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. Recent studies show that syntactic-based clone detection methods could achieve better accuracy compared to token-based approaches. However, token-based methods have their advantages because they are inherently language-independent and highly scalable.

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 detection tool Deckard, with the execution time reduced by more than an order of magnitude.

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 [11]. 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 [25], 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.

Many code clone detection approaches have been proposed in the literature. Generally, they can be classified into four categories: textual approaches, token-based approaches, syntactic approaches and semantic approaches. This classification is made according to the level of analysis applied to the source code. For example, textual approaches [23] usually apply little transformation on the source code, and the comparisons are made based on the raw source code. Meanwhile, semantic approaches [15] use program dependency graphs (PDGs) to represent the program, and the clones are detected by finding isomorphic subgraphs of corresponding PDGs, which is an NP-complete problem.

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. For example, they need to construct abstract syntax trees (ASTs) or PDGs, and perform other compiling-related tasks. During this process, many intermediate data need to be recorded for comparison.

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. The key idea of Boreas is to collect characterizing information of each variable, and represent them as Count Vectors(CVs). For every code segment, variable information, such as how and where they are used, reflects the features of the code segment. Each CV records only counting information without variable names, such that we could easily deal with variable renaming clones. However, we differentiate variables with different names in the CVs collected, such that we could provide a more accurate description of the program segment

compared to previous approaches that treat all identifiers as the same. By combining all the CVs, Boreas composes a Count Matrix(CM) to represent the characteristics of a code fragment. In addition to variables, Boreas also generates CVs for both keywords and punctuations to better characterize a code segment.

In order to compare the CMs and CVs effectively, Boreas employs two different functions to calculate the similarities between two code segments. Several optimizations are also introduced during comparison to further speed up the detecting process. As a language-independent approach, we perform experiments on JDK and Linux source code, and compare Boreas to a state-of-the-art approach Deckard. Results show that Boreas is able to match the clone detection abilities of Deckard, while reducing the comparison time by an order of magnitude.

This paper is organized as follows. In Section 2, we introduce the background and related work of code clone and previous techniques. In Section 3, we give an overview of Boreas. The notion of Count Vector and Count Matrix is introduced in Section 4, and we describe two similarity functions and comparison method in Section 5. After that, we introduce clone clustering techniques in Section 6. The experimental results are presented in Section 7, and we conclude with Section 8.

2. BACKGROUND AND RELATED WORK

2.1 Code Clones

Previous researchers have classified code clones into four types based on textual and functional similarities [5, 8, 16]:

1. **Type-1**: Identical code fragments except for variations in whitespace, layout and comments.
2. **Type-2**: Syntactically identical fragments except for variations in identifiers, literals, types, whitespace, layout and comments.
3. **Type-3**: Copied fragments with further modifications such as changed, added or removed statements, in addition to variations in identifiers, literals, types, whitespace, layout and comments.
4. **Type-4**: Two or more code fragments that perform the same computation but are implemented by different syntactic variants.

Type-1 and Type-2 are relatively simple cases, while Type-3 is more difficult. A representative case of Type-3 is swapping two adjacent lines in a program. Many detection techniques cannot detect this scenario. For Type-4, it does not require the code fragments to have any similar code, but only the same computation, which seems out of the scope of most clone detection research. Actually, Type-4 clone detection problem is the same as proving the equivalence of two programs, which is fundamentally undecidable.

In the context of this paper, we consider code clones as code copied from the original copy, with or without minor modifications. If there are more than a few modifications, it becomes a derivation or innovation of the original, not a clone. Thus we consider only code clones of Type-1, Type-2, Type-3. Of course, Type-4 clones are inherently hard for most techniques as well. To the best of our knowledge, no existing approaches can detect Type-4 clones effectively.

2.2 Clone Detection Techniques

Most code clone detection techniques can be classified into four categories [25]:

(1) *Textual approaches* (or text-based techniques) use little or no transformation on the source code before the actual comparison, and in most cases raw source code is used directly in the clone detection process. Examples: SDD [19], NICAD [24], Simian[1], etc.

(2) *Lexical approaches* (or token-based techniques) begin by transforming the source code into a sequence of lexical "tokens" using compiler-style lexical analysis. The sequence is then scanned for duplicated subsequences of tokens and the corresponding original code is returned as clones. Lexical approaches are generally more robust over minor code changes such as formatting, spacing, and renaming than textual techniques. Examples: Dup [2, 3], CCFinder [13], CP-Miner [20], etc.

(3) *Syntactic approaches* use a parser to convert source programs into parse trees or abstract syntax trees which can then be processed using either tree matching or structural metrics to find clones. Examples: CloneDr [4], Deckard [10], CloneDigger [6], Semantic-web Based Technique [26], Intermediate Representation [27], Tree Kernel Based Technique [7], etc.

(4) *Semantics approaches* have also been proposed, using static program analysis to provide more precise information than simply syntactic similarity. In some approaches, the program is represented as a program dependency graph (PDG). The nodes of this graph represent expressions and statements, while the edges represent control and data dependencies. Examples: Duplix [17], GPLAG [21], Incremental PDG Based Technique [9],etc.

