A Novel Technique for Retrieving Source Code Duplication

Yoshihisa Udagawa
Computer Science Department, Faculty of Engineering
Tokyo Polytechnic University
Atsugi-city, Kanagawa, Japan
udagawa@cs.t-kougei.ac.jp

***Abstract*—In this paper we propose a new approach for the detection of clones in source code for improving safety of software systems. The main contributions of this paper are development of a mining algorithm to explore program structure and the definition of a similarity measure that is tailored to sequentially structured texts for retrieving similar source code fragments. Retrieval experiments were conducted using Apache-Tomcat 7, which is a large-size open source Java program. The results show that the proposed mining algorithm finds a set of frequent sequences within one minute, and the proposed similarity measure is a better indicator than the Sorensen-Dice index.**

Keywords- Java source code, Control statement, Method identifier, Similarity measure, Derived sequence retrieval model, Sorensen-Dice index

I. INTRODUCTION

Reusing source code via copy and pasting rather than rewriting a similar code from scratch is a common activity in software development because it is very easy and can significantly reduce programming effort and time. Hence, software often contains duplicated code, known as a software clone. Previous research shows that between 7% - 23% of source code in a typical software system can be cloned code [1][11].

A software clone may have an adverse impact on the quality, productivity, reusability, and maintainability of a software system [10][12][13]. Tool support is necessary to facilitate code change tasks, because the number of source code may reach several hundred thousand lines for a maintenance engineer based on the author's experience in the industry.

Code clone detection has been actively researched for approximately two decades. Many approaches for identifying similar code fragments have been proposed in the literature. Generally, these techniques can be classified into four main groups, i.e., text-based, token-based, structure-based, and metrics-based.

(1) Text-based approaches

In text-based approaches, the target source program is considered as a sequence of strings. Baker [1] described an approach that identifies all pairs of matching "parameterized" code fragments. Johnson [6] proposed an approach to extract repetitions of text and a matching mechanism using fingerprints on a substring of the source code. Although these methods achieve high performance, they are sensitive to lexical aspects such as formatting and renaming of identifiers, including variables.

(2) Token-based approaches

In the token-based detection approach, the entire source code is transformed into a sequence of tokens and control statements, which is then analyzed to identify duplicate subsequences. A sub-string matching algorithm is generally used to find common subsequences. CCFinder [19] adopts the token-based technique to detect "copy and paste" code clones efficiently. In CCFinder, a similarity metric between two sets of source code files is defined based on the concept of "correspondence." CP-Miner [10] uses a frequent subsequence mining technique to identify a similar sequence of tokenized statements. Token-based approaches are typically more robust against code changes compared to text-based approaches.

(3) Structure-based approaches

In this approach, a program is parsed into an abstract syntax tree (AST) or program dependency graph (PDG). ASTs and PDGs contain structural information about the source code; thus, sophisticated methods can be applied to ASTs and PDGs for clone detection. CloneDR [2] is a pioneer among AST-based clone techniques. Wahler et al. [18] applied frequent itemset data mining techniques to ASTs represented in XML to detect clones with minor changes. DECKARD [5] also employs a tree-based approach in which certain characteristic vectors are computed to approximate the structural information within ASTs in Euclidean space.

Typically, a PDG is defined to contain the control flow and data flow information of a program. An isomorphic subgraph matching algorithm is applied to identify similar subgraphs. Komondoor et al. [7] have also proposed a tool for C programs that identifies clones. They use PDGs and a program slicing technique to find clones. Krinke [9] uses an iterative approach (k-length patch matching) to determine maximal similar subgraphs. Structure-based approaches are generally robust to code changes such as reordered, inserted, and deleted codes. However, they are not scalable to large programs.

(4) Metrics-based approaches

Metrics-based approaches calculate metrics from code fragments and compare these metric vectors rather than directly comparing with the source code. Mayrand et al.[11] proposed several function metrics that are calculated using ASTs for each functional unit of a program. Kontogiannis et al. [8] developed an abstract pattern matching tool to measure similarity between two programs using Markov

 ISBN: 978-1-61208-319-3

models. Some common metrics in this approach include a set of software metrics called "fingerprinting" [6], a set of method-level metrics including cyclomatic complexity, and a characteristic vector to approximate the structural information in ASTs.

