

# 極大頻出系列検出を用いたコードクローンの検出

宇田川 佳久<sup>\*1</sup>

## Detecting Code Clone Using Maximal Frequent Sequence Mining

Yoshihisa Udagawa<sup>\*1</sup>

**Abstract** This paper describes a software clone detection technique using an Apriori-based sequential data mining algorithm. Generally, a frequent sequential data mining algorithm extracts vast numbers of sequential patterns when a threshold named minimum support (minSup) is small, creating an obstacle to the detection of code clones. The proposed method reduces the number of extracted frequent sequences by incorporating pruning processes that depend on characteristics of the source code, techniques to extract the maximal frequent sequences, and curbing measures for repetitive subsequences. The result shows that the proposed sequential data mining algorithm maintains the performance at a practical level until the minSup reaches two.

### 1. はじめに

系列データは、同質のデータをその発生順に直列に並べたデータであり、商品の購買履歴、プログラムのソースコード、株価などの金融データ、DNA に関するデータなどがその例である。頻出系列検出手法は、1994 年に Agrawal らが発表した効率的な頻出アイテム(item)集合を検出する Apriori アルゴリズム[1] を契機として研究されてきた。Apriori アルゴリズムでは、最小支持度とよばれる閾値（以降 minSup と表記）を使って、検出対象とするアイテムがデータベース（トランザクションの集合）内に minSup 回以上出現するとき、そのアイテムを頻出アイテム集合(frequent item set)として検出する。

Apriori アルゴリズムに端を発する頻出系列検出手法は、購買履歴のパターンを対象とすることから、購買品（アイテム）の集合を並べた系列から構成されるデータベースを処理対象としている。頻出系列データを効率的に検出する汎用的な手法として、Apriori アルゴリズムに基づく PrefixSpan[2]、飽和頻出系列(Closed Frequent Sequence)を検出する CloSpan[3]、極大頻出系列(Maximal Frequent Sequence)を検出する VMSP[4] など数多くのアルゴリズムが開発され公開されている[5]。

頻出系列検出手法をソフトウェアのコードクローン（類似したソースコード片）に適用する研究も行われている[6][7][8]。コードクローンとは、2 つ以上の類似したソースコードである[9]。コードクローンは、開発効率を高めるとい点では、有効なものであるが、ソフトウェアの保守の場面では有害であるとされている。例えば、コピー元のソースコードにバグが含まれていた場合、ソースコードをコピーする度にバグが拡散する。このようにして発生したバグを修正するためには、関連するすべてのコードクローン

を検出する必要があり、コードクローンの検出はソフトウェアエンジニアリングにおいて重要な研究領域となっている。

ソースコードに含まれるコードクローンは、2 個以上の類似したソースコード片であることから、頻出系列データのマイニング手法では、minSup が 2 以上の系列を検出する必要がある。一般に、頻出系列は、系列を構成する要素の種類が多く、minSup が小さくなるほど大量の頻出系列を検出する。このことは、Apriori アルゴリズムの基本的な問題点として指摘されていることであり、抽出した頻出系列の数を減らす研究が行われてきた。極大頻出系列(Maximal Frequent Sequence) [4][10][11] は、実質的に有効な頻出系列の数を抽出する手法であり、本研究でも極大頻出系列を採用した。

コードクローンは、構文上、以下の 3 種類に分類される[9]。

タイプ 1：コメント、空白、タブの有無、括弧の位置などを除き、ソースコードとして完全に一致するソースコード

タイプ 2：タイプ 1 のソースコードの内、変数名、リテラル、メソッド名などのユーザ定義名、および、変数の型などの予約語だけが異なるソースコード

タイプ 3：タイプ 2 のソースコードの内、コピー&ペースト後に文の変更、追加および削除が行われた結果によって生成されたソースコード

ソフトウェア開発では、ソースコード片をコピーした後に、行の追加や削除が行われるため、実用的にタイプ 3 クローンの検出が重要である。タイプ 3 クローンを検出するためには、要素間に間欠 (Gap: 対応関係がない要素) を含む系列を検出する必要があり、汎用的な頻出系列の検出手法を適用する研究が行われてきた。Li, Z.氏らは[6]、

<sup>\*1</sup> 東京工芸大学工学部コンピュータ応用学科教授  
2017 年 9 月 26 日 受理

CloSpan アルゴリズムの機能を拡張し、間欠を含むコードクローンの検出を行い、処理時間とメモリ消費量、および、系列の長さと同数のパターン個数などについて実験結果を報告した。石尾氏らは[7]、PrefixSpan アルゴリズムを用いて、コーディングパターンを検出する手法を提案し、6 種類の Java プログラムに対して minSup を 10 に設定した実験を行い、最長で 24 の頻出系列を検出した。El-Matarawy, A.氏らは[8]、独自に極大頻出系列を検出するアルゴリズムを開発し、タイプ 1 から 3 のコードクローンの検出実験結果を報告した。

