ and $\bar{v}_2$ in dimension i. We use the absolute value of $d_{1i} - d_{2i}$ as the total attribute difference of $\bar{v}_1$ and $\bar{v}_2$ in all dimensions. $\max(d_{1i}, d_{2i})$ is the max attribute difference between $\bar{v}_1$ and $\bar{v}_2$ in dimension i. Calculate the sum of all the attribute difference of $\bar{v}_1$ and $\bar{v}_2$ in all dimensions. Use the practical attribute difference divide the max attribute difference, getting difference proportion. The smaller the difference proportion is, the higher the similarity is, so use one minus the difference proportion, getting attribute similarity $s2 \in [0,1]$.

4.3.3. Depth Similarity: Depth similarity is the similarity between different depth function callings. Generally speaking, the smaller the depth of function-calling tree nodes, the more important the function calling of corresponding nodes in origin program is, and with increasing in the depth of the nodes increasing, the importance would decrease gradually.

Suppose depth similarity of N_1 (in T_1) and N_2 (in T_2 ,) is h_1 and h_2 respectively, we designate the depth similarity of h_1 and h_2 by $s3$. Then the definition of $s3$ is:

$$s3 = \begin{cases} 1- \left| \dfrac{\ln h_1 - \ln h_2}{\max(\ln h_1, \ln h_2)} \right|, h_1 \neq h_2 \\ 1, \qquad h_1 = h_2 \end{cases} \tag{3}$$

When h_1 is not equal to h_2, the natural logarithm of depth is used to calculate the similarity; when h_1 is equal to h_2, the depth similarity is 1. Thus, $s3 \in [0,1]$.

When h_1 and h_2 changes, the value of $s3$ is showed in Figure 3. It could be known that when h is equal to 1 and any value except 1 have no similarity, which means depth similarity of root nodes and other nodes is 0, and with the increasing of depth, the influence of depth difference on depth similarity is decreasing, consistent with actual situation.

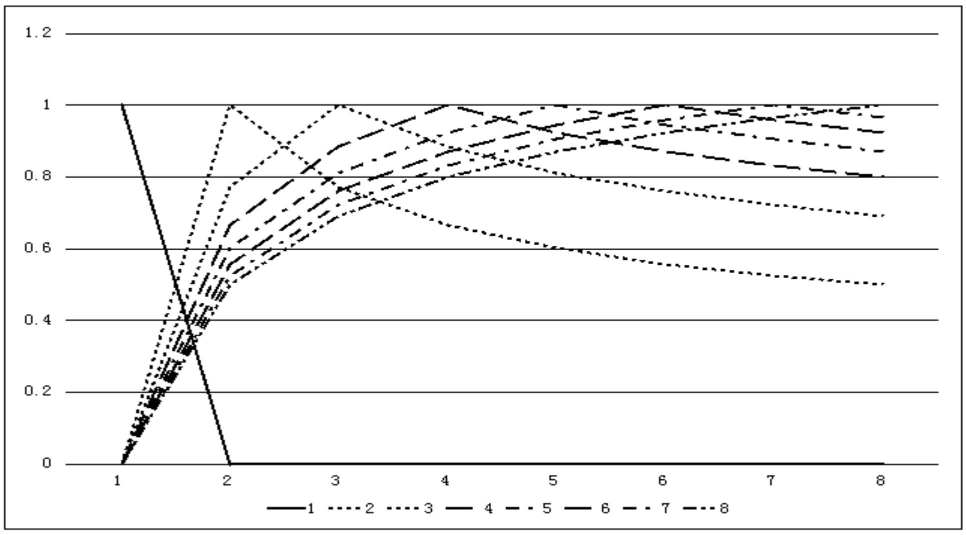


Figure 3. Changing Curves of Similarity of Depth

Using formula(1), (2) and (3), $s1$, $s2$ and $s3$ could be calculated. The formula of calculating similarity s of N_1 and N_2 is:

$$s = s1 \times s2 \times s3 \tag{4}$$

Using above calculating progress, the algorithm of tree T_1 's node mapping is showed in Algorithm2 as followed, and it is the same with T_2 .

Algorithm 2. Node mapping

Step 1 Traverse T_1 . If there are not-traverse nodes in T_1 , traverse the next node, and execute step2; otherwise, finish mapping; the end of the algorithm.

Step 2 Traverse T_2 .If there are not-traverse nodes in T_2 , traverse the next node, and execute step3; otherwise, go to step1.

Step 3 Calculate similarity. Use formula(4) to calculate the similarity s' of T_1 and T_2 in step2; if similarity $s < s'$, then $s = s'$, and record the serial number of nodes in T_2 . Return to step2.