Many new techniques have been proposed recently, focusing on new applications such as instant search on large scale systems [14, 18]. The trend shows that it is important to investigate faster approaches that are able to scale to large code bases.

2.3 Counting-based Approaches

Counting is a great idea for code clone detection, because it makes the approach resistant to minor modifications of programs, and produces a characteristic vector for each code fragment, which simplifies the comparison process.

Deckard is one of the state-of-the-art tree-based algorithms proposed by Jiang et al. [10], which computes certain characteristic vectors to approximate structural information within ASTs and then uses locality sensitive hashing (LSH) to cluster similar vectors. It adopts the idea of counting, as the dimensions of the characteristic vectors are occurrence counts of the relevant nodes.

Boreas collects counting information based on token streams in a program, which is much more scalable and language-independent. Since we do not need ASTs to obtain syntactic information, Boreas is much faster than Deckard. Boreas uses different granularity from Deckard, so the results produced by them are different.

Our previous studies have shown that counting-based techniques could improve the accuracy of token-based clone detection and are effective in some applications such as detecting programming bugs and plagiarisms [28]. But it only

[1] `http://www.redhillconsulting.com.au/products/simian/`

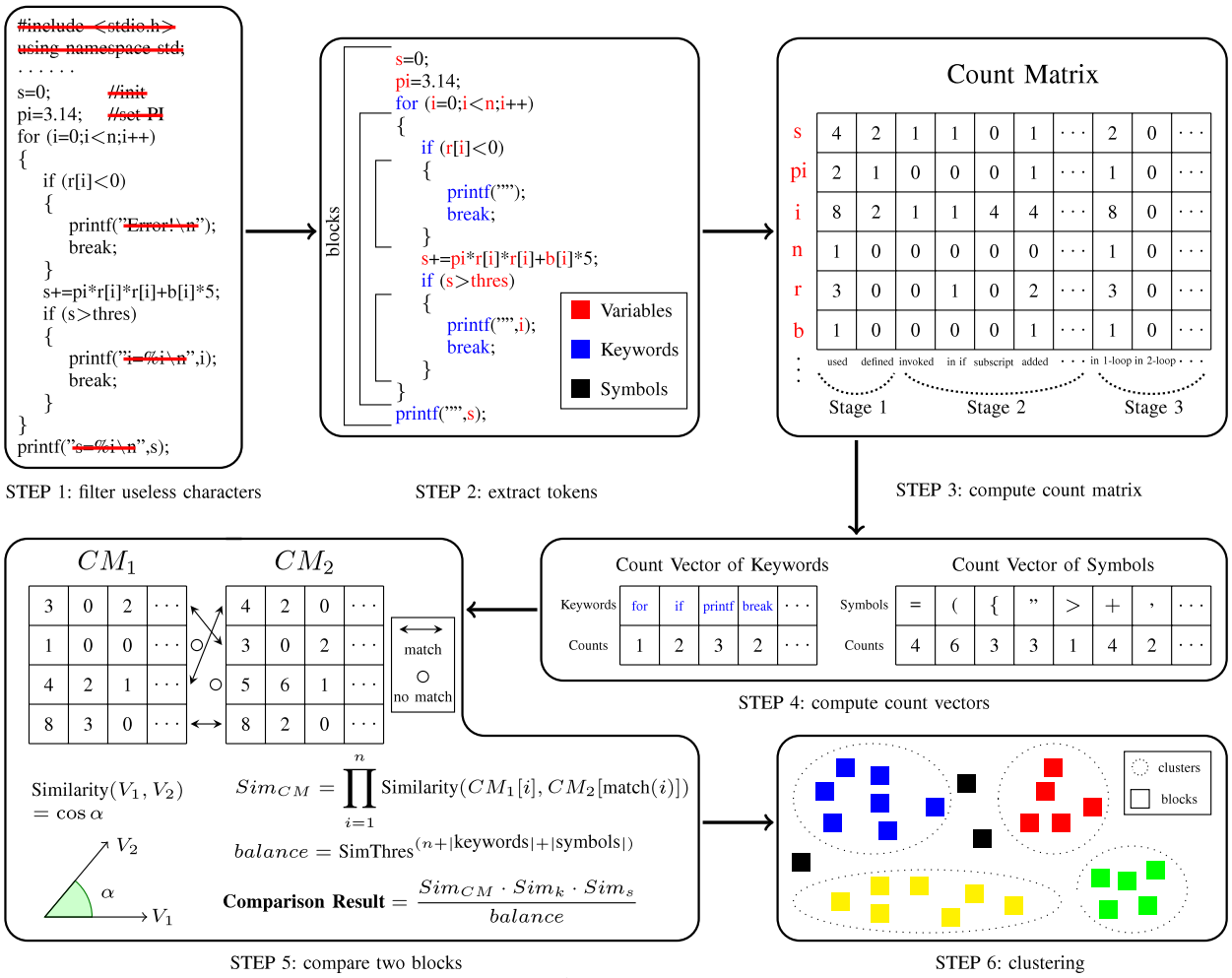


Figure 1: An overview of Boreas

applies to Java and uses bipartite graph matching algorithm to find clones, which is not scalable. Boreas adds a count vector for keywords and punctuations to improve accuracy, and uses two different similarity functions for comparison. Besides, Boreas introduces Quick Separation and clustering optimization to construct code clone clusters efficiently.