本論文では、Apriori アルゴリズムに基づく頻出系列の検出手法を用いたコードクローンの検出手法について述べる。本研究では、プログラミング言語における構文の影響を最小限に抑えるために、制御文とメソッド（関数）呼び出し文の系列で構成されるプログラム構造を対象とした。当面の解析対象としては、多くのソースコードが入手可能な Java を採用した。したがって、手順としては、Java の文法に即した構文解析を行ってプログラム構造を抽出し、このプログラム構造に対し独自に開発した頻出系列の検出手法を適用してタイプ 3 クローンの検出を行う。

この研究の手法としての特徴を以下に示す。

- 1) 制御文と変数名の影響を除去したメソッド呼び出しに基づくプログラム構造を対象とすることで、プログラミング言語への依存性を低減する。
- 2) コードクローンを効率的に検出するために、ソースコードの特性を反映した刈り込み処理を行う。
- 3) タイプ 3 コードクローンを検出するために独自の頻出系列検出アルゴリズムを開発した。なお、このアルゴリズムは、検索条件として与えられた系列が一つの系列内で自己参照する場合にも対応している。
- 4) 最長共通部分列(LCS)アルゴリズム[12] を使って、系列の一致と間欠の度合いを計算する。
- 5) 極大頻出系列を使って、最もコンパクトなコードクローンの集合を検出する。

これらの特徴により、本研究で開発した頻出系列の検出手法は minSup が 2 である場合でも実用的な処理時間でコードクローン検出できる。

## 2. 研究の概要

### 2.1 コードクローンの検出処理の流れ

図 1 に本研究におけるコードクローンの検出処理の流れを示す。“プログラム構造の検出”では、本研究で対象とするプログラム構造を検出する。プログラムの構成要素の文字列としての長さはさまざまであり、間欠を含む文字列の一致度合いを LCS アルゴリズムで計算するためにはプログラム構成要素を同じ長さの文字列に変換する必要がある。この処理を行うのが“抽出した構造の 32 進数への変換”である。

“間欠を含む頻出系列の検出”処理では、独自に開発した頻出系列検出アルゴリズムを、32 進数 3 文字に変換し

た文字列に適用する。“極大頻出系列の検出”処理では、抽出した頻出系列から、極大頻出系列[4][10][11] を検出する。なお、極大頻出系列とは、“頻出系列であり、かつ、それよりも 1 つ長い系列が頻出系列でない系列(A sequence is maximal frequent if none of its immediate super-sequence is frequent)”と定義される[4]。直感的には、頻出系列と非頻出系列を分ける境界を構成する系列の集合である。

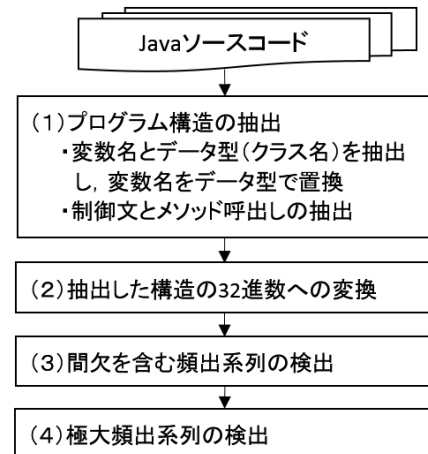


図1 コードクローンの検出処理の流れ

### 2.2 プログラム構造の検出

プログラム構造の検出では、Java ソースコードを以下の処理により、正規化した文字列に変換する[13]。

- (a) 段付けタブ、コメント、空白、空行などを削除する。
- (b) 変数名と変数名に対応するデータ型(クラス名またはデータ型)を検出し、変数名をデータ型に置き換える。
- (c) メソッドの宣言、および、メソッド呼び出しを検出する。
- (d) 制御文を検出する。

(a)の処理により、タイプ 1 クローンに対応できる。(b)の処理により変数の違いを除去し、タイプ 2 クローンに対応している。なお、本研究では、定数、変数宣言などのリテラル情報は処理対象としていない。

(c)と(d)の処理は、プログラム構造を検出する処理であり、クラス内に定義されたメソッドごとに、メソッドの宣言、メソッド呼び出しと制御文を検出する。メソッドの宣言は、メソッドが定義されているクラス名とともに、下記の構文で検出する。