The proposed approach is classified as a structure-based comparison [15]. It features a sequence of statements as a retrieval condition. We have developed a lexical parser to extract source code structure, including control statements and method identifiers. The extracted structural information is input to an extended Sorensen-Dice model [3][14] and the proposed source code retrieval model, named the "derived sequence retrieval model" (DSRM). The DSRM takes a sequence of statements as a retrieval condition and derives meaningful search conditions from the given sequence. Because a program is composed of a sequence of statements, our retrieval model improves the performance of source code retrieval.

In comparison with our previous paper [15], the main contribution of this paper is the development of a mining algorithm to explore a program's structure. Without knowledge of the frequency of a sequence of statements, we could not issue a query to the point. The other contribution is a set of experiments using Apache-Tomcat 7 source code, which is considered as large-scale software.

The remainder of this paper is organized as follows. In Section 2, we present a source code pre-process to extract interesting fragments. In Section 3, we present an algorithm for mining program structures and define source code similarity metrics. Experimental results are discussed in Section 4. Section 5 presents conclusions.

II. EXTRACTING SOURCE CODE SEGMENTS

At the beginning of our approach, a set of Java source codes [4] is partitioned into methods. Then, the code matching statements are extracted for each method. The extracted fragments comprise class method signatures, control statements, and method calls.

(1) Class method signatures

Each method in Java is declared in a class. Our parser extracts class method signatures in the following syntax.

<class identifier>::<method signature>

Our parser extracts a method declared in an anonymous class in the following syntax.

<class identifier>:<anonymous class identifier>:
<method signature>

Generic data types are widely used in Java to facilitate the manipulation of data collections. Our parser also extracts generic data types according to Java syntax. For example, List<String> and List<Integer> are extracted and treated as different data types.

(2) Control statements

Our parser also extracts control statements with various levels of nesting. A block is represented by the "{" and "}" symbols. Hence, the number of "{" symbols indicates the number of nesting levels. The following Java keywords for control statements [4] are processed by our parser:

if, else if, else, switch, while, do, for, break, continue, return, throw, synchronized, try, catch, and *finally*.

(3) Method calls

From the assumption that a method call characterizes a program, our parser extracts a method identifier called in a Java program. Generally, the instance method is preceded by a variable whose type refers to a class object to which the method belongs. Our parser traces the type declaration of a variable and translates a variable identifier to its data type or class identifier, i.e.,

<variable>.<method identifier>

is translated into

<data type>.<method identifier>

or

<class identifier>.<method identifier>.

We selected Apache-Tomcat 7.0.42 as our target because Apach-Tomacat [17] is one of the most popular Java web application servers. We estimated the volume of the source code using file metrics. Typical file metrics are as follows:

Number of Java Files	----	1,100
Number of Classes	----	1,681
Number of Methods	----	10,640
Number of Code Lines	----	177,724
Number of Comment Lines	----	108,167
Number of Blank Lines	----	50,344
Number of Total Lines	----	334,457

Apache-Tomcat 7.0.42 consists of 334,457 lines of source code. Relative to the number of lines, Apache-Tomcat 7.0.42 is classified as large-scale software in the IT industry.

III. RETRIEVING SIMILAR SOURCE CODE

A. Code Retrieval Approach

Our experiments consist of two stages: (1) mining structures in the whole extracted program structures; (2) performing retrievals for the mined structures using the DSRM similarity model, which are defined in Subsection III-C.

B. Mining Structures in Source Code

Initially, we mine the structures of source code using the algorithm shown in Figure 1. This algorithm shares many concepts with the well-known *Apriori* algorithm for mining frequent itemsets [16][18]. It takes the minimum support number minSup as an argument, and has its control structures similar to those of the *Apriori* algorithm. However, our algorithm essentially deals with a sequence of statements, while the *Apriori* algorithm deals with a set of items.

The major difference between the two algorithms can be found in the candidate generation process. In the *Apriori* algorithm, new candidate k-itemsets are generated based on the (k−1)-itemsets found in the previous iteration. The order of the items is ignored because the *Apriori* algorithm

 ISBN: 978-1-61208-319-3

focuses on finding a set of itemsets that occur in a dataset or transactions having a frequency greater than a given threshold, i.e., minSup.

Our algorithm is designed to find a set of sequences that occur frequently. It should be noted that several matchings can be detected in a sequence for a sub-sequence given as a matching condition. For example, the two matchings of a sub-sequence A→B are detected in a sequence A→B→A→C→A→B→D. The *Retrieve_Cand (MS, T_k)* function shown in Figure 1 finds a set of sequences of (k+1)-statements that includes k-statements found in the previous iteration in method structures extracted from Java source code "MS."

