Sequence Data Mining Approach for Detecting Type-3 Clones

Yoshihisa Udagawa and Mitsuyoshi Kitamura
Computer Science Department, Faculty of Engineering,
Tokyo Polytechnic University
Atsugi-city, Kanagawa, Japan
e-mail: {udagawa, kitamura}@cs.t-kougei.ac.jp

Abstract— **Code clones are introduced to source code by changing, adding, and/or deleting statements in copied code fragments. Thus, the problem of finding code clones is essentially the detection of strings that partially match. The proposed algorithm is based on the well-known apriori principle in data mining and is tailored to detect code clones represented as sequences of strings. However, the apriori principle may generate too many sequential patterns. The proposed algorithm finds a compact representation of sequential patterns, known as maximal frequent sequential patterns, which is often two orders of magnitude smaller than frequent sequential patterns. Early experiments using the *Java SDK 1.7.0.45 lang* package demonstrate the number of extracted patterns and elapsed time in several contexts.**

Keywords-component; Code clone; Maximal frequent sequence; Longest common subsequence(LCS) algorithm; Java source code.

I. INTRODUCTION

Copying and pasting similar code (or code clones) is very common in large software since it can significantly reduce programming effort and time. However, code clones complicate software maintenance. For example, when an error is identified in one copy, the same error can occur in the code clones. Thus, a maintenance programmer must check all code clones to ensure parallel changes.

Generally, the detection process for code clones comprises two phases, i.e., transformation and matching [1].

(1) Transformation phase: parts of interest of the source code are transformed to another intermediate representation for ease of matching.
(2) Matching phase: the intermediate representation units are compared to find a match.

Because copying and modifying statements are common programming practices, finding code clones that partially match is a challenging task from both practical and technical perspectives. Partially-matching code clones are referred to as type-3 clones, gapped clones, and near-miss clones in code clone detection literature [1]. The term "gap" refers to nothing-match or non-match elements that comprise two code clone candidates.

A number of approaches have been developed for detecting code clones. State-of-the-art research is divided into two categories. The first category is dedicated to code clone detection. Ducasse et al. [2] defined and assessed six degrees of transformation with regard to varying gap sizes of zero, one, and two. Because of limitations of scalability, they restricted themselves to a gap size of zero in some case studies. Roy et al. [3] proposed a near-miss clone detection method called Accurate Detection of Near-miss Intentional Clones (NICAD). NICAD combines language-sensitive parsing with language-independent similarity analysis using an optimized longest common subsequence (LCS) algorithm [4] to detect code clones. Murakami et al. [5] proposed a new token-based method that detects gapped code clones using a local sequence alignment algorithm, i.e., the Smith–Waterman algorithm. They discussed a sophisticated trace back algorithm tailored for code clone detection.

The other category focuses on frequent sequence mining techniques to detect code clones and code change patterns. CP-Miner [6] employs an extended version of CloSpan [7] to support gap constraints in frequent subsequences. It tolerates one to two statement insertions, deletions, or modifications in copy-pasted code. Negara et al. [8] developed a sophisticated data mining algorithm that effectively detects frequent code change patterns. They also identified 10 types of popular high-level code change patterns from mined code change patterns.

The main idea of the proposed approach is a combination of frequent sequence mining and the LCS algorithm to detect type-3 clones.

A sequence is called a frequent sequence if it appears in a given sequence database with a frequency no less than a user-specified threshold (i.e., minSup). Although several algorithms have been proposed for frequent sequence mining, such as CloSpan, ClaSP, and CM-ClaSP [9], one of the drawbacks of these algorithms is that they can present a very large number of sequential patterns. A sequential pattern is maximal if immediate super-sequences are frequent [10]. The maximal sequential patterns are generally a small subset of frequent sequential patterns. The proposed approached employs maximal sequential pattern mining to discover a compact set of clone candidates.

