An Empirical Study on Retrieving Structural Clones Using Sequence Pattern Mining Algorithms

Yoshihisa Udagawa
Faculty of Engineering,
Tokyo Polytechnic University
1583 Iiyama, Atsugi-city, Kanagawa
243-0297 Japan
udagawa@cs.t-kougei.ac.jp

ABSTRACT

Many clone detection techniques focus on fragments of duplicated code, i.e., simple clones. Structural clones are simple clones within a syntactic boundary that are good candidates for refactoring. In this paper, a new approach for detection of structural clones in source code is presented. The proposed approach is parse-tree-based and is enhanced by frequent subsequence mining. It comprises three stages: preprocessing, mining frequent statement sequences, and fine-matching for structural clones using a modified longest common subsequence (LCS) algorithm. The lengths of control statements in a programming language and method identifiers differ; thus, a conventional LCS algorithm does not return the expected length of matched identifiers. We propose an encoding algorithm for control statements and method identifiers. Retrieval experiments were conducted using the Java SWING source code. The results show that the proposed data mining algorithm detects clones comprising 51 extracted statements. Our modified LCS algorithm retrieves a number of structural clones with arbitrary statement gaps.

Categories and Subject Descriptors

D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement–*Restructuring, reverse engineering, and reengineering*, H.2.8 [**Database Management**]: Database Applications –*Data mining*, H.3.3 [**Information Storage and Retrieval**]: Information Search and Retrieval –*Retrieval models*, K.6.3 [**Management of Computing and Information Systems**]: Software Management–*Software development, Software maintenance*

General Terms

Algorithms, Experimentation, Measurement, Management

Keywords

Structural clone, Java source code, Control statement, Method identifier, Frequent subsequence mining

1. INTRODUCTION

Previous work in this field has asserted that cloning code is generally a bad practice, and that cloned code should be removed or at least detected. Several studies have suggested that as much as 20–50% of large software systems comprise cloned code [1][2]. Programming through copy-and-paste is the fastest way to develop software; thus, programmers prefer reusing code rather than rewriting similar code from scratch. However, cloned code results in significantly more complicated and costly software maintenance. For example, when an error is identified in one copy, the programmer must check all other copies to make parallel changes. In addition, cloned code can make software systems more difficult to understand because the essential difference in two pieces of code may be unclear.

In many clone detection studies, cloned code resulting from a copied code fragment has been compared based on textual similarities. The following types of cloned code have been identified [3].

Type 1 is an exact copy without modification, with the exception of layout and comments.

Type 2 is a syntactically identical copy, with the exception of variations in identifiers, literals, types, layout, and comments.

Type 3 is a copy with modifications, such as changed, added, or removed code, in addition to variations in identifiers, literals, types, layout, and comments.

Structural clones can be any of the three types (i.e., simple clones) including the syntactic boundaries of a particular language [3] [4]. In Java, these boundaries are as follows.

Declaration: class, method, constructor
Control statement: *if*-statement, *for*-statement, *while*-statement, *try*-statement, *synchronized*-statement
Block range surrounded with '{' and '}'

Parse-tree-based or fine-grained clone detection approaches [5][6][7] are heavily dependent on fine-grained parsers and find clones on abstract syntax tree (AST). Most of these approaches aim to find Type 3 clones. However, they are not scalable to large programs. Li et al. [8] proposed a token-based approach named CP-Miner. CP-Miner modifies a frequent subsequence mining algorithm to find Type 3 clones. Roy and Cordy [9] proposed a reasonably lightweight language specific parser-based clone detection approach to detect Type 3 clones at the block level. Their approach uses pretty-printing to standardize formatting and a longest common subsequence (LCS) algorithm to compare the text of potential clones. Basit et al. [4] introduced the concept of structure clones and proposed mining techniques to detect potential clones in software. In their approach, simple source code clones are extracted and then fed to a finding frequent closed itemset process to detect recurring groups of simple clones in different files or methods.

The proposed method is a parse-tree-based approach enhanced by frequent subsequence mining that comprises three steps: preprocessing, mining frequent statement sequences, and fine-matching for structural clone retrieval using a modified LCS algorithm. The proposed algorithm was applied to the Java SWING source code. Our experimental results show that the SWING source code includes cloned code consisting of up to 51 extracted statements. To the best of our knowledge, this paper is the first to report detection of clones of such length and a number of Type 3 clones.