**クラス名::メソッド名(引数)**

メソッドの中で呼び出しているメソッド名は、一般に、**変数名.メソッド名** という構文で出現する。変数名は(b)の処理により、変数名を該当する クラス名または データ型に置き換え、**クラス名.メソッド名** または **データ型.メソッド名** に置き換える。これにより、変数名の違いによる影響を取り除いている。従って、本研究では、メソッド名が同じでも異なるクラスに定義されているものは、異なるメソッドとして扱われる。なお、Java のクラスとメソッドに定義されているアクセス修飾子、[public | protected |

private]は、処理対象としていない。

(d)の処理では、Java の制御文をネスト構造とともに検出する。ブロック構造は、"{" と "}" で表現する。従って、"{"の数がネストの深さを表す。Java で定義されている以下の制御構造を検出対象とする。

- ・ if 文 (else, else if のバリエーションを含む)
- ・ try 文 (catch, finally のバリエーションを含む)
- ・ switch 文                      ・ while 文
- ・ do while 文                    ・ for 文
- ・ break 文                      ・ continue 文
- ・ return 文                      ・ throw 文
- ・ synchronized 文

## 2.3 本研究で用いた Java ソースコードについて

本研究で対象としたのは Java SDK 1.8.0.101 の SWING パッケージである。このソースコードに関する主なメトリックスは下記の通りである。

ファイル数:            737 個  
 クラス数:             1,864 個  
 ソース総行数:        372,186 行

このメトリックスから Java SDK 1.8.0.101 の SWING は、大規模システムと位置付けられるものである[14]。図 2 は、`javax.swing.JApplet.java` ファイルの `JApplet::setRootPane(JRootPane root)` メソッドのソースコードから検出したプログラム構造である。

```
JApplet::setRootPane(JRootPane root)
# 0 0 15
{
    if{
        remove()
    }
    if{
        isRootPaneCheckingEnabled()
        try{
            setRootPaneCheckingEnabled()
            add()
        }
    }
    finally{
        setRootPaneCheckingEnabled()
    }
}
```

図 2 `JApplet::setRootPane` メソッドの検出構造

## 2.4 プログラム構造の 32 進 3 文字での表記

LCS アルゴリズムは、二つの文字系列に共通する最長の文字列を検出するため、例えば if 文と synchronized 文のように文字列の長さが違う文に対して、異なる値を返す。この性質は、プログラム構造の類似検索には適さないものである。なぜなら、文字列の長さは、ソースコードにおける文の役割とは関係がないためである。if 文と synchronized 文は、類似するプログラム構造の検索においては同じ重みとする必要がある。本研究では、文字列の長さの影響を排

除するために、前処理で検出したプログラムを構成するすべての識別名を 3 桁の 32 進数に変換している。この変換により、最大で 32,768 個を識別可能であり、今回の実験対象の数倍程度の規模のソースコードでも処理可能である。識別名を 3 桁の 32 進数に変換する処理は、以下の 2 段階で行われる。

- (1) 識別名の一覧を作成し、それぞれの識別名に一意な 3 桁の 32 進数を割り当てる。
- (2) コードシーケンスのそれぞれの識別名を一意な 3 桁の 32 進数に置き換える。

図 3 は、3 桁の 32 進数と識別名の一覧表の一部を示している。図 4(A)に示す系列は、図 2 に示したプログラム構造に対応するものであり、図 4(B)は、図 4(A)の系列を 32 進数で表記したものである。なお、通常の LCS アルゴリズムは 1 文字を単位として比較するが、当該研究では、3 文字を単位として比較するように改良した。

001, {	05E, remove()
002, if{	...
003, synchronized{	0AG, finally{
004, Boolean.valueOf()	...
005, }	0D7, add()
006, return	...
007, putValue()	11D, setRootPaneCheckingEnabled()
...	...
015, try{	11M, isRootPaneCheckingEnabled()
...	...