The main contributions of this paper are as follows: (1) development of a code transformation parser that extracts code matching statements; (2) development of a matching algorithm that efficiently detects type-3 clones using a tailored sequential pattern mining algorithm; (3) evaluation of the proposed algorithm using the *Java SDK 1.7.0.45 lang* package with several parameters; and (4) performance comparison of the proposed algorithm to previous methods.

The remainder of this paper is organized as follows. Section 2 presents an overview of the proposed approach. Section 3 describes the proposed algorithm, which discovers clone candidates using maximal frequent sequence mining. Section 4 presents the results of an experimental study using the *Java SDK lang* package. Finally, Section 5 concludes the paper and provides suggestions for future work.

 ISBN: 978-1-61208-458-9

II. Overview of Our Approach

Our goal is to detect method pairs or sets that share common code fragments. 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 [11]. Our approach consists of the following four steps (Figure 1).

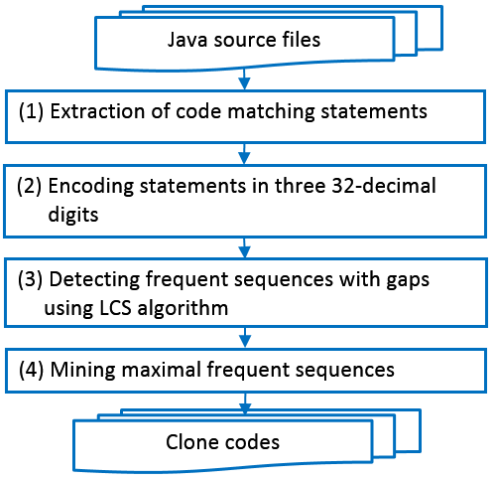

Figure 1. Overview of the proposed approach.

A. Extraction of code matching statements

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 as follows.

<variable>.<method identifier>
is translated into
<data type>.<method identifier>
or
<class identifier>.<method identifier>.

We have developed a parser that 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 15 control statements are processed by the proposed parser.

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

We selected the *Java SDK 1.7.0.45 lang* package as our target. The number of total lines is 67,677. Figure 2 shows an example of the extracted structure of the *encode(char[] ca, int off, int len)* method in the *StringCoding.java* file of the *java.lang* 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 the structure of the source code.

```
StringEncoder::encode(char[] ca, int off, int len)
#     2     0     25
{
scale()
        if{
        return
        }
        if{
        return
        }
        else{
        CharsetEncoder.reset()
        ByteBuffer.wrap()
        CharBuffer.wrap()
                try{
                CharsetEncoder.encode()
                        if{
                        CoderResult.throwException()
                        }
                CharsetEncoder.flush()
                        if{
                        CoderResult.throwException()
                        }
                }
                catch{
                Error()
                }
        return
        }
}
```

Figure 2. Example of the extracted structure.

In this study, we only deal with Java. However, extraction of code matching statements can allow our approach to be independent of programming languages, such as C/C++ and Visual Basic.

B. Encoding statements in three 32-decimal digits

The conventional LCS algorithm takes two given strings as input and compares each character of the strings. However, the length of statements in program code differs; thus, the conventional LCS algorithm does not work effectively. In other words, for short statements, such as *if* 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 LCS algorithm returns large LCS values.

We have developed an encoder that converts a statement to three 32-decimal digits, which results in a fair base for a similarity metric in clone detection. Figure 3 shows the encoded statements that correspond to the code shown in Figure 2. Figure 4 shows the mapping table between three 32-decimal digits and a code matching statement extracted from the original source files.

```
StringEncoder::encode(char[] ca, int off, int len)→001→
13V→005→004→003→005→004→003→00C→14F→
141→142→00V→14G→005→144→003→14H→005→
144→003→003→011→07F→003→004→003→003
```

Figure 3. Encoded statements corresponding to Figure 2.

