Effects of an Apriori-based Data-mining Algorithm for Detecting Type 3 Clones

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

***Abstract*—A code clone is a fragment of source code that appears at least twice in software source code. Code clones introduce difficulties in software maintenance because an error in one fragment is reproduced in code clones. It is significant to detect every code clones for making software maintenance easy and reliable.**

This paper describes software clone detection techniques using an *Apriori*-based sequential data mining algorithm. The *Apriori*-based algorithm is used because it is designed to find all frequent items that occur no less than a user-specified threshold named the minimum support (minSup). Since clones are slightly modified by adding, removing, or changing source code in general, the algorithm for detecting code clones has to deal with both match and mismatch portions of source code. The essential idea of the proposed approach is a combination of a partial string match using the longest-common-subsequence (LCS) and an *Apriori*-based algorithm for finding frequent sequences.

Generally, *Apriori*-based algorithms extract vast numbers of frequent sequences especially when the minSup is small, creating an obstacle to the detection of code clones. The novelties of our approach include pruning processes that depend on characteristics of a programming language, techniques to reduce the number of frequent sequences, and functions to control repetitive subsequences. We evaluate the effectiveness of the proposed algorithm based on experimental results using the source code of the *Java SDK SWING* graphics package. The results show that the proposed sequential data mining algorithm maintains the performance at a practical level until the minSup reaches two. This paper also shows some mined sequences and source code to demonstrate that the proposed algorithm works from short sequences to long ones.

***Keywords*—Code clone; Apriori-based algorithm; Maximal frequent sequence; Longest common subsequence(LCS) algorithm; Java source code.**

I. INTRODUCTION

Two or more fragments of source code that are identical or similar to each other are named code clones. Programmers can reuse software code to speed up development, especially when similar functionality is already implemented in programs. Code clones are very common in large software, because they can significantly reduce programming effort and shorten programming time. However, code clones are believed to be harmful in the quality management across the software life cycle, especially in the maintenance phase. Code clones complicate software maintenance, because an error in a cloned fragment is reproduced in every copy. In other words, if there are many code clones in software source code, and a bug is found in one code clone, a programmer must check and update all of its instances consistently.

Techniques detecting similar code patterns including code clones help programmers understand the software code with less pain. Accumulated code patterns can lead to programming knowledge about a particular application. Techniques to detect code clones can be an essential part of programming knowledge extraction. The programming knowledge can enhance the performance of a project team enabling to provide high-quality software on schedule.

Since code clones are a set of similar fragments that appear at least twice in source code, the problem of finding code clones is essentially the detection of a set of string sequences that partially match and appear at least twice. The previous studies of Udagawa [1][2] propose a sequential data mining algorithm for string sequences based on an *Apriori* principle [3]. This paper enhances the previous studies through applying the algorithm to a large scale software, i.e., the *Java SDK SWING* graphics package.

A number of approaches have been developed to detect code clones based on textual similarity, three types of cloned code have been identified [4]. Type 1 is an exact copy without modification, with the exception of layout and comments. Type 2 is a slightly different copy typically resulting from renaming of variables or constants. Type 3 is a copy with further modifications typically resulting from adding, removing, or changing code units. Since Type-3 clones are generated by modifying original code units, there are mismatch portions of code when the clones and its original code units are compared. The mismatch portion is referred to as a "gap" in this paper.

Research on Type 3 clones has been conducted in recent decades, because there are substantially more significant clones of Type 3 than those of Types 1 or 2 in software for industrial applications. Our approach also focuses on finding Type 3 clones.

The following issues have to be addressed in finding this type of clone.

(1) How to handle gaps in pattern matching.
There are many algorithms that are tailored to handle gaps in similarity measures, such as sequence alignment,

dynamic pattern matching, tree-based matching, and graph-based matching techniques [4][5].

(2) How to find frequently occurring patterns.
The detection of frequently occurring patterns in a set of sequence data has been researched thoroughly, as reported in the sequential pattern-mining literature [3][6]-[10]. There are several studies [11]-[13] using an *Apriori*-based algorithm to discover code clones in source code.

Code clones are defined as a set of syntactically and/or semantically similar fragments of source code [4][5]. Since source code consists of a sequence of statements, finding clone code can be achieved by finding similar sequences that occur at least twice. *Apriori*-based sequential pattern-mining algorithms are worth studying, because they are especially designed to detect a set of frequently occurring sequences. The algorithms take a positive integer threshold set by a user called "minimum support" or "minSup" for short. The choice of minSup controls the level of frequency [3][10].

In previous studies [1][2], Udagawa shows that repeated structures in a method adversely affect the performance, especially when the minSup is two or three using *Java SDK 1.8.0_101 awt* [1] and *Apache Struts 2.5.2 Core* [2]. This paper builds on those studies using the large-scale software *Java SDK 1.8.0_101 SWING* and analyzes to what extent the minSup affects the number of retrieved sequences and time performance. For this purpose, the proposed *Apriori*-based sequential mining algorithm is properly revised to deal with the repeated structures in a method.

The contributions of this paper are as follows:

(I) design and implementation of a code transformation parser that extracts code matching statements, including control statements and typed method calls;

(II) design and implementation of a sequential data-mining algorithm that maintains performance at a practical level until the threshold minSup reaches two;

(III) evaluation of the proposed algorithm using the *Java SDK 1.8.0_101 SWING* graphics package with respect to a minSup of two to ten and a gap size of zero to three. In addition to time performance, the number of mined sequences is analyzed for each length of sequences showing that the number of repeated structures in a method accounts for a large part of the mined sequences, especially in the case when the minSup is two.

The remainder of the paper is organized as follows. After presenting some basic definitions and terminology on frequent sequence mining techniques in Section II, we gave an overview of the proposed approach in Section III. Section IV describes the proposed algorithm for discovering clone candidates using an *Apriori*-based maximal frequent sequence mining technique. Section V presents the experimental results using the *Java SDK 1.8.0_101 SWING* package with some mined sequences and source code. Section VI presents some of the most related work. Section VII concludes the paper with our plans for future work.

II. BASIC DEFINITIONS

Definition 1 (sequence and sequence database). A sequence is an enumerated collection of items in which repetitions are allowed. A sequence database is a set of the sequences. Formally, they are defined as follows.

Let $I = \{ i_1, i_2, \ldots, i_h \}$ be a set of items (symbols). A sequence s_x is an ordered list of items $s_x = x_{j1} \rightarrow x_{j2} \rightarrow \ldots \rightarrow x_{jn}$ such that $x_{jk} \subseteq I$ $(1 \leq jk \leq h)$. A sequence database (SDB) is a set of sequences $SDB = \langle s_1, s_2, \ldots, s_p \rangle$ having sequence identifiers (SIDs) 1, 2, …, p.

Definition 2 (sequence containment). The notion of containment between two sequences is a key concept to characterize matching of the sequences. The sequence containment is defined by the identical or match items that are consecutively paired between two sequences. The concept of sequence containment is formalized as follows.

A sequence $s_a = a_1 \rightarrow a_2 \rightarrow \ldots \rightarrow a_n$ is said to be contained in a sequence $s_b = b_1 \rightarrow b_2 \rightarrow \ldots \rightarrow b_m$ $(n \leq m)$ iff there exists a strictly increasing sequence of integers q taken from [1, n], $1 \leq q[1] < q[2] < \ldots < q[n] \leq m$ such that $a_1 = b_{q[1]}$, $a_2 = b_{q[2]}$, …, $a_n = b_{q[n]}$ (denoted as $s_a \sqsubseteq s_b$).