図 3 3 桁の 32 進数と識別名のリスト

```
JApplet::setRootPane(JRootPane root) → {if{ → remove()
→ } → if{ → isRootPaneCheckingEnabled() → try{ →
setRootPaneCheckingEnabled() → add() → } → finally{ →
setRootPaneCheckingEnabled() → }
```

(A)処理前 (図 2 に対応する系列)

```
JApplet::setRootPane(JRootPane root)
→ 001 → 002 → 05E → 005 → 002 → 11M → 015 → 11D → 0D7 → 005 → 005
→ 0AG → 11D → 005 → 005
```

(B)処理後

図 4 3 桁の 32 進数によるシーケンスの表現

## 3. 頻出系列抽出アルゴリズムの概要

### 3.1 頻出系列抽出について

頻出するアイテムの集合を見つけ出す問題は、大量のデータから有用なパターンを見つけるための基本手法であり、Apriori アルゴリズムに関する研究を契機として、急速に発展した[1]。Apriori アルゴリズムが対象とするのは、アイテムの集合で構成されるデータベースである。処理対象とするデータベース  $D$  が、 $D = \{t_1, t_2, \dots, t_n\}$ 、アイテムの集合  $t_i$  が  $\{I_1, \dots, I_m\}$  であるとする。Apriori アルゴリズム

は、このデータベース  $D$  において、与えられた  $\text{minSup}$  以上の頻度で発生する集合を効率よく列挙することができる。

$X$  をアイテムの集合とし、 $\text{support}(X)$  でデータベース  $D$  に出現する  $X$  の頻度を表すものとする。一般に、アイテム集合  $X, Y$  が  $X \subset Y$  であれば、 $\text{support}(X) \geq \text{support}(Y)$  が成り立つ。すなわち、あるアイテムを含む集合( $Y$ )の発生頻度は、その部分集合( $X$ )の発生頻度より低い。Apriori アルゴリズムは、この原理を利用している。Apriori アルゴリズムの原理は、アイテムの集合の系列にも適用できる。これまでに、PrefixSpan, CloSpan, ClaSP を含む、数多くの頻出系列検出アルゴリズムが実装されている[5]。

### 3.2 頻出系列抽出について

参考文献[5]で公開されているアルゴリズムは、“アイテムの集合の系列”を対象としている。一方、本研究はプログラムを構成する文(文字列)の頻出系列を検出することを目的としていることから“文字列の系列”を対象としている。Apriori アルゴリズムと同様に、本アルゴリズムでも頻出であることを判別するための  $\text{minSup}$  と最大許容間欠長(以降  $\text{maxGap}$  と表記)をパラメータとしている。2つの系列の比較には、LCS アルゴリズムを使って一致する要素数と間欠数を計算する。

次の 3.2.1 項で述べるように、本アルゴリズムは、与えられた部分系列を含むすべての系列を検出するために、一つの系列に与えられた部分系列が複数回含まれる場合にも対応している。なお、本アルゴリズムの実装では、部分系列を複数回検出する Self OK モードと、1 回しか検出しない Self NG モードをサポートしている。Self OK モードと Self NG モードの処理性能については、4 章で述べる。

#### 3.2.1 本研究で開発したアルゴリズムの概要

本研究で開発したアルゴリズムの動作を図 5 に示した系列データベースを使って説明する。図 5 の各系列の最初の文字列は、Java プログラムのメソッド名に対応するものである。ここでは、MTHD1 などと簡略化している。メソッド名に続く文字列がプログラムを構成する要素に相当するもので、図 5 では 3 文字で一つのプログラム要素を符号化している。記号“→”は、文の前後の繋がりを示す。

図 6 は、 $\text{maxGap}$  が 0 で、 $\text{minSup}$  が 2 以上(図 5 のデータベースが 4 個の系列で構成されているので、率で表したとき  $\text{minSup}$  は 50%) の場合の頻出系列の検出結果を示している。要素 00F は、MTHD2 と MTHD4 で 1 箇所ずつ、合計 2 箇所で開催している ( $N=2$ )。要素 00E は MTHD1 と MTHD3 では 1 箇所、MTHD2 と MTHD4 では 2 箇所の合計 6 箇所で開催している。MTHD2 と MTHD4 のように、検索対象とする系列が、データベース内の系列に複数回出現することを“自己参照”(repetitive)と呼ぶ[15]。要素 00C は 6 箇所で開催している。要素の系列 00E→00C→は、MTHD1 で 1 箇所、MTHD4 で 2 箇所の合計 3 箇所で開催している。

MTHD1→00E→00C
MTHD2→00E→00F→00C→00E→00G
MTHD3→00E→00A→00C→00C
MTHD4→00E→00C→00H→00F→00E→00C

図 5 系列データベースの例

00F→	N=2 (2 4)
00E→	N=6 (1 2+2 3 4+4)
00C→	N=6 (1 2 3+3 4+4)
00E→00C→	N=3 (1 4+4)