 ISBN: 978-1-61208-458-9

```
001, {                          13V, scale()
002, super()                    ...
003, }                          141, ByteBuffer.wrap()
004, return                     142, CharBuffer.wrap()
005, if{                        143, CharsetDecoder.decode()
...                             144, CoderResult.throwException()
00C, else{                      ...
...                             14F, CharsetEncoder.reset()
00V,try{                        ...
...
```

Figure 4. Mapping table between three 32-decimal digits and a code matching statement.

C. *Detecting frequent sequences with gaps*

We have developed a mining algorithm to find frequent sequences based on the apriori principle [12]. 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 proposed algorithm detects the two matches of subsequence A→B in sequence A→B→A→C→A→B→D.

The LCS algorithm is tailored to match three 32-decimal digits as a unit. The LCS algorithm can match two given sequences even if "gaps" (nothing-match or non-match elements) exist. Given two sequences of matching strings S1 and S2, let |lcs| be the length of their longest common subsequence, and let |S1| and |S2| be the length of S1 and S2, respectively. The "gap size" gs is defined as gs= |lcs| − min(|S1|, |S2|).

D. *Mining maximal frequent sequences*

Frequent sequences mining can result in a very large number of sequential patterns, which makes it difficult for users to analyze the results. Mining maximal frequent sequences addresses a drawback of frequent sequences mining [10]. We have developed an algorithm to discover maximal frequent sequences. Note that our approach deals with gapped sequences; thus, it requires a tailored technique to filter non-maximal frequent sequences.

III. PROPOSED FREQUENT SEQUENCE MINING

This section outlines the proposed frequent sequence mining algorithm and shows some examples that demonstrate how the algorithm works.

A. *Proposed Frequent Sequence Mining Algorithm*

The proposed approach is based on frequent sequence mining. A subsequence is considered frequent when it occurs no less than a user-specified minimum support threshold (i.e., minSup) in the sequence database. Note that a subsequence is not necessarily contiguous in an original sequence.

We assume that a sequence is a list of items, whereas several algorithms for sequential pattern mining [9] deal with a sequence that consists of an ordered list of "itemsets." Our assumption is rational because we focus on detecting code clones. In addition, the assumption simplifies the implementation of the proposed algorithm, which makes it possible to achieve high performance (Section 4).

The proposed frequent sequence mining algorithm comprises two methods, i.e., GProve (Figure 5) and Retrieve_Cand (Figure 6). It follows the key idea behind apriori; if a sequence S in a sequence database appears at least N times, so does every subsequence R of S.

```
1  GProve(String[] args){
2     k= 1;
3     Initialization: Set the 15 control statements of Java to LinkedList<String> Sk
4     do {
5        Retrieve_Cand();   // Find a set of sequences of length k+1 that matches Sk.
6                           // Store the set of sequences in Ck.
7        k= k+1;
8        Sk.clear();        //  Clear Sk in order to store frequent sequences of length k.
9        while( For all elements e in Ck ){
10          if ( Frequency of  e  >= minSup ){
11                Print e and sequence ids of e in the database to output file.
12                Add e and the sequence ids to Sk;
13                Scan the database to find a set of gap synonyms of e;
14                Add each gap synonym and the sequence ids to Sk;
15                }
16          }
17       }
18    } while (Sk.size() > 0);
19 }
```

Figure 5. Frequent sequence detection of the proposed algorithm.

```
1  Retrieve_Cand(){
2     Ck.clear();
3     for (each element s in Sk) {
4        for (each element t in the sequence database) {
5           for ( each position p that s matches in t){
6              Compute LongestCommonSubsequence between s and t at position p;
7              if (match count >= k && gap count  <=  maxGap ){
8                 Put s and frequency of s to Ck;
9                 Extract gap synonym g of s from t.
10                Put g and frequency of s to Ck;
11                }
12             }
13          }
14       }
15 }
```

Figure 6. Candidate sequences retrieval for the next repetition.