Because most of important methods are invoked in a control structure, the first element of the sequence T_1 is assumed to be a set of the "control statements," i.e., *if, else if, else, switch, while, do, for, break, continue, return, throw, synchronized, try, catch*, and *finally* statements. The assumption is considered as customization for retrieving source code duplication.

Figure 2 shows the number of retrieved sequences and elapsed time in milliseconds for each minSup. The total number of methods is 10,640; thus, for example, minSup 0.0070 corresponds to approximately 75 methods. We measured the elapsed time using the following experimental environment:

CPU: Intel Core i3 540 3.07 GHz
Main memory: 4.00 GB
OS: Windows 7 64 Bit
Programming Language: Visual Basic for Applications

Table I shows the 18 sequences mined with minSup 0.0070, which were used as retrieval conditions of the code similarity retrieval experiments.

```
// Mining statement sequences that begin with a control statement.
List<statement> MS;    // MS is a list of a "sequence" of statements
                       // that are extracted by a lexical parser.
int k;                 // k defines the number of repetition
int minSup;            // minSup defines the minimun number of occurences
List<statement> LS;    // LS is a list of a "sequence" of statements
List<statement> Tk;    // Tk is a list of a "sequence" of statements
Map<statement, int> Ck;    // Ck is a map of a "sequence and
                           // the number of occurrences of statements
    LS= Φ;
    k=1;
    Tk= { i | i ∈{Control Statements} };
    repeat
        k=k+1
        Ck= Retrieve_Cand( MS, Tk );  // A set of sequences that begins Tk
        Tk= Φ;
        for ( each c∈Ck.key() ) {
            if ( at least one element of c is not a control statement
                and Ck.value() ≥ N× minSup ) {
                LS.add( c );
                Tk.add( c ) ;
            }
        }
    until Tk= Φ;
    Result= LS;
```

Figure 1. Algorithm for mining frequent sequences

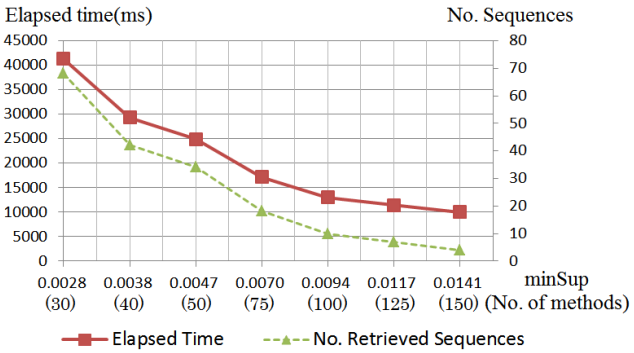

Figure 2. Number of retrieved sequences and elapsed time for each minSup

TABLE I. MINED SEQUENCES USED AS RETRIEVAL CONDITIONS

No	Sequences	Number of occurences
1	if{ → Log.debug	267
2	if{ → Log.debug → }	259
3	if{ → StringBuilder.append	218
4	catch{ → Log.error	150
5	catch{ → ExceptionUtils.handleThrowable	130
6	if{ → IllegalArgumentException	127
7	if{ → IllegalArgumentException → }	127
8	catch{ → Log.error → }	112
9	if{ → org.apache.juli.logging.Log.debug	101
10	if{ → StringBuilder.append → }	100
11	if{ → org.apache.juli.logging.Log.debug → }	99
12	if{ → StringBuilder.append → StringBuilder.append	97
13	if{ → IOException	95
14	if{ → IOException → }	95
15	catch{ → MBeanException	94
16	catch{ → MBeanException → }	94
17	if{ → StringBuilder.append → StringBuilder.append → }	92
18	catch{ → Log.error → } → }	75

C. Extending Sorensen-Dice Index

The Sorensen-Dice index is originally defined by two sets and formulated as follows.

$$\mathrm{Sim}_{\text{Sorensen-Dice}}(X_1, X_2) = \frac{2|X_1 \cap X_2|}{2|X_1 \cap X_2| + |X_1 \cap \neg X_2| + |\neg X_1 \cap X_2|} \quad (1)$$

Here, $|X_1 \cap X_2|$ indicates the number of elements in the intersection of sets X_1 and X_2.

