Boreas: An Accurate and Scalable Token-Based Approach to Code Clone Detection

Yang Yuan and Yao Guo
Key Lab of High-Confidence Software Technologies (Ministry of Education)
Department of Computer Science, School of EECS, Peking University
Beijing 100871, P. R. China
{yangyuan, yaoguo}@pku.edu.cn

ABSTRACT

Detecting code clones in a program has many applications in software engineering and other related fields. In this paper, we present Boreas, an accurate and scalable token-based approach for code clone detection. Boreas introduces a novel counting-based method to define the characteristic matrices, which are able to describe the program segments distinctly and effectively for the purpose of clone detection. We conducted experiments on JDK 7 and Linux kernel 2.6.38.6 source code. Experimental results show that Boreas is able to match the detecting accuracy of a recently proposed syntactic-based tool Deckard, with the execution time reduced by more than an order of magnitude.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging

General Terms

Algorithms, Experimentation

Keywords

Code clone detection, count vector, count matrix

1. INTRODUCTION

In software development, programmers frequently reuse code fragments by copy-paste operations. Those code fragments, which are similar or identical, are called *code clones*. Code clones could bring many problems to the software systems [4]. For example, if many cloned instances exist in a software system and a bug was found in the cloned code, one need to find and fix all of them. It would produce unpredictable results if inconsistent modifications are made to these clones. According to [10], a significant part of large software system source code is cloned, typically ranging from 7%-23%. If these code clones could be efficiently and accurately detected, the problems they brought might be easily solved or at least properly controlled.

ASE'12, September 3–7, 2012, Essen, Germany

Many code clone detection approaches have been proposed in the literature. Generally, they can be classified into four categories: textual approaches [9], token-based approaches [5,7], syntactic approaches [1,3,11] and semantic approaches [6, 8]. This classification is made according to the level of analysis applied to the source code.

The traditional belief is that more complicated approaches would have better understanding about the structure of the programs, so that they can identify code clones more accurately. Although syntactic approaches could produce better results compared to previous token-based work in general, these high level approaches have many shortcomings. On one hand, syntactic and semantic approaches normally require that the source code are syntactically correct or even compilable, which makes them inapplicable to incomplete code. On the other hand, high level approaches usually demand more computation and storage resources.

Token-based approaches are inherently language-independent and low-cost. They work faster because they only need to transform the source code into tokens, without the need to construct ASTs or PDGs. They are more language-independent compared to higher-level approaches as they are much easier to migrate to other languages. They also need less resources because they process low level data, which in turn makes them more scalable to large-scale software systems. However, previous token-based approaches, which are mostly based on the sequence of tokens and with variable names ignored, have their own limitations. Because they are too focused on tokens, they could easily lose the big picture, thus typically they cannot detect those clones with swapped lines or added/removed tokens.

In this paper, we propose a new token-based code clone detection approach called Boreas, which introduces a novel counting-based characteristic matrix definition to overcome the shortcomings of traditional token-based techniques.

2. OVERVIEW

The key of a code clone detection approach is to generate precise abstractions for each code fragment. After abstraction, code fragments can be compared efficiently, using a variety of comparing or clustering techniques.

As a token-based approach, Boreas matches the variables, rather than matching sequences or structures. Using this idea, the similarity of two code segments is decided by the proportion of variables that could be matched based on their characteristics.

We introduce the notion of *Counting Environments*(CE), and use these CEs to describe the patterns of variables. A

