A Scalable and Accurate Approach Based on Count Matrix for Detecting Code Clones

Yang Yuan

Department of Computer Science
Peking University, Beijing, China
yangyuan@pku.edu.cn

Abstract

In this paper, we introduce a new token based algorithm for code clone detection. Count Environment(CE) is certain scenario related to variables. Count Vector(CV) for one variable is consisted of counting occurrences of this variable in different CEs. Count Matrix(CM) for one code fragment is consisted of different CVs of all variables in the code fragment. We use CVs to depict variables, and use CM to represent a code fragment. Two code fragments will be compared by their corresponding CMs, and during the comparison, two heuristics are used. Experimental results show that our algorithm is significantly faster than Deckard, a state-of-the-art syntactic technique for detecting code clones.

Categories and Subject Descriptors Software [*SOFTWARE ENGINEERING*]: Distribution, Maintenance, and Enhancement

General Terms Algorithm, Experimentation

Keywords Code Clone, Count Matrix, Token Based

1. Introduction

Code clone are those code fragments made by copy-paste operations during software development, which might be a little different from the original copy. According to Jürgens et al. [2], code clones are harmful to software systems, because they usually make the source code more intricate and increase the maintenance cost. According to previous researchers [5], a significant fraction of large software system is cloned. So code clone detection is an important problem.

To evaluate a code clone detection algorithm, two aspects are important: scalability and accuracy. In this paper, we propose a new token-based algorithm, which uses count matrix to represent code fragments, and produces satisfying results on the test data of JDK 7 source files.

2. Background

Previous code clone techniques can be divided into four categories according to the level of analysis to source code: textual, token-based, syntactic, and semantic. Among them, Deckard [1], CP-Miner[3] and GPLAG[4] represent the state-of-the-art. However, previous token-based algorithms usually focus on token sequence rather than variables, and when positions of some tokens are changed, they might fail to find the clones. Meanwhile, those high level techniques, both syntactic and semantic, need a lot of time to analysis the source code, and are difficult to migrate to other programming languages.

We have described a preliminary version of this algorithm called CMCD and some conducted experiments about it in an early paper [6]. CMCD is based on Soot, and uses bipartite graph matching to compare two code fragments.

3. Count Matrix

For computers, the function or the structure of the source code are difficult to understand, we choose to analyze the variables instead. Count Environment(CE) is used for describing patterns of variables. Generally, CEs are related to how and where each variable is used. For example, whether the variable appears in a if-predicate, or whether the variable is a parameter in a function-call. In terms of the required analysis of the source code, we can divide CEs into three stages. Stage1 only needs to match the tokens; Stage2 needs the information of the statement in which the corresponding variable appears; Stage3 is more aggressive, which needs to analyze many statements in a single count.

In our implementation, we use the following CEs:

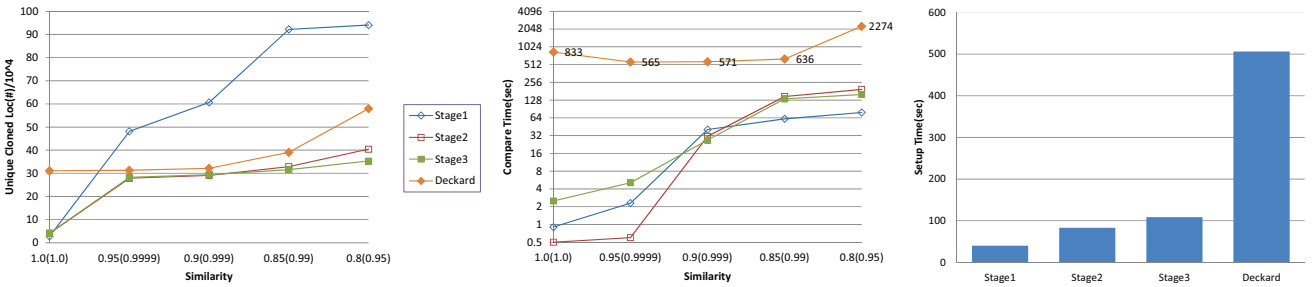
Stage 1:
- The variable is used
- The variable is defined

Stage 2:
- The variable is in an if-predicate
- The variable is added or subtracted
- The variable is multiplied or divided
- The variable is an array subscript
- The variable is defined by an expression with constants

Stage 3:
- The variable is in a first-level loop
- The variable is in a second-level loop
- The variable is in a third-level loop (or deeper)

Generally, more kinds of CEs can be added in, as long as they are representative and easy to be identified. We investigate each variable by counting its occurrences in each CE, and combining the results together to get a Count Vector(CV) for the variable. The length of CV is m, representing m CEs defined. In each code fragment, there are many variables, and each has a CV. By combining these CVs, we will get a Count Matrix(CM) for the code fragment, which has n rows and m columns, representing n variables and m CEs in the code fragment.