In software, the Sorensen-Dice index is known to experimentally produce better results than other indexes such as a simple matching index that counts the number of features absent in both sets [3][14]. The absence of a feature in the two entities does not necessarily indicate similarity in software source code. For example, if two classes do not include the same method it does not mean that the two classes are similar. Our study takes the Sorensen-Dice index as a basis for defining the similarity measure between source codes. The extension of the Sorensen-Dice index on N sets is straightforward and is expressed as follows.

 ISBN: 978-1-61208-319-3

$$\mathrm{Sim}_{\text{Sorensen-Dice}}(X_1, X_2, \ldots, X_n) = \frac{n \mid X_1 \cap X_2 \ldots \cap X_n \mid}{\sum_{r=0}^{r=n-1} (n-r) \mid SetComb(X_1 \cap X_2 \ldots \cap X_n, r) \mid} \quad (2)$$

The function *SetComb*($X_1 \cap X_2 \cap \ldots \cap X_n$, r) defines intersections of sets {X_1, X_2, …, X_n} whose r elements are replaced by elements with the negation symbol. The summation of r = 0 to n−1 of *SetComb*($X_1 \cap X_2 \cap \ldots \cap X_n$, r) generates the power set of sets X_1, X_2, …, X_n, excluding the empty set. (n−r) indicates the number of sets without the negation symbol. $|X_1 \cap X_2, \ldots, \cap X_n|$ indicates the number of tuples $<x_1, x_2, \ldots, x_n>$ where $x_1 \in X_1$, $x_2 \in X_2$, …, $x_n \in X_n$.

D. Similarity Metric for Source Codes

In our study, the similarity measure has been tailored to measure the similarity of sequentially structured text. We first define the notion of a sequence. Let S_1 and S_2 be statements extracted by the structure extraction tool. $[S_1 \rightarrow S_2]$ denotes a sequence of S_1 followed by S_2. In general, for a positive integer n, let S_i (i ranges between 1 and n) be a statement. Thus, $[S_1 \rightarrow S_2 \rightarrow \ldots \rightarrow S_n]$ denotes a sequence of n statements.

The similarity of the DSRM can be considered the same as the extended Sorensen-Dice index except for symbols, i.e., using the → symbol in place of the ∩ symbol. The DSRM similarity between two sequences is defined as follows.

$$\mathrm{Sim}_{DSRM}([S_1 \rightarrow S_2 \rightarrow \cdots \rightarrow S_m], [T_1 \rightarrow T_2 \rightarrow \cdots \rightarrow T_n]) = \frac{n \mid [S_1 \rightarrow S_2 \rightarrow \cdots \rightarrow S_m], [T_1 \rightarrow T_2 \rightarrow \cdots \rightarrow T_n] \mid}{\sum_{r=0}^{r=n-1} (n-r) \mid\mid [S_1 \rightarrow S_2 \rightarrow \cdots \rightarrow S_m], SqcComb([T_1 \rightarrow T_2 \rightarrow \cdots \rightarrow T_n], r) \mid\mid} \quad (3)$$

Here, without loss of generality, we can assume that m ≥ n. In the case m < n, we replace the sequence $[S_1 \rightarrow S_2 \rightarrow \ldots \rightarrow S_m]$ with $[T_1 \rightarrow T_2 \rightarrow \ldots \rightarrow T_n]$.

The numerator of the definition, i.e., $| [S_1 \rightarrow S_2 \rightarrow \ldots \rightarrow S_m], [T_1 \rightarrow T_2 \rightarrow \ldots \rightarrow T_n] |$ indicates the number of statements in the sequence where $S_{j+1} = T_1$, $S_{j+2} = T_2$, … , $S_{j+n} = T_n$ for some j ($0 \le j \le m-n$). The denominator of the definition indicates the iteration of the sequence match that counts the sequence of statements from r = 0 to n−1. Note that the first sequence $[S_1 \rightarrow S_2 \rightarrow \ldots \rightarrow S_m]$ is renewed when the sequence match succeeds, i.e., replacing the matched statements with a not applicable symbol "n/a." *SqcComb* ($[T_1 \rightarrow T_2 \rightarrow \ldots \rightarrow T_n]$, r) generates a set of sequence combinations by replacing the r ($0 \le r < n$) statements with the negation of the statements.