The remainder of this paper is organized as follows. In Section 2, we present a source code preprocess to extract relevant Java code fragments. In Section 3, we present a mining algorithm for probing program structures. In Section 4, we present a modified LCS algorithm with a back-tracking function for detecting Type 3 clones. Experimental results are discussed in Section 5, and Section 6 presents conclusions.

2. PREPROCESSING

In the proposed approach, Java source code is initially partitioned into methods. Then, code matching statements are extracted for each method. The extracted statements comprise class method signatures, control statements, and method calls.

(1) Class method signatures
Each Java method is declared in a class. The proposed parser extracts class method signatures using the following syntax.

<class identifier>::<method signature>

An anonymous class, which is a local class without a class declaration, is often used when a local class is used only once. An anonymous class is defined and instantiated in a single expression using the new operator to produce concise code. The proposed parser extracts a method declared in an anonymous class using the following syntax.

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

Arrays and generics are widely used in Java to facilitate manipulation of data collections. The proposed parser also extracts arrays and generic data types according to Java syntax. For example, Object[], String[][], List<String>, and List<Integer> are extracted and treated as unique data types.

(2) Control statements
The proposed parser also extracts control statements with various levels of nesting. A block is represented by the "{" and "}" symbols. Thus, the number of "{" symbols indicates the number of nesting levels. The following Java keywords for control statements are processed by the proposed parser.

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

(3) Method calls
Under the assumption that a method call characterizes a program, the proposed 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. The proposed parser traces a type declaration of a variable and translates a variable identifier to its data type or class identifier:

<variable>.<method identifier>

is translated into

<data type>.<method identifier>
or
<class identifier>.<method identifier>.

The Java SDK is an integrated development environment for writing Java applications, including applets, beans, and security. The Java SDK was developed by Sun Microsystems. We selected the Java SDK 1.7.0.45 SWING package as our target because the size of the source code is fairly large. The major software metrics are as follows:

Number of Java Files	----	739
Number of Classes	----	1,883
Number of Code Lines	----	191,646
Number of Comment Lines	----	147,137
Number of Blank Lines	----	36,284
Number of Total Lines	----	372,807

Relative to the number of lines, the SWING package is classified as large-scale software in the IT industry.

Figure 1 shows an example of the extracted structure of *DebugGraphics::drawImage (Image img, int x, int y, Image Observer observer)* in the *DebugGraphics.java* file of the *javax.swing* package. The three numbers preceded by the # symbol are the number of comments, and blank and code lines, respectively. The extracted structures include control statement nesting depth; thus, they provide sufficient information for retrieving methods using source code substructure.

```
DebugGraphics::drawImage
  (Image img, int x, int y, ImageObserver observer)
# 5        3         36
{
         if{
         DebugGraphicsInfo.log()
         }
         if{
                  if{
                  debugGraphics()
                  Graphics.drawImage()
                  Graphics.dispose()
                  }
         }
         else_if{
         img.getSource()
         FilteredImageSource()
         Toolkit.getDefaultToolkit()
         DebugGraphicsObserver()
                  for{
                  loadImage()
                  Graphics.drawImage()
                  Toolkit.getDefaultToolkit()
                  sleep()
                  }
         }
         return
}
```

Figure 1. Example of extracted structures

3. MINING FREQUENT PROGRAM STRUCTURES

Initially, we mine source code structures using the algorithm shown in Figure 2. This algorithm shares many concepts with the well-known *apriori* algorithm for mining frequent itemsets [10] (Figure 3). The proposed algorithm takes the minimum support number *minSup* as an argument and has control structures that are similar to those of the *apriori* algorithm. However, the proposed algorithm essentially deals with a sequence of statements while the *apriori* algorithm deals with a set of items.

```
/* Mining Statement Sequences that begin with a control statement.  */
List<statement[]> MS;    // MS is a list of statement sequences
                         // that are extracted by our lexical parser.
int k;                   // k defines the number of repetitions.
int minSup;              // minSup defines the minimum number of occurrences.
List<statement[]> LS;   // LS is a sequence list as the result of the algorithm.
List<statement[]> F ;   // F is a sequence list for a given repetition.
List<statement[]> C ;   // C is a candidate sequence list for a given repetition.
Map<statement c, int n> ;  //  c ∈$C_k$  is candidate sequence of $C_k$ , and
                           //   n is the number of  c's occurrences.
 LS= Φ;
 k=1;
 $F_k$= { i | i ∈{Control Statements} };
 repeat
    k=k+1
    $C_k$= Retrieve_Candidates( MS, $F_{k-1}$ ); // A set of sequences that begins $F_{k-1}$
    $F_k$= Φ;
    for ( each c ∈ $C_k$ && substring(c) ∈ MS  ) {
          Map( substring(c), n+1 );    //  count the number of  its occurrences
    }
    if ( at least one element of c is not a control statement
       && |c| ≧ minSup) {
          LS.add(c);
          $F_k$.add(c) };
    }
 } until $F_k$= Φ;
 Result= LS;
```

