CMCD: Count Matrix based Code Clone Detection

Yang Yuan and Yao Guo
National Engineering Research Center for Software Engineering
Key Laboratory of High-Confidence Software Technologies (Ministry of Education)
School of Electronics Engineering and Computer Science, Peking University, Beijing 100871, China
Email: {yangyuan, yaoguo}@pku.edu.cn

***Abstract*—This paper introduces CMCD, a Count Matrix based technique to detect clones in program code. The key concept behind CMCD is Count Matrix, which is created while counting the occurrence frequencies of every variable in situations specified by pre-determined counting conditions. Because the characteristics of the count matrix do not change due to variable name replacements or even switching of statements, CMCD works well on many hard-to-detect code clones, such as swapping statements or deleting a few lines, which are difficult for other state-of-the-art detection techniques. We have obtained the following interesting results using CMCD: (1) we successfully detected all 16 clone scenarios proposed by C. Roy et al.; (2) we discovered two clone clusters with three copies each from 29 student-submitted compiler lab projects; (3) we identified 174 code clone clusters and a potential bug from JDK 1.6 source files.**

***Keywords*-Code clone detection; count matrix; bipartite graph matching**

I. Introduction

In software development, it is common to reuse some code fragments by copying with or without minor modifications. This kind of code fragments are called *code clones*. Code clones are shown to be harmful in software maintenance and evolution [11]. However, studies show that a significant amount (typically ranging from 7%-23%) of code is cloned in large software system [21], [20], [7]. Thus, it is important to detect the clones in the source code accurately, which requires a great deal of work. Besides, as some code clones are not exact copy of the original one, comparing their similarity and discrepancy will help both programmers and researchers in program understanding, code quality analysis, or bug detection. A more serious problem is code plagiarism, which often occurs in educational environment or relates to legal cases. Therefore, detecting code clones is both important and worthwhile in many fields.

Many clone detection approaches have been proposed in the literature. Roy et al. proposed scenario-based evaluation to compare almost all existing clone detection techniques in [23]. They presented four kinds of scenarios. Among them, Scenario 1 is the easiest, while Scenario 4 is the hardest, which is related to switching statements, deleting a few lines, and other minor changes. Based on their evaluation, every known technique has its limitation in discovering clones in certain scenario, especially in Scenario 4.

This paper presents a new Clone Detection algorithm based on Count Matrix (CMCD) that could successfully detect all the scenarios mentioned in the above paper (including the hardest Scenario 4). The key idea of CMCD is the language-independent *Count Matrix*, which can accurately model the uniqueness of a code fragment, while achieving both high accuracy and low false positive rate.

In the proposed technique, we use *Count Matrix* (CM) to represent the characteristics of a code segment. A CM is composed of n *Count Vectors* (CVs), while n is the number of variables in the code segment. A CV represents the occurrence counts of a certain variable in different circumstances. For instance, a CV will record how many times the variable is used, defined, multiplied and called in the given code segment. Different variables will have different CVs since their occurrence counts in different circumstances usually differ. By comparing the matrix using bipartite graph matching algorithm, we can obtain the similarity between different code segments.

Our algorithm has two major advantages. First, CMCD is language-independent because it depends only on variable counts. In this paper, we choose Java as the working language to illustrate our approach, but the algorithm can be easily adapted to almost all high-level programming languages as well. Second, CMCD can detect clones obtained by switching statements, adding or removing a few lines, and other minor changes both in structure and in context. Previous evaluations have shown that these clones are difficult to detect for the state-of-the-art techniques.

CMCD can deal with methods as small as three to five statements, and false positives are rare in our experiments. Since we simply count the occurrence time of each variable, rather than analyzing the interior structure of the code specifically, it is not surprising that the execution time of the proposed approach is relatively short compared to other syntactic approaches.

In our experiments, we will show that CMCD not only works perfectly in the scenario-based tests, but also highly practical in real world applications. The results we have achieved so far include: (1) we successfully detected all 16 clone scenarios proposed by C. Roy et al.; (2) we discovered two clone clusters with three copies each from 29 student-submitted compiler lab projects; (3) we identified 174 code

1530-1362/11 $26.00 © 2011 IEEE
DOI 10.1109/APSEC.2011.13

clone clusters and a potential bug from JDK 1.6 source files.