The variable k indicates the count of the repetition (line 2, Figure 5). LinkedList < String > Sk is initialized to hold 15 control statements. The Retrieve_Cand method (line 5, Figure 5) discovers a set of sequences of length k+1 from a sequence database that matches statement sequences in Sk. The while loop (lines 9–17) finds frequent sequences and sequence IDs in a sequence database.

Lines 12–14 maintain the frequent sequences. Note that the proposed algorithm handles gapped sequences, and both a frequent sequence and its "gap synonyms" are prepared for the next repetition. Here, "gap synonyms" means a set of sequences that match a given subsequence under a given gap constraint.

Generally, the Retrieve_Cand() method in Figure 6 works as follows. HashMap <String, Integer> Ck holds a sequence (String) and its frequency (Integer). First, Ck is cleared (line 2, Figure 6). The three for loops examine all possible matches between an element in Sk and sequences in a sequence database. The longest common subsequence algorithm is tailored to compute the match count and gap

 ISBN: 978-1-61208-458-9

count (line 6, Figure 6). The if statement, (line 7, Figure 6) screens a sequence based on the match count and gap count. Lines 8–10 maintain the frequency of sequences and its "gap synonyms."

B. Extracting Frequent Sequences

In our approach, we assume a program structure is represented as a sequence of statements preceded by a class-method ID. Each statement is encoded to three 32-decimal digits so that the LCS algorithm correctly works regardless of the length of the original program statement. The proposed algorithm is illustrated for the given sample sequence database in Figure 7. MTHD# is an abbreviated notation for a class-method ID.

```
MTHD1→005→003
MTHD2→005→00A→003→003
MTHD3→005→003→00F→006→005→003
MTHD4→005→006→003→005→00C
```

Figure 7. Example sequence database.

Figure 8 shows the result of the frequent sequences for a gap of 0 and minSup of 50%, which is equivalent to a minSup count that equals 2. "005" is a frequent sequence with a minSup count of 6 because "005" occurs once in the first and second sequences and twice in the third and fourth sequences. The proposed algorithm maintains an ID-List, which indicates the positions a frequent sequence appears in a sequence database. The ID-List for "005" is 1|2|3+3|4+4.

Similarly, 005→003→ is a frequent sequence with a minSup count of 3, i.e., the ID-List for 005→003→ is 1|3+3.

```
005→          N=6（1|2|3+3|4+4）
005→003→      N=3（1|3+3）
```

Figure 8. Result of the frequent sequences (gap, 0; minSup, 50%).

Figure 9 shows the result of the frequent sequences for a gap of 1 and minSup of 50%. "005" is a frequent sequence with a minSup count of 6, which is the same in the case of a gap of 0.

Similarly, 005→003→ is a frequent sequence with a minSup count of 5. In addition to the consecutive sequence 005 → 003 → , the proposed algorithm detects gapped sequences. In the case of 005→003→, the algorithm detects 005→00A→003→ in the second sequence and 005→006→003→ in the fourth sequence. Thus, the ID-List for 005→003→ is 1|2|3+3|4.

```
005→          N=6（1|2|3+3|4+4）
005→003→      N=5（1|2|3+3|4）
```

Figure 9. Result of the frequent sequences (gap, 1; minSup, 50%).

Figure 10 shows the result of the frequent sequences for a gap of 2 and minSup of 50%. In addition to 005→ and 005→003→, 005→006→ is detected as a frequent sequence because 005→003→00F→006→ in the third sequence matches 005→006→ with a gap of 2, and 005→006→ in the fourth sequence with a gap of 0. Thus, the ID-List for 005→006→ is 3|4.

```
005→          N=6（1|2|3+3|4+4）
005→003→      N=5（1|2|3+3|4）
005→006→      N=2（3|4）
```

Figure 10. Result of the frequent sequences (gap, 2; minSup, 50%).

