On the Effect of Minimum Support and Maximum Gap for Code Clone Detection
－ An Approach Using Apriori-based Algorithm －

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

***Abstract*— Software clones are introduced to source code by copying and slightly modifying code fragments for reuse. Thus, detection of code clones requires a partial match of code fragments. 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 mining for finding frequent sequences. The novelty of our approach includes the maximal frequent sequences to find the most compact representation of sequential patterns. After outlining the proposed methods, the paper reports on the results of a case study using *Java SDK 1.8.0_101 awt* graphics package with highlighting the effect analysis on thresholds of the proposed algorithm, i.e., a minimum support and a maximum gap. The results demonstrate the proposed algorithm can detect all possible code clones in the sense that code clones are similar code segments that occur at least twice in source code under consideration.**

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

I. INTRODUCTION

Two fragments of source code are called software clones if they are identical or similar to each other. Software clones are very common in large software because they can significantly reduce programming effort and shorten programming time. However, many researchers in clone code detection point out that software clones introduce difficulties in software maintenance and cause bug propagation. For example, if there are many copy-pasted code fragments in software source code and a bug is found in one code clone, the bug has to be detected within a piece of software thoroughly and fixed consistently.

Different types of software clones exist depending on the degree of similarity between two code fragments [1][2]. Type 1 is an exact copy without modification, with the exception of layout and comments. Type 2 is a slightly different copy typically due to renaming of variables or constants. Type 3 is a copy with further modifications typically due to adding, removing, or changing code units of at least one code unit.

Research on Type 3 clones has been conducted in recent decades because there are substantially more significant clones of Type 3 than there are of Types 1 or 2 in software for industrial applications. Our approach also focuses on finding Type 3 clones. To find such type of clone, the following problems must be addressed.

(1) How to handle gaps in a context of similarity.
There are many algorithms that are tailored to handle gaps in similarity measure such as sequence alignment, dynamic pattern matching, tree-based matching and graph-based matching techniques [2].
(2) How to find frequently occurring patterns.
The detection of frequently occurring patterns in a set of sequence data has been conducted intensively, as reported in sequential pattern mining literature [3]-[8]. There are several studies [9]-[12] using the *apriori*-based algorithm to discover software clones in source code.

Code clones are defined as a set of syntactically and/or semantically similar fragments of source code [1][2]. Since source code is represented by a sequence of statements, finding clone code is a problem of finding similar sequences that occur at least twice. *Apriori*-based sequential pattern mining algorithms are worth studying because they are 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 minSup controls the level of frequency [3][8].

In [12], Udagawa shows that repeated structures in a method adversely affect the performance especially when a minSup is two or three. This paper pushes forward the study using a large scale software, i.e., *Java SDK 1.8.0_101 aw*t, and analyzes to what extent a minSup affects the number of retrieved sequences and time performance. For this purpose, a 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) the design and implementation of a code transformation parser that extracts code matching statements, including control statements and typed method calls;
(II) the design and implementation of a sequential data mining algorithm that maintains performance at a practical level until a threshold minSup reaches down to two;
(III) the evaluation of the proposed algorithm using *Java SDK 1.8.0_101 awt* with respect to minSup of two to ten and gap size of zero to three. In addition to time performance, the number of retrieved sequences is analyzed for each length of sequences showing that the number of repeated structures in a method accounts for a large part on numbers especially in the case when minSup is two.

 ISBN: 978-1-61208-553-1

The remainder of the paper is organized as follows. After presenting some basic definitions and terminologies on frequent sequence mining technique in Section II, we overview 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 *Java SDK 1.8.0_101 awt* package. Section VI presents some of the most related work. Section VII concludes the paper with our plans for future work.

II. BASIC DEFINITIONS

Defnition 1 (sequence and sequence database). 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 list of sequences $SDB = \langle s_1, s_2,\ldots, s_p\rangle$ having sequence identifiers (SIDs) 1, 2,…,p.