II. Background and related work

A. Clone Types

Previous researchers classified the code clones into four types based on both the textual and functional similarities, which are two main kinds of similarity between code fragments [5], [8], [15]:

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 of a program, and many detection techniques cannot find that they are similar. We believe it is the hardest case of code clone detection. 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 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. This is the base ground of this paper: 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, there is no existing approach can detect Type-4 clones effectively.

B. Clone Detection Techniques

There are mainly four types of code clone detection techniques [23]:

(1) Textual approach: 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[17], NICAD[22], Simian[1], etc.
(2) Lexical approach: 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[18], etc.
(3) Syntactic approaches: 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], etc.
(4) Semantic approaches: Semantics-aware 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[16], GPLAG[19], etc.

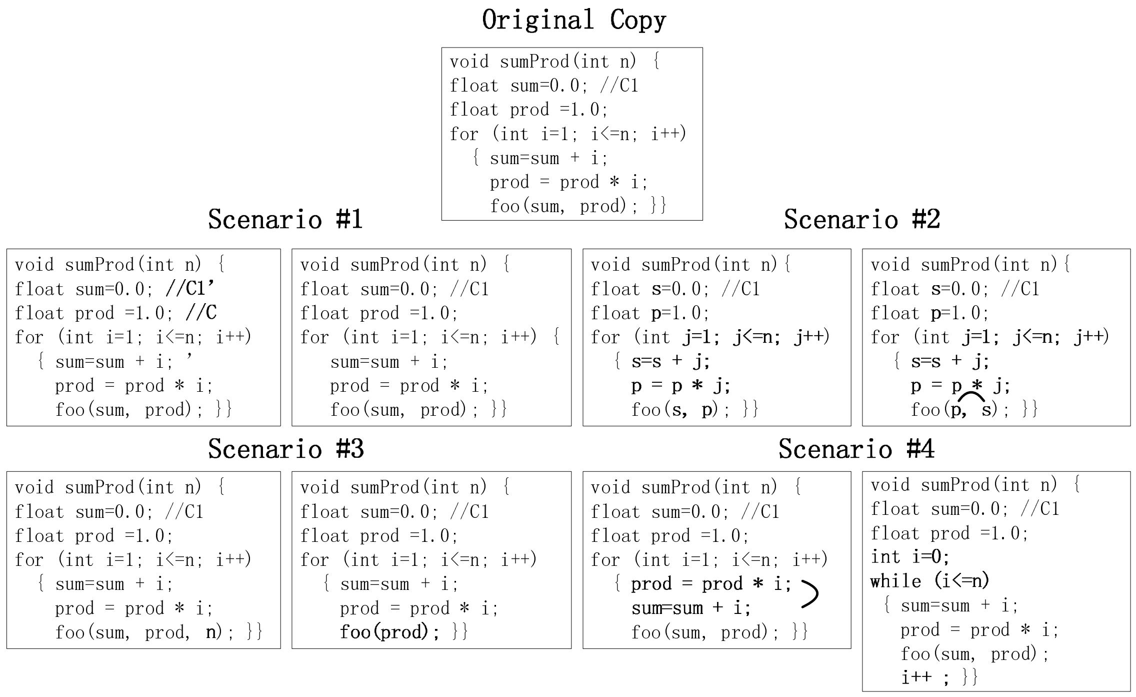
[1]http://www.redhillconsulting.com.au/products/simian/

C. Scenario-based Evaluation

In the survey paper by Roy et al. [23], they proposed a qualitative approach called *scenario-based evaluation* to compare and evaluate almost all well-known existing clone detection techniques, which includes both classical and state-of-the-art techniques. Actually, they tried to present benchmarks for clone detection techniques, by taking a predictive, scenario-based approach. They designed a small set of hypothetical program editing scenarios representative of typical changes to copy/pasted code in the form of a top-down editing taxonomy.

It seems difficult at the first glance to "choose" some scenarios to be test cases to evaluate all the techniques, for there are so many aspects need to be checked and compared. However, there are mainly four kinds of scenarios, and some kinds of scenarios are inherently difficult for certain clone detection techniques. Thus, the scenarios actually represent typical obstacles for the clone detection techniques, rather than being specific invariable benchmarks. Generally, Scenario 1 adds or deletes some irrelevant characters, such as spaces or comments. Scenario 2 makes small differences such as changing the names of variables, or swapping the positions of two variables. Scenario 3 deletes some statements, or changes the call of procedure. Scenario 4 swaps two statements of the code fragment. Figure 1 shows examples of different scenarios.