Definition 3 (gapped sequence containment). A gap is a non-identical or mismatch item that consists either or both of the sequences. The concept of a gapped sequence containment is essential for detecting Type-3 clones because they are generated through modifications of adding, removing, or changing code units that cause gaps between two sequences. The gapped sequence containment is formally defined using a threshold, i.e., maxGap, that specifies the maximum length of non-identical or mismatch items.

Let maxGap be a threshold set by a user. A sequence $s_a = a_1 \rightarrow a_2 \rightarrow \ldots \rightarrow a_n$ is said to be contained in a sequence $s_b = b_1 \rightarrow b_2 \rightarrow \ldots \rightarrow b_m$ with respect to maxGap iff $a_1 = b_{q[1]}$, $a_2 = b_{q[2]}$, …, $a_n = b_{q[n]}$ and $q[j]–q[j–1]–1 \leq maxGap$ for all $2 \leq j \leq n$.

Definition 4 (prefix and postfix with respect to maxGap). A sequence can be divided into two subsequences, i.e., prefix and postfix, according to the concept of the gapped sequence containment.

A sequence $s_a = a_1 \rightarrow a_2 \rightarrow \ldots \rightarrow a_n$ is called a prefix of a sequence $s_b = b_1 \rightarrow b_2 \rightarrow \ldots \rightarrow b_m$ $(n \leq m)$ iff s_a is a gapped sequence containment of s_b with maxGap. A subsequence $s'_b = b_{n+1} \rightarrow \ldots \rightarrow b_m$ is called a postfix of s_b with respect to prefix s_a, denoted as $s_b = s_a \rightarrow s'_b$.

Definition 5 (support count with respect to maxGap). The support count of a sequence is defined by the number of its occurrences that appear in a sequence database. Since gaps in a sequence increase the chance of matching, the support count depends on maxGap.

Given a value of maxGap, the support count of a sequence s_b in a sequence database SDB with respect to maxGap is defined as the number of sequences $s \in SDB$ such that s_b is a gapped sequence containment of s with respect to maxGap and is denoted by $sup_{maxGap}(s_b)$.

Definition 6 (multi-occurrence mode and single-occurrence mode). Given a value of maxGap and a sequence $s_b = b_1 \rightarrow b_2 \rightarrow \ldots \rightarrow b_m$ with a prefix s_a, the

sequence s_b has a support of $sup_{maxGap}(s_b)$ that is greater than zero.

When the prefix s_a is contained in a postfix of s_b, i.e., $s'_b = b_{n+1} \rightarrow \ldots \rightarrow b_m$, the support is calculated as $sup_{maxGap}(s_b) + 1$.

This calculation is applied recursively for each postfix of s_b to determine the support number. The support number calculated recursively is referred to as the support number in *multi-occurrence mode* in this paper. This mode is critical when dealing with long sequences, such as nucleotide DNA sequences [6][7], and periodically repeated patterns over time [8]. The support number without the calculation of the postfix of s_b is referred to as the support number in *single-occurrence mode*. The algorithm proposed in the paper supports both of these modes.

Definition 7 (frequent sequences with maxGap). Let maxGap and minSup be thresholds set by a user. A sequence s_b is referred to as a frequent sequence with respect to maxGap iff $sup_{maxGap}(s_b) \geq$ minSup. The problem of sequence mining on a sequence database SDB is to discover all frequent sequences for given integers maxGap and minSup.

Definition 8 (closed frequent sequence). A closed frequent sequence is defined to be a frequent sequence for which there exists no super sequence that has the same support count as the original sequence [10][14].

Definition 9 (maximal frequent sequence). A maximal frequent sequence is defined to be a frequent sequence for which none of its immediate super sequences are frequent [9][10]. The maximal frequent sequence is valuable, because it provides the most compact representation of frequent sequences [14].

The closed frequent sequence is widely used when a system is designed to generate an association rule [10] that is inferred from a support number of a frequent sequence.

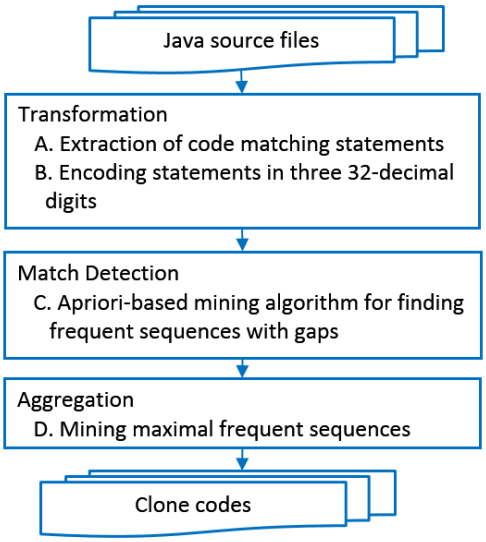

Figure 1. Overview of the proposed approach

III. Overview of Proposed Approach

Fig. 1 provides an overview of the proposed approach. According to the terminology in the survey [4], our approach can be summarized in three steps, transformation, match detection, and aggregation.

A. Extraction of code matching statements

Under the assumption that method calls and control statements characterize a program, the proposed parser extracts them 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. The translation allows us to deal with Type 2 clones.

<variable>.<method identifier>

is translated into

<data type>.<method identifier> or

<class identifier>.<method identifier>.

The parser extracts control statements with various levels of nesting. A block is represented by the "{" and "}" symbols. Thus, the number of "{" symbols indicate 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 1.8.0_101 SWING* package as the target of our study. The total number of lines is 372,186, qualifying the *SWING* package as a kind of large-scale software in the industry.

Fig. 2 shows an example of the extracted structure of the *paintContentBorderBottomEdge(Graphics g, int tabP, int sIndex, int x, int y, int w, int h)* method in the *BasicTabbedPaneUI.java* file of the *javax.swing.plaf.basic* package. The three numbers preceded by the # symbol are the numbers of comments, blank lines, and code lines, respectively. Fig. 3 shows the source code of the *paintContentBorderBottomEdge()* method whose matching statements are shown in Fig. 2.

```
BasicTabbedPaneUI::paintContentBorderBottomEdge
        (Graphics g, int tabP, int sIndex, int x, int y, int w, int h)
#               5       0       24
{
getTabBounds()
g.setColor()
        if{
        g.drawLine()
        g.setColor()
        g.drawLine()
        }
        else{
        g.drawLine()
        g.setColor()
        g.drawLine()
                if{
                g.setColor()
                g.drawLine()
                g.setColor()
                g.drawLine()
                }
        }
}
```

Figure 2. Example of an extracted structure

```
protected void paintContentBorderBottomEdge(Graphics g,
                                    int tabPlacement,
                                     int selectedIndex,
                                     int x, int y, int w, int h) {
      Rectangle selRect = selectedIndex < 0? null :
                        getTabBounds(selectedIndex, calcRect);
      g.setColor(shadow);
      // Draw unbroken line if tabs are not on BOTTOM, OR
      // selected tab is not in run adjacent to content, OR
      // selected tab is not visible (SCROLL_TAB_LAYOUT)
      //
      if (tabPlacement != BOTTOM || selectedIndex < 0 ||
          (selRect.y - 1 > h) ||
          (selRect.x < x || selRect.x > x + w)) {
         g.drawLine(x+1, y+h-2, x+w-2, y+h-2);
         g.setColor(darkShadow);
         g.drawLine(x, y+h-1, x+w-1, y+h-1);
      } else {
         // Break line to show visual connection to selected tab
         g.drawLine(x+1, y+h-2, selRect.x - 1, y+h-2);
         g.setColor(darkShadow);
         g.drawLine(x, y+h-1, selRect.x - 1, y+h-1);
         if (selRect.x + selRect.width < x + w - 2) {
            g.setColor(shadow);
            g.drawLine(selRect.x + selRect.width, y+h-2, x+w-2, y+h-2);
            g.setColor(darkShadow);
            g.drawLine(selRect.x + selRect.width, y+h-1, x+w-1, y+h-1);
         }
      }
}
```

Figure 3. Source code corresponding to Fig. 2

In this study, we deal only with *Java*. However, a slight modification of the parser allows us to apply the proposed approach to other languages such as C/C++ and Visual Basic.