AOSD'12, March 25-30, 2012, Potsdam, Germany.
ACM 978-1-4503-1222-6/12/03.



(a) Cloned LoC (b) Comparing Time (c) Setup Time

Figure 1. Results for Boreas(using different settings) and Deckard on JDK 7

4. Comparison

When comparing two code fragments, we want to find the matching of the variables in them. If most variables can be properly matched, these two fragments might be clones.

Firstly, given two CVs a and b, we define the Cosine Vector Similarity(CVS) as cosine of the angle between them. If the CVS of two CVs is small, they are considered similar:

$$CosSim = \cos(\alpha) = \frac{a \cdot b}{||a||||b||} = \frac{\sum_{i=1}^{m} a_i \times b_i}{\sqrt{\sum_{i=1}^{m} a_i^2} \times \sqrt{\sum_{i=1}^{m} b_i^2}}$$

Secondly, we discuss the method to find matching between the variables. The bipartite graph matching algorithm is accurate, but too slow. Our solution is to sort the variables according to their total used times, and try to match each variable a of block A to those variables of block B whose ranks are close to the rank of a. Duplicated matches are allowed, that is, although every variable of block A must match exact one variable of block B, there are no such restrictions on the variables of block B. Moreover, we designed a heuristic called Quicksep, which skips comparing those pairs that are very different in terms of some general information, such as number of variables, number of lines, etc. These two heuristics greatly simplify and speed up our implementation.

5. Experiments

Our algorithm is implemented in C++, and is able to process C, C++ and Java language. In this section, we use the test data of JDK 7. The experiments were conducted with Core 2 Duo T9400 and 6 GB RAM on Ubuntu 11.04.

We use 3 versions of our algorithm: Stage1 only uses CEs in first stage; Stage2 only uses CEs in first two stages; Stage3 uses CEs from all stages. We also compare our results with Deckard[1]. We set different similarity thresholds for these techniques (1.0, 0, 95, 0.9, 0.85, 0.8 for our algorithm, and 1.0, 0.9999, 0.999, 0.99, 0.95 for Deckard).

We investigate three aspects of these techniques: scalability, clone quantity, and clone quality. The running time of these techniques was split into setup time and comparing time in Figure 1(c) and Figure 1(b). The clone quantity is measured by counting the unique LoC of the result found by each technique, see Figure 1(a). And the clone quality is measured by the false positive rates, which are computed by randomly picking up 100 cloned pairs, and then checking the correctness of them manually, see Table 1. These results show that our algorithm is significantly faster than Deckard, and the results are as good as Deckard.

[1] http://www.mysmu.edu/faculty/lxjiang/research.html

Table 1. False Positive Rates

Simi	Stage1	Stage2	Stage3	Deckard
1.00 (1.00)	5%	0%	0%	0%
0.95(0.9999)	65%	0%	0%	0%
0.90 (0.999)	72%	1%	1%	0%
0.85 (0.99)	67%	9%	4%	0%
0.80 (0.95)	82%	33%	19%	51%

6. Conclusion

We introduce a new algorithm for code clone detection in this paper. The main idea is to construct Count Matrix for code fragments based on token sequence, and compare two code fragments by comparing their CMs. Experimental results show that our algorithm is both scalable and accurate.

References

[1] L. Jiang, G. Misherghi, Z. Su, and S. Glondu. DECKARD: Scalable and accurate tree-based detection of code clones. In *ICSE*, pages 96–105. IEEE Computer Society, 2007.

[2] E. Jürgens, F. Deissenboeck, B. Hummel, and S. Wagner. Do code clones matter? In *ICSE*, pages 485–495. IEEE, 2009.

[3] Z. Li, S. Lu, S. Myagmar, and Y. Zhou. CP-miner: Finding copy-paste and related bugs in large-scale software code. *IEEE Trans. Software Eng*, 32(3):176–192, 2006.

[4] C. Liu, C. Chen, J. Han, and P. S. Yu. GPLAG: Detection of software plagiarism by program dependence graph analysis, 2006.

[5] C. K. Roy, J. R. Cordy, and R. Koschke. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Sci. Comput. Program*, 74(7):470–495, 2009.

[6] Y. Yuan and Y. Guo. CMCD: Count Matrix based Code Clone Detection. In *Proceedings of the 18th Asia-Pacific Software Engineering Conference (APSEC 2011), to appear*, 2011.