Given the formal definition of each scenario, Roy et al. [23] compared the performance of each clone detection techniques in each scenario. The results show that Scenario 1 is the easiest one, while scenario 4 is hard for most



```
void sumProd(int n) {
float sum=0.0; //C1
float prod =1.0;
for (int i=1; i<=n; i++)
  { sum=sum + i;
    prod = prod * i;
    foo(sum, prod); }}
```

```
void sumProd(int n) {
float sum=0.0; //C1'
float prod =1.0; //C
for (int i=1; i<=n; i++)
  { sum=sum + i; '
    prod = prod * i;
    foo(sum, prod); }}
```

```
void sumProd(int n) {
float sum=0.0; //C1
float prod =1.0;
for (int i=1; i<=n; i++) {
   sum=sum + i;
   prod = prod * i;
   foo(sum, prod); }}
```

```
void sumProd(int n){
float s=0.0; //C1
float p=1.0;
for (int j=1; j<=n; j++)
  { s=s + j;
    p = p * j;
    foo(s, p); }}
```

```
void sumProd(int n){
float s=0.0; //C1
float p=1.0;
for (int j=1; j<=n; j++)
  { s=s + j;
    p = p * j;
    foo(p, s); }}
```

```
void sumProd(int n) {
float sum=0.0; //C1
float prod =1.0;
for (int i=1; i<=n; i++)
  { sum=sum + i;
    prod = prod * i;
    foo(sum, prod, n); }}
```

```
void sumProd(int n) {
float sum=0.0; //C1
float prod =1.0;
for (int i=1; i<=n; i++)
  { sum=sum + i;
    prod = prod * i;
    foo(prod); }}
```

```
void sumProd(int n) {
float sum=0.0; //C1
float prod =1.0;
for (int i=1; i<=n; i++)
  { prod = prod * i;
    sum=sum + i;
    foo(sum, prod); }}
```

```
void sumProd(int n) {
float sum=0.0; //C1
float prod =1.0;
int i=0;
while (i<=n)
  { sum=sum + i;
    prod = prod * i;
    foo(sum, prod);
    i++ ; }}
```

Figure 1. Taxonomy of editing scenarios for different clone types.

techniques. No existing techniques can perform very well in all the given scenarios.

III. Approach

During the process, we first split the source code into classes, and then split classes into methods. Using a lexical analyzer, we can obtain the count matrix (CM) for each method. Based on the CMs, we will construct a bipartite graph for two methods, and do bipartite graph matching on the graph. Similarity between two methods is closely related to the size of the matching. Using the similarities between methods, the similarity between classes can be easily calculated using the same method. A false positive elimination step is performed after matching to eliminate some obvious false positive cases based on heuristics. In the end, methods or classes with strong similarities are detected as clones.

A. Main Idea

One of the key ideas behind our approach is to avoid using syntactic properties (such as structure, data flow, etc.) of the program, because they are both complex to analyze and difficult to compare. As we have seen, using certain fixed syntactic properties of the program may prevent the technique from detect some special cases of clones, especially in the cases when the property is changed (for example, switching statements). Instead, we try to analyze lexical-only properties to overcome the limits. The properties we choose in this paper is variable counts.

We introduce *Counting Conditions*, which are used to decide when to count. That is, if we decide to count the occurrences of variables in certain circumstance with special criteria, this criteria is called a counting condition. When current circumstance meets the condition, we increase the count by one. Our approach uses different counting conditions heavily to analyze the lexical properties. With a large number of counting results, we can achieve a concrete representation of the code.

There are two advantages using counting conditions. First, it is easy and fast to analyze the code properties by counting. Second, since we choose a large number of properties to count, it will be highly unlikely, if not impossible, for some special cases to influence all of the count conditions, thus our approach is robust in almost all cases.

If two code segments are essentially different, it is unlikely that all of their counting results will be similar; meanwhile, if they are clones, most of their counting aspects will match. The more aspects we investigate, the more accurate our approach will be. Since the idea behind our approach is simple, new kinds of count conditions can also be added to the counting algorithm easily.

B. Count Vector

Since programs written in high-level programming languages are based on operations for variables, we choose to investigate the behavior of variables. Our algorithm currently include 13 counting conditions, which are related to the occurrence counts of variables in the following circumstances:

1	used
2	added or subtracted
3	multiplied or divided
4	invoked as a parameter
5	in an if-statement
6	as an array subscript
7	defined
8	defined by add or subtract operation
9	defined by multiply or divide operation
10	defined by an expression which has constants in it
11	in a third-level loop(or deeper)
12	in a second-level loop
13	in a first-level loop

Notice that we analyze 13 aspects for *every variable*, and these 13 aspects will construct a *count vector* (CV) for the variable: these are the only information about the variable after this step. So, neither the structure of the code nor other information such as the names of the variables will remain after the counting step.