B. *Encoding statements in three base-32 digits*

The conventional longest-common-subsequence (LCS) algorithm [15] takes two given strings as input and returns values depending on the number of matching characters of the strings. Because the length of statements in program code differs, 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 base-32 digits (to cope with 32,768 identifiers), resulting in a fair base for a similarity metric in clone detection. Fig. 4 shows the encoded statements that correspond to the matching statements shown in Fig. 2. Fig. 5 shows a portion of the mapping table between three base-32 digits and the matching statements extracted from the original source files.

```
BasicTabbedPaneUI::paintContentBorderBottomEdge(Graph
ics g:int tabPlacement:int selectedIndex:int x:int y:int w:int
h)→001→4F2→07F→002→07G→07F→07G→005→009
→07G→07F→07G→002→07F→07G→07F→07G→005→
005→005
```

Figure 4. Encoded statements corresponding to Fig. 2

001, {	...
002, if{	07D, g.getColor()
...	07E, g.translate()
005, }	07F, g.setColor()
006, return	07G, g.drawLine()
007, putValue()	07H, Border.paintBorder()
008, this()	07I, Border.getBorderInsets()
009, else{	07J, g.drawRect()
...	...

Figure 5. Mapping table between three base-32 digits and a code matching statement used to encode statements in Fig. 4

C. *Apriori-based mining algorithm for finding frequent sequences with gaps*

We have developed a mining algorithm to find frequent sequences based on the *Apriori* principle [3][10], i.e., *if an itemset is frequent, then all of its subsets must be frequent*.

Frequent sequence mining is essentially different from itemset mining, because a subsequence can repeat not only in different sequences but also within each sequence. For example, given two sequences C→C→A and B→C→A→B →A→C→A, there are three occurrences of the subsequence C→A. The repetitions within a sequence [6]-[8] are critical when dealing with long sequences such as protein sequences, stock exchange rates, or customer purchase histories.

Note that the proposed algorithm is implemented to run in two modes, i.e., *multi-occurrence mode* to find all subsequences included in a given sequence, and *single-occurrence mode* to find a subsequence in a given sequence even if there exist several subsequences. As described in Section V, the multi-occurrence mode detects so many code matchings that it has an adverse effect on performance, especially when the minSup is two, and the maxGap is one to three.

The LCS algorithm is also tailored to match three base-32 digits as a unit. That algorithm can match two given sequences even if there is a "gap." Given two sequences of matching strings S1 and S2, let |lcs| be the length of their longest common subsequence, and let |common (S1, S2)| be the common length of S1 and S2 from a back trace algorithm. The "gap size" gs is defined as gs = |common (S1, S2)| – |lcs|.

D. *Mining maximal frequent sequences*

Frequent sequence mining tends to result in a very large number of sequential patterns, creating difficulty for users in analyzing the results. Closed and maximal frequent sequences are two representations for alleviating this drawback. A closed frequent sequence needs to be used in the case in which a system under consideration is designed to deal with an association rule [3][10][14] that plays an important role in knowledge discovery.

A maximal frequent sequence is a sequence that is frequent in a sequence database and that is not contained in any other longer frequent sequences. Maximal frequent

sequences comprise a subset of the closed frequent sequences. Such a sequence is representative in the sense that all sequential patterns can be derived from it. Because we are interested only in finding a set of frequent sequences that are representative of a code clone, we developed an algorithm to discover the maximal frequent sequences.

IV. Proposed Frequent Sequence Mining Algorithm

We have developed two algorithms for detecting code clones with gaps. The first is for mining frequent sequences, and the second is for extracting the maximal frequent sequences from a set of frequent sequences.

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 a sequence database. Note that a subsequence is not necessarily contiguous in an original sequence since the proposed algorithm deals with Type-3 clones.

We assume that a sequence is "a list of items," whereas several algorithms for sequential pattern mining [6]-[9] deal with a kind of sequence that consists of "a list of sets of items." Our assumption is reasonable, because we focus on detecting code clones that consist of "a list of statements." In addition, the assumption simplifies the implementation of the proposed algorithm, enabling it to achieve high performance as described in Section V.

The proposed frequent sequence-mining algorithm contains two methods, GProbe (Fig. 6) and Retrieve_Cand (Fig. 7). It follows the key idea behind the *Apriori* principle;

if a sequence S in a sequence database appears N times, so does every subsequence R of S at least.

The algorithm takes two arguments, minSup and maxGap (the allowable maximal number of gaps).

Fig. 6 shows the pseudocode of the algorithm GProbe. The algorithm takes two arguments, minSup and maxGap (the allowable maximal number of gaps). Let S_k be the set of frequent sequences of size k, and C_k the set of pairs of candidate sequences with frequency k. Let $CSyn_k$ be the set of gap-synonyms of a candidate sequence $c \in C_k$. The gap-synonym of a candidate sequence c is a sequence that matches c with no more than maxGap gaps.

- The algorithm initializes k = 1 and ANS = Φ. S_1 is initialized to hold 15 control statements of *Java*, based on the assumption that an important code clone is preceded by at least one control statement (lines 2 and 3).
- Next, the algorithm iteratively generates candidate k-sequences using the frequent (k–1)-sequences found in the previous iteration (line 5). Candidate generation is implemented using the function, Retrieve_Cand (S_k), which is described in Fig. 7.
- Then, the algorithm eliminates all candidate sequences whose support counts are less than minSup (line 8).
- If the support count of a candidate sequence *c* satisfies the minSup condition, then *c* is merged into ANS, which always contains all frequent sequences discovered thus far (line 9).
- S_k is reconstructed by merging *c* and its gap-synonym $CSyn_k$ (line 10).
- The algorithm terminates when there are no new frequent sequences generated, i.e., $S_k = \Phi$ (line 13).

```
1  GProbe()
2  k= 1;  ANS= Φ;
3  S1 = { 15 control statements of Java };
4  repeat
5      Ck, CSynk = Retrieve_Cand(Sk);
6      k= k+1;  Sk= Φ;
7      for each element c in Ck
8          if ( frequency of  c  >= minSup ){
9              ANS= ANS ∪ { c };
10             Sk= Sk ∪ { c } ∪ CSynk;
11         }
12     end for
13 until Sk = Φ;
```

Figure 6. Pseudocode of the algorithm to find frequent sequences

Briefly, the Retrieve_Cand(S_k) method in Fig. 7 works as follows:

- C and CSyn are initialized to empty (line 2).
- The three *for* loops examine all possible matches between a sequence s_k in S_k and sequences in a sequence database (lines 3, 4, and 5).
- The LCS algorithm is executed to compute the match and gap counts (line 6).
- The *if* statement screens a sequence based on the match and gap counts (line 7).
- If a sequence s_k satisfies the match- and gap-count conditions, then a sequence s_{k+1}, i.e., a sequence s_k extended by one statement of program code, is merged into the candidate sequences C (line 8). Line 8 also implies counting the frequency of a sequence s_{k+1}.
- A gap-synonym of the candidate sequence s_{k+1} is maintained (line 9).
- The Retrieve_Cand (S_k) method returns C and CSyn as the results of execution for S_k (line 15).

```
1  Retrieve_Cand(Sk);
2  C = Φ;  CSyn = Φ;
3  for each element sk in Sk
4      for each element t  in the sequence database
5          for each position p that sk matches in t
6              Compute the LCS between sk and t at position p ;
7              if ( match count >= k  &  gap count  <=  maxGap ){
8                  C = C ∪ { sk+1 };  // put element sk+1 and count frequency.
9                  CSyn = CSyn ∪ { sk+1' };  // sk+1' (≠ sk+1) is a sequence that
10                     // matches with sk+1 under minSup and maxGap conditions.
11             }
12         end for
13     end for
14 end for
15 return C, CSyn
```

Figure 7. Pseudocode of the algorithm for retrieving candidate sequences for the next repetition

B. Extracting Frequent Sequences

In our approach, we assume that a program structure is represented as a sequence of statements preceded by a class-method ID. Each statement is encoded as three base-32 digits to enable the LCS algorithm to work correctly, regardless of the length of the original program statement.

The proposed algorithm is illustrated for the given sample sequence database in Fig. 8. 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 8. Example sequence database

Fig. 9 shows the result for the frequent sequences in the multi-occurrence mode for a gap of zero and a minSup of two, which can be expressed as a minSup of 50% since the total number of sequences is 4. "005" is a frequent sequence with a minSup count of six, 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 at which 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 three, 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 9. Result of the frequent sequences (gap, 0; minSup, 2)

Fig. 10 shows the result of the frequent sequences for a gap of one and minSup of two. "005" is a frequent sequence with a minSup count of six, which is the same in the case of a gap of zero.

Similarly, 005→003→ is a frequent sequence with a minSup count of five. 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 10. Result of the frequent sequences (gap, 1; minSup, 2).

Fig. 11 shows the result of the frequent sequences for a gap of two and a minSup of two. 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 two, and 005→006→ in the fourth sequence with a gap of zero. 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 11. Result of the frequent sequences (gap, 2; minSup, 2)

C. Extracting Maximal Frequent Sequences

A frequent sequence is defined to be a maximal frequent sequence if it has no super (or longer) sequence that is a frequent sequence. Such a sequence is representative, because it can be used to recover all frequent sequences. Several algorithms for finding the maximal frequent sequences and/or itemsets use sophisticated search and pruning techniques to reduce the number of sequence and/or item set candidates during the mining process [9].

However, we wish to compare the effects of a maximal frequent sequence with those of a frequent sequence; therefore, the proposed algorithm first mines a set of frequent sequences and then extracts the maximal frequent sequences.

Screening maximal frequent sequences from frequent sequences with a gap of zero is fairly simple. Given a set of frequent sequences Fs, the set of maximal frequent sequences MaxFs is defined by the following formula:

$$MaxFs = \{x \in Fs \mid \forall y \in Fs\ (x \not\subset y) \wedge (|x| + 1 = |y|)\}.$$

$x \not\subset y$ says that a sequence x is not included in a sequence y. Since a gap has length zero, the length of the immediate super sequence is $|x| + 1$.

The proposed algorithm is described using the sample sequence database in Fig. 12.

```
001→
002→
003→
004→
001→002→
004→003→
001→002→004→
004→002→003→001→
001→008→002→055→004→
```

Figure 12. Example frequent sequences

Fig. 13 shows a set of maximal frequent sequences. The frequent sequence 001→ is not a maximal frequent sequence, because there is a frequent sequence 001→002→ that includes a sequence 001 and whose length is two. In the same manner, we see that the sequences 002→, 003→, and 004→ are not maximal frequent sequences. 001→002→ is not a maximal frequent sequence, because the sequence 001→002→004→ includes 001→002→.

On the other hand, the sequence 004→003→ is a maximal frequent sequence, because there are no frequent sequences that exactly include this sequence. In the same manner, we see that the sequences 001→002→004→, 004→002→003→001→ and 001→008→ 002→055→004→ are maximal frequent sequences.

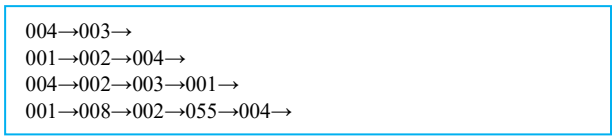

Figure 13. Result of maximal frequent sequences (gap, 0)

Now, we extend the definition of maximal frequent sequences for gaps greater than zero. Let maxGap be the maximal gap under consideration.

$$MaxFs_{maxGap} = \{ x \in Fs \mid \forall y \in Fs\ (x \not\subset_{maxGap} y) \wedge |x| + 1 + maxGap = |y| \}$$

$x \not\subset_{maxGap} y$ says that a sequence x is not included in a sequence y under the gap constraint maxGap.

Fig. 14 shows a set of maximal frequent sequences for a maxGap of one. 004→003→ is not a maximal frequent sequence because 004→003→ is included in the sequence 004→002→003→001→ for a maxGap of one.

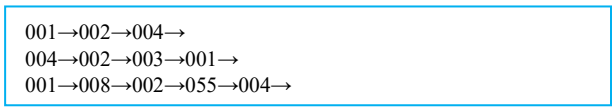

Figure 14. Result of maximal frequent sequences (gap, 1)

Fig. 15 shows a set of maximal frequent sequences for a maxGap of two. In this case, 001→002→004→ is not a maximal frequent sequence, because 001→002→004→ is included in 001→008→002→055→004→ for a maxGap of two.

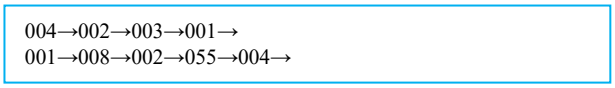

Figure 15. Result of maximal frequent sequences (gap, 2)

V. Experimental Results

This section presents a statistical evaluation of experimental results. *Java* provides a wide range of functions including GUI, network, security, image and sound programming. Among them, *Java SDK 1.8.0_101 SWING* is a set of program components that provide the capability to process image data in various formats, to create graphical user interface (GUI) components, to handle events generated from a user, and to paint graphics of various shapes, etc. Because GUI components such as buttons, text fields, etc., and events such as mouse action, keystroke handling, etc., share functional commonalities, the *SWIMG* package is expected to include various code clones.

The source code of *Java SDK 1.8.0_101 SWING* package is input to the proposed parser to extract matching statements, i.e., method calls and control statements. Then, they are encoded to translate matching statements to three base-32 digits, making match detection successful. A statement sequence is generated for each method.

The total number of source code lines is 372,186. The extracted statement sequences comprise 9,234 lines that roughly correspond to the number of methods calls. The number of extracted unique IDs is 6,310. Since method calls in *Java* source code are preceded by a data type and/or a class identifier, the methods with the same method signatures that are defined in different classes are treated as distinguished ones. The method calls preceded by a data type and/or a class identifier allow *Java* to implement the overriding of methods that play an important role in object-oriented programming.

We performed the experiments using the following PC environment:

CPU: Intel Core i7-6700 (3.40 GHz)
Main memory: 8 GB
OS: Windows 10 HOME 64 Bit
Programming Language: *Java 1.8.0_101*.

The experiments are performed for minSup of two to ten, and maxGap of zero to three. minSup of two means that all possible clones are detected, if a code clone is defined as a fragment of source code that appears at least twice in the package. maxGap of zero means identical or un-gapped sequence matching. Comparison with the results of experiments on maxGap of zero to three shows the functionality and performance of the proposed algorithm.