図 6  $\text{maxGap}=0$  の場合の頻出系列の検出例

図 7 は、 $\text{maxGap}$  が 1 で、 $\text{minSup}$  が 2 の場合の頻出系列を示している。要素 00F, 00E, 00C は、 $\text{maxGap}$  が 0 の場合と同じである。要素の系列 00E→00C→は、 $\text{maxGap}$  が 0 の場合と異なり 5 箇所で開催している。出現する箇所は、MTHD1, MTHD 2, MTHD3 と MTHD4 であり、MTHD 2 と MTHD3 では、間欠 1 で一致している。さらに、要素の系列 00F→00C→と 00F→00E→は、MTHD 2 と MTHD4 において間欠 1 で一致している。

00F→	N=2 (=2 4)
00E→	N=6 (1 2+2 3 4+4)
00C→	N=6 (1 2 3+3 4+4)
00E→00C→	N=5 (1 2 3 4+4)
00F→00C→	N=2 (2 4)
00F→00E→	N=2 (2 4)

図 7  $\text{maxGap}=1$  の場合の頻出系列の検出例

図 8 は、 $\text{maxGap}$  が 2 で、 $\text{minSup}$  が 2 の場合の頻出系列を示している。 $\text{maxGap}$  が 1 の場合に比べて、間欠 2 で一致する要素の系列 00C→00E→が加わっている。系列 00C→00E は、MTHD 2 では間欠 0 で、MTHD 4 では間欠 2 で一致する。

00F→	N=2	IDs=2 4
00E→	N=6	IDs=1 2+2 3 4+4
00C→	N=6	IDs=1 2 3+3 4+4
00E→00C→	N=5	(1 2 3 4+4)
00F→00C→	N=2	(2 4)
00F→00E→	N=2	(2 4)
00C→00E→	N=2	(2 4)

図 8  $\text{maxGap}=2$  の場合の頻出系列の検出例

#### 3.2.2 本研究で開発したアルゴリズムの特徴

本研究で開発した頻出系列検出アルゴリズム(図 9)は、Apriori アルゴリズムの原理に基づくものである。頻出系列検出アルゴリズムも Apriori アルゴリズムも、 $\text{minSup}$  を引数として取り、 $\text{minSup}$  回以上の頻度で出現するデータを検出する点が共通している。ただし、前者は Java ソースコードを対象にした系列、後者はアイテムの集合の系列を対



象にしていることから、以下に示す相違がある。

#### (A) 初期値

Apriori 準拠アルゴリズムでは、処理の初期値は、1 要素から成る値の集合である。一方、頻出系列検出アルゴリズムの初期値は 1 個の制御文である。すなわち、if, else if, else, switch, while, do, for, break, continue, return, throw, synchronized, try, catch, finally の 15 個が初期値である。これは、有意なソースコードであれば、制御文に先導されているという前提に基づくものである。

#### (B) 次候補の生成

Apriori アルゴリズムでは、 $k$  ( $k > 1$ ) 回目の頻出集合の候補  $C_k$  は、ひとつ前の頻出集合  $F_{k-1}$  を構成する要素の数学的な意味での“組合せ”によって生成される。

これに対し、本研究で開発した頻出系列検出アルゴリズムでは、系列を構成する要素の出現順序を保存する必要があることから、基本的に、ひとつ前の頻出系列  $S_{k-1}$  を基準にして、系列データベースを検索することにより頻出系列の候補  $C_k$  を生成する。ただし、系列データベースを構成する 1 つの系列に、複数の（頻出）系列が含まれることがある。例えば、 $A \rightarrow B \rightarrow A \rightarrow C \rightarrow A \rightarrow B \rightarrow D$  という系列には、 $A \rightarrow B$  という（頻出）系列が 2 個含まれている。そのため、与えられた 1 つの系列から、一致する可能性のあるすべての部分系列を生成する。例えば、 $A \rightarrow B \rightarrow A \rightarrow C \rightarrow A \rightarrow B \rightarrow D$  からは、 $A \rightarrow B$  に加え、 $A \rightarrow C \rightarrow A \rightarrow B \rightarrow D$  という部分系列を生成し、これらの部分系列に対して出現数をカウントする処理を行っている。

本アルゴリズムでは、検索条件として与えられた系列が一つの系列内で自己参照を発生することで  $\text{minSup}$  以上出現している場合に対応するため、頻出系列とする自己参照許可モード(SelfOK)と頻出系列としない自己参照不許可モード(SelfNG)を実装している。