C. Extracting Maximal Frequent Sequences

A frequent sequence is a maximal frequent sequence and no supersequence of it is a frequent sequence. The set of maximal frequent sequence is often several orders of magnitude smaller than the set of all sequential patterns. In addition, it is representative because it can be used to recover all frequent sequences. Several algorithms for finding maximal frequent sequences and/or itemsets employ sophisticated search and pruning techniques to reduce the number of sequence and/or itemset candidates during the mining process.

However, we wish to measure the effects of a maximal frequent sequence; therefore, the proposed algorithm first extracts a set of frequent sequences and then detects a set of maximal frequent sequences.

Note that, since we deal with a gapped sequence, screening a maximal frequent sequence is required to check that none of the immediate super-sequences of gap synonyms, which are a set of sequences that match a given subsequence under the gap constraint, is a frequent sequence.

IV. Experimental Results

We present some measures on frequent sequences or candidate clones, time analysis, and findings relating to the *Java SDK 1.7.0.45 lang* package. Some features of the code are as follows.

1. After screening methods without control statements or method calls, the normalized code consist of 2,522 methods, 18,205 identifiers, and 1,286 unique identifiers.
2. The maximum length method is *isCallerSensitiveMethod()* (127 lines), which is obtained from the *java.lang.invoke.MethodHandleNatives.java* file.
3. The maximum method nesting level is seven, which is obtained from the *getEnclosingMethod()* method of the *java.lang.Class.java* file.

A. Maximum Length of Retrieved Sequences

The proposed algorithm can retrieve sequences that satisfy an arbitrary gap size specified by the user. Figure 11 summarizes the maximum lengths of the retrieved sequences for each minSup and gap size. As minSup decreases, the filtering condition lessens; thus, the maximum lengths increase. As the gap size increases, the matching condition lessens; thus, the maximum lengths increase. The results in Figure 11 show that the maximum

 ISBN: 978-1-61208-458-9

lengths reach 120 when minSup is 2 and gap size is no less than 2.

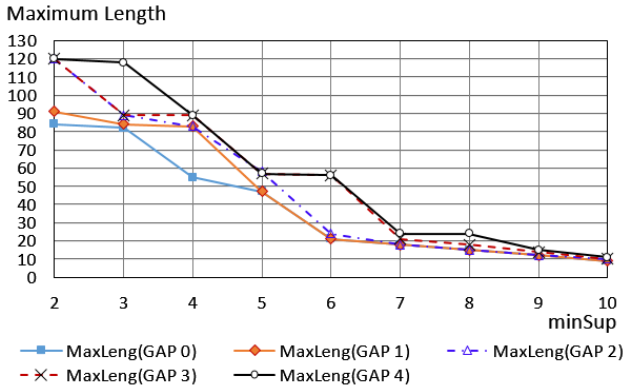

Figure 11. Maximum length of retrieved sequences for each minSup and gap size.

B. *Numbers of Retrieved Sequences*

Figure 12 shows the number of retrieved frequent sequences with respect to gap (0 to 4) and minSup (2 to 10). As expected, the number of retrieved frequent sequences increases as gap increases and minSup decreases. The proposed algorithm can find frequent sequences that occur at least twice in the sequence database, which is necessary for finding all possible code clones. Note that the numbers of retrieved frequent sequences for a gap of 0 are plotted on the right secondary axis because they are 1/7 to 1/60 of the numbers of retrieved frequent sequences for a gap of 1 to 4.

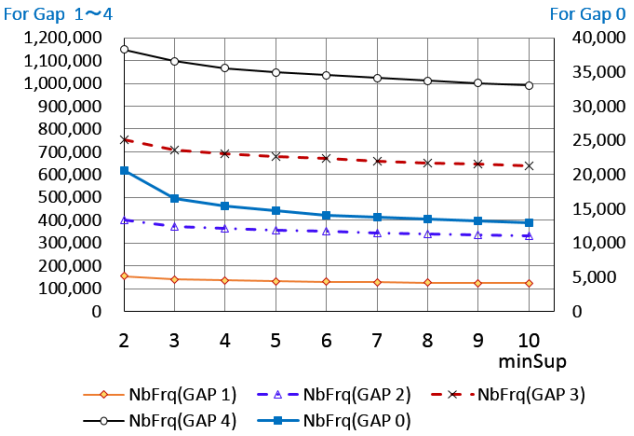

Figure 12. Numbers of retrieved frequent sequences (gap size, 0 and 1-4; minSup, 2-10).