3. OVERVIEW

We first introduce the key idea behind the proposed approach, then present an overview architecture of Boreas.

3.1 Key Ideas

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.

The high level approaches are proposed because researchers want to dig into the internal structure or exact meaning of the source code, which limits the running speed. Although much faster than high level approaches, previous token-based approaches did not fully utilize the information obtained from the tokens. They usually focus on the structure of the token sequence, such as the longest common substring of sequences of two code fragments. However, the exact positions of tokens are not that important, as the effect of many programs will not change after swapping some statements. Moreover, since most token-based techniques erase the variable name information, the precision of description of the original code fragment, using the token sequence, is inherently limited.

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 *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 [8], including clones made by swapping lines

```
            doLog( Level.FINE, "UTIL.
                classCastExceptionInLoadStub",
                parameters, UtilSystemException.
                    class, exc ) ;
        }
        return exc ;
    }
    public BAD_OPERATION
        classCastExceptionInLoadStub(
        CompletionStatus cs ) {
        return classCastExceptionInLoadStub( cs,
            null ) ;
    }
    public BAD_OPERATION
        classCastExceptionInLoadStub( Throwable t
        ) {
        return classCastExceptionInLoadStub(
            CompletionStatus.COMPLETED_NO, t ) ;
```

Figure 2: A typical mismatched clone found by Deckard

or adding/removing tokens, where most existing code clone detection approaches were unable to detect effectively.

3.2 Overview of Boreas

The overall process of Boreas can be divided into six steps, as shown in Figure 1. we first remove the strings, declarations, and headers (Step 1), then extract blocks and statements and identify three different kinds of tokens: the variables, keywords and punctuations (Step 2). In step 3, a CV will be generated for each variable by counting the corresponding occurrences in three different stages. In step 4, CVs for keywords and punctuations are generated by counting their total occurrences in the code fragment. In step 5, CMs and CVs will be compared correspondingly, and the similarity between two code fragments will be computed. Finally, the blocks are merged into clone clusters.

3.3 Clone Granularity

Clone granularity is a very important feature for clone detection techniques. Many clone detection approaches [1, 22] choose natural program structures such as begin-end blocks as the clone granularity. Because this kind of granularity represents meaningful program structures, they are easy to identify and could have more potential applications.

Other approaches might choose different granularities, which could potentially produce much different clone results. For example, Deckard [10] uses fixed number of AST nodes as its granularity. ASTs help Deckard slice the programs into pieces, with which the sliding window algorithm can be applied. However, this kind of granularity could produce mismatched clones, such as the one shown in Figure 2. Although these kind of clones sometimes are useful in detecting bugs or other special purposes, they are not useful in other activities such as analyzing software systems, or detecting interesting aspects.

In order to produce meaningful clones that have wide applications, Boreas chooses code blocks separated by natural punctuation marks as the basic clone granularity. Larger granularities such as methods and classes can also be formed easily if necessary. The task of finding code fragments can be as simple as choosing blocks (or larger pieces such as methods and classes), which could be easily done during the token scanning process.

4. COUNT MATRIX

4.1 Classification of the Counting Environments

As mentioned before, Counting Environments (CEs) are used for describing patterns of variables, and play a key role in Boreas. We divide the CEs into three stages, with each stage providing 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. If we use CEs from this stage to construct Count Vectors for variables, two variables are matched if they have similar "used" and "defined" occurrence counts.

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. Other CEs include add-environment (the variable is added), subtract-environment (the variable is subtracted), call-environment (the variable is a parameter in a procedure-call), subscript-environment (the variable is an array subscript), etc. A statement might satisfy the condition of one or more CEs, and a variable can appear in different CEs simultaneously.

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 this case, we need to find out how many loops have included the variable.

In our implementation, we choose the following CEs from all the three stages:

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)

Although we only use CEs mentioned above in Boreas, other well-defined CEs can also be included if they could help represent the code segment more comprehensively.

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.

4.2 Count Matrix for Blocks

Using m CEs, we can generate an m-dimensional CV for each variable. But the 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

Algorithm 1 Merge CM of One Block with Its Subblock

```
1: function Merge(fa, son, loopLevel)
2:   AddKeyV(keyV[fa], keyV[son])
3:   for i = 1 to variableTotal[son] do
4:     pos = FindName(fa, variName[son][i])
5:     if pos = NULL then
6:       pos ← NewVariableID(fa)
7:     end if
8:     AddStage1&2(CM[fa][pos], CM[son][i])
9:     AddStage3(CM[fa][pos], CM[son][i], loopLevel)
10:  end for
11: end function
```

in Boreas. This might become a limitation of Boreas, because in a large block, two different variables in different sub-blocks may have the same name with different declaration statements in the corresponding blocks. However, according to our experimental results, variables with the same name usually have similar functions, thus this simplification is generally acceptable.