A. Numbers of Retrieved Frequent Sequences

Fig. 16 compares the number of retrieved frequent sequences with respect to maxGap (zero to three) and minSup (two to ten). For comparison, Fig. 16 shows the number of retrieved frequent itemsets of the *Java* implementation of the *Apriori* algorithm [16]. The proposed algorithm for a maxGap of zero is comparable to the *Apriori* algorithm for a minSup of five to ten. The *Apriori* algorithm fails to generate frequent itemsets for a minSup of two, because it never completes the process within six hours.

As expected, the number of retrieved frequent sequences increases as maxGap increases and minSup decreases. The proposed algorithm can find frequent sequences that occur at least twice (minSup of two) in the sequence database, which is necessary for finding all possible code clones. One of the important findings of the experiment is that the effect of repetitions within a sequence becomes conspicuous when the minSup equals two. A detailed analysis of the retrieved frequent sequences is discussed in Subsection "*C. Sequence Length Analysis w.r.t minSup and maxGap.*"

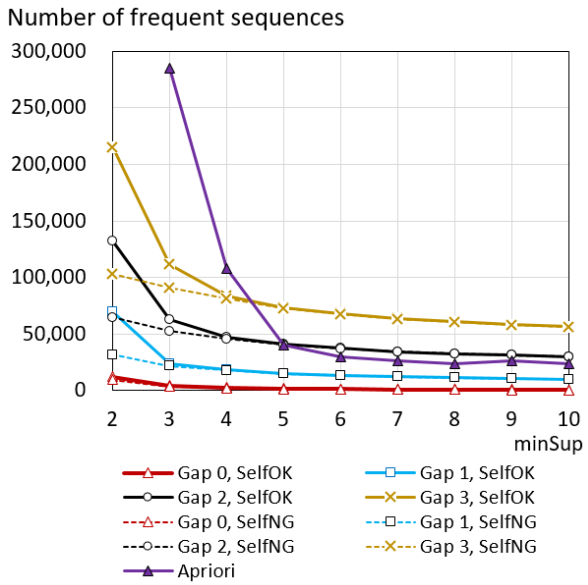

Figure 16. Numbers of retrieved frequent sequences (gap size, 0 and 1–3; minSup, 2–10) and frequent itemsets for the *Apriori* algorithm

Fig. 17 shows the ratio of the number of maximal frequent sequences to the number of frequent sequences. In most of the cases, the ratio decreases as minSup values decrease. This can be explained by the fact that decreasing minSup values probably have a negative effect on the relevance of frequent sequences. Thus, redundant frequent sequences are likely mined as minSup values decrease, resulting in the low ratio of the number of maximal frequent sequences to the number of frequent sequences.

The ratios are generally smaller in the multi-occurrence mode than in the single-occurrence mode. This might be because the single-occurrence mode suppresses extraction of frequent subsequences caused by repetitions within a sequence.

The results show that the gap size affects the ratio by approximately 8.0% for a minSup of two.

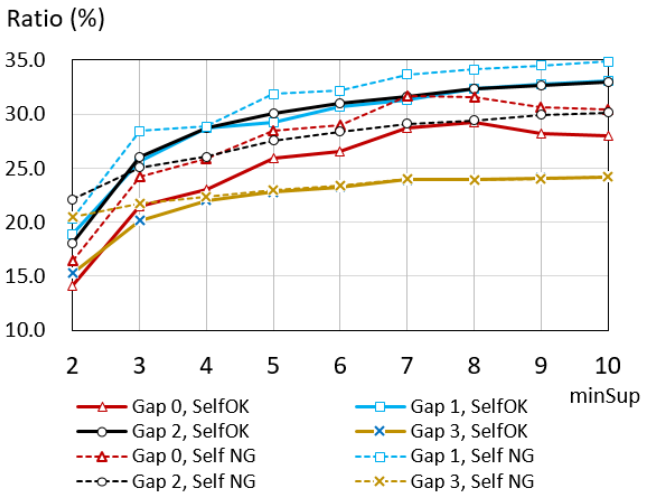

Figure 17. Ratio of the number of maximal frequent sequences to the number of frequent sequences (gap size, 0 and 1–3; minSup, 2–10)

B. *Time Analysis*

Fig. 18 shows the elapsed time in seconds for retrieving frequent sequences for a minSup of two to ten. The proposed algorithm for a maxGap of zero is comparable to the *Apriori* algorithm for a minSup of three to ten as for performance. However, the *Apriori* algorithm fails to find frequent itemsets for a minSup of two within six hours.

The proposed algorithm can retrieve frequent sequences fairly efficiently. For example, it takes 1,437 seconds to identify 31,825 frequent sequences for a maxGap of one and a minSup of two in the single-occurrence mode. Note that elapsed time increases as maxGap increases. This tendency is obvious especially for a minSup of two and three for a maxGap of zero to two, and a minSup of two to six for a maxGap of three. As for differences between the multi-and single-occurrence modes, the ratios of elapsed time range from 1.27 (for a maxGap of zero) to 3.06 (for a maxGap of one). Some reasons for performance degradation are analyzed in the next subsection.

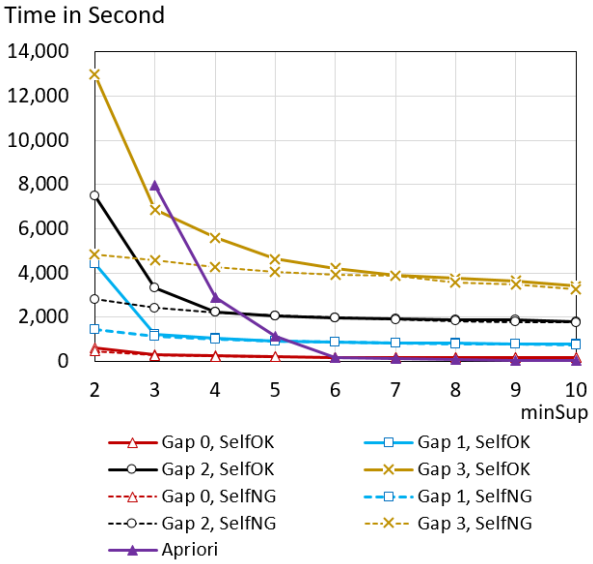

Figure 18. Elapsed time for retrieving frequent sequences (gap size, 0–3; minSup, 2–10) and frequent itemsets for the *Apriori* algorithm

Fig. 19 shows the elapsed time in seconds (Y-axis) for extracting maximal frequent sequences. The result shows that the elapsed time for extracting maximal frequent sequences obviously increases when maxGap is one to three. This can be explained by the observation that the number of maximal frequent sequences increases as maxGap increases, as expected from the expression $|x| + 1 + \text{maxGap} = |y|$ in the definition of maximal frequent sequences, $\text{MaxFs}_{\text{maxGap}}$, defined in Section IV.