4.4. Similarity Calculating

Similarity of nodes and mapping nodes have part and whole relationships with the two codes. Hence, the similarities of nodes and mapping nodes could be combined together to get the similarity of the whole tree. Method used in this paper is calculating from the leave nodes, adopting the method from the bottom to the top. Related similarity's definition is proposed here.

Definition 5 (Related similarity) Related similarity is the similarity of one code corresponds of another code, changing with the comparison code.

Suppose there are codes P_1 , P_2 , P_3 , generating function-calling trees T_1 , T_2 , T_3 . Mapping results between T_1 and T_2 are different from mapping nodes that are between T_1 and T_3 . In the first situation, the results of similarity combination between nodes and mapping in T_1 is noted as similarity of T_1 corresponding to T_2 , the same with the second situation.

Suppose TC is the child tree, TC_N is a tree whose root node is N and S_{TC} is the relative similarity of TC . Then:

$$S_{TC_N} = \sum_{i=1}^{n} (S_{TC_{NC}} - 0.5) \times \frac{len_i \times times}{len_a} + \lambda s \tag{5}$$

where $S_{TC_{NC}}$ is the relative similarity of the child tree whose root nodes is from N 's child nodes NC , which is the similarity of leave nodes and mapping nodes at the beginning. When NC is leave nodes, the value of $S_{TC_{NC}}$ lies in $[0,1]$, so the value of $(S_{TC_{NC}} - 0.5)$ is in $[-0.5, 0.5]$. At this time, 0 is the boundary point, and when $S_{TC_{NC}}$ is less than 0.5, it will bring negative gain, and in vice, it will bring positive gain for the whole. The value of $S_{TC_{NC}}$ is in $[-0.5, \lambda+0.5]$ after calculating. Changing the range of $S_{TC_{NC}}$ into [0,1] by using formula(6) is needed every time after calculating $S_{TC_{NC}}$.

$$S_{TC_N} = \frac{S_{TC_N} + 0.5}{\lambda + 1} \tag{6}$$

In formula (5), times is the $times$ of nodes N calling the same child nodes NC . When child nodes is recursive calling, through experiment testing, accurate results can be get by supposing the value of times set as 2 could get accurate results. len_i is the length of words after child nodes' participles. len_a is the word length of the origin program. λ is the adjustment factor when multiple child trees combining with the similarity of father nodes and s is the similarity of N and the mapping node of N .

Calculation of relative similarity starts from leave nodes, from the bottom to the top and repeat the above calculation progress, until $TC = T$, getting relative similarity of T . Finally, getting the similarity of T_1 and T_2 is get by using calculating the average:

$$S_{T_1T_2} = \frac{S_{T_1} + S_{T_2}}{2} \tag{7}$$

5. Experiment

At present, more popular code clone detection tools include JPlag, MOSS, YAP3, etc. JPlag is better than MOSS in the aspect of detection effect, and its performance is superior to YAP3 at the same time [25]. Therefore, this paper selects JPlag as the representation to compare detection results, including two groups of different conditions.

The first group is conducted in an ideal condition. We separate the eleven kinds of clone approaches described above, and only use an approach for clone at a time. The average length of programs is 180 lines, while the modified parts account for less than 25% of the original programs. The experimental results are shown in Figure 4.

Experimental results of the first group indicate that the method described in this paper and JPlag both get good detection effects using clone approaches labeled as ①②③④⑤ which only involve text level. However, for clone approaches labeled as ⑥⑦⑧⑨⑩⑪, which deepen into program structure level, JPlag shows poor performance due to changes in program structure to a certain extent, while our method is able to accurately identify the clone approaches because our method deeply takes the internal structure of programs into consideration.