If a block contains n variables, the CM for the block will have n lines, which correspond to the CVs of these n variables. CMs can be calculated in $O(L + knm)$ time, where L is the code length of the block, and k is the number of sub-blocks in the block. During the computation of CM for a block, we compute the CMs of its sub-blocks and then merge all of them. The merge step is not only add those matrices together; CEs like loop-levels need to be re-computed during merging, because sub-blocks might be in a for-loop of the larger block. Besides, if there are variables declared inside the sub-block, the size of CM might increase after merging. See Algorithm 1 for details.

4.3 Count Vector for Keywords and Punctuations

Like variables, keywords and punctuations (such as '[', ']', '+', '*', etc) may also have their own CMs and CEs. However, there is a big difference: variables in different programs can have different names and the same functions, but keywords and punctuations cannot. In other words, the keywords and punctuations are easier to match during the comparison. Moreover, because it is unnecessary to design environments for keywords and punctuations, we only count the occurring times of keywords and punctuations, and produce a single Count Vector for them.

5. COMPARISON

Because there are two kinds of vectors: CVs for variables and CVs for keywords and punctuations, we choose different similarity functions to compare them.

5.1 Cosine Similarity Function

Cosine similarity is a measure of similarity between two vectors by measuring the cosine of the angle between them.[2] 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.

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

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

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

A drawback of the cosine similarity function is that if two vectors with different lengths pointing to the same direction, their cosine similarity will be 1. Boreas also multiplies the cosine similarity with the quotient of the length of the shorter vector dividing the length of the longer one to get the final similarity such that vectors with different lengths will have similarity less than 1.

5.2 Proportional Similarity Function

In order to compare the CVs of keywords and punctuations, we need to choose a different similarity function, because intuitively one dimension of the CV here, which represents a keyword or symbol, deserves more weight than one dimension of the CV in the CM, which represents only $1/m$ fraction of one variable.

Given two numbers, what is their similarity? The most natural idea is to use the smaller one to divide the bigger one. However, sometimes certain keyword occurs in one code fragment but not in the other, which means the corresponding pair of occurrence counts of this keyword will be a non-zero number and zero. In this case, the similarity will be zero, which is not acceptable, because after multiplication with other similarities, it will make the final similarity of two blocks become zero.

Boreas uses an improved proportional similarity function to prevent incorrect zero similarity. Given two occurrence counts a and $b (a >= b)$, their proportional similarity is defined as

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

If a equals to b, their $ProSim$ will be 1; if b equals to 0, their $ProSim$ will be $1/(a+1)$, which continually approaches zero as a grows (which means the difference between a and b grows).

5.3 Calculating Similarity

The similarity of two blocks is the product of the similarity of their CVs of the keyword and punctuations, and the similarity of their CMs. The similarity of CVs is computed using the proportional similarity function, while the similarity of CMs relates to the matching of the variables.

In our previous work, we use Bipartite Graph Matching to compute the similarity of CMs. This is a straightforward idea, because each variable of one block can be matched to only one variable of the other block, and we should pick the matching with minimum total matching cost (or maximum total matching similarity). However, the $O(n^3)$ time complexity of the matching algorithm is unacceptable. In this paper, we choose a fast and mostly accurate algorithm to make Boreas scalable.

The solution, described in Algorithm 2, is to 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:

Algorithm 2 Find Clones for Block A

```
1: function FindSimilar(a)
2:   for i = 1 to blockTotal do
3:     if (line[a] < miniLine) ⋁(line[i] < miniLine)
       ⋁(Abs(lines[a] − lines[i]) > lineDif)
       ⋁(Abs(variables[a] − variables[i]) > variDif)
       ⋁(QuickSep(a, i, variAttr, variSep)) then
4:       return
5:     end if
6:     dif ← PropDif(lines[a], lines[i])
7:     dif ← dif ∗ PropDif(varibles[a], variables[i])
8:     for j = 1 to keyTotal do
9:       dif ← dif ∗ PropDif(keyV[a][j], keyV[i][j])
10:    end for
11:    for j = 1 to variableTotal do
12:      best ← 0
13:      for k = j − range to j + range do
14:        if (validNum((k)) ⋀
           (VecDif(CM[a][j], CM[i][k]) > best) then
15:          best ← VecDif(CM[a][j], CM[i][k])
16:        end if
17:      end for
18:      dif = dif ∗ best / similarity
19:    end for
20:    if dif > stdThres then
21:      Clustering(a, i)
22:    end if
23:  end for
24: end function
```

Algorithm 3 Quick Separation for Two Blocks

```
1: function QuickSep(a, b, attr, minSep): boolean
2:   for i = 1 to attributeSize do
3:     if (Abs(attr[a][i] − attr[b][i]) > minSep[i][0])
       ⋁ (attr[a][i]/attr[b][i] > minSep[i][1])
       ⋁ (attr[b][i]/attr[a][i] > minSep[i][1]) then
4:       return true
5:     end if
6:   end for
7:   return false
8: end function
```

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. According to our experimental results, this simple algorithm is both fast and accurate.

5.4 Optimization: Quick Separation

Since we use count matrices instead of vectors to represent the blocks, many previous techniques for fast comparison or clustering can not be applied here. Meanwhile, quadratic pairwise comparisons are computationally expensive, so we introduce an optimization heuristic called **Quick Separation**.