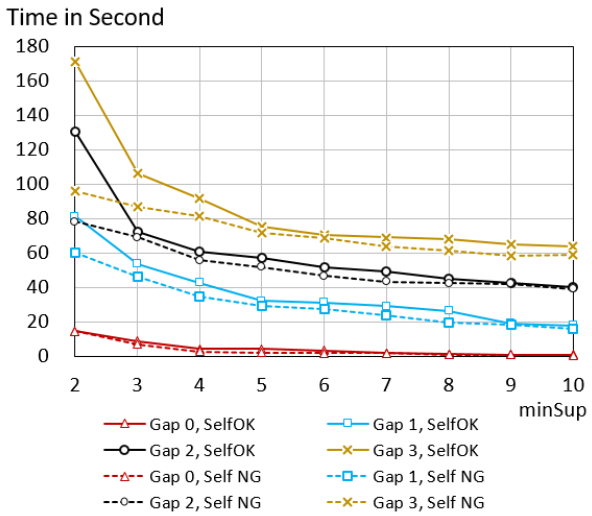

Figure 19. Elapsed time for extracting maximal frequent sequences

C. *Sequence Length Analysis w.r.t minSup and maxGap*

Fig. 20 shows the number of retrieved maximal frequent sequences the first time for each minSup for a maxGap of zero to three. For example, S2-S3 in Fig. 20 indicates the difference between the results of a minSup of two and those of a minSup of three. The vertical axis of Fig. 20 for S2-S3 indicates the number of maximal frequent sequences that is found the first time when the minSup is two.

The numbers are mostly affected by the modes of the proposed algorithm, i.e., multiple or single occurrences with a minSup of two. The ratios of the number of sequences in multi-occurrence to single-occurrence modes for a minSup of two are 1.3 (for a maxGap of zero), 3.3 (for a maxGap of one), 4.4 (for a maxGap of two), and 5.4 (for a maxGap of three).

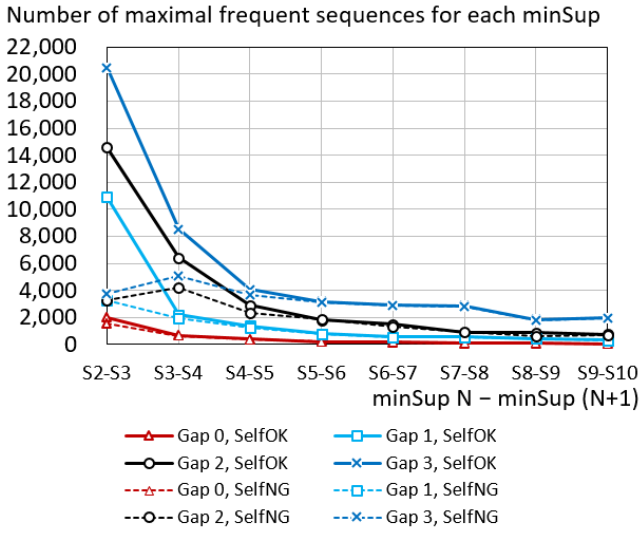

Figure 20. Number of frequent sequences first found for each minSup

Fig. 21 shows the number of maximal frequent sequences for each sequence length (two to 30) in the multi-occurrence mode and a maxGap of three with a minSup ranging from two to five. The maximum length of a sequence is 150 in the multi-occurrence mode. Note that Fig. 21 omits the results on sequences of length 31–150. The number of maximal frequent sequences for each length reaches a peak around a sequence length of eight to ten for each minSup of two to five. This suggests that code clones of length eight to ten occur most frequently.

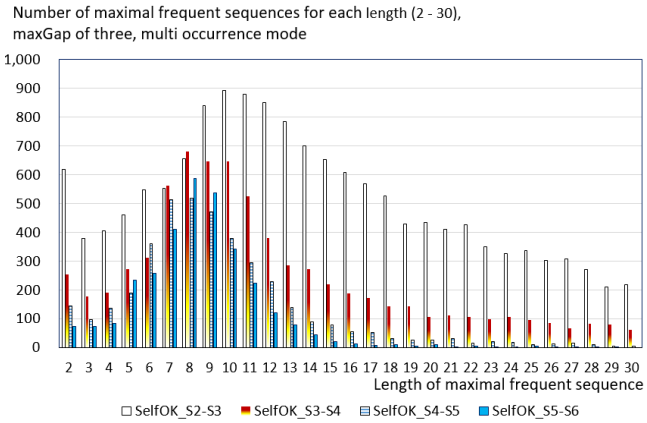

Figure 21. Number of retrieved sequences for each length in multi occurrence mode and a maxGap of three

Fig. 22 shows the number of maximal frequent sequences for each length in the single-occurrence mode. The maximum length of the sequences is 54 in the single-occurrence mode. The number of sequences is substantially decreased owing to the suppression of repetitive subsequences, analyzed in the following section.

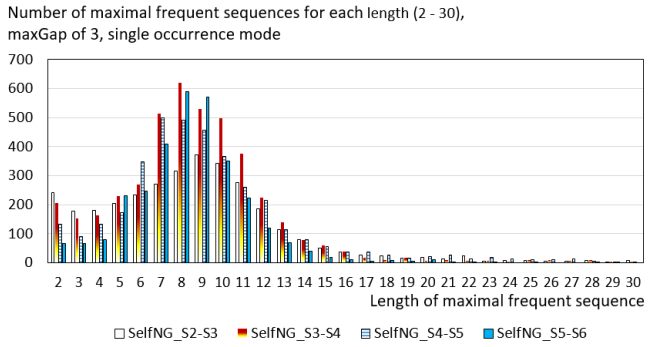

Figure 22. Number of retrieved sequences for each length in single-occurrence mode and a maxGap of three

D. *Source Code Findings*

TABLE I shows two sample sequences that include key sequences repetitively. The subsequences in bold and underlined text indicate the key sequences.

The first sequence in TABLE I includes the key sequence 002→4NM→002→4MI→005→005→ twice. The sequence is mined only in multi-occurrence mode with a minSup of two and with gap of one. The second sequence in TABLE I

includes the key sequence 065→04Q→4HU→4HV →4HV→065→04Q→ three times. The first subsequence **04Q→4HU→4HV →4HV→*065→04Q→*** matches the key sequence with gap of one, i.e., the first 065→ is mismatched. In the second subsequence, ***065→04Q→*** appears as the heads of the second subsequence, i.e., ***065→04Q→***4HU→ 4HV→4HV →065→04Q→, which fully matches the key sequence. The third subsequence follows after the second subsequence sharing ***065→04Q→*** at the top of the third subsequence.

The results of the experiments show that there are many repetitive subsequences of statements in a method. Since these repetitive subsequences are found in a method, programmers are expected to find them easily in the screen of a program editor. Based on this observation, it can be safely said that the single-occurrence mode is preferable to clone mining from the programmer's point of view.

TABLE I. SAMPLE SEQUENCES INCLUDE KEY SEQUENCE REPETITIVELY

No.	Method signiture and statement sequences
1	Handler::repaintDropLocation(JTable.DropLocation loc)→001→002→006→005→**002→4MH→002→4MI→005**→006→005→**002→4NM→002→4MI→005→005**→002→4NM→002→4MI→005→005→005
2	ScrollableTabSupport::updateView()→001→4G9→4GF→4HS→4HM→4HN→4HT→002→04P→04Q→04Q→002→005→065→04Q→04Q→002→005→065→005→005→4GB→04P→**04Q→4HU→4HV→4HV→*065→04Q→*4HU→4HV→4HV→*065→04Q→*4HU→4HV→4HV→065→04Q→**00M→4HU→4HV→4HV→005→005

TABLE II shows a sample set of methods that include the key sequence 002→07F→07J→07F→07G→07G→ in single-occurrence mode with gap of two. Actually, 07F, 07G, and 07J are symbols for the methods *setColor(), drawLine(),* and *drawRect(),* respectively. All of these methods are concerned with paint graphics.