Figure 13 shows the number of maximal retrieved frequent sequences with respect to a gap of 0 to 4 and minSup of 2 to 10. As expected, the number of maximal retrieved frequent sequences is a compact representation of the set of frequent sequences, which is approximately one to two orders of magnitude smaller than that of frequent sequences. The ratio of the number of frequent sequences to the number of maximal frequent sequences increases as gap increases. For example, the ratio is approximately 100, which is the largest obtained ratio, when gap is 4 and minSup is 2.

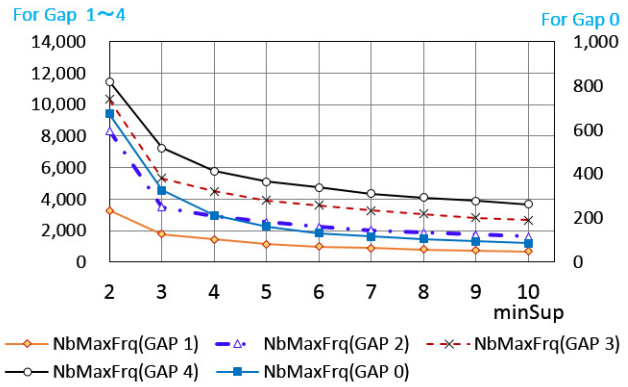

Figure 13. Numbers of retrieved maximal frequent sequences (gap size, 0 and 1-4; minSup, 2-10).

C. *Time Analysis*

Figure 14 shows the elapsed time in milliseconds for retrieving frequent sequences. The x-axis indicates minSup. Note that the elapsed time for a gap of 0 is plotted on the right secondary axis. We measured elapsed time using the following experimental environment.

CPU: Intel Core i3-540 3.07 GHz
Main memory: 8 GB
OS: Windows 7 64 Bit
Programming Language: Java 1.7.0

The proposed algorithm can retrieve frequent sequences fairly efficiently. For example, it takes 289,481 milliseconds to identify 154,789 frequent sequences for a gap of 1 and minSup of 2. Note that elapsed time increases as the gap increases. The results show that the elapsed time is approximately 4.8 × N × t, where N is the number of gaps, and t is the elapsed time for a gap of 0.

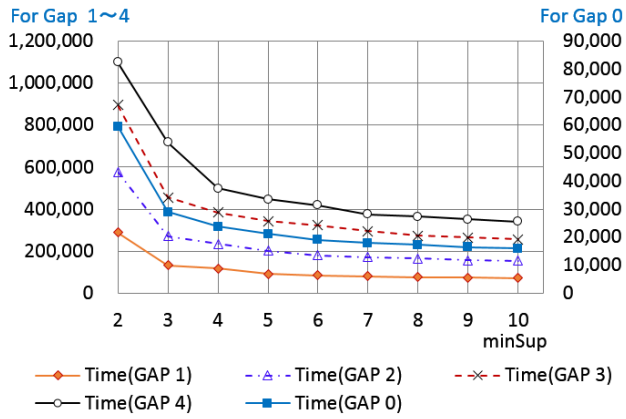

Figure 14. Elapsed time (milliseconds) for retrieving frequent sequences.

The maximal sequential pattern (MaxSP) and vertical mining of maximal sequential patterns (VMSP) produce a set of maximal sequential patterns. However, the runnable code of MaxSP downloaded from an open-source data mining library [9] fails to process the sequence database due to an overly long process time. For VMSP, it processes in less than 100 ms with a very small set of maximal sequential patterns that is approximately three orders of magnitude smaller than the expected set of patterns.