The basic idea behind Quick Separation is that if two blocks are similar, most of their attributes will not differ much. Typically, these attributes include the total lines, the total number of variables, the maximum used times of variables, the total occurrence counts of keywords, etc. These attributes will not be affected much with minor modifications, and thus are typically stable. If two blocks are "very different" in one attribute, we will mark them as not similar, and finish comparison early.

The meaning of "very different" includes two aspects. First, the two numbers should not differ by a large constant (e.g. 15), such that 35 and 20 will be regarded as "very different". Second, the two numbers should not differ by a large ratio (for example 1.5), so 8 and 4 will be regarded as "very different". The algorithm is given in Algorithm 3.

Thus, for every attribute, there will be two thresholds: the constant and the ratio. These thresholds need to be set carefully, because if the gap is too large, few pairs are rejected and the heuristic has little effect. Meanwhile, if the gap is too small, some pairs that are actually similar would be rejected at the beginning.

In our experiments, the thresholds are pre-determined using a training phase. We first set a wide gap and then gradually shorten the gap as long as Quick Separation has very little effect on the final results. The pre-determined thresholds are trained with Java, however they are also proved effective for C in our experiments.

At first glance, Quick Separation does not improve the running speed of Boreas much, as each pair of blocks is still processed. However, Boreas improves it by sorting all the blocks according to the number of their variables. Since the number of variables is an important attribute used in Quick Separation, every block will only compare with those blocks located near it, which has similar number of variables. Thus, the running complexity becomes $O(tn)$, while t is a parameter related to the threshold of Quick Separation, which equals to the number of nearby blocks for each block in terms of total variables.

6. CLONE CLUSTERING

Similar to many other clone detection approaches, Boreas also use clone clusters to represent the clone detection results. In Boreas, we require that all clusters must be disjoint sets, because they have very efficient merge operations.

As a result, each cluster can appear in only one cluster. If one block is similar to blocks in two (or more) clusters, we will choose to either merge the two clusters or keep them separated in different clusters, according to the algorithms described below. Moreover, because the clustering process requires comparing the similarities of every pair of code blocks, we also want to improve the clustering cost by reducing the number of comparisons between blocks.

6.1 Clustering Algorithm

Boreas introduces a *cluster average* metric to describe the representative features of each cluster, which is the *average* value of a selected set of important attributes of all the blocks inside the cluster. Initially, each cluster has only one block in it, so the cluster average is represented as the attributes of this block. After merging two clusters, the cluster average of the new cluster will be computed as the weighted (based on their sizes) average of the two cluster averages.

Algorithm 4 shows the algorithms related to cluster averages. When we identify two similar code blocks, a and b, we will attempt to merge their respective clusters. The quick separation method described earlier is applied to compare their cluster averages. (Note that the separation thresholds used here are stricter than those used for block comparison.) If they can not be quickly separated, we will merge them, and calculate the new cluster average for the merged clus-

Algorithm 4 Block Clustering

```
1: function Clustering(a, b)
2:     a ←FindFather(a)
3:     b ←FindFather(b)
4:     if (a = b) ⋁
       (QuickSep(a, b, clustAvgAttr, clustSep)) then
5:         return
6:     end if
7:     father[b] ← a
8:     size[a] ← size[a] + size[b]
9:     AdjustClusterAvgAttribute(a, b)
10: end function
```

ter. Otherwise the two blocks will be kept in their respective clusters (i.e., they are not considered clones of each other).

6.2 Clustering Optimization: Merge Twice

Ideally, all the code blocks in a clone cluster are similar to each other. However, because similarity is not always transitive, two blocks similar to the same block could be dissimilar. Due to this constraint, many potential large clusters will be split into much smaller ones in practice. On the other hand, if we add a block to a cluster when the block is similar only to one block in the cluster, the final cluster would be very large, and contain essentially different blocks.

If we transform the cluster into a graph that each code fragment is represented as a vertex, and we link two vertices if the corresponding code fragments are clones, we can use *connectivity* in graph theory to represent the overall similarity in a clone cluster.

For a cluster, if the connectivity of its corresponding graph is k, we say this cluster is k-connected. If in the cluster, all n blocks are similar to each other, this cluster is n–1-connected; if there exists some block that has only one clone in the cluster, this cluster is considered as 1-connected. We could set the k value to acquire clone clusters with different connectivities. Thus, in order to add a block to a certain cluster, we only need to find k similar blocks.

Boreas introduces an optimization called *Merge Twice*, which essentially requires the connectivity be set to at least 2. During clustering, if current block has found two similar blocks, further comparisons for this block will not be conducted. In this way, the clusters will be at least 2-connected, but in practice the connectivities of the final clusters are much larger than 2. Experimental results show that this optimization could reduce the clustering cost significantly, without affecting the clustering results.

7. EXPERIMENTAL RESULTS

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

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. The experiments were conducted with Core 2 Duo CPU T9400 and 6GB DDR3 RAM on Ubuntu 11.04.

In our implementation, we set $range = 5, miniLine = 5$ (see Algorithm 2). That means, each variable in the Count Matrix will be compared with 11 variables (from $j - 5$ to $j+5$), and only those blocks with at least 5 lines will be compared. For Count Vectors of keywords and punctuations, we construct a 34 dimension vector, including all representative keywords and punctuations.