A simplified version of the algorithm used for computing the DSRM similarity is shown in Figure 3. It takes a set of method structures M and a sequence $[T_1 \rightarrow T_2 \rightarrow \ldots \rightarrow T_n]$ as arguments, and returns an array of similarity values for the set of method structures.

We assumed that the *getMethodStructure(j)* function returns a structure of the *j*-th method extracted by the structure extraction tool. The function abstracts the implementation of the internal structure of the method, which is represented as a sequence of statements.

The *Count*(MS, TN, R) function returns the number of "positive statements" that matches the (n−R)-combinations of statement sequences TN in the method_structure MS. The *SqcComb*(TN, R) function generates (n−R)-combinations of statement sequences that replace the R statements with the negation of the statements in the sequence TN.

```
/* Datatype "method_structure" is a set of "sequence" of statements.
*  A "sequence" is represented by an array of statements.
*     Input:  set_of_method_structure M;
*     Input:  sequence [T1→T2→...→Tn];
*     Output: Sim[M.length];                    */
/*  --- Definition of the SimDSRM function  --- */
double[] SimDSRM( set_of_method_structure M, sequence [T1→T2→...→Tn])
{
 double Sim[M.length];
 int Nume;
 int Deno;
 for ( int j=0; j < M.length; j++ ) {
    Nume= Count( getMethodStructure(j), [T1→T2→...→Tn], 0 );
    Deno= 0;
    for ( int r=1; r < [T1→T2→...→Tn].length; r++ ){
       Deno= Deno + Count( getMethodStructure(j), [T1→T2→...→Tn], r );
    }
    if ( (Nume + Deno) == 0 ) { Sim[j]= -1; }
    else { Sim[j]= (double) Nume / (double)( Nume + Deno ); }
 }
 Return Sim;
}

/* --- Definition of the Count function. --- */
int Count( method_structure MS, sequence TN, int R)
{
 statement[] S;       // Type of S is a "sequence" of statements
 statement[][] DS;    // Type of DS is a set of a "sequence" of statements
 statement[] SV;      // Type of SV is a "sequence" of statements
 int CT=0;
 // Generate derived sequence replacing R statements with negations
 DS= SqcComb( TN, R );
 for ( each S[] ∈ MS){
   for ( int j=1; j<=MS.length–TN.length; j++ ){
     for ( each SV[] ∈ DS ){
       for ( int k=1; k<= TN.length–R; k++ ){
         if ( S[j] = SV[k] for all k-th elements that are not negative )
                 { CT= CT + (TN.length–R) };
       }
     }
   }
 }
 Return CT;
}
```

Figure 3. Algorithm to compute the similarity for the sequence $[T_1 \rightarrow T_2 \rightarrow \ldots \rightarrow T_n]$

IV. EXPERIMENTAL RESULTS

Table II shows omits some of the results of the retrieval experiments owing to space limitations. The retrieval condition is *if{ → IOException → }* (No.14 in Table I). Let a "boundary method" be a retrieved method whose DSRM similarity is greater than 0 and whose extended Sorensen-Dice index is the minimum among retrieved methods. The

 ISBN: 978-1-61208-319-3

read() method in the *NioBlockingSelector* class, which is shown at No.15 in Table II, is the boundary method. The number of retrieved methods for the extended Sorensen-Dice index is defined by the boundary method. The number of retrieved methods for DSRM similarity is defined by the number of methods with similarity greater than 0. The degree of improvement of DSRM over extended Sorensen-Dice index is calculated by the following formula.

$$\frac{(No.methods\ by\ Sorensen{-}Dice)-(No.methods\ by\ DSRM)}{(No.methods\ by\ Sorensen{-}Dice)} \quad (4)$$

For the retrieval condition *if{ → IOException → }*, the number of retrieved methods for the extended Sorensen-Dice index is 89, whereas the number of retrieved methods for DSRM similarity is 71. The degree of improvement is (89 −