Denition 2 (sequence containment). 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 the strictly increasing sequence of integers q taken from [1, n], $1 \leq q[1] < q[2] < \ldots < q[n] \leq m$ such that $a_1=b_{q[1]}$, $a_2=b_{q[2]},\ldots,$ $a_n=b_{q[n]}$ (denoted as $s_a \sqsubseteq s_b$).

Definition 3 (gapped sequence containment). Let maxGap be a threshold set by the 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 we have $a_1=b_{q[1]}$, $a_2=b_{q[2]},\ldots,$ $a_n=b_{q[n]}$ and $q[j] - q[j-1] - 1 \leq \text{maxGap}$ for all $2 \leq j \leq n$.

Denition 4 (prefx and postfix with respect to maxGap). A sequence $s_a = a_1 \rightarrow a_2 \rightarrow \ldots \rightarrow a_n$ is called a prefix of a sequence $s_b = b_1 \rightarrow b_2 \rightarrow \ldots \rightarrow b_m$ iff s_a is a gapped sequence containment of maxGap. A subsequence $s'_b = b_{n+1} \rightarrow \ldots \rightarrow b_m$ is called postfix of s_b with respect to prefix s_a donoted as $s_b = s_a \rightarrow s'_b$.

Denition 5 (support with respect to maxGap). Given a maxGap, the support 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 $s_b \sqsubseteq s$ with respect to maxGap and is denoted by $\sup_{maxGap}(s_b)$.

Denition 6 (multi occurrence mode and single occurrence mode). Given a 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 the 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 recursively applied for each postfix of s_b to count the support number. The support number recursively calculated is named the support number in *multi occurrence mode* in this paper. This mode is critical when dealing with long sequences such as nucleotide DNA sequences [4] [5] and periodically repeated patterns over time [6]. On the other hand, the support number without the calculation of the postfix of s_b is named the support number in *single occurrence mode*. The algorithm proposed in the paper supports both of the modes.

Denition 7 (frequent sequences with maxGap). Let maxGap and minSup be a threshold set by the user. A sequence s_b is called a frequent sequences with respect to maxGap iff $\sup_{maxGap}(s_b) \leq$ 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 [5][8].

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 [7][8].

The closed frequent sequence is widely used when a system is designed to generate an association rule [3][8] that is inferred from a support number of a frequent sequence. On the other hand, the maximal frequent sequence is valuable, because it provides the most compact representation of frequent sequences [7][13].

III. OVERVIEW OF PROPOSED APPROACH

Fig. 1 depicts an overview of the proposed approach [12]. According to the terminology in the survey [1], our approach can be summarized in three steps, i.e., transformation, match detection and formatting, and aggregation.

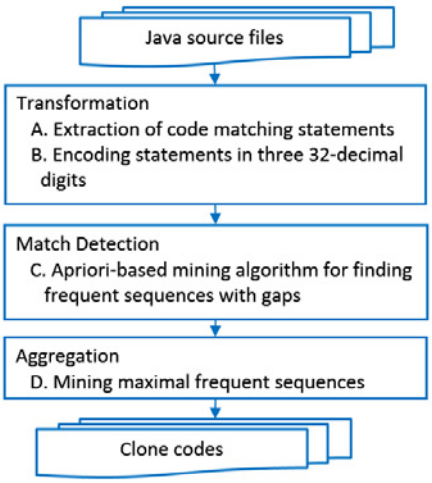

Figure 1. Overview of the proposed approach.

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 clone.

<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 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 1.8.0_101 awt* package as our target of the study. The number of total lines is 166,016, which means the *awt* package is a kind of large scale software in industry.

Fig. 2 shows an example of the extracted structure of the *getFlavorsForNatives(String[] natives)* method in the *SystemFlavorMap.java* file of the *java.awt.datatransfer* package. The three numbers preceded by the # symbol are the number of comments, and blank and code lines, respectively.

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

```
SystemFlavorMap::getFlavorsForNatives(String[] natives)
  #     5     8   12
{
      if{
            getNativesForFlavor()
            List.toArray()
      }
      HashMap()
      for{
            getFlavorsForNative()
            HashMap.put()
      }
      return
}
```

Figure 2. Example of the extracted structure.