Here are some reasons behind the choices. The name of the variable is meaningless, because code could be cloned by changing the name of the variables. The structure of the code and the relationship between statements are important, but they are hard to analyze and compare. Although these 13 numbers seem abstract, the count vector they form could represent the variable satisfactorily for the purpose of clone detection.

Different variables in different functions will have different CVs. For example, loop counter **i** will often appear in first-level loops, loop counter **j** will often appear in second-level loops, and variables which save max/min data will often appear in if-statements. Temporary variables will be used for only a few times, while important variables are used more frequently. By comparing their CVs, we can easily distinguish one variable from another.

If one or two of the conditions are missing, it might not affect the results significantly. You can also add more counting conditions which are easy to calculate and helpful to represent the variables: the more aspects of certain variable used in the algorithm, the more accurate detection results will be achieved.

Note that it is fully acceptable to use more or fewer aspects here. It is also interesting to further investigate the influences of each aspect, but we do not discuss this in this paper due to page limitations.

Figure 2 shows an example for calculating the CVs for variables. In the program, the variable tot appears only in the first line, so its CV will be $\langle 0,0,0,0,0,0,1,1,1,0,0,0,0\rangle$, because it is defined by both subtract operation and multiply operations. The variable i appears in the loop, and its CV is $\langle 4,1,0,0,1,3,1,0,0,1,0,4,1\rangle$. Similarly, the CV of variable j is $\langle 3,0,0,0,1,3,1,1,0,1,0,4,0\rangle$.

Note that during the implementation of the program, the variables i and j should compare with $n-1$ or n in

A sort program:

```
1: $tot$=$n$*$n$-Find($n$)
2: for $i = 1$ to $n - 1$ do
3:     for $j = i + 1$ to $n$ do
4:         if $a[i] > a[j]$ then
5:             $k = a[i]$
6:             $a[i] = a[j]$
7:             $a[j] = k$
8:             $swap = swap$+1
9:         end if
10:    end for
11: end for
```

Count matrix:

tot	0	0	0	$\cdots$	0
i	4	1	0	$\cdots$	1
j	3	0	0	$\cdots$	0
a	4	0	0	$\cdots$	0
k	1	0	0	$\cdots$	0
n	5	2	1	$\cdots$	2
$swap$	1	1	0	$\cdots$	0

Figure 2. A sample program and the corresponding count matrices.

the loop test expressions. Under this interpretation, some dimensions of the CVs should be modified accordingly. But it actually does not matter, because we just want to explain the main idea of CV here. In practice, whether an occurrence of a variable is counted in a certain circumstance is not important either, as long as the counting condition is clear and consistent throughout the implementation.

After calculating CVs for all variables, we obtain n vectors, which form a count matrix (CM), with n lines and 13 columns. If two code fragments are cloned, their CMs will be very similar. Intuitively, even if the clone changes a small part of the original copy, their variable occurrence counts will not change greatly from a statistical perspective, thus their CMs will remain similar. Using an appropriate method to compare their CMs, we can compare the similarity between two code fragments.

C. Metrics and Matching

We compare the CVs in the Euclidean space. The difference between two vectors is determined by the Euclidian Distance between them in the space, i.e.,

$$D(v_1, v_2) = ||v_1 - v_2||_2 = \sqrt{\sum_{i=1}^{13}(v_{1i} - v_{2i})^2}$$

However, the similarity of two CVs should be related to their lengths. That is, if two vectors are very long, their distance are likely to be longer, but they might be similar as well. In the meantime, if the distance between two short vectors is as long as the distance between two long vectors, the two short vectors are less similar. Thus, we will calculate the difference in a piecewise-defined function: if the lengths of vectors are small, we calculate their distance; otherwise, their difference equals their distance divided by their lengths.