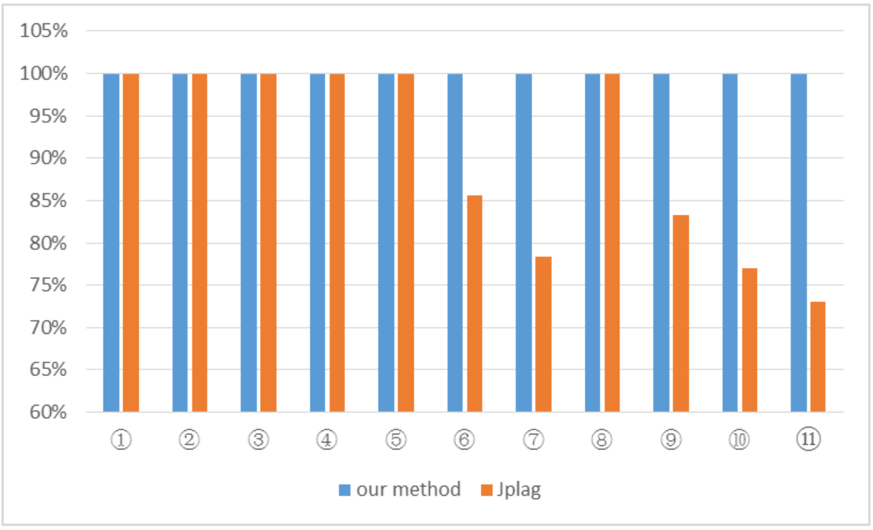


Figure 4. Comparison between Eleven Clone Approaches

The second group uses benchmark provided by http://www.sei.buaa.edu.cn/buaasim to detect code combining clone approaches. The benchmark dataset is divided into three groups, and programs being cloned are respectively cross-ref00.c, cross-ref10.c and cross-ref20.c. The first group's benchmark dataset is simulated according to eleven kinds of clone approaches in this paper, while benchmark dataset of this group derives from the clones after thoughtful thinking, using advanced clone approaches as far as possible, which has practical significance. The comparison results as shown in Figure 5-7.

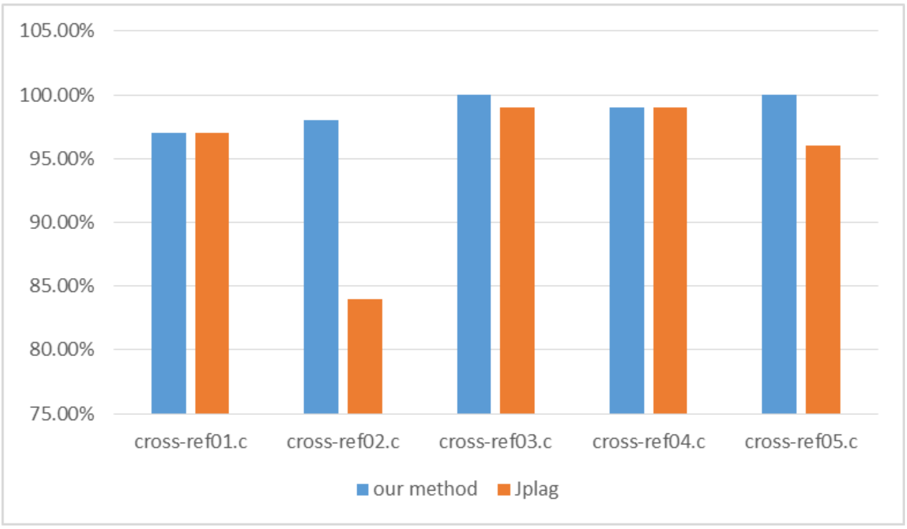


Figure 5. Comparison with Cross-ref00.c

Experimental results of the second group indicate that when the clones use unknown and advanced clone approaches as much as possible, detection results of the method described in this paper can always maintain a higher accuracy, while JPlag gets larger errors in a few contrasts. This indicates our method shows excellent performance in the practical application.