#### (C) 間欠を含むマッチング処理

間欠を含むマッチング処理は LCS アルゴリズムを用いている。LCS アルゴリズムは、その名のとおり、2 つの文字系列で一致する最大の長さを検出する。間欠を含むマッチング処理には、文字系列のグローバルアラインメントを求める Needleman-Wunsch のアルゴリズム[16] や、ローカルアラインメントを求める Smith-Waterman のアルゴリズム[17] がある。これらのアルゴリズムは、不一致と間欠に対するペナルティ値を設定する必要があり、このペナルティ値によって異なる結果が生成され得る。一方、LCS アルゴリズムは、ペナルティ値の設定が不要であること、および、2 つの文字系列で一致する最大の長さを検出できることから、本研究では、LCS アルゴリズムを採用した。

### 3.3 頻出系列検出アルゴリズムの処理概要

図 9 は、本研究で開発した頻出系列検出アルゴリズム GProbe の概要を示している。図 9 の変数  $k$  は、検索の回数を示す。4 行目の変数  $S_k$  は、 $k$  回目の頻出系列を記憶するための記憶領域であり、初期値は Java の制御文である。これは、有意なソースコードであれば、制御文が先行する

という前提を反映したものである。なお、間欠を許容する場合、変数  $S_k$  には、指定された間欠の範囲で一致する頻出系列（頻出同義系列）が記憶される。

6 行目の Retrieve\_Cand() メソッドは、 $S_k$  に含まれる各要素に対し、その要素を含む系列データベース内の系列の行番号と検出数を記憶するメソッドであり、詳細は次項で説明する（図 10）。7 行目では次の検索に向けた変数  $k$  の更新、8 行目では次の回の頻出系列を記憶するために変数  $S_k$  を初期化している。9 行目から 18 行目までは、頻出系列を検出している。10 行目と 11 行目の条件を満たしたものが頻出系列であり、12 行目で系列と検出個数、検出行の番号リストとともに結果ファイルに書き出す。13 行目と 14 行目では、1 つ長い頻出候補系列を検索するための準備として、変数  $S_k$  に頻出同義系列を設定している。19 行目では、当該アルゴリズムの終了判定を行っている。

```

1 GProbe(String[] args){
2     k = 1;
3     自己参照NGモードの設定(true または false)
4     LinkedList<String> Sk に探索の初期値(制御文)をセットする
5     do {
6         Retrieve_Cand(); // 今回の検索キーよりも1個長い系列検索する
7         k = k+1;
8         Sk.clear(); // 次回の頻出系列を記憶するために、初期化する
9         while( Ck のすべての要素 e に対し){
10            if ( Ck[e] の検出個数 >= minSup ){
11                if ( !(検出行の番号リストが自己参照だけ & 自己参照NGモード) ){
12                    Ck[e] + 検出個数 + 検出行の番号リストを結果ファイルに出力する
13                    while( CkSyn から Ck[e] に対応する 頻出同義系列を取り出す ){
14                        LinkedList<String> Sk に 頻出同義系列を追加する;
15                    }
16                }
17            }
18        }
19    } while (Sk.size() > 0);
20 }

```

図 9 頻出系列アルゴリズム

図 10 は、間欠を考慮した頻出候補系列を検索する Retrieve\_Cand() メソッドの概要を示している。記憶領域として使用されている  $C_k$ ,  $C_kMD$ ,  $C_kSyn$  は、Java の HashMap を使って実装している。6 行目と 8 行目の for 文で、検索の対象とする系列が、データベース内の 1 個の系列に複数回発生することに対する処理を行っている。9 行目では、LCS アルゴリズムを使って 2 つの系列  $s$  と系列  $t$  の先頭から  $p$  番目から始まる系列との最長共通部分列を計算する。最長共通部分列を確実に動作させるため、系列  $s$  には  $k$  回目の頻出系列を、系列  $t$  にはデータベースに記憶されている系列をセットする。最長共通部分列の長さを  $lcs$  とし、系列  $s$  の長さを  $|s|$  と表記するとき、間欠の長さは  $gap = lcs - |s|$  で計算する。10 行目から 14 行目では、頻出候補系列  $s$  と  $k$  個以上一致し、かつ、指定された間欠数以下である条件を満たすデータベース内の系列 ( $k+1$  回目の頻出候補系列) に関する情報を  $C_k$ ,  $C_kMD$ ,  $C_kSyn$  に記憶する。

```

1 Retrieve.Cand()
2   Ck.clear(); // 頻出候補系列の 検出個数を記憶する領域
3   CkMD.clear(); // 頻出候補系列の 検出行番号を記憶する領域
4   CkSyn.clear(); // 頻出候補系列にGap数の範囲で一致する系列を記憶する領域
5   for (すべての Sk の要素 s に対し) {
6     for (データベース内のすべての系列 t に対し) {
7       t と s の先頭の要素と一致する箇所 p を検索する(自己参照への対応)
8       for (すべての p に対し) {
9         LongestCommonSubsequence( s, t, p );
10        if (一致した要素数 >= k && Gap数 <= maxGap) {
11          Ck.put( s, 検出個数 );
12          CkMD.put( s, 検出行の番号リスト );
13          t から一致したサブシーケンスを切り出す(= 頻出同義系列 )
14          CkSyn.put( 頻出同義系列, s );
15        }
16      }
17    }
18  }
19 }