The function itself is heuristic; we can still add lots of special conditions in it. For example, if two vectors differ in seven or more dimensions, the variables they represent are unlikely to be similar. If the length of one vector is twice the length of another, they will not match either. We design an integrated function to determine the difference of two vectors based on the above considerations. The function reads two vectors, and produce an output of a float number ranged in

Algorithm 1 Measure the similarity between two vectors

```
Input: two vectors $u$ and $v$, Len($u$) >Len($v$)
Output: the similarity between $u$ and $v$
1: if (Len($u$) $>$ 2*Len($v$)) or (dif($u, v$) $\geq$ 7) then
2:     return 0
3: end if
4: $maxlen \leftarrow$Max(Len($u$),Len($v$))
5: if $maxlen \leq$SmallLen then
6:     return Normalize(EucDist($u, v$))
7: else
8:     return Normalize(EucDist($u, v$)$/maxlen$)
9: end if
```

Algorithm 2 Compare two methods: using a threshold

```
Input: two matrices $A$ and $B$
Output: the similarity between $A$ and $B$
1: construct bipartite graph $G$ for $A, B$
2: add zero node to $G$
3: if similarity($A[i], B[j]$) $\leq$ Threshold then
4:     $e_{i,j} \leftarrow 0$
5: else
6:     $e_{i,j} \leftarrow 1$
7: end if
8: $MiniWeight \leftarrow$AugmentingPath($G$)
9: $MiniWeight \leftarrow$Normalize($MiniWeight$,size($G$))
10: return $MiniWeight$
```

$[0, 1]$, where 0 means not similar at all, and 1 means they are the same.

A CM is composed of n count vectors, where n is the number of variables within the method. When we are comparing two methods, we cannot just compare each row of their matrices and "add" the result together as the answer, because the variable represented by the first row of the first matrix may match the variable represented by the fourth row of the second matrix, while the variable represented by the first row of the second matrix may not match any variables in the first method at all. If we match the i_{th} rows of two matrices for all $1 \leq i \leq n$, the result is certainly unconvincing.

Hence, we will construct a bipartite graph from the matrices, and do maximum matching on the graph. A *bipartite graph* is a graph whose vertices can be divided into two disjoint sets U and V such that every edge connects a vertex in U to one in V, i.e., U and V are independent sets. A *matching* M in G is a set of pair wise non-adjacent edges. A *maximum matching* is a matching that contains the largest possible number of edges. A *perfect matching* is a matching which matches all vertices of the graph.

Maximum weighted bipartite matching is defined as a perfect matching where the sum of the values of the edges in the matching has a maximal value. If the graph is not complete bipartite, missing edges are inserted with value zero. There are efficient polynomial algorithms solving both maximum matching in bipartite graphs and maximum weighted bipartite matching problems.

Suppose the created graph is $G = \{U, V, E\}$, where U, V are vertices sets and E is a weighted edge set. Every variable in the first (second) method will be represented by a vertex in $U(V)$, and for every vertex $x \in U$ and every vertex $y \in V$, there is a weighted edge $e \in E$ connecting them, whose weight is the similarity between the two variables which x and y represented. After using the KM algorithm[2], we will get a perfect matching with maximum weight. If the total numbers of the variables in the methods are not same, we can add some "zero node" to the graph, whose CVs are zeros.

During the matching process, every variable is matched to its most likely corresponding variable in the other method, in the sense that the total similarity between each pairs of variables is maximized. We use the total weight of the matching to represent the similarity between two methods.

Of course, we should take the total number of the variables in the methods and the size of the methods into consideration as well, because large methods are more likely to have bigger matching weights, while in contrast, matching weights of small methods will not be considerable even the methods are essentially different. Hence, the similarity between two methods should be normalized with the consideration of method-size and variable-count.

We also want to mention a modified version of this algorithm, which is more practical and is used heavily in our implementation. We define a threshold for the similarity between two variables. That means, if the similarity between two variables is below the threshold, based on the function we designed, we treat them as essentially different variables; otherwise, we treat them as similar variables. With the threshold, the edges no longer need to be weighted, and we can use an augmenting path algorithm to find bipartite graph maximum matching on the graph. In this way, the similarity between two methods is the number of the matched variables divided by the total number of the variable of the method. The standard algorithm is more accurate and consumes more time, and this modified algorithm is time-efficient, thus more favorable.