Figure 2. Algorithm for mining frequent sequences

```
/*  APRIORI Algorithm  */
Set<statement[]> T ;     //  T  is given itemsets (Transaction).
int minSup;              //  minSup defines the minimum number of occurrences.
int k;                   //  k defines the number of repetitions.
Set<statement[]> LS ;    // LS is itemset as the result of the algorithm.
Set<statement[]> F ;     // F is itemset for a given repetition.
Set<statement[]> C ;     // C is candidate itemset for a given repetition.
Map<statement c, int n> ;  //  c ∈$C_k$  is a candidate item of $C_k$ , and
                           //   n is the number of  c's occurrences.
LS= Φ;
k= 1
$F_k$= { i | i ∈ {frequent items of length 1} };
repeat
     k = k+1;
     $C_k$ = apriori_gen($F_{k-1}$ );
     $F_k$= Φ;
     for ( each c ∈ $C_k$ && c ∈ T  ) {
          Map( c, n+1 );    //  count the number of  its occurrences
     }
     if (  c ∈ T  &&  |c| ≧ minSup ) {
          LS.add(c);
          $F_k$.add(c) };
     }
}  until $F_k$ = Φ;
Result = LS;
```

Figure 3. *Apriori* algorithm

There are three primary differences between the two algorithms. First, the initial condition is different. In the proposed algorithm, the initial sequence F_1 is set by a set of "control statements," i.e., *if, else if, else, switch, while, do, for, break, continue, return, throw, synchronized, try, catch*, and *finally* statements. This concept is derived from the assumption that most important methods are called in a control structure. Thus, we set "control statements" as an initial sequence. In the *apriori* algorithm, the initial set comprises frequent items of length 1 because it deals with a set of items.

The second difference relates to the generation of next candidates. In the proposed algorithm, next candidates are retrieved from a given set of program structures using a current sequence as a key. In contrast, the *apriori* algorithm generates next candidates by combining all possible elements.

The third difference relates to screening processes. In the proposed algorithm, screening conditions are at least one element of a candidate sequence that is not a control statement with a sequence frequency that is greater than *minSup*. In the *apriori* algorithm, the only screening condition is that the frequency of an item must be greater than a given *minSup*. These differences are considered a customization of source code sequence retrieval.

The proposed algorithm is designed to find a set of frequently occurring sequences. Note that several matches can be detected in a sequence for a subsequence given as a matching condition. For example, the two matches of subsequence A→B are detected in sequence A→B→A→C→A→B→D. The *Retrieve_Candidates (MS, F_{k-1})* function shown in Figure 2 finds a set of sequences of k statements that includes (k–1) statements found in the previous iteration in method structures extracted from Java source code "MS."

4. FINE-MATCHING USING LCS ALGORITHM

LCS is an algorithm commonly used in clone detection. Roy and Cordy [9] used an LCS to define a similarity metric. Duszynski et al.[11] used an LCS algorithm to compute information about common code elements. The proposed method employs an LCS algorithm to detect a structural clone precisely, i.e., it is used to identify relevant statements between given statement sequences.

The conventional LCS algorithm [12] takes two given strings as input and compares each single character of the strings. However, the length of statements in program code differs; thus, the conventional LCS algorithm does not work well. In other words, for short statements, such as *if, for,* and *try* statements, the LCS algorithm returns small LCS values for matching. For long statements, such as *synchronized* statements or a long method identifier, the conventional LCS algorithm returns large LCS values, which result in an unfair base for a similarity metric in clone detection.