The third and fourth methods are fairly said to be code clones, because their entire statement sequences are completely matched, and they are defined in the same *BasicTabbedPaneUI.java* file. The third method in TABLE II, i.e., the *paintContentBorderBottomEdge()* method, is the method shown in Figs. 2 and 3.

The other methods cannot be safely said to be clones, because they are defined in different files, and they are partially matched with the key sequence. However, the sequence patterns of *setColor(), drawLine(),* and *drawRect()* are informative to programmers for implementing paint graphics. The proposed mining algorithm can serve to find all of the related sequence patterns within a specified gap, viz., maxGap. In addition, these methods are worth checking in the event of bug fixes.

TABLE II. SAMPLE SET OF METHODS

No.	Method signiture and statement sequences
1	SwatchPanel::paintComponent(Graphics g) →001→07F→07S→00O→00O→0GU→07F→002→005→009→005→07S→07F→07G→07G→**002→07F→07G→07G→**07G→07G→07G→07G→005→005→005→005
2	RolloverButtonBorder::paintBorder(Component c, Graphics g, int x, int y, int w, int h) →001→2PO→2PP→002→005→002→07D→07E→**002→07F→07J→07F→07G→07G→**005→009→07F→07J→07F→07G→07G→005→07E→07F→005→005
3	BasicTabbedPaneUI::paintContentBorderBottomEdge(Graphics g, int tabP, int sIndex, int x, int y, int w, int h) →001→4F2→07F→002→07G→07F→07G→005→009→07G→07F→07G→**002→07F→07G→07F→07G→**005→005→005
4	BasicTabbedPaneUI::paintContentBorderRightEdge(Graphics g, int tabP, int sIndex, int x, int y, int w, int h) →001→4F2→07F→002→07G→07F→07G→005→009→07G→07F→07G→**002→07F→07G→07F→07G→**005→005→005
5	MetalComboBoxUI::paintCurrentValueBackground(Graphics g, Rectangle bounds, boolean hasFocus) →001→**002→07F→07J→07F→**07J→002→07F→32T→002→07S→005→002→07S→005→005→005→018→074→005→005
6	Handler::paintDraggedArea(Graphics g, int rMin, int rMac, TableColumn draggedColumn, int distance) →001→29D→4MH→4MH→2C2→07F→07S→07F→07S→**002→07F→07G→07G→**005→00O→4MH→3J9→002→07F→4MH→07G→005→005→005
7	ToolBarBorder::paintBorder(Comp c, Graphics g, int x, int y, int w, int h) →001→002→006→005→07E→002→002→5AL→002→5AM→005→009→5AM→005→005→009→5AL→5AM→005→005→**002→07F→07G→07F→07G→**005→07E→005
8	MetalPropertyListener::paintTrack(Graphics g) →001→002→5LQ→006→005→5LR→5BD→07E→002→5LO→005→009→002→005→009→5LP→5LP→005→005→**002→07F→07J→07F→07G→07G→**07F→07G→07G→005→009→07F→07J→005→002→002→002→005→009→005→005→009→002→005→009→005→005→**002→07F→07G→07G→**07F→07S→005→009→07F→07S→005→005→07E→005

TABLE III shows a pair of methods that share a rather long key sequence:
1T2→1SB→1T3→002→1T4→05Q→002→1T5→005→0BC→005→005→015→1T6→002→1T7→005→005→017→005→017→005→017→005→002→015→005→017→005→005→002→17K→005→1SD→002→006→005→.

The length of the key sequence is 37. This epitomizes the effectiveness of a minSup of two, because there are only two methods that share this key sequence in the *Java SDK 1.8.0_101 SWING* package. Though their statement sequences are not fully matched, they are considered to be clones, because they are defined in the same *Java* file and they share a large amount of functionality.

TABLE III. METHODS SHARING A KEY SEQUENCE OF LENGTH 37

No.	Method signiture and statement sequences
1	JOptionPane::showInternalOptionDialog(Component parentComponent, Object message, String title, int optionType, int messageType, Icon icon, Object[] options, Object initialValue) →001→1S5→1T1→0RO→1SG→**1T2→1SB→1T3→002→1T4→05Q→002→1T5→005→0BC→005→005→015→1T6→002→1T7→005→005→017→005→017→005→017→005→002→015→005→017→005→005→1SH→002→17K→005→002→006→005**→002→002→006→005→006→005→00O→002→006→005→005→006→005
2	JOptionPane::showInternalInputDialog(Component parentComponent, Object message, String title, int messageType, Icon icon, Object[] selectionValues, Object initialSelectionValue) →001→1S5→1T1→0RO→1S6→1S7→1S8→**1T2→1SB→1T3→002→1T4→05Q→002→1T5→005→0BC→005→005→015→1T6→002→1T7→005→005→017→005→017→005→017→005→002→015→005→017→005→005→002→17K→005→1SD→002→006→005**→006→005

TABLE IV shows a pair of methods that share the longest key sequence of length 54 in single-occurrence mode with a minSup of two and with gap of zero. 001→ is not a component of the key sequence, because 001→ corresponds to the statement "{" that is not a control statement and thus is excluded from the statement at the initialization process of the proposed algorithm. Though they are defined in the different *Java* files, they are considered to be code clones, because they have entirely the same statement sequences in the same context of similar *SWING* components, i.e., Button and Label.

TABLE IV. METHODS SHARING THE LONGEST KEY SEQUENCE OF LENGTH 54

No.	Method signiture and statement sequences
1	AccessibleAbstractButton::getAfterIndex(int part, int index)→ 001→**002→006→005→04P→04Q→002→006→005→015→006→005→017→006→005→04Q→015→01G→04R→04S→04T→002→006→005→04T→002→006→005→006→005→017→006→005→04Q→015→01G→04U→04S→04T→002→006→005→04T→002→006→005→006→005→017→006→005→00M→006→005→005**
2	AccessibleJLabel::getAfterIndex(int par, int index)→ 001→**002→006→005→04P→04Q→002→006→005→015→006→005→017→006→005→04Q→015→01G→04R→04S→04T→002→006→005→04T→002→006→005→006→005→017→006→005→04Q→015→01G→04U→04S→04T→002→006→005→04T→002→006→005→006→005→017→006→005→00M→006→005→005**