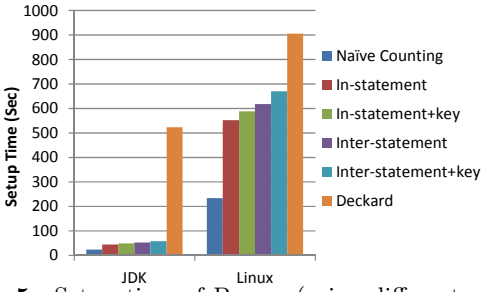


Figure 5: Setup time of Boreas (using different settings) and Deckard on JDK and the Linux kernel

Table 1: Space requirements

	JDK	Linux
Boreas	52MB	249MB
Deckard	378MB	5120MB

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 punctuations (represented as "+key"). However, Naïve Counting Stage has comparatively poor performance, so we only keep the Naïve Counting Stage without CVs of keywords and punctuations, which is the simplest version of Boreas. Hence, we evaluated a total of five versions of Boreas.

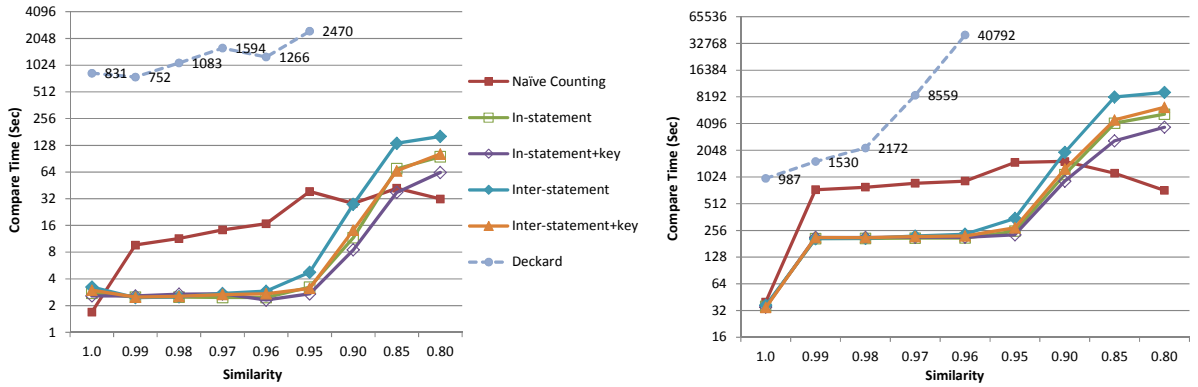
In the experiments, we use Deckard 1.2.1 for comparison. The parameters of Deckard were set as $mint = 50, stride = 2$, which are the default parameter settings of Deckard and also the settings used in the paper [10]. We notice that Deckard is sensitive to the parameter *stride*; different *stride*s may produce very different answers. However, it is extremely time-consuming to attempt all possible *stride*s, and will also bring much more false positives, so we only compare with the default parameter settings of Deckard.

Deckard uses a different similarity function from Boreas, which makes precise comparison impossible. According to our observation, the proper similarity range for Deckard is $[0.95, 1.0]$, because both false positive rate and running time of Deckard become unacceptable when the similarity is below 0.95. For Boreas, similarity can be set to as low as 0.8. Thus, the similarities of two approaches can not be directly compared. Although we represent experimental results based on similarities, we want to put emphasis on the trends of different aspects of the two approaches.

Many other recent clone detection techniques have shown satisfactory results, such as CC-Finder [12], CP-Miner [20] and CloneDR [4]. However, since Deckard has compared with the previous techniques including CloneDR and CP-Miner, we only compare Boreas to Deckard in this paper.

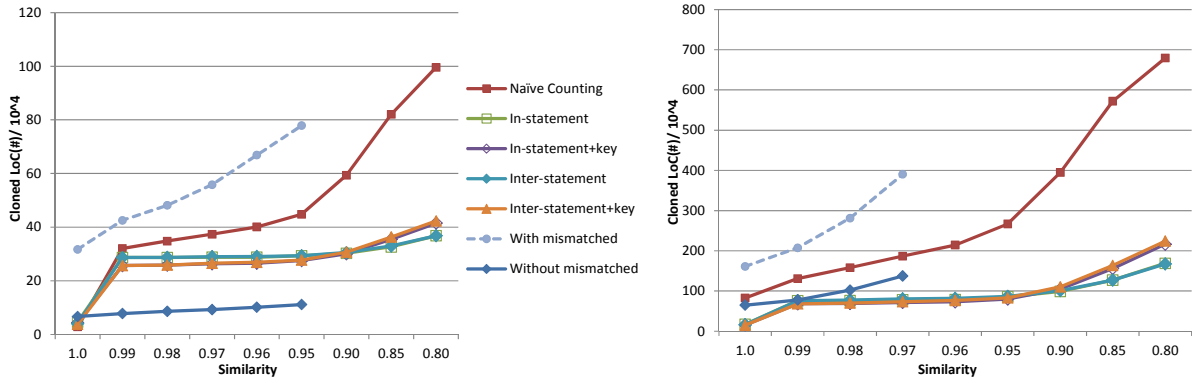
7.1 Scalability

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



(a)JDK (b)Linux

Figure 3: Comparison time of Boreas (using different settings) and Deckard on JDK and the Linux kernel (Note: The vertical scale is logarithmic)



(a)JDK (b)Linux

Figure 4: Cloned LoC found by Boreas (using different settings) and Deckard on JDK and the Linux kernel (Note: Only unique lines are counted here)

also has vector generation period and vector clustering period, so we compare them correspondingly.