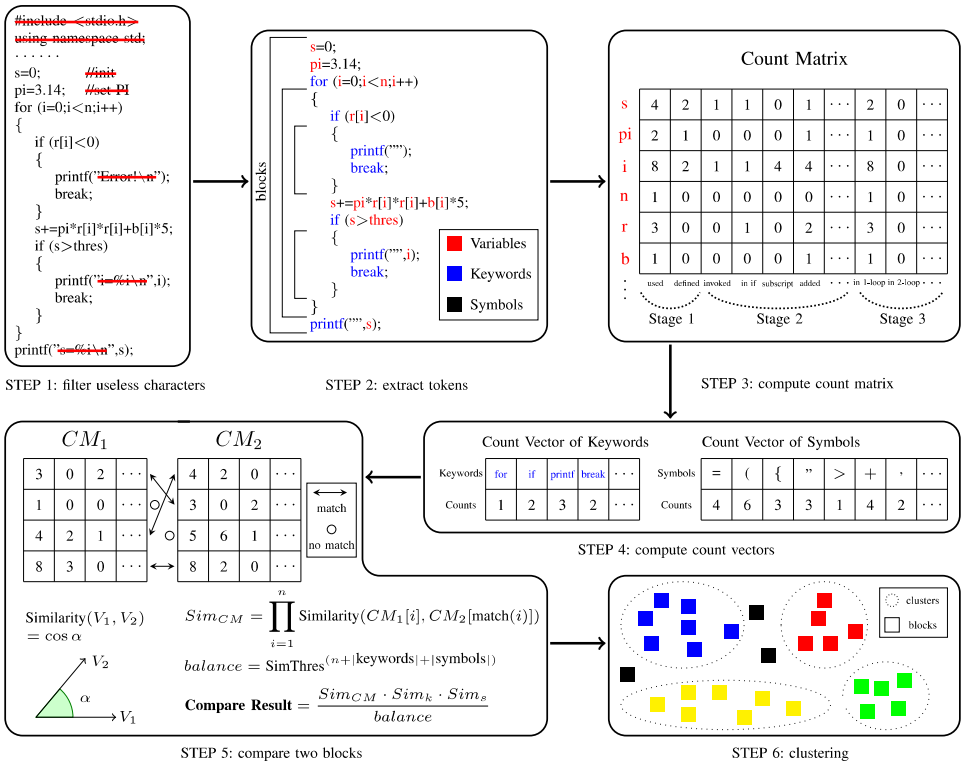


Figure 1: An overview of Boreas

count vector(CV) for one variable is a combination of occurrence counts in different CEs. Since each CV represents one variable in the code fragment, by combining all the CVs together, we build a *count matrix* (CM), which forms a comprehensive description of the code fragment.

We choose variable occurrences for three reasons. First, it is easy to implement. Second, it does reflect the pattern of variables: the description of being in one CE for many times and never appear in another CE, does help us identify the corresponding variable. Finally, which is the most interesting and important, counting-based representation has high tolerance to minor modifications, making Boreas effective on type-3 clones [2], including clones made by swapping lines or adding/removing tokens, where most existing code clone detection approaches were unable to detect effectively.

The overall process of Boreas can be divided into six steps, as shown in Figure 1.

3. COUNT MATRIX

3.1 Counting Environments

Counting Environments (CEs) are used for describing patterns of variables, and play a key role in Boreas. We divide the CEs into three stages, while each stage provides a more concrete and distinct description for the chosen variables.

The first stage, *Naïve Counting*, includes the used and defined environments. The CEs of this stage are easy to discover, because we only need to find the variables in the code fragments, with little analysis. The variables that have an "=" (or "+=", "-=", etc.) token right after them are treated as being defined.

A slightly higher stage, *In-statement Counting*, includes CEs that should be identified using information from the statements in which the variable appears. For example, if the statement starts with "if", we know that this is a if-statement, and the if-environment count of the corresponding variable will be increased by 1.

The third stage, *Inter-statement Counting*, involves some environments that need the information of multiple statements to identify. A typical example is the nested loop-level of variable. That is, to decide the variable is in a first-level loop, a second-level loop, or a deeper level loop.

In summary, the following CEs are used in this paper:

Naïve Counting Stage:
The variable is used
The variable is defined
In-statement Counting Stage:
The variable is in an if-predicates
The variable is added or subtracted
The variable is multiplied or divided
The variable is an array subscript
The variable is defined by expression with constants
Inter-statement Counting Stage:
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)

To clarify, we do use some syntactic techniques while identifying CEs. However, the simple analysis involves only basic analysis without the construction of ASTs, so Boreas still belongs to the category of token-based techniques.

3.2 Count Vector and Count Matrix

Using m CEs, we can generate an m-dimensional Count Vector(CV) for each variable, where the i-th dimension of

the CV is the occurrence count of the variable in the i-th CE. So the CV is a characteristic vector for the variable. We notice that variables are not easy to distinguish only by names. To avoid syntactic analysis, we simply treat all the variables with the same name as the same variable in Boreas. According to our experimental results, variables with the same name usually have similar functions, thus this simplification is generally acceptable.

After CVs are computed, we combine all n CVs in a code fragement (with n variables), and obtain a $n \times m$ Count Matrix(CM). CM is a abstraction for the code fragments, and we compare two code fragments by comparing their CMs.

In order to better characterize a code segment, Boreas also generates CVs for both keywords and punctuations, which uses similar methods as variables. The details are omitted due to space limitations.

4. COMPARISON

During comparision, we use Cosine similarity to compare the CVs for variables, while using a modified proportional similarity function to compare the CVs for keywords and punctuations.