```

図 10 候補系列の生成アルゴリズム

### 3.4 極大頻出系列の抽出

Apriori 準拠アルゴリズムでは、系列（またはアイテム集合）の長さが短いほど頻出しやすく、反対に長いほど頻出しにくくなる。系列が頻出であるか否かを  $\text{minSup}$  だけで判断すると大量の頻出系列が検出されることが知られており、その中から有意な頻出系列を検出する技法が研究されてきた。そのひとつが極大頻出系列[4][10][11]である。

系列の長さを一つずつ成長させてゆき、検出したすべての頻出系列の中で、最も長い頻出系列を極大頻出系列と呼ぶ。頻出系列集合  $F_s$  の極大頻出系列の集合  $\text{MaxFs}$  は以下の式で定義される。ここで、 $|x|$  は頻出系列  $x$  の長さを示し、 $x \not\subset y$  は頻出系列  $y$  が  $x$  を含まないことを表す。

$$\text{MaxFs} =$$

$$\{x \in F_s \mid \forall y \in F_s (x \not\subset y) \wedge |x|+1 = |y|\} \quad (1)$$

図 11 に示した頻出系列集合に対する極大頻出系列集合を図 12 に示す。長さ 1 の系列  $00A \rightarrow$ ,  $00B \rightarrow$ ,  $00C \rightarrow$ ,  $00D \rightarrow$  は、長さ 2 の系列  $00A \rightarrow 00B \rightarrow$ ,  $00A \rightarrow 00C \rightarrow$ ,  $00D \rightarrow 00A \rightarrow$  に含まれるので、極大頻出系列ではない。同様に、 $00A \rightarrow 00B \rightarrow$ ,  $00D \rightarrow 00A \rightarrow$  も極大頻出系列ではない。

```

00A→
00B→
00C→
00D→
00A→00B→
00A→00C→
00D→00A→
00A→00B→00C→
00D→00A→00B→
00D→005→00A→00B→
00A→00B→002→005→00C→

```

図 11 頻出系列集合の例

```

00A→00C→
00A→00B→00C→
00D→00A→00B→
00D→005→00A→00B→
00A→00B→002→005→00C→

```

図 12 図 11 に対応する極大頻出系列集合

極大頻出系列集合を拡張して、間欠を考慮した場合の極大頻出系列集合を以下の式で定義する。

$$\text{MaxFs maxGap} = \{x \in F_s \mid \forall y \in F_s (x \not\subset \text{maxGap } y) \wedge |x|+1+\text{maxGap} = |y|\} \quad (2)$$

ここで  $\text{maxGap}$  は、許容される間欠の長さの最大値 ( $\text{maxGap}$ ) を示す。図 13 は、 $\text{maxGap}$  が 1 の場合の図 11 に対応する極大頻出系列集合を示す。系列  $00A \rightarrow 00C \rightarrow$  は、間欠が 1 で系列  $00A \rightarrow 00B \rightarrow 00C \rightarrow$  と一致するので、極大頻出系列ではない。

```

00A→00B→00C→
00D→005→00A→00B→
00A→00B→002→005→00C→

```

図 13 極大頻出系列集合( $\text{maxGap}=1$ )

図 14 は、 $\text{maxGap}$  が 2 の場合の図 11 に対応する極大頻出系列集合を示す。系列  $00A \rightarrow 00B \rightarrow 00C \rightarrow$  は、間欠が 2 で系列  $00A \rightarrow 00B \rightarrow 002 \rightarrow 005 \rightarrow 00C \rightarrow$  と一致するので、極大頻出系列ではない。

```

00D→005→00A→00B→
00A→00B→002→005→00C→

```

図 14 極大頻出系列集合( $\text{maxGap}=2$ )

## 4. 実験結果

### 4.1 頻出系列数と処理時間

本研究で開発した、頻出系列検出アルゴリズムの特性を確認するために Java SDK 1.8.0.101 の SWING パッケージから検出したプログラム構造を使った実験を行った。実験に用いた系列データの主なメトリックスは以下の通りである。