B. Encoding statements in three 32-decimal digits

The conventional longest-common-subsequence (LCS) algorithm takes two given strings as input and returns values depending on the number of matching characters of the strings. Due to fact that 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 32-decimal digits (to cope with 32,768 identifiers), which results in a fair base for a similarity metric in clone detection. Fig. 3 shows the encoded statements that correspond to the code shown in Fig. 2. Fig. 4 shows a part of the mapping table between three 32-decimal digits and a code matching statement extracted from the original source files.

```
SystemFlavorMap::getFlavorsForNatives (String[] natives)
→001→004→0VH→0VQ→003→044→04E→0VI→0VR
→003→009→003
```

Figure 3. Encoded statements corresponding to Fig. 2.

001, {	04E, for{
002, this()	...
003, }	0VH, getNativesForFlavor()
004, if{	0VI, getFlavorsForNative()
...	...
044, HashMap()	0VQ, List.toArray()
...	0VR, HashMap.put()

Figure 4. Mapping table between three 32-decimal digits and a code matching statement used to encode statements in Fig. 3.

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][8], 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 [4]-[6] are critical when dealing with long sequences such as protein sequences, stock exchange rates, 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 exists several subsequences.

As described in Section V, the multi occurrence mode detects so many code matching that it has an adverse effect on performance especially when a minSup is two and a maxGap is one to three.

The LCS algorithm is also tailored to match three 32-decimal 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|.

 ISBN: 978-1-61208-553-1

D. Mining maximal frequent sequences

Frequent sequence mining tends to result in a very large number of sequential patterns, making it difficult for users to analyze the results. A closed and maximal frequent sequences are two representations for alleviating this drawback. The closed frequent sequence needs to be used in case a system under consideration is designed to deal with an association rule [3][8] that plays an important role for knowledge discovery. The maximal frequent sequence is such a sequence that are frequent in a sequence database and that is not contained in any other longer frequent sequence. It is a subset of the closed frequent sequence. It is representative in the sense that all sequential patterns can be derived from it [7]. Because we are just interested in finding a set of frequent sequences that are representative of code clone, we developed an algorithm to discover the maximal frequent sequences.

IV. PROPOSED FREQUENT SEQUENCE MINING

We have developed two algorithms for detecting software 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.

We assume that a sequence is "a list of items," whereas several algorithms for sequential pattern mining [4]-[7] deal with a sequence that consists of "a list of sets of items." Our assumption is rational 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, which makes it possible to achieve high performance as described in Section V.

The proposed frequent sequence mining algorithm comprises two methods, i.e., GProbe (Fig. 5) and Retrieve_Cand (Fig. 6). It follows the key idea behind *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).

```
1  GProbe(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, Fig. 5). LinkedList < String > Sk is initialized to hold 15 control statements. The Retrieve_Cand method (line 5, Fig. 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. Thus, 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.

Briefly, the Retrieve_Cand() method in Fig. 6 works as follows. HashMap <String, Integer> Ck holds a sequence (String) and its frequency (Integer). First, Ck is cleared (line 2, Fig. 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 count (line 6, Fig. 6). The *if* statement (line 7, Fig. 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 works correctly, regardless of the length of the original program statement.

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

 ISBN: 978-1-61208-553-1

Fig. 8 shows the result of the frequent sequences in the multi occurrence mode for a gap of 0 and minSup of 50%, which is equivalent to a minSup count of 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 where 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%).

Fig. 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%).

Fig. 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 super sequence of it is a frequent sequence. 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.

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:

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

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

The proposed algorithm is described using the sample sequence database in Fig. 11.