TABLE II. SAMPLE OF RETRIEVAL EXPERIMENTS

No	Method Name	Derived Sequence Retrieval Model			Ext. Sorensen-Dice Model		
		Similarity	Exact Match	Partial Match	Similarity	Exact Match	Partial Match
1	SSIServletExternalResolver::getAbsolutePath()	0.857	6	1	0.857	6	1
2	InputBuffer::readByte()	0.750	3	1	0.750	3	1
4	WsOutbound::flush()	0.750	3	1	0.750	3	1
5	UpgradeAprProcessor::write()	0.750	3	1	0.750	3	1
6	SecureNioChannel::close()	0.667	6	3	0.667	6	3
7	Conversions::byteArrayToLong()	0.600	3	2	0.600	3	2
8	BioReceiver::start()	0	0	11	0.545	6	5
9	FileUtils::forceDelete(File file)	0	0	9	0.333	3	6
10	InputBuffer::skip(long n)	0.200	3	12	0.200	3	12
11	ClassParser::parse()	0.158	3	16	0.158	3	16
12	MemoryUserDatabase::save()	0.097	3	28	0.290	9	22
13	InternalAprInputBuffer::fill()	0	0	23	0.261	6	17
14	SecureNioChannel::read()	0	0	19	0.158	3	16
15	NioBlockingSelector::read()	0.103	3	26	0.103	3	26

TABLE III. SUMMARY OF 18 RETRIEVAL EXPERIMENTS

No	Retrieval Sequence	Number of Methods by Derived Sequence Retrieval Model	Number of Methods by Extended Sorensen-Dice Model	Degree of improvements
1	if{ → Log.debug	143	151	5.3%
2	if{ → Log.debug → }	141	151	6.6%
3	if{ → StringBuilder.append	102	143	28.7%
4	catch{ → Log.error	112	131	14.5%
5	catch{ → ExceptionUtils.handleThrowable	97	111	12.6%
6	if{ → IllegalArgumentException	107	133	19.5%
7	if{ → IllegalArgumentException → }	107	133	19.5%
8	catch{ → Log.error → }	87	128	32.0%
9	if{ → org.apache.juli.logging.Log.debug	70	73	4.1%
10	if{ → StringBuilder.append → }	66	141	53.2%
11	if{ → org.apache.juli.logging.Log.debug → }	68	71	4.2%
12	if{ → StringBuilder.append → StringBuilder.append	41	140	70.7%
13	if{ → IOException	71	89	20.2%
14	if{ → IOException → }	71	89	20.2%
15	catch{ → MBeanException	30	31	3.2%
16	catch{ → MBeanException → }	30	30	0.0%
17	if{ → StringBuilder.append → StringBuilder.append → }	37	135	72.6%
18	catch{ → Log.error → } → }	66	116	43.1%
			Average	23.9%

 ISBN: 978-1-61208-319-3

71) / 89 = 20.2%.

Note that there are some methods whose DSRM similarity is 0, whereas the extended Sorensen-Dice index similarity is greater than 0. This occurs when a program structure does not include a given sequence of statements but includes some elements of the statements. This means that the DSRM imposes a more severe retrieval condition than the extended Sorensen-Dice model. Consequently, the results of the DSRM are a subset of the results of the extended Sorensen-Dice model.

Table III shows a summary of 18 retrieval experiments using the retrieval conditions shown in Table I. Column three of Table III represents the number of methods retrieved by the DSRM with similarity values greater than 0. Column four shows the number of methods retrieved by the extended Sorensen-Dice model.

The degree of improvement ranges from 0% to 72.6% and is 23.9% on average over the extended Sorensen-Dice model. The 0% improvement occurs for the case No.16 in Table III. The retrieval condition for the case No.16 includes the term "*catch{ → MBeanException → }*," which is so rare in the collection of code that all *MBeanExceptions* are preceded by the *catch* clause. Thus, both retrieval models produce the same results. With the exception of No.16, the DSRM similarity outperformed the extended Sorensen-Dice index.

V. CONCLUSIONS

Many different similarity measures have been proposed to detect similar source code fragments. However, defining similarity measures should be carefully performed because the similarity measures may influence the detection of similar fragments more than other processes such as parsing structures, and normalizing identifiers.

Source code is essentially a sequence of statements; therefore, we have defined a similarity measure that is tailored to sequentially structured text to retrieve similar source code fragments. We also developed a mining algorithm to mine a set of sequences of statements with a frequency greater than a given threshold. Prior to similar source code retrieval, determining the frequency sequence of statements was essentially performed to issue a query to the point.

Our similarity measure was evaluated using Apache-Tomcat 7, which is a large-size open source Java program. The results show that the degree of improvement over the extended Sorensen-Dice model is on average 23.9% for the 18 retrieval conditions detected by our mining algorithm.