4.1 Cosine Similarity Function

Cosine similarity is a measure of similarity between two vectors by measuring the cosine of the angle between them.[1] The cosine of 0 is 1, and less than 1 for any other angles. The cosine of the angle between two vectors thus determines whether two vectors are pointing to roughly the same direction. Cosine Similarity is a perfect choice for comparing Count Vectors, because CVs represent the patterns of variables, and the closeness of two variables can be approximated by the cosine of vectors in high dimensional spaces.

For two vectors a and b with the angle α between them, their cosine similarity is defined as

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

4.2 Proportional Similarity Function

Boreas uses an improved proportional similarity function to to compare the CVs of keywords and punctuation marks. The function is modified to prevent incorrect zero similarity. Given two occurrence counts a and $b(a >= b)$, their proportional similarity is defined as

$$ProSim = \frac{1}{a+1} + \frac{b}{a+1}.$$

4.3 Calculating Similarity

The similarity of two blocks is the product of the similarity of their CMs, and the similarity of their CVs of the keyword and punctuation marks (as computed using the above proportional similarity function).

We sort the variables according to their used frequencies, and then 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. This greatly simplifies the implementation and computation of the comparison: we only need to search a small range of variables for each variable of block A, and pick up the most similar one as the similarity value for each variable, and then compute the product of these similarities.

[1] `http://en.wikipedia.org/wiki/Cosine_similarity`

5. EXPERIMENTAL RESULTS

In this section, we evaluate Boreas in terms of three aspects: scalability, clone quantity and clone quality.

We have implemented Boreas in C++, and use the same scanner generated by lex to process both Java and C/C++ code. It can be easily migrated to other languages. We use the source code of Java SE Development Kit 7 (7,492 files, 2,260,946 LoC) and Linux kernel 2.6.38.6 (35,856 files, 10,068,963 LoC) as the test data. Due to limited space, we mainly present the results on JDK here. The experiments were conducted with Core 2 Duo CPU T9400 and 6GB DDR3 RAM on Ubuntu 11.04.

We use three different stages of CEs in the experiments: Naïve Counting Stage, In-statement Counting State and Inter-statement Counting Stage. Higher level stage also includes CEs of the lower level stages. These three stages may also be combined CVs of keywords and punctuation marks (represented as "+key"). However, Naïve Counting Stage has comparatively poor performance, so we only keep the Naïve Counting Stage without CVs of keywords and punctuation marks, which is the simplest version of Boreas. Hence, we evaluated a total of five versions of Boreas.

In the experiments, we use Deckard 1.2.1 for comparison. The parameters of Deckard were set as $mint = 50, stride = 2$, which are the default parameter settings of Deckard and also the settings used in the paper [3].

5.1 Execution Time

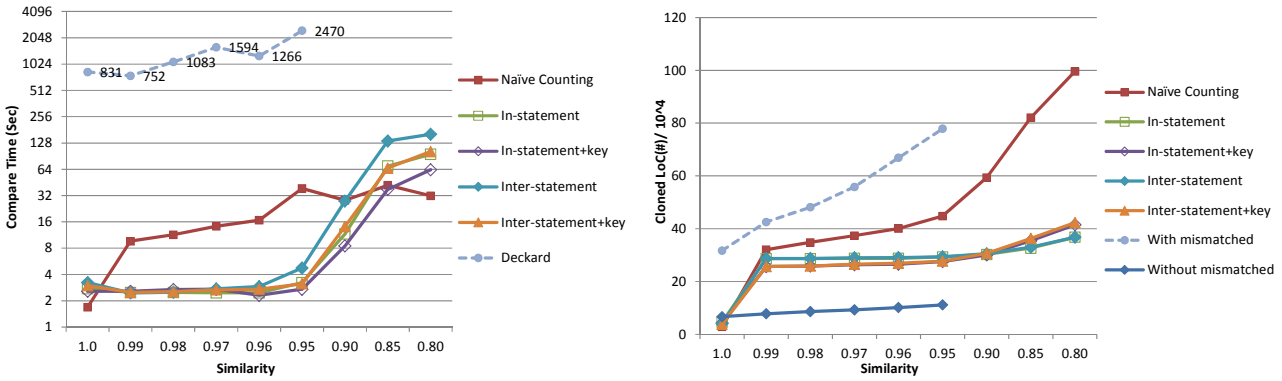
We investigate the scalability of Boreas in terms of its running time and space requirements. The running time of Boreas is split into two parts: setup time and comparison time. The setup time is the time consumed by Boreas to perform lexical analysis, and compute CMs and CVs; the comparison time is the time consumed by comparison. Deckard also has vector generation period and vector clustering period, so we compare them correspondingly.