```
001→
003→
004→
005→
001→005→
004→003→
004→003→005→
004→001→004→003→
```

Figure 11. Example frequent sequences.

Fig. 12 shows a set of maximal frequent sequences. The frequent sequence 001→ is not a maximal frequent sequence because there is a frequent sequence 001→005→ that includes a sequence 001 and whose length is two. For the same reason, 003→, 004→, 005→ are not maximal frequent sequences. In this manner, we see that the sequence 004→003→ is not a maximal frequent sequence. However, 001→005→ is a maximal frequent sequence because there are no super-sequences that exactly include 001→005→. 004→003→005→ and 004→001→004→003→ are maximal frequent sequences.

```
001→005→
004→003→005→
004→001→004→003→
```

Figure 12. Result of maximal frequent sequences (gap, 0).

The definition of the maximal frequent sequence is simply extended to those dealing with gaps, as described in [12].

V. EXPERIMENTAL RESULTS

This section shows statistical evaluation of experimental results using *Java SDK 1.8.0_101 awt* package. The number of total source code lines is 166,016. The extracted statement sequences comprise 5,108 lines which are roughly corresponding to the number of methods in the package. The number of extracted unique IDs is 3,175. We performed the experiments using the following environment:

 ISBN: 978-1-61208-553-1

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.

A. *Numbers of Retrieved Frequent Sequences*

Fig. 13 compares the number of retrieved frequent sequences with respect to maxGap (0 to 3) and minSup (2 to 10) with the number of retrieved frequent itemsets for the *apriori* algorithm [14]. The proposed algorithm for a maxGap of zero is comparable to the *apriori* algorithm for a minSups of six to ten. The *apriori* algorithm fails to generate frequent itemsets for a minSup of two, due to it never completes the process in three 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 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 a minSup equals two. A detailed analysis of the retrieved frequent sequences is discussed in Subsection "*C. Sequence Length Analysis*."

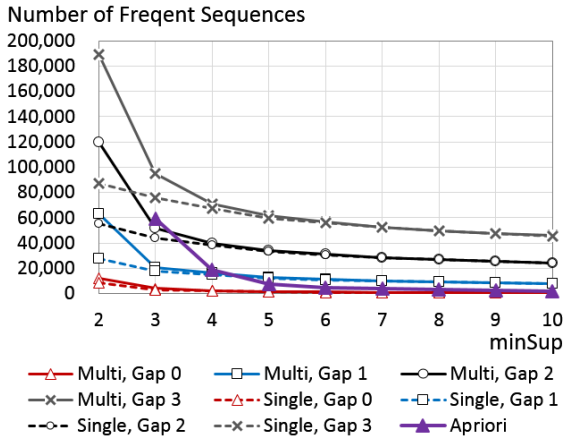

Figure 13. Numbers of retrieved frequent sequences (gap size, 0 and 1-3; minSup, 2-10) and frequent itemsets for *apriori* algorithm.

Fig. 14 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 has 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. It can be a fair explanation that 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 up to approximately 5.55% for a maxGap of two.

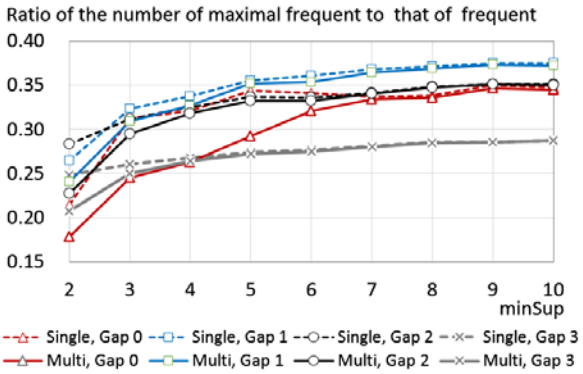

Figure 14. Ratio of the number of maximal frequent sequences to the number of frequent sequences (gap size, 0 and 1-3; minSup, 2-10).

B. *Time Analysis*

Fig. 15 shows the elapsed time in milliseconds 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 five to ten as for performance.

The proposed algorithm can retrieve frequent sequences fairly efficiently. For example, it takes 816,534 milliseconds to identify 27,435 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 for a minSup ranging from two through ten. As for a minSup of two in the multi occurrence mode, the elapsed time jumps up from 2.36 (for a maxGap of three) to 4.65 (for a maxGap of one) times of those for a minSup of three in the multi occurrence mode. A reason for performance degradation is analyzed in the next subsection.

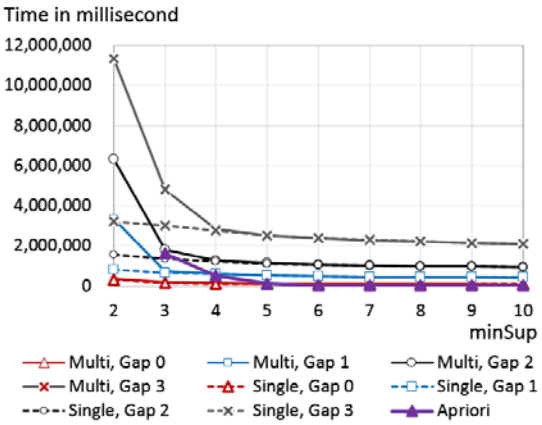

Figure 15. Elapsed time (milliseconds) for retrieving frequent sequences (gap size, 0-3; minSup, 2-10) and frequent itemsets for *apriori* algorithm.