Fig. 23 shows the source code of the *getAfterIndex()* method in the *javax/swing/AbstractButton.java* file. It is somewhat surprising that the two methods share not only the statements corresponding to the key sequence, but also every character of the comments.

```
/**
 * Returns the String after a given index.
 *
 * @param part the AccessibleText.CHARACTER, AccessibleText.WORD,
 * or AccessibleText.SENTENCE to retrieve
 * @param index an index within the text &gt;= 0
 * @return the letter, word, or sentence, null for an invalid
 *  index or part
 * @since 1.3
 */
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 23. Source code of *getAfterIndex()* method

.

VI. RELATED WORK

Zhu and Wu [6] propose an *Apriori*-like algorithm to mine a set of gap-constrained sequential patterns that can be found in a long sequence, such as stock exchange rates, DNA, and protein sequences. Ding et al. [7] discuss an algorithm for mining repetitive gapped subsequences and apply the proposed algorithm to program execution traces. Kiran et al. [8] propose a model for mining periodic-frequent patterns that occur at regular intervals or gaps in long sequences. Fournier-Viger et al. [9] discuss the importance of maximal

sequential pattern mining and propose an efficient algorithm for finding the maximal patterns.

Wahler et al. [11] propose a method for detecting clones of Types 1 and 2 that are represented as abstract syntax trees in the Extensible Markup Language (XML) by applying a frequent itemset mining technique. Their tool uses the *Apriori* algorithm to identify features as frequent itemsets in large numbers of software program statements. They devise an efficient link structure and a hash table for achieving efficiency in practical applications.

Li et al. [12] propose a tool, CP-Miner, which uses the closed frequent patterns mining technique [10][14] to detect frequent subsequences including statements with gaps. CP-Miner shows that a frequent subsequence mining technique can avert redundant comparisons, leading to improved time performance.

El-Matarawy et al. [13] propose a clone detection technique based on sequential pattern mining. Their method treats source code lines as transactions and statements as items. Their algorithm is applied to discover frequent itemsets in the source code that exceed a given frequency threshold, viz., minSup. Finally, their method finds the maximum frequent sequential patterns [9][10][14] of code clone sequences.

Their approach deals with each program statements including variable declaration, arithmetic calculation, etc, to detect code clones. Each non-reserved word in the source code is replaced by the same letter "X". In addition, any data types are replaced by the same letter "T". Thus the identity of the statements seem to be greatly lessen. They don't discuss the effect of the transformation of replacing by the same letters in their experiments. In this study, because we extract method calls preceded by a data type and/or a class identifier, the matching statements extracted from source code reserve full identity of them. The method calls with a data type and/or a class identifier code enable the overriding of methods that is essential in *Java* as an object-oriented programming language. The results of experiments, Fig. 23 for instance, show that identified method calls and control statements provides sufficient information for detecting code clones.

As for calculation of gaps, El-Matarawy et al. don't clearly describe processes and parameters on gaps. According to the paper [13], their *Apriori*-based algorithm generates candidate code clones of length i+1 by using Cartesian product of $CC_i \times F$, where CC_i is a set of code clones of length i, and F is a set of frequent statements of length one. Then, the dedicated check process examines the presence of code clones, which seems to handle gaps. In this study, we use an LCS algorithm for systematic handling of gaps of similar sequences as described in Section IV.

Accurate detection of near-miss intentional clones (NICAD) [17] is a text-based code clone detection technique. NICAD uses a parser that extracts functions and performs pretty-printing to standardize code format and an LCS algorithm [15] to compare potential clones with gaps. Unlike an *Apriori*-based approach, NICAD compares each potential clone with all of the others. Regarding LCS, Iliopoulos and Rahman [18] introduce the idea of a gap constraint in LCS to address the problem of extracting multiple sequence alignments in DNA sequences.

Murakami et al. [19] propose a token-based method. Their method detects gapped code clones using a well-known local sequence-alignment algorithm, the Smith-Waterman algorithm [20]. They discuss a sophisticated backtracking algorithm tailored for code clone detection.

VII. Conclusions and Future Work

In this paper, we presented an attempt to identify Type 3 code clones. Our approach consists of four steps, extraction of code matching statements, encoding statements in base-32 digits, detecting frequent sequences with gaps, and mining the maximal frequent sequences. The paper deals primarily with the last two steps.

Experiments using *Java SDK 1.8.0_101 SWING* package source code show that the proposed algorithm works successfully for finding clones with respect to a maxGap of zero through three and a minSup of two through ten.

As long as code clones are defined syntactically as similar code segments that occur at least twice, the proposed algorithm achieves 100% recall and 100% precision [13] as a result of the nature of *Aprior*-based data mining with a minSup of two.

In this study, all matching statements, i.e., control statements and typed method calls, are treated equally because we focus on the detection of code clones. In practice, however, each statements should be weighted to reflect their relative importance from the programmer's point of view.

Future work is planned to develop functions to cluster code clones according to the weighted parameters set to each matching statements. The functions aim at providing practical guidance and insights to facilitate understanding and maintenance of large scale software.

ACKNOWLEDGMENT

The authors would like to thank the anonymous reviewers for their invaluable feedback. This research is supported by the JSPS KAKENHI under grant number 16K00161.

References

[1] Y. Udagawa, "On the Effect of Minimum Support and Maximum Gap for Code Clone Detection — An Approach Using Apriori-based Algorithm —", Proc. 3rd international Conference on Advances and Trends in Software Engineering (SOFTENG 2017), April 23 - 27, 2017, pp. 66-73.

[2] Y. Udagawa, "Maximal Frequent Sequence Mining for Finding Software Clones," Proc. 18th International Conference on Information Integration and Web-based Applications & Services (iiWAS 2016), Nov. 2016, pp. 28-35.

[3] R. Agrawal, T. Imielinski, and A. Swami "Mining association rules between sets of items in large databases," Proc. ACM SIGMOD International Conference on Management of Data, June 1993, pp. 207-216.

[4] 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.

[5] A. Sheneamer and J. Kalita. “A survey of software clone detection techniques,” International Journal of Computer Applications, vol.v137, issue 10, Mar. 2016, pp. 1-21.

[6] X. Zhu, and X. Wu “Mining complex patterns across sequences with gap requirements,” Proc. 20th International Joint Conference on Artifical Intelligence(IJCAI'07), Jan. 2007, pp. 2934-2940.

[7] B. Ding, D. Lo, J. Han,and S-C. Khoo “Efficient Mining of Closed Repetitive Gapped Subsequences from a Sequence Database,” Proc. 25th IEEE International Conference on Data Engineering (ICDE 2009), March 2009, pp. 1024-1035.

[8] R. U. Kiran, M. Kitsuregawa, and P. K. Reddy “Efficient discovery of periodic-frequent patterns in very large databases,” Journal of Systems and Software, vol. 112, issue C, Feb. 2016, pp. 110-121.

[9] P. Fournier-Viger, C-W. Wu, A. Gomariz, and V. S-M. Tseng “VMSP: Efficient Vertical Mining of Maximal Sequential Patterns,” Proc. 27th Canadian Conference on Artificial Intelligence (AI 2014), May 2014, pp. 83-94.

[10] P-N. Tan, M. Steinbach, and V. Kumar “Introduction to Data Mining,” Addison-Wesley, March 2006.

[11] V. Wahler, D. Seipel, J. Wolff, and G. Fischer “Clone detection in source code by frequent itemset techniques,” Proc. IEEE International Workshop on Source Code Analysis and Manipulation, Oct. 2004, pp. 128-135.

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

[13] A. El-Matarawy, M. El-Ramly, and R. Bahgat “Code clone detection using sequential pattern mining,” International Journal of Computer Applications, vol. 127, issue 2, Oct. 2015, pp. 10-18.

[14] R. Verma, “Compact Representation of Frequent Itemset,” http://www.hypertextbookshop.com/dataminingbook/public_version/contents/chapters/chapter002/section004/blue/page001.html, 2009.

[15] J. Hunt, W. and Szymanski, T. G. “A fast algorithm for computing longest common subsequences,” Comm. ACM, vol. 20, issue.5, May 1977, pp. 350-353.

[16] M. Monperrus, N. Magnus, and S. Yibin “Java implementation of the Apriori algorithm for mining frequent itemsets,” GitHub, Inc., https://gist.github.com/monperrus/7157717, 2010.

[17] 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.

[18] C. S. Iliopoulos and M. S. Rahman “Algorithms for computing variants of the longest common subsequence problem,” Theoretical Computer Science vol. 395, issues 2–3, May 2008, pp. 255–267.

[19] H. Murakami, K. Hotta, Y. Higo, H. Igaki, and S.Kusumoto “Gapped code detection with lightweight source code analysis,” Proc. IEEE 21st International Conference on Program Comprehension (ICPC), May 2013, pp. 93-102.

[20] “Smith–Waterman algorithm,” https://en.wikipedia.org/wiki/Smith%E2%80%93Waterman_algorithm, Aug. 2016.