Comparison time is shown in Figure 2(a) (the vertical scale is logarithmic). As expected, Boreas is much faster than Deckard. The setup times are in general stable for different versions of Boreas. Deckard uses different parsers for different languages (which is not language-independent), and the parser for Java is very slow. The setup time of Deckard is about 1-5 times more than the setup time of Boreas, because it requires to construct ASTs, which is much more time consuming.

As another indication of scalability, we compare the space requirements of Boreas and Deckard. The source code of Linux kernel is only 401 MB, but Deckard requires 5 GB temporary files, which is much more than the requirement of Boreas (249 MB).

5.2 Clone Quantity

We measure the clone quantity by counting the number of lines of code (LoC) within the detected cloned code fragments. We choose to count the Unique Cloned LoC, that is, each line of the code will be counted for only once, no matter how many times they appear in the identified clones. In this paper, "LoC" always represents unique cloned LoC.



(a)Comparison time on JDK

(b)Cloned LoC on JDK

Figure 2: Comparison time and cloned LoC of Boreas (using different settings) and Deckard on JDK

Moreover, we found a large proportion of the clones identified by Deckard is trivial, including self-clones and import or package clones. Here, self clones refers to two code fragments having a large shared fragment, and they are reported as clones because Deckard found that their shared fragment is "similar". Import or package clones are those clones consisting of only "import" and "package" statements (mainly in JDK). Since Boreas does not include these kinds of clones, we perform a post-process to remove all the self-clones and import or package clones from the results of Deckard.

As mentioned before, since Boreas and Deckard use different granularities, Deckard produces clones with mismatched brackets, which Boreas does not produce. On both JDK and Linux, Deckard finds more clone lines than Boreas when counting mismatched clones. However, when mismatched clones are removed, Boreas can find more clone lines than Deckard in JDK, while the two methods are very close in Linux. Results are shown in Figure 2(b).

5.3 Clone Quality

Clone quality is an important metric, but it is very difficult to measure the false positive rates automatically. Instead, we use manual inspection to detect false positives. For each set of results, we randomly picked 100 cloned pairs and inspected them manually. Due to the small set of samples inspected, this false positive rate might not be very accurate, but it could still reflect the clone quality of the corresponding technique to some degree.

According to our results, Naïve Counting has the highest false positive rate; and the false positive rates of different versions will grow when *similarity* decreases. The results show that Boreas is able to maintain a relative low false positive rate (<5%) when similarity is as low as 0.95 (0.90 for JDK), while the false positive rate of Deckard reaches above 10% when similarity is lower than 0.96. Thus when comparing the numbers of cloned lines in the previous subsections, we also need to consider the false positive rates at different similarity settings.

6. CONCLUSION

In this paper, we propose a new approach called Boreas for detecting code clone clusters. Boreas introduces a novel counting-based characteristic matrix to represent the patterns of variables. With two fast similarity functions, Boreas is able to perform faster clone detection than previous approaches. The experimental results show that Boreas is able to match the clone detection capability of by one of the state-of-the-art approach Deckard with much faster execution time.

7. ACKNOWLEDGEMENTS

This work is supported in part by the National Basic Research Program of China under Grant No. 2009CB320703, the Science Fund for Creative Research Groups of China under Grant No. 60821003, National High-Tech R&D Program under Grant No. 2011AA01A202, and the National Natural Science Foundation of China under Grant No.61103026.

We also want to thank Prof. Lingxiao Jiang for helping us with the Deckard software.

8. REFERENCES

[1] I. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier. Clone detection using abstract syntax trees. In *ICSM*, 1998.

[2] M. Gabel, L. Jiang, and Z. Su. Scalable detection of semantic clones. In *ICSE 2008*, pages 321–330, 2008.

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

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

[5] T. Kamiya, S. Kusumoto, and K. Inoue. CCFinder: A multilinguistic token-based code clone detection system for large scale source code. *IEEE Trans. Software Eng*, 28(7):654–670, 2002.

[6] J. Krinke. Identifying similar code with program dependence graphs. In *WCRE*, pages 301–309, 2001.

[7] 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.

[8] C. Liu, C. Chen, J. Han, and P. S. Yu. Gplag: detection of software plagiarism by program dependence graph analysis. In *KDD '06*, pages 872–881, 2006.

[9] C. K. Roy and J. R. Cordy. NICAD: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization. In *ICPC*, pages 172–181, 2008.

[10] 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.

[11] Y. Yuan and Y. Guo. CMCD: Count Matrix based Code Clone Detection. In *APSEC*, pages 250–257, 2011.