C. *Sequence Length Analysis*

Fig. 16 shows the number of retrieved sequences for each length of sequences in the multi occurrence mode and a maxGap of three with a minSup ranging from two to five. The maximum length of the retrieved sequence is 244. Note

that Fig. 16 omits the results on 31 to 244 of the length of sequence. The number of retrieved sequences reaches peaks 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.

The sequence length of 244 is extracted from the *GetLayoutInfo()* method in *GridBagLayout.java* file of *java.awt* package, consisting of 569 source lines including comments and blank lines. The sequence is detected as a frequent sequence, because the sequence includes "*if{ * }*" statements 244 times caused by repetitions within the sequence of *GetLayoutInfo()* method. It is clear that the detection is not preferable for finding code clone detection.

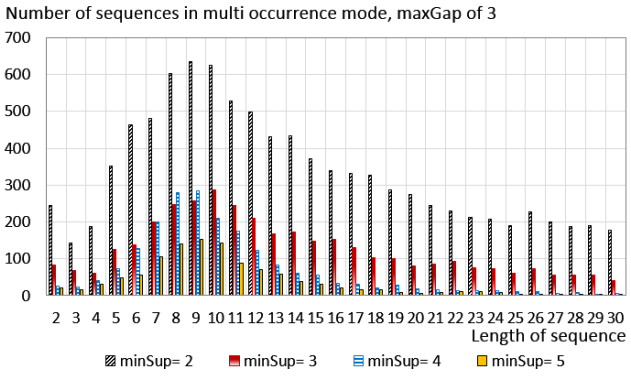

Figure 16. Number of retrieved sequences for each length in multi occurrence mode and maxGap of three.

Fig. 17 shows the number of retrieved sequences for each length of sequence in the single occurrence mode and a maxGap of three with a minSup ranging from two to five.

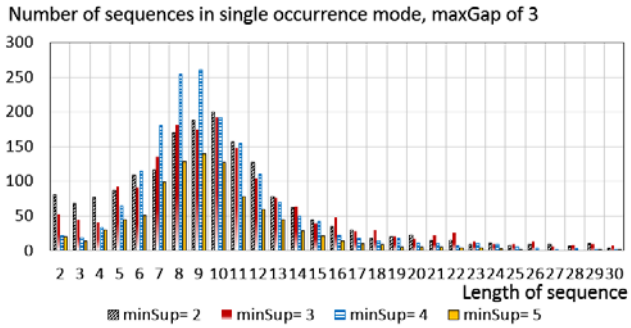

Figure 17. Number of retrieved sequences for each length in single occurrence mode and maxGap of three.

The maximum length of the retrieved sequence is 53 in the single occurrence mode. The sequence of length 53 is extracted from the *getDataElements()* method in *Banded SampleModel.java* file of *java.awt.image* package and the *getDataElements()* method in *ComponentSample Model.java* file. The two methods are the same except for minor syntactic structure, e.g., *if* <*single statement*> and *if {*<*single statement*>*}*, which suggests that they are code clone. Fig. 18 shows the encoded sequence of *getDataElements()* method in *BandedSampleModel.java* file.

BandedSampleModel::getDataElements(int x:int y:Object obj:DataBuffer data) →
001→004→003→24A→24B→007→008→004→003→006→
003→04E→24C→003→00M→008→008→004→003→006→
003→04E→24D→003→00M→008→004→003→006→003→
04E→24E→003→00M→008→004→003→006→003→04E→
24F→003→00M→008→004→003→006→003→04E→24G→
003→00M→003→009→003

Figure 18. Encoded sequence of *getDataElements()* method in *BandedSampleModel.java* file.

VI. RELATED WORK

Zhu and Wu [4] propose an *apriori*-like algorithm to mine a set of gap constrained sequential patterns which can be found in a long sequences such as stock exchange rates, DNA and protein sequences. Ding et al. [5] discuss an algorithm to mine repetitive gapped subsequence and apply the proposed algorithm to program execution traces. Kiran et al. [6] propose a model to mine periodic-frequent patterns that occurs at regular intervals or gaps. Fournier-Viger et al. [7] discuss the importance of the maximal sequential pattern mining and propose an efficient algorithm to find the maximal patterns.

Wahler et al. [9] propose a method to detect clones of the Types 1 and 2 which are represented as an abstract syntax tree (AST) 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 amounts of software program statements. They devise an efficient link structure and a hash table for achieving efficiency for practical applications.

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