We have developed an encoder that converts a statement to three 32-decimal digits, which allows encoding of up to 32,768 identifiers. The encoder comprises two steps: (1) gathering all unique identifiers and numbering them in three 32-decimal digits and (2) replacing each identifier in a statement sequence with a number encoded in three 32-decimal digits.

Figure 4 shows the fragments of a "3-digit identifier" list generated from the Java SWING source code. Figure 5 shows an example of an encoded statement sequence that corresponds to the extracted structures shown in Figure 1.

We modified the conventional LCS algorithm to handle three characters as a unit of matching. We also implemented a backtrack function that returns the positions of the matched identifiers in the sequences.

001, {	0J4, getColor()
002, if{	0J5, Toolkit.getDefaultToolkit()
003, synchronized{	0J6, sleep()
004, Boolean.valueOf()	…
005, }	0JP, DebugGraphicsInfo.log()
006, return	0JQ, Graphics.drawImage()
…	0JR, img.getSource()
00O, for{	0JS, FilteredImageSource()
…	0JT, DebugGraphicsObserver()
08J, Graphics.dispose()	0JU, loadImage()
0J2, debugGraphics()	0JV, ImageIcon.loadImage()
0J3, Graphics.drawRect()	

Figure 4. Fragments of "3-digits identifier" list

```
if{→DebugGraphicsInfo.log()→}→if{→if{→debugGraphics()→G
raphics.drawImage()→Graphics.dispose()→}→}→else_if{→img.g
etSource()→FilteredImageSource()→Toolkit.getDefaultToolkit()→
DebugGraphicsObserver()→for{→loadImage()→Graphics.drawIm
age()→Toolkit.getDefaultToolkit()→sleep()→}→}→return→}
```

(A) Before encoding (corresponding to Figure 1)

```
002→0JP→005→002→002→0J2→0JQ→08J→005→005→018
→0JR→0JS→0J5→0JT→00O→0JU→0JQ→0J5→0J6→005→
005→006→005
```

(B) After encoding

Figure 5. Example of encoded statement sequence

5. EXPERIMENTAL RESULTS

5.1 Code Normalization

We conducted experiments that comprised three stages: (1) preprocessing to normalize source code, (2) mining frequent statement sequences in the normalized code, and (3) fine-matching for a structural clone with arbitrary gaps using the modified LCS algorithm.

The target source code is the Java SDK 1.7.0.45 SWING package. Through the preprocessing described in Section 2, statements that include control statements and original source code method identifiers are extracted to form normalized code that is a collection of statement sequences. Some features of the normalized code are as follows:

The maximum length of the method is 248, which is obtained from the *parseTag()* method of the *javax.swing.text.html.parser* package.

The maximum nesting level of the method is nine, which is obtained from the *JComponent.java* method of the *javax.swing* package.

After screening methods without control statements or method calls, the normalized code consist of 9,234 methods and 6,310 unique identifiers.

5.2 Mining Frequent Sequences

Table 1 shows the total number and maximum length of the retrieved sequences, as well as the elapsed time in milliseconds for each *minSup* for the number of sequences (2–40). The results show that the elapsed time is roughly proportional to the total number of retrieved sequences. We measured elapsed time using the following experimental environment:

CPU: Intel Core i7-3770 3.40 GHz
Main memory: 16 GB
OS: Windows 8 64 Bit
Programming Language: Java 1.7.0

Table 1. Summary of frequent sequence mining

minSup	Total No.Sequences	Max Length	Elapsed Time
2	5229	51	25,221
3	1704	24	8,567
4	1017	24	5,696
5	580	24	3,354
6	413	24	2,713
7	259	17	1,749
8	224	14	1,755
9	182	11	1,419
10	173	11	1,334
11	153	10	1,274
12	138	10	1,155
13	130	10	1,170
14	125	10	1,087
15	120	10	1,064
16	117	10	1,059
17	112	10	1,002
18	110	10	1,015
19	106	10	968
20	99	10	931
21	98	10	919
22	91	10	928
23	88	10	881
24	83	10	911
25	81	10	864
30	66	8	771
35	17	4	490
40	10	4	460