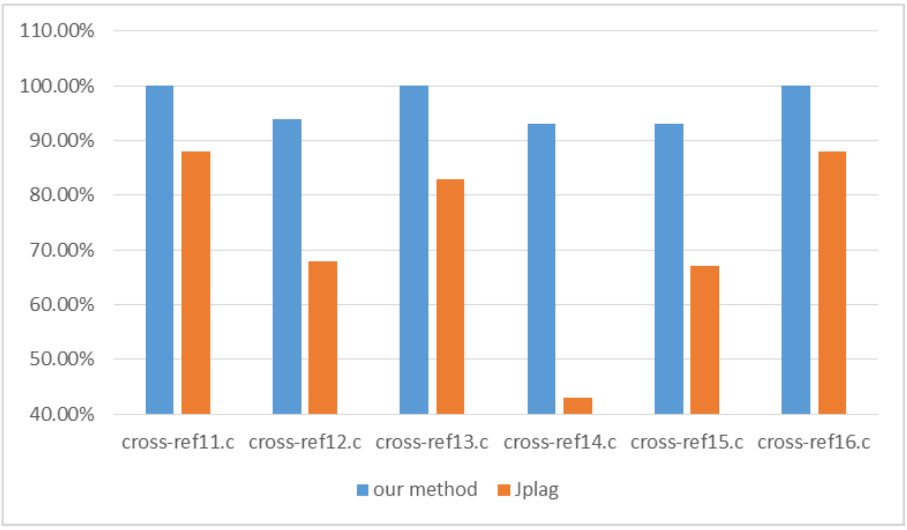


Figure 6. Comparison with Cross-ref10.c

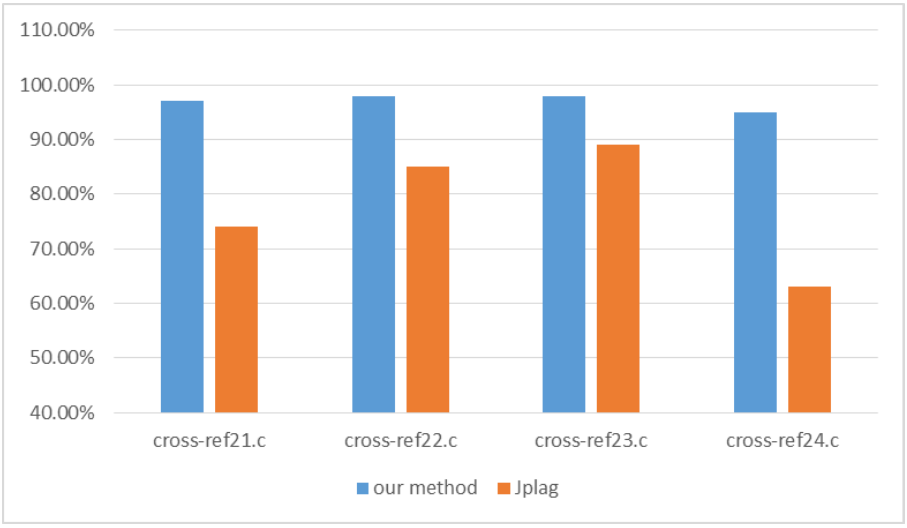


Figure 7. Comparison with Cross-ref20.c

6. Conclusion

Compared with traditional methods, code clone detection method described in this paper considers both properties and structure of programs, which makes clone detection results more reliable. Without considering overall semantic of programs and syntax tree established completely, detection accuracy is worse than that of detection methods based on syntax tree in some cases, but traditional detection methods based on syntax tree are not suitable for large-scale code clone detection because of high cost in detection.

Our method is only tested in C language, but it can be extended to C++, Java and other advanced languages. In addition, due to much information stored in function definitions, information contained in extracted feature vectors is not equal to that contained in original function definitions. In the process of two function-calling trees node mapping, it is critical to decide the value of λ when adding parent node with child node. This paper only gets optimal value from detection results, and we can future research how to decide the value of λ in detail in different tree structure in order to further improve the accuracy of detection.

Acknowledgments

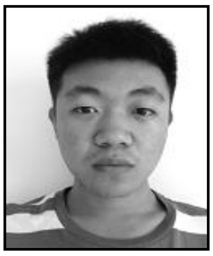
This work is supported by Beijing Higher Education Reform (No. 2013-ms047), Beijing Forestry University Resources Sharing Course (Data Structure), Beijing Undergraduate Training Programs for Innovation and Entrepreneurship(No. S201410022070), and Beijing Forestry University Special Research of Campus Informatization (Construction of the Teaching Resources Sharing Platform for Program Design and Algorithms Courses).

References

[1] T. Kamiya, S. Kusumoto and K. Inoue, "CCFinder: A Multi-Linguistic Token Based Code Clone Detection System for Large Scale Source Code", IEEE Transactions on Software Engineering, vol. 44, no. 8, **(2002)**, pp. 654-670.
[2] L. Chao, C. Chen and H. Jia-Wei, "GPLAG: DetecTionof Software Plagiarism by Program Dependence GraphAnalysis", Proceedings of ACM SIGKDD, **(2006)**, pp. 872-881.
[3] X. Hao, Y. Haihua and G. Tao, "Code Similarity Detection: A Survey", Computer Science, vol. 37, no. 8, **(2010)**, pp. 9-14.
[4] E. L. Jones, "Metrics based plagiarism monitoring", Proceedings of the 6th Annual CCSC Northeastern Conference on the Journal of Computing in SmallColleges, vol. 16, no. 4, **(2001)**, pp. 253-261.