The results are sufficiently promising to warrant further research. In future, we intend to improve our algorithms by combining Java-specific information such as inheritance of a class and method overloading. We also plan to develop an improved user interface and conduct experiments using various types of open source programs available on the Internet.

REFERENCES

[1] B. S. Baker, "Parameterized Pattern Matching: Algorithms and Applications," Journal of Computer and System Sciences, Vol. 52, no. 1, February 1996, pp. 28-42.

[2] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier, "Clone Detection Using Abstract Syntax Trees," Proc. 14th International Conference on Software Maintenance, 1998, pp. 368-377.

[3] S. S. Choi, S. H. Cha, and C. C. Tappert, "A Survey of Binary Similarity and Distance Measures," Journal of Systemics, Cybernetics and Informatics ISSN 1690-4532, Vol. 8, no. 1, 2010, pp. 43-48.

[4] J. Gosling, B. Joy, G. Steele, and G. Bracha, "The Java Language Specification," 3rd Edition, Addison-Wesley, 2005.

[5] L. Jiang, G. Misherghi, Z. Su, and S. Glondu, "DECKARD: Scalable and Accurate Tree-based Detection of Code Clones," Proc. 29th International Conference on Software Engineering, May 2007, pp. 96-105.

[6] J. H. Johnson, "Identifying Redundancy in Source Code Using Fingerprints," Proc. 1993 Conference of the Centre for Advanced Studies Collaborative Research, October 1993, pp. 171-183.

[7] R. Komondoor and S. Horwitz, "Using Slicing to Identify Duplication in Source Code," Proc. 8th International Symposium on Static Analysis, LNCS Vol.2126, July 2001, pp. 40-56.

[8] K. A. Kontogiannis, R. Demori, E. Merlo, M. Galler, and M. Bernstein, "Pattern Matching for Clone and Concept Detection," Journal of Automated Software Engineering Vol. 3, June 1996, pp. 77-108.

[9] J. Krinke, "Identifying Similar Code with Program Dependence Graphs," Proc. 8th Working Conference on Reverse Engineering, October 2001, pp. 301-309.

[10] Z. Li, S. Lu, S. Myagmar, and Y. Zhou, "CP-Miner: Finding Copy-Paste and Related Bugs in Large-Scale Software Code," IEEE Transactions on Software Engineering, Vol. 32, no. 3, 2006, pp. 176-192.

[11] J. Mayrand, C. Leblanc, and E. M. Merlo, "Experiment on the Automatic Detection of Function Clones in a Software System Using Metrics," Proc. 12th International Conference on Software Maintenance, November 1996, pp. 244-253.

[12] J. R. Pate, R. Tairas, and N. A. Kraft, "Clone Evolution: A Systematic Review," Technical Report no. SERG-2010-01.R2, Department of Computer Science, the University of Alabama, August 2011, pp. 1-24.

[13] C. K. Roy, J. R. Cordya, and R. Koschkeb, "Comparison and Evaluation of Code Clone Detection Techniques and Tools: A Qualitative Approach, " Science of Computer Programming, Vol. 74, Issue 7, May 2009, pp. 470-495.

[14] O. Maqbool and H .A. Babri, "Hierarchical Clustering for Software Architecture Recovery," IEEE Transactions on Software Engineering, Vol. 33, Issue 11, November 2007, pp. 759-780.

[15] Y. Udagawa, "Source Code Retrieval Using Sequence Based Similarity," International Journal of Data Mining & Knowledge Management Process (IJDKP), Vol. 3, no. 4, July 2013, pp. 57-74.

[16] P. N. Tan, M. Steinbach, and V. Kumar, "Introduction to Data Mining, " 2006, Addison-Wesley.

[17] The Apache Software Foundation. Apache Tomcat. http://tomcat.apache.org/, November 2013.

[18] V. Wahler, D. Seipel, J. Wolff v. Gudenberg, and G. Fischer, "Clone Detection in Source Code by Frequent Itemset Techniques," Proc. 4th IEEE International Workshop Source Code Analysis and Manipulation, September 2004, pp. 128-135.

[19] T. Yamamoto, M. Matsushita, T. Kamiya, and K. Inoue, "Measuring Similarity of Large Software Systems Based on Source Code Correspondence," Proc. 6th International Conference on Product Focused Software Process Improvement, June 2005, pp. 530-544.

 ISBN: 978-1-61208-319-3