Given any two methods, we can compare them efficiently. Using the *Scale-up approach*, we can compare two classes or two packages of classes. First, we construct a bipartite graph for the two classes: each vertex represents a method in the class, and the edge in the graph connecting two vertices is weighted by the similarity between the methods them represent. After constructing the graph, we use the KM algorithm to find the maximum weighted bipartite

[2]http://en.wikipedia.org/wiki/Hungarian_algorithm

Algorithm 3 Compare two classes in a scale-up manner

```
Input: two classes $C_1$ and $C_2$
Output: the similarity between $C_1$ and $C_2$
1: construct bipartite graph $G$ for $C_1, C_2$
2: $m_{ij} \leftarrow$the $j_{th}$ method in $C_i$
3: add zero node to $G$
4: $w_{i,j} \leftarrow$similarity$(m_{1i}, m_{2j})$
5: $MiniWeight \leftarrow$KM$(G)$
6: $MiniWeight \leftarrow$Normalize$(MiniWeight$,size$(G))$
7: return $MiniWeight$
```

matching. The threshold also works here. And from classes to packages, we can do this process again. Finally, we will get a float number ranged from $[0, 1]$, representing the similarity between two packages of codes. In Algorithm 3 we describe the algorithm used to compare two classes from the result of the similarities between methods in these classes in a scale-up manner. The algorithm for comparing two packages is similar.

IV. Implementation and results

In the implementation of the CMCD prototype, we first use Soot to convert the Java code to Jimple. Jimple is a 3-address intermediate representation that has been designed to simplify analysis and transformation of Java bytecode introduced by Vallée-Rai and Hendren[24].

Jimple has a smaller language set, thus are more easily to be analyzed. And it will break some complex statements in Java into several basic ones, which helps us count the occurrence times of the variables more easily. If there are only class files, Soot is still able to convert them to Jimple files without the Java source files, which is another advantage of using Jimple to analyze the code.

Besides, while simplifying the implementation, this transformation has little influence on the results the algorithm might produce, because the Jimple language can do as many things as the Java language. The transformation does not change the functionality of the program; it just changes the language it uses. Since our approach is language-independent, it applies well to Jimple.

A. Scenario-based Evaluation

Based on the survey by Roy et al. [23], we have tested the CMCD prototype implementation on all the 16 scenarios listed in it. These scenarios are comparatively easy for our algorithm because there are only a few modifications between the original copy and the modified copies. Our algorithm successfully detected all the clones, which is very good since the authors believe that it is hard to solve all the cases "very well". Actually, according to their rating, there are no known algorithms can solve all the cases well. We make a small table listing the performance of some state-of-the-art techniques [21], [12], [18], [4], [10], [1], [19] quoted from the survey [23]. From the table, we can see that Scenario 4 is the most difficult: many popular techniques fail here, since this scenario has swapped or deleted statements, which are difficult for previous techniques, but relatively easy for our count based algorithm.

Note in the table, we use "✓" to denote that the technique can detect the clone in the scenario ("medium" or higher in the original table), and use "✗" to denote that it can not ("low" or lower in the original table).

For our approach, Scenario 4 is just as easy as other scenarios. Take the first modified program of Scenario 4 in Figure 1 as an example, i.e., swapping the first and the second line in the for-loop. Since our approach does not consider the order of statements, the similarity between the two copies should be 1. And in the second program, the for-loop is changed into while-loop. Since our approach converts the Java language into Jimple, while after transformation, there will be just goto-statements and if-statements. Neither for-statement nor while-statement remains, thus these two copies are similar in Jimple.

B. Detecting Plagiarism

In order to test the applicability of CMCD, we performed an experiment to detect plagiarisms in student-submitted compiler lab projects. The data are collected from the previous year's submissions (which won't affect students' grades). We tested 29 project submissions, and each of them has class quantities ranging from 106 to 251, and their LoC (lines of code) ranging from 7,825 to 38,086. The LoC for all the 29 projects is 585,508. In our experiments, we used an Intel(R) Core(TM)2 Duo CPU T9400 processor with Windows 7, it takes 123 minutes to compare all the projects. According to the results, we find 2 clusters of code clones, while each has 3 copies in it. We have confirmed with the professor that each cluster consists of two student submissions copied from one original submission. In one case, all the class names (including package/folder names) are replaced, which is difficult to identify through only manual examination.

Notice that all the 29 project submissions implement the same functions using similar algorithms, but different students has different writing styles, thus we did not find any false positives in this experiment. Although the results did not affect the student grades because the experiments are considered confidential, but we have implemented it as a Web service at this year's submission website, such that each student can check whether their code is similar to an existing submission before they submit their final project. Of course it is impossible to detect more delicate plagiarisms in this way, however, we believe that if a student can pass the automatic plagiarism check, they have spent enough time modifying their code so that these code becomes their "own" code.