Comparison time is shown in Figure 3. As expected, Boreas is much faster than Deckard, especially for JDK. Boreas uses less than 3 minutes to process JDK, and less than 3 hours to process the Linux kernel. As a matter of fact, when $similarity = 0.95$, Deckard failed to produce results for the Linux kernel in more than 24 hours.

In Figure 3, the Naïve Counting Stage has decreasing running time as *similarity* decreases. It is because the Merge Twice is very effective. In the Naïve Counting Stage, when similarity decreases, it becomes easier for Boreas to find a match, and jump out of the loop. So the running time is not always growing.

We compare the setup time for each scenario in Figure 5. Higher level stages usually require more setup time because they perform more analysis on the source code. But 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 apparently the parser for Java is very slow. In both cases, the setup time of Deckard is significantly longer 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 in Table 1. 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).

7.2 Clone Quantity

We measure the clone quantity by counting the number of lines of code (LoC) within the detected cloned code fragments. In the Deckard paper [10], LoC are counted for all cloned pairs, which results in some code fragments to be counted for multiple times if they belongs to code clone pairs of different sizes. As a result, Deckard reports 1,943,777 cloned LoC from JDK 1.4.2, which has only 2,418,767 LoC (which means that more than 80% of lines are cloned). We believe this counting method for computing LoC is not reasonable, so 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.

Moreover, we found a large proportion of the clones identified by Deckard is trival, 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". Since they actually refer to the same code fragment, they are not clones (so called self clones). Import or package clones are those clones consisting of only "import"

Table 2: False positive rates for different techniques

JDK	Naïve	In-statement no key	In-statement key	Inter-statement no key	Inter-statement key	Deckard
1.00	2%	0%	0%	0%	0%	0%
0.99	8%	0%	0%	0%	0%	0%
0.98	25%	0%	0%	0%	0%	4%
0.97	36%	0%	0%	0%	0%	3%
0.96	40%	0%	0%	0%	0%	14%
0.95	66%	0%	0%	0%	0%	34%
0.90	90%	0%	0%	0%	0%	-
0.85	89%	7%	6%	4%	9%	-
0.80	89%	20%	26%	20%	19%	-
Linux	Naïve	no key	key	no key	key	Deckard
1.00	0%	0%	0%	0%	0%	0%
0.99	28%	1%	0%	0%	0%	0%
0.98	56%	3%	0%	0%	0%	5%
0.97	68%	3%	0%	0%	0%	10%
0.96	65%	5%	0%	1%	0%	-
0.95	76%	6%	3%	5%	3%	-
0.90	89%	10%	5%	10%	11%	-
0.85	94%	20%	15%	22%	27%	-
0.80	95%	40%	36%	37%	45%	-

and "package" statements (mainly in JDK). While they are indeed clones, they do not have any practical implications. 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. According to our calculation, about 36% Cloned LoC in JDK and 50% Cloned LoC in the Linux kernel produced by Deckard are trival.

The comparison of identified clone LoCs is shown in Figure 4. As mentioned before, since Boreas and Deckard use different granularities, Deckard produces clones with mismatched brackets, which Boreas does not produce. Only about 12% Cloned LoC in JDK and 18% Cloned LoC in the Linux kernel produced by Deckard are matched clones. So we show two cloned LoC results for Deckard: one with mismatched clones, and another without. In both cases, we removed the self clones and import/package clones from the Deckard results. In 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. This shows that when comparing blocks or methods to detect clones, Boreas performs at least as effective as Deckard.

But cloned LoC can be deceptive sometimes. For example, in both Figure 4(a) and Figure 4(b), the Naïve Counting version of Boreas has found the most LoCs. However, most clones it found are false positives, which we will discuss in the next subsection. Taking clone quality into consideration, we will later find that cloned LoC of higher level stage versions of Boreas are more reliable, which is also comparable to cloned LoC of Deckard.

7.3 Clone Quality

Although clone quality is also an important metric, 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[3] , this false positive rate might not be very accurate, but it could still reflect the clone quality of the corresponding technique to some degree.

The numbers of false positives are presented in Table 2. We can see that 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 (<10%) when similarity is as low as 0.90 (0.85 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.

Table 3: Number of clusters found by Boreas and Deckard

Boreas(simi=0.9)	JDK	Linux
inter-statement	123	423
inter-statement+key	114	355
Deckard(simi=0.97)	JDK	Linux
With mismatched	111	829
Without mismatched	67	551

7.4 Clone Clusters

Next we inspect the clone clusters found by Boreas. In order to remove the trivial clones, we only consider clusters with at least 10 clones.