El-Matarawy et al. [11] 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, i.e., minSup. Finally, their method finds the maximum frequent sequential patterns [7][8] of code clone sequences. Their method is fairly similar to ours except for a code transformation parser and systematic handling of gaps of similar sequences based on an LCS algorithm.

Accurate detection of near-miss intentional clones (NICAD) [15] is a text-based code clone detection technique. NICAD uses a parser that extracts functions and performs pretty-printing to standardize code format and the longest-common-subsequence (LCS) algorithm [16] 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 [17] introduce the idea of gap constraint in LCS to address the problem of extracting multiple sequence alignment in DNA sequences.

 ISBN: 978-1-61208-553-1

Murakami et al. [18] propose a token-based method. The method detects gapped software clones using a well-known local sequence-alignment algorithm, i.e., the Smith-Waterman algorithm [19]. They discuss a sophisticated backtracking algorithm tailored for code clone detection.

VII. CONCLUSIONS AND FUTURE WORK

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

Through the experiments using *Java SDK 1.8.0_101 awt* package source code, the proposed algorithm works out successfully for finding clones with respect to a maxGap of zero through three and a minSup of two through ten.

Because a minSup of two poses heavy process loads for the proposed algorithm, we analyze the effect of the repeated subsequences in a method and conclude that the repeated subsequences have adverse effects on both performance and the quality of retrieved code clone especially lower minSup, i.e., minSup of two or three.

So long as code clone is syntactically defined as similar code segments that occur at least twice, the proposed algorithm achieves 100% recall and 100% precision due to the nature of the *aprior*-based data mining with a minSup of two [11]. However, we do not believe that the situation is so simple that syntactically defined recall and precision evaluate the quality of mined code clones. Actually, we find a large number of mined code sequences that mainly consist of control statements. Many of these sequences are not clone from programmer's point of view. We are still only halfway to detecting code clones for industry use especially regarding the quality of mined code clones.

Future work will include the development of functions for clustering and ranking mined code clones for the programmer's sake, and the improvement of the transformation for extracting code matching statements.

ACKNOWLEDGEMENT

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] 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] A. Sheneamer and J. Kalita. “A survey of software clone detection techniques,” International Journal of Computer Applications, Vol.137, Issue 10, Mar. 2016, pp.1-21.

[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] 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.

[5] 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.

[6] 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.

[7] 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.

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

[9] 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.

[10] 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.

[11] 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.

[12] 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.

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

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

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

 ISBN: 978-1-61208-553-1