Table 2. Summary of frequent itemset mining

minSup	Total No.Item Sets	Max Cardinality	Elapsed Time
2	N/A	N/A	> 40Hr
3	209,234	16	650,732
4	146,739	16	305,901
5	76,601	15	112,667
6	49,871	15	66,266
7	5,636	10	20,806
8	4,086	10	13,960
9	3,558	10	6,694
10	3,200	10	5,317
11	2,887	10	3,989
12	2,675	10	3,504
13	2,507	10	3,004
14	2,347	10	2,582
15	2,240	10	2,566
16	2,142	10	2,301
17	2,016	10	1,941
18	1,952	10	1,815
19	1,896	10	1,815
20	1,080	8	1,407
21	1,037	8	1,267
22	977	8	1,159
23	925	8	1,252
24	893	8	1,049
25	845	8	1,018
30	520	6	750
35	787	6	656
40	348	6	512

Table 2 shows a summary of the results for the frequent itemsets mining using the *apriori* algorithm implemented in the Java language [13]. The table shows the total number of frequent itemsets and maximum cardinality of the itemsets, as well as elapsed time in milliseconds for each *minSup* in the number of sequences (2–40). In the frequent itemsets mining, two statements, i.e., { and } are excluded because these statements appear in all methods; thus, they do not work well for mining frequent itemsets.

Compared to the *apriori* algorithm, the proposed algorithm for mining frequent sequences demonstrates comparable performance when the minimum support number *minSup* is greater than 25. However, when *minSup* is less than 25, the proposed algorithm outperforms the *apriori* algorithm. The *apriori* algorithm fails to mine frequent itemsets for *minSup* of 2, even after running for 40 hours in the experimental environment.

Figure 6 shows a mined sequence whose length is 51. The sequence has two method occurrences, i.e., the *getAfterIndex(int part, int index)* method in the *AbstractButton.java* file and the *getAfterIndex(int part, int index)* method in the *JLabel.java* file of the *javax.swing* package. These two methods are exactly the same, with the exception of the class names or Java file names.

```
switch{→case→if{→return→}→try{→return→}→catch{→
return→}→case→try{→getText()→BreakIterator.getWordIn
stance()→BreakIterator.setText()→BreakIterator.following()
→if{→return→}→BreakIterator.following()→if{→return→
}→return→}→catch{→return→}→case→try{→getText()→
BreakIterator.getSentenceInstance()→BreakIterator.setText()
→BreakIterator.following()→if{→return→}→BreakIterator.
following()→if{→return→}→return→}→catch{→return→}
→default→return→}→}
```

Figure 6. Mined sequence (length is 51)

Figure 7 shows the source code for the *getAfterIndex* methods. Note that the source code comprises 53 lines whereas the mined sequence comprises 51 lines. This indicates that the control statements and method calls characterize a Java program, which was the initial assumption of our study.

Figure 8 shows a mined sequence whose length is 24. The sequence has six method occurrences in the *DebugGraphics.java* file of the *javax.swing* package, i.e., *drawImage(Image img, int x, int y, ImageObserver observer), drawImage(Image img, int x, int y, int width, int height, ImageObserver observer), drawImage(Image img, int dx1, int dy1, int dx2, int dy2, int sx1, int sy1, int sx2, int sy2, Color bgcolor, ImageObserver observer),* etc.

Figure 9 shows the source code of *drawImage(Image img, int x, int y, ImageObserver observer)*. These six methods in *DebugGraphics.java* are considered clones because, to implement method overloading, the arguments of the methods called in each *drawImage* method differ only slightly.

```
public String getAfterIndex(int part, int index) {
        if (index < 0 || index >= getCharCount()) {
            return null;
        }
        switch (part) {
        case AccessibleText.CHARACTER:
            if (index+1 >= getCharCount()) {
              return null;
            }
            try {
                return getText(index+1, 1);
            } catch (BadLocationException e) {
                return null;
            }
        case AccessibleText.WORD:
            try {
                String s = getText(0, getCharCount());
                BreakIterator words =
                    BreakIterator.getWordInstance(getLocale());
                words.setText(s);
                int start = words.following(index);
                if (start == BreakIterator.DONE || start >= s.length()) {
                    return null;
                }
                int end = words.following(start);
                if (end == BreakIterator.DONE || end >= s.length()) {
                    return null;
                }
                return s.substring(start, end);
            } catch (BadLocationException e) {
                return null;
            }
        case AccessibleText.SENTENCE:
            try {
                String s = getText(0, getCharCount());
                BreakIterator sentence =
                    BreakIterator.getSentenceInstance(getLocale());
                sentence.setText(s);
                int start = sentence.following(index);
                if (start == BreakIterator.DONE || start > s.length()) {
                    return null;
                }
                int end = sentence.following(start);
                if (end == BreakIterator.DONE || end > s.length()) {
                    return null;
                }
                return s.substring(start, end);
            } catch (BadLocationException e) {
                return null;
            }
        default:
            return null;
        }
    }
```