Boreas is able to identify 114 clone clusters from JDK and 355 clusters from the Linux kernel (using Inter-statement +key, $similarity = 0.90$). Based on our manual inspection, most of the clusters are correct. Moreover, Boreas is able to find large clones, which could be potentially more useful. For example, 6 clusters with at least 100 clones are identified from JDK, with the largest one containing 491 cloned blocks[4].

As Deckard also produces clone clusters, we should be able to compare the clustering results. However, Deckard is able to produce significantly more clone clusters due to self-clones and mismatched clones. If we remove these kinds of clone clusters, we found that Boreas could identify more clusters than Deckard on JDK, and comparable number of clusters on Linux (See Table 3).

In the meantime, the clusters found by Boreas are more meaningful and interesting because they are bounded by natural boundaries such as brackets. Since all the code fragments in the same cluster are performing similar functions, it might be of future interests to analyze the design patterns, or find aspects within these clusters.

7.5 Effects of Optimizations

In this subsection, we investigate the effect of our optimization heuristics using two scenarios. Scenario A includes the whole JDK source files, and Scenario B includes the first 1000 JDK source files. We run Boreas on these scenarios, with $similarity = 0.85$.

From the table, Merge Twice improves 20%-50% of the running time, and Quick Separation improves about 100X of the running time. Without Quick Separation, Boreas even fails to produce the result within one hour for Scenario A.

[3]We have investigated more than 9,000 cloned pairs. All of them with fragment ID and label (true or false) are available on our website `http://www.callowbird.com/boreas.html`. Anyone can download them and inspect it.

[4]We also put the information and source code of the identified large clusters on our website.

Table 4: Experimental results of Boreas with and without the optimization heuristics

	QSep	Twice	CTime	Clusters	Max	LOC
Scenario A	yes	yes	68.5s	869	491	339517
	yes	no	83.0s	868	491	339564
	no	yes	>60min	-	-	-
	no	no	>60min	-	-	-
Scenario B	yes	yes	1.6s	155	179	39563
	yes	no	2.5s	155	179	39563
	no	yes	131.7s	135	373	42814
	no	no	155.1s	129	613	42904

Meanwhile, The results produced (such as clusters and LoC) are not significantly affected.

Acknowledgment

The authors would like to thank Prof. Lingxiao Jiang for helping us with our questions on the Deckard paper and software.

8. 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, keywords and punctuations. With two fast similarity functions and optimization heuristics, 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.

9. REFERENCES

[1] G. Antoniol, U. Villano, E. Merlo, and M. D. Penta. Analyzing cloning evolution in the linux kernel. *Information & Software Technology*, 44(13):755–765, 2002.

[2] B. S. Baker. On finding duplication and near-duplication in large software systems. In *WCRE*, pages 86–95, 1995.

[3] B. S. Baker. Parameterized pattern matching: Algorithms and applications. *J. Comput. Syst. Sci.*, 52(1):28–42, 1996.

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

[5] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. Merlo. Comparison and evaluation of clone detection tools. *IEEE Trans. Softw. Eng.*, 33:577–591, 2007.

[6] P. Bulychev and M. Minea. Duplicate code detection using anti-unification. In *SYRCOSE*, 2008.

[7] A. Corazza, S. Di Martino, V. Maggio, and G. Scanniello. A tree kernel based approach for clone detection. In *ICSM*, pages 1 –5, 2010.

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

[9] Y. Higo, U. Yasushi, M. Nishino, and S. Kusumoto. Incremental code clone detection: A pdg-based approach. In *WCRE*, pages 3 –12, 2011.

[10] 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.

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

[12] T. Kamiya. The official ccfinderx website. http://www.ccfinder.net/ccfinderx.html.

[13] 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.

[14] I. Keivanloo, J. Rilling, and P. Charland. Internet-scale real-time code clone search via multi-level indexing. In *WCRE*, pages 23 –27, 2011.

[15] H. Kim, Y. Jung, S. Kim, and K. Yi. Mecc: memory comparison-based clone detector. In *ICSE*, pages 301 –310, 2011.

[16] R. Komondoor and S. Horwitz. Using slicing to identify duplication in source code. In *SAS*. Springer-Verlag, 2001.

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

[18] M.-W. Lee, J.-W. Roh, S.-w. Hwang, and S. Kim. Instant code clone search. In *FSE '10*, pages 167–176, 2010.

[19] S. Lee and I. Jeong. SDD: high performance code clone detection system for large scale source code. In *OOPSLA*, pages 140–141, 2005.

[20] 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.

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

[22] E. Merlo, G. Antoniol, M. Di Penta, and V. Rollo. Linear complexity object-oriented similarity for clone detection and software evolution analyses. In *ICSM 2004*, pages 412 – 416.

[23] C. K. Roy and J. R. Cordy. An empirical study of function clones in open source software. In *WCRE*, pages 81–90. IEEE, 2008.

[24] 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.

[25] 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.

[26] P. Schugerl. Scalable clone detection using description logic. In *IWSC '11*, pages 47–53, 2011.

[27] G. Selim, K. C. Foo, and Y. Zou. Enhancing source-based clone detection using intermediate representation. In *WCRE*, pages 227 –236, 2010.

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