Approaches	S1a	S1b	S1c	S2a	S2b	S2c	S2d	S3a	S3b	S3c	S3d	S3e	S4a	S4b	S4c	S4d
BasicNICAD	√	√	√	×	×	×	×	√	√	√	√	√	×	×	×	×
FullNICAD	√	√	√	√	√	√	√	√	√	√	√	√	×	×	×	×
Simian	√	√	×	√	√	√	×	×	×	×	×	×	×	×	×	×
CCFinder	√	√	√	√	√	√	×	×	×	×	×	×	×	×	×	×
CP-Miner	√	√	√	√	√	√	×	√	√	√	√	√	×	×	×	×
CloneDr	√	√	√	√	√	√	×	√	√	×	√	√	×	×	×	×
Deckard	√	√	√	√	√	√	×	√	√	√	√	√	×	×	×	×
Antoniol	√	√	√	√	√	√	√	√	√	×	×	×	√	√	×	×
GPLAG	√	√	√	√	√	√	×	×	×	√	√	√	√	√	×	√
CMCD	√	√	√	√	√	√	√	√	√	√	√	√	√	√	√	√

Figure 3. Ratings of techniques

C. Analyzing JDK 1.6 Source Code

In a larger-scale experiment, we choose to analyze JDK 1.6.0_18 (which contains 7,197 java files and 2,079,166 LoC). Our objective is to identify similar code as a code cluster: these code clusters can be used as aspect candidates to promote aspect-oriented programming (AOP) [14], or they can also be used to detect potential bugs when considering the differences in each cluster [18].

We pick every pair of two methods in the source code, and calculate the similarity between them. If two methods are similar, we will put them into the same cluster. Some methods, such as $init()$ or $getName()$, $getObject()$, are relatively small methods and are easily to be similar. For example, most of methods called $getXXX()$, has only one line as its content, that is, "return xx.yy.zz;". Of course they are code clones, but it is meaningless to point them out. So we focus on the larger methods, which produces more meaningful results.

We have found 786 similar methods in 174 cluster in the JDK source code in 163 minutes. Here we list three methods in one cluster in Figure 4, which are similar but not exactly the same. Many of the methods we discovered are in this pattern, and they usually differ from swapping or deleting some lines, which are typical scenario 4 cases. We believe this kind of examples reveals the applicability and scalability of our approach: it performs well in real cases.

The example in Figure 4 raises another interesting point: we actually discovered a potential bug in it. All three methods implement the Singleton design pattern [9]. It is used to implement the mathematical concept of a singleton, by restricting the instantiation of a class to one object. To be thread-safe, a classical solution to implement this design pattern is to use synchronization to lock the mutual exclusion, and one needs to add if-statement both before and after the synchronized-statement, which ensure current class has no instantiation yet.

```
Method 1: (in com.sun.corba.se.impl.ior.iiop.JavaSerializationComponent)
public static JavaSerializationComponent singleton() {
if (singleton == null) {
    synchronized (JavaSerializationComponent.class) {
        singleton =
            new JavaSerializationComponent(Message.JAVA_ENC_VERSION);
    }
}
return singleton;
}
Method 2: (in com.sun.corba.se.impl.ior.iiop.SyncFactory)
public static SyncFactory getSyncFactory(){
 if(syncFactory == null){
 synchronized(SyncFactory.class) {
    if(syncFactory == null){
        syncFactory = new SyncFactory();
    } //end if
 } //end synchronized block
 } //end if
 return syncFactory;
}
Method 3: (in javax.swing.JComponent)
static Set<KeyStroke> getManagingFocusBackwardTraversalKeys() {
    synchronized(JComponent.class) {
        if (managingFocusBackwardTraversalKeys == null) {
            managingFocusBackwardTraversalKeys = new HashSet<KeyStroke>(1);
            managingFocusBackwardTraversalKeys.add(KeyStroke.getKeyStroke(
                    KeyEvent.VK_TAB,InputEvent.SHIFT_MASK|InputEvent.CTRL_MASK));
        }
    }
    return managingFocusBackwardTraversalKeys;
}
```

Figure 4. A clone example found in JDK 1.6.0

The three methods found in the JDK source code have different implementation: the first one has no if-statement inside the lock; the second one is the classical implementation; the third one has no if-statement outside the lock.