Figure 7. Source code of the *getAfterIndex* methods

```
if{→DebugGraphicsInfo.log()→}→if{→if{→debugGraphics
()→Graphics.drawImage()→Graphics.dispose()→}→}→else
_if{→img.getSource()→FilteredImageSource()→Toolkit.get
DefaultToolkit()→DebugGraphicsObserver()→for{→loadIm
age()→Graphics.drawImage()→Toolkit.getDefaultToolkit()→
sleep()→}→}→return→}
```

Figure 8. Mined sequence (length is 24)

```
public boolean drawImage(Image img, int x, int y,
                              ImageObserver observer) {
   DebugGraphicsInfo info = info();
   if (debugLog()) {
      info.log(toShortString() +
            " Drawing image: " + img +
            " at: " + new Point(x, y));
   }
   if (isDrawingBuffer()) {
      if (debugBuffered()) {
         Graphics debugGraphics = debugGraphics();
         debugGraphics.drawImage(img, x, y, observer);
         debugGraphics.dispose();
      }
   } else if (debugFlash()) {
      int i, count = (info.flashCount * 2) - 1;
      ImageProducer oldProducer = img.getSource();
      ImageProducer newProducer
         = new FilteredImageSource(oldProducer,
                  new DebugGraphicsFilter(info.flashColor));
      Image newImage
         = Toolkit.getDefaultToolkit().createImage(newProducer);
      DebugGraphicsObserver imageObserver
         = new DebugGraphicsObserver();
      Image imageToDraw;
      for (i = 0; i < count; i++) {
         imageToDraw = (i % 2) == 0 ? newImage : img;
         loadImage(imageToDraw);
         graphics.drawImage(imageToDraw, x, y,
                     imageObserver);
         Toolkit.getDefaultToolkit().sync();
         sleep(info.flashTime);
      }
   }
   return graphics.drawImage(img, x, y, observer);
}
```

Figure 9. Source code of a *drawImage* method

5.3 Structural Clone Retrieval Using LCS

The frequent statement sequences indicate the existence of the given sequences with the number of occurrences, as well as their location in the source code. However, programmers would need to know all fragments that partially match a statement sequence. To address a partial match of a statement sequence, we modified the LCS as described in Section 4.

Table 3 shows the experimental results obtained using the modified LCS. The retrieval condition or query is as follows:

if{→JComponent.getWriteObjCounter()→
JComponent.setWriteObjCounter()→if{→ui.installUI().

We know from frequent sequence mining that the query statement sequence has 34 occurrences. The modified LCS allows us to retrieve the other five fragments with up to 24 gaps. The similarity values in Table 3 were computed using the following formula:

$$\text{Similarity} = \frac{Length\ of\ query}{Length\ of\ query + Gap}$$

Here, the query length is 5.

Table 3. Experimental results using modified LCS

No. Occurences	Gap	Total Length	Similarity
34	0	5	1.000
0	1	6	0.833
0	2	7	0.714
1	3	8	0.625
0	4	9	0.556
0	5	10	0.500
0	6	11	0.455
0	7	12	0.417
0	8	13	0.385
1	9	14	0.357
0	10	15	0.333
1	11	16	0.313
0	12	17	0.294
0	13	18	0.278
0	14	19	0.263
0	15	20	0.250
0	16	21	0.238
0	17	22	0.227
0	18	23	0.217
0	19	24	0.208
0	20	25	0.200
0	21	26	0.192
1	22	27	0.185
0	23	28	0.179
1	24	29	0.172

Note that the proposed algorithm requires no input, with the exception of the query and a collection of normalized code. The maximum length of the normalized method in Java SWING is 248 (Section 5.1); thus, a desktop PC has sufficient memory to compute an LCS of this size.

If a maintenance programmer does not wish to retrieve a long gapped clone, the proposed method can cope with such a requirement with only slight modification to the proposed LCS algorithm.

6. CONCLUSINONS

We have presented a method to identify similar structural code. The proposed approach is parse-tree-based and comprises three steps: preprocessing, mining frequent statement sequences, and fine-matching using a modified LCS algorithm.