- 1 行以上の制御文またはメソッド呼出しを含むメソッド数（系列数）： 9,234 個
- 識別子の種類： 6,310 個
- 識別子の総数： 79,095 個

図 15 は、 $\text{minSup}$  が 2 から 10、 $\text{maxGap}$  が 0 から 2 について、本研究で開発した頻出系列検出アルゴリズム GProbe が検出した頻出系列の数をグラフ表示したものである。GProbe は、1 つの系列に複数の部分系列を含む“自己参照”も検出する。実験では、特定の一つの系列だけから抽出された“部分系列”のみによって検出された頻出系列を検出する Self OK モードと検出しない Self NG モードについて実験した。また比較のために、系列を構成する識別子の集合で構成されるデータベースを作成し、Apriori アルゴリズム[18] を使って頻出集合を検出した結果を併記した。

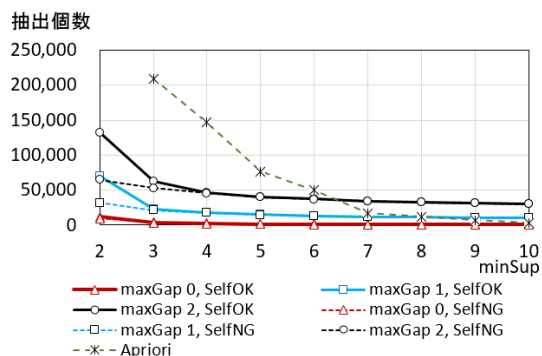


図 15 頻出系列の検出数

一般的に指摘されているように、Apriori の原理に基づくアルゴリズムでは、minSup が小さくなるほど、検出される頻出集合の大きさが増大する。この実験においても、minSup が小さくなるほど検出した頻出系列の数が増大する。minSup が 2 または 3 になると自己参照を検出する頻度も増大することから、Self OK モードと Self NG モードの差異が顕著になる。特に、minSup が 2 で、maxGap が 1 および 2 では、検出数が 2 倍程度になることが確認できた。一方で、Apriori アルゴリズムは、minSup が 6 以下になると、頻出集合の大きさが急激に増大する。

図 16 は、minSup が 2 から 10、maxGap が 0 から 2 についての GProbe の処理時間と、Apriori アルゴリズムおよび参考文献[5] で公開されている PrefixSpan アルゴリズムの処理時間を併記したものである。なお、PrefixSpan では、minSup が 10 で検出した頻出系列が 1,000 万件以上であったことから、図 15 には検出数を掲載していない。Apriori は minSup が 2 で 4 時間以上の処理時間を要し、PrefixSpan も minSup が 5 で 4 時間以上の処理時間を要したため、実行を中断した。

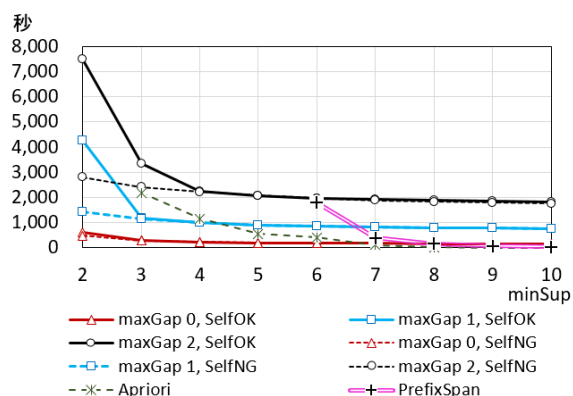


図 16 頻出系列を検出するのに要した時間

GProbe は、Self OK モードで maxGap が 1 または 2、minSup が 2 または 3 である場合に急激に処理性能が劣化する。これは図 15 に示した頻出系列の検出数の増加に呼応している。一方、Self NG モードでは性能劣化が抑えられ、minSup が 4 以上の場合の延長上にあることが確認できる。

## 4.2 極大頻出系列数と処理時間

極大頻出系列は、検出したすべての頻出系列の中で最も長い頻出系列であり、コードクローン検出の観点からは、実質的なコードクローンの候補である。図 17 は、極大頻出系列と頻出系列の検出数の比率を示している。minSup が小さくなると、検出数の比率も小さくなる傾向がある。これは、minSup が小さくなると、検出される頻出系列が急激に増大する一方で極大頻出系列の検出数はそれほど増大しないことから、比率としては小さくなるためである。なお、極大頻出系列と頻出系列との検出数の比率は、minSup が 2 または 3 である場合に、Self OK モードと Self NG モードで差異が見られる。