Figure 15 shows a comparison of elapsed time for the proposed algorithm with respect to gap 0, as well as ClaSP and CM-ClaSP, which produce the best available results. Note that ClaSP and CM-ClaSP terminate with a stack overflow error when minSup is less than 7.

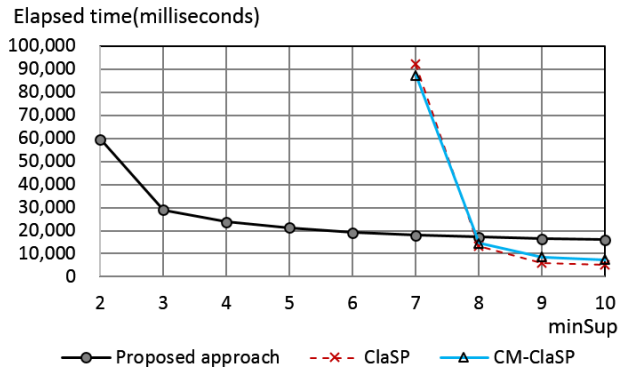

Figure 15. Comparison of elapsed time.

Figure 16 shows elapsed time in milliseconds for retrieving maximal frequent sequences. The input is a list of retrieved frequent sequences, and the output is a list of maximal frequent sequences. The elapsed time is nearly proportional to the number of maximal retrieved frequent sequences in Figure 13. The elapsed time for extracting a list of maximal frequent sequences is 1/160 of that for retrieving a set of frequent sequences and is nearly independent of gap size.

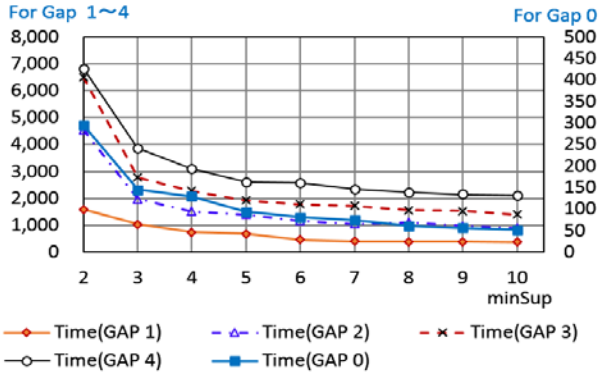

Figure 16. Elapsed time for retrieving maximal frequent sequences.

D. *Source Code Findings*

By increasing gap size to greater than 2, we can relax the gap constraint; however, this is detrimental to the relevance of retrieved sequence occurrences. We limit ourselves to a gap size of 1 to simplify analysis.

1	StringDecoder::decode(byte[] ba, int off, int len)→001→13V→005→004→003→005→004→003→00C→140→141→142→00V→143→005→144→003→145→005→144→003→003→011→07F→003→004→003→003
2	StringDecoder::decode(Charset cs, byte[] ba, int off, int len)→001→13T→13V→005→004→003→005→005→0E0→003→003→14B→005→004→003→00C→141→142→00V→143→005→144→003→145→005→144→003→003→011→07F→003→004→003→003
3	StringEncoder::encode(char[] ca, int off, int len)→001→13V→005→004→003→005→004→003→00C→14F→141→142→00V→14G→005→144→003→14H→005→144→003→003→011→07F→003→004→003→003
4	StringEncoder::encode(Charset cs, char[] ca, int off, int len)→001→14E→13V→005→004→003→005→005→0E0→003→003→14J→005→004→003→00C→141→142→00V→14G→005→144→003→14H→005→144→003→003→011→07F→003→004→003→003

Figure 17. Four clone candidate methods.