In an experiment using the Java SDK 1.7.0.45 SWING source code, we extracted 9,234 methods and 6,310 unique identifiers. The proposed sequence mining algorithm successfully locates cloned code. To the best of our knowledge, this paper is the first to report detection of long clones (length of 51). The experimental results show that the elapsed time required to mine frequent statement sequences is roughly proportional to the total number of retrieved sequences. Experimental results show that, with *minSup* in the number of sequences equaling 2, the proposed method could mine all possible clones. Although this case incurs a very heavy processing load, the proposed algorithm detected 5,229 clone candidates within 26 seconds.

After identifying simple Type 1, Type 2, and Type 3 clones, a higher-level structural clone could be composed to find design level similarities. In this sense, our work is still in progress, and some obstacles remain. First, higher-level structural clones may include patterns of recurring components at the architectural level, i.e., source code patterns applied repeatedly by programmers to solve similar problems. An algorithm to relate simple clones to higher-level structural clones is a challenging subject, especially for data mining and clustering techniques. Second, higher-level structural clones may coexist among multiple programming languages. Currently, only Java source code can be analyzed; however, extending the proposed system to support other languages could be implemented easily. In the proposed approach, the only language-dependent components are lexical analysis and code normalization. Thus, once normalized code is generated, the remaining processes would be identical for different programming languages.

7. REFERENCES

[1] Baker, B. S. 1995. On finding duplication and near-duplication in large software systems. In *Proceedings of the 2nd Working Conference on Reverse Engineering* (July 1995), 86-95,

[2] Ducasse, S, Rieger, M., and Demeyer, S. 1999. A language independent approach for detecting duplicated code. In *Proceedings of the International Conference on Software Maintenance* (Sep. 1999), 109-118.

[3] Roy, C.K. and Cordy J.R. 2007. A survey on software clone detection research. *Queen's Technical Report:541.* Queen’s University at Kingston, Ontario, Canada (Sep.2007), 1-115.

[4] Basit, H.A. and Jarzabek, S. 2009. A Data Mining Approach for Detecting Higher-level Clones in Software. *IEEE Transactions on Software Engineering* 35, 4 (Feb. 2009), 497-514.

[5] Baxter, I. D., Yahin, A., Moura, L., Sant'Anna, M., and Bier, L. 1998. Clone Detection Using Abstract Syntax Trees. In *Proceedings of the 14th International Conference on Software Maintenance* (Nov 1998), 368-377.

[6] Krinke, J. 2001. Identifying Similar Code with Program Dependence Graphs. In *Proceedings of the 8th Working Conference on Reverse Engineering* (Oct. 2001), 301-309.

[7] Koschke, R., Falke, R., and Frenzel, P. 2006. Clone Detection Using Abstract Syntax Suffix Trees. In *Proceedings of the 13th Working Conference on Reverse Engineering* (Oct. 2006), 253 - 262.

[8] Li, Z., Lu, S., Myagmar, S., and Zhou, Y. 2004. CP-Miner: A Tool for Finding Copy-paste and Related Bugs in Operating System Code. In *Proceedings of the 6th Symposium on Operating System Design and Implementation* (Dec, 2004), 289-302.

[9] Roy, C.K. and Cordy J.R. 2008. NICAD: Accurate Detection of Near-Miss Intentional Clons Using Flexible Pretty-Printing and Code Normalization. In *Proceedings of the 16th IEEE International Conference on Program Comprehension* (June 2008), 172-181.

[10] Agrawal, R., Imielinski, T., and Swami, A.N. 1993. Mining association rules between sets of items in large databases. In *Proceedings of the 1993 ACM SIGMOD international conference on Management of data* (1993), 207-216.

[11] Duszynski, S., Knodel, J., and Becker, M. 2011. Analyzing the Source Code of Multiple Software Variants for Reuse Potential. In *Proceedings of the 18th Working Conference on Reverse Engineering* (Oct. 2011), 303-307.

[12] Longest common subsequence, 2014. http://rosettacode.org/wiki/Longest_common_subsequence, (Sept. 2014).

[13] Nathan Magnus and Su Yibin. 2009. Apriori Implementation. http://www2.cs.uregina.ca/~dbd/cs831/notes/itemsets/itemset_prog1.html.