Note that the third method is inefficient, but correct, because once there is a singleton instance available, we do not need to acquire the monitor again and again as it is expensive. But the first method is **incorrect**, because it is possible that two threads enter the first if-statement at the same time when there is no singleton available. Then, because there is no-statement inside the lock, both of these two threads will enter the lock sequentially. Thus we could get two instantiations.

This method is located in the class $com.sun.corba.se.impl.ior.iiop$, and also in the newest JDK version(jdk1.6.0_22). Because it is not fixed yet, we have reported it as a potential bug to the Java Developer Support (the assigned Bug ID is 6999537, and the bug report can be found at the website[3]).

V. Concluding Remarks

In this paper, we have presented a count-based, language-independent clone detection approach called CMCD, which is able to detect a class of hard-to-detect code clones. CMCD compares code segments using their count matrices, which is composed by count vectors for each variable in a given code segment. By constructing bipartite graphs and perform matching on the graphs, we can compare the similarities between two methods, classes or packages. The results shows that CMCD performs well in scenario-based evaluation. In larger-scale evaluations, CMCD is able to detect code plagiarism in students' homework, and also able to identify a potential bug in the JDK source code.

Acknowledgment

This work is supported by the National Basic Research Program of China (973) under Grant No. 2009CB320703, the Science Fund for Creative Research Groups of China under Grant No. 60821003, National Science and Technology Major Project under Grant No.2011ZX01043-001-002, and the National Natural Science Foundation of China (NSFC) under Grant No.61103026.

We want to thank Prof. Qiangxiang Wang for providing the source code from his compiler lab class, and Prof. Wenxin Li for allowing us to use the Java programs submitted by the users of the POJ (PKU Online Judge) system.

References

[1] G. Antoniol, G. Casazza, M. D. Penta, and E. Merlo. Modeling clones evolution through time series. In *Proceedings of ICSM*, pages 273–280, 2001.

[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, Feb. 1996.

[4] I. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier. Clone detection using abstract syntax trees. In *Proc. of the Int'l. Conf. on Software Maintenance*, 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, September 2007.

[6] P. Bulychev and M. Minea. Duplicate code detection using anti-unification. In *Proc. of the SyRCoSE Workshop*, 2008.

[7] S. Ducasse, M. Rieger, and S. Demeyer. A language independent approach for detecting duplicated code. In *Proceedings ICSM'99 (International Conference on Software Maintenance)*, pages 109–118. IEEE, 1999.

[8] M. Gabel, L. Jiang, and Z. Su. Scalable detection of semantic clones. In W. Schäfer, M. B. Dwyer, and V. Gruhn, editors, *30th International Conference on Software Engineering (ICSE 2008)*, pages 321–330. ACM, 2008.

[9] Gamma, Helm, Johnson, and Vlissides. *Design Patterns Elements of Reusable Object-Oriented Software*. Addison-Wesley, Massachusetts, 2000.

[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. IEEE, 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 multi-linguistic token-based code clone detection system for large scale source code. *IEEE Trans. Software Eng*, 28(7):654–670, 2002.

[14] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. marc Loingtier, and J. Irwin. Aspect-oriented programming. In *ECOOP*. Springer-Verlag, 1997.

[15] R. Komondoor and S. Horwitz. Using slicing to identify duplication in source code. In *Proceedings of the 8th International Symposium on Static Analysis (SAS)*. Springer-Verlag, 2001.

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

[17] S. Lee and I. Jeong. SDD: high performance code clone detection system for large scale source code. In *Companion to the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2005*, pages 140–141. ACM, 2005.

[18] 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.

[19] C. Liu, C. Chen, J. Han, and P. S. Yu. GPLAG: Detection of software plagiarism by program dependence graph analysis. In *Proc. 2006 ACM SIGKDD Int. Conf. on Knowledge Discovery and Data Mining (KDD'06),*. ACM Press, 2006.

[20] J. Mayrand, C. Leblanc, and E. M. Merlo. Experiment on the automatic detection of function clones in a software system using metrics. In *International Conference on Software Maintenance (ICSM)*, pages 244–253, 1996.

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

[22] C. K. Roy and J. R. Cordy. NICAD: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization. In *The 16th IEEE International Conference on Program Comprehension, ICPC*, pages 172–181. IEEE, 2008.

[23] 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.

[24] R. Vallee-Rai, L. Hendren, V. Sundaresan, P. Lam, E. Gagnon, and P. Co. Soot - a Java optimization framework. In *Proceedings of CASCON 1999*, pages 125–135, 1999.

[3] http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=6999537