Figure 17 shows a set of four methods that match a sequence 005 → 004 → 003 → 00C → 141 → 142 → 00V → within a gap of 1. These four methods are defined in the *StringCoding.java* file of the *java.lang* package. They are considered clones because the arguments in the *encode* and *decode* methods differ only slightly in order to implement method overloading. In addition, they share the sequence 005→004→003→00C→141→142→00V→. Note that only the *encode(char[] ca, int off, int len)* method (Figure 2) matches the sequence 005→004→003→00C→141→142→00V→ with one gap (i.e., "14F" or *CharsetEncoder.reset()*), as shown in the third row of Figure 17. This difference is considered to be worthy of checking by the implementer of the methods.

V. CONCLUSINONS AND FUTURE WORK

We have presented an approach to identify type-3 clones using a source code parsing technique to extract matching code statements, a maximal frequent sequence mining algorithm, and a modified LCS algorithm for computing the matching degree and gaps of corresponding code segments.

Early experiments using the *Java SDK 1.7.0.45 lang* package indicate that the algorithms can identify type-3 clones in a reasonable elapsed time. The experimental results show that the ratio of the number of the frequent sequences to the number of maximal frequent sequences reaches approximately 100. However, the proposed algorithm still generates thousands of maximal frequent sequences. Therefore, we plan to improve the proposed algorithm in future.

 ISBN: 978-1-61208-458-9

REFERENCES

[1] C. K. Roy and J. R. Cordy, "A survey on software clone detection research," Queen's Technical Report:541. Queen's Uni-versity at Kingston, Ontario, Canada, Sep. 2007, pp.1-115.

[2] S. Ducasse, O. Nierstrasz, and M. Rieger, "On the effectiveness of clone detection by string matching," Journal of Software Maintenance and Evolution Research And Practice, Jan. 2006, pp.37–58.

[3] C. K. Roy and J. R. Cordy, "NICAD: Accurate detection of near-miss intentional clons using flexible pretty-printing and code normalization," Proc. 16th IEEE International Conference on Program Comprehension, June 2008, pp.172-181.

[4] J. W. Hunt and T. G. Szymanski, "A fast algorithm for computing longest common subsequences," Comm. ACM, Vol.20, Issue.5, May 1977, pp.350-353.

[5] H. Murakami, K. Hotta, Y. Higo, H. Igaki, and S. Kusumoto, "Gapped code clone detection with lightweight source code analysis," May 2013, pp.93-102.

[6] Z. Li, S. Lu, S. Myagmar, and Y. Zhou, "CP-Miner: A tool for finding copy-paste and related bugs in operating system code," Proceedings of the 6th Symposium on Operating System Design and Implementation, Dec, 2004, pp.289-302.

[7] X. Yan, J. Han, and R. Afshar, "CloSpan: Mining closed sequential patterns in large datasets," Proc. 3rd SIAM International Conference on Data Mining (SDM'03), May, 2003, pp.166-177.

[8] S. Negara, M. Codoban, D. Dig, and R. E. Johnson: Mining Fine-Grained Code Changes to Detect Unknown Change Patterns, Proc. 36th International Conference on Software Engineering(ICSE 2014), May 2014, pp.803-813.

[9] "An Open-Source Data Mining Library," http://www.philippe-fournier-viger.com/spmf/index.php, v0.96r20, August 2015.

[10] P. Fournier-Viger, C.-W. Wu, A. Gomariz, and V. S. Tseng, "VMSP: Efficient vertical mining of maximal sequential patterns," Proc. 27th Canadian Conference on Artificial Intelligence (AI 2014), Springer, Advances in Artificial Intelligence, Lecture Notes in Computer Science,Vol. 8436, May 2014, pp. 83-94.

[11] Y. Udagawa, "A novel technique for retrieving source code duplication," Proc. 9th International Conference on Systems (ICONS 2014), Vol. 9, Feb. 2014, pp.172-177.

[12] R. Agrawal and R. Srikant, "Fast algorithms for mining associa-tion rules," Proc. 20th International Conference on Very Large Data Bases(VLDB), 1994, pp.487-499.

 ISBN: 978-1-61208-458-9