[5] Z. Changhai, Y. Haihua and J. Maozhong, "Approach based on compiling optimization and disassembling to detect program similarity", Journal of Beijing University of Aeronautics and Astronautics, vol. 34, no. 6, **(2008)**, pp. 711-715.
[6] K. J. Ottenstein, "An Algorithmic Approach to the Detection and Prevention of Plagiarism", SIGCSEBulletin, vol. 8, no. 4, **(1976)**, pp. 30-41.
[7] H. M. Halstead, "Elements of Software Science", Elsevier, **(1977)**.
[8] H. L. Berghel and D. L. Sallaeh, "Measurements of program similarity in identical task environments", Sigplan Notices, vol. 19, **(1984)**, pp. 65-76.
[9] K. J. Otixnstein, "An algorithmic approach to the detection and prevention of plagiarism", ACM SIGSCE Bulletin, vol. 8, no. 4, **(1976)**, pp. 30-41.
[10] S. Grier, "A tool that detects plagiarism in Pascal programs", ACM SIGCSE Bulletin. ACM, vol. 13, no. 1, **(1981)**, pp. 15-20.
[11] J. A. W. Faidhi and S. K. Robinson, "An empirical approach for detecting program similarity and plagiarism within a university programming environment", Computers & Education, vol. 11, no. 1, **(1987)**, pp. 11-19.
[12] D. McCabe, "Levels of Cheating and Plagiarism Remain High", Center for Academic Integrity, Duke University, **(2005)**.
[13] J. L. Donaldson, M. Lancaster and P. H. Sposato, "A plagiarism detection system", ACM SIGCSE Bulletin. ACM, vol. 13, no. 1, **(1981)**, pp. 21-25.
[14] I. D. Baxter, A. Yahin and L. Moura, "Clone detection using abstract syntax trees", Software Maintenance, Proceedings, International Conference on. IEEE, **(1998)**, pp. 368-377.
[15] P. Clough, "Plagiarism in natural and programming languages: an overview of current tools and technologies", Research Memoranda: CS-00-05, Department of Computer Science, University of Sheffield, UK, **(2000)**, pp. 1-31.
[16] S. Engels, V. Lakshmanan and M. Craig, "Plagiarism detection using feature-based neural networks", ACM SIGCSE Bulletin. ACM, vol. 39, no. 1, **(2007)**, pp. 4-38.
[17] M. F. Zibran and C. K. Roy, "Towards flexible code clone detection, management, and refactoring in IDE", Proceedings of the 5th International Workshop on Software Clones, ACM, **(2011)**, pp. 75-76.
[18] A. Aiken, "Moss (measure of software similarity) plagiarism detection system", **(2000)**.
[19] L. Prechelt, G. Malpohl and M. Philippsen, "Finding plagiarisms among a set of programs with JPlag", Journal of Universal Computer Science, vol. 8, no. 11, **(2002)**, pp. 1016-1038.
[20] M. J. Wise, "Detection of similarities in student program: YAP'ing may be preferable to Plague'ing", Proceedings of 23th SIGCSE Technical Symposium, Kansas City, USA, **(1992)**, pp. 268-271.
[21] M. J. Wise, "YAP3: Improved Detection of Similarities in Computer Program and other Texts", ACM SIGCSE Bulletin. ACM, vol. 96, **(l996)**, pp. 130-134.
[22] C. K. Roy, J. R. Cordy and R. Koschke, "Comparison and evaluation of code clone detection techniques and tools: A qualitative approach", Science of Computer Programming, vol. 74, no. 7, **(2009)**, pp. 470-495.
[23] G. Whale, "Plague: plagiarism detection using program structure", Dept. of Computer Science Technical Report 8805, University of NSW, Kensington, Australasian, **(1988)**.
[24] H. Kikuchi, T. Goto and M. Wakatsuki, "A source code plagiarism detecting method using alignment with abstract syntax tree elements", Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD), 15th IEEE/ACIS International Conference on. IEEE, **(2014)**, pp. 1-6.
[25] L. Precheh, G. Malpohl and M. Philippsen, "Finding plagiarisms a-monga set of programs with JPlag", Journal of Universal Computer Science, vol. 8, no. 11, **(2002)**, pp. 1016—1038.

Authors



Wei Li, he is an undergraduate student in School of Information and Technology, Beijing Forestry University. He joined Institute of Artificial Intelligence in Beijing Forestry University in 2013 as a research assistant. His main research interests include clone analysis and software reconstruction.

Dongmei Li, she received the M.S. degree from Institute of Software, Chinese Academy of Sciences and the Ph.D. degree from Beijing Jiaotong University. Currently she is an associate professor in School of Information and Technology, Beijing Forestry University. Her main research interests include artificial intelligent, knowledge engineering and semantic web.

Chengjing Qiu, she received the B.S. degree from Northeast Normal University. Her research interests include software engineer and artificial intelligence.

Jiajia Hou, she received the bachelor degree from Beijing forestry University. Currently she is a master student in School of Information, Renmin University of China. Her main research interests include intelligent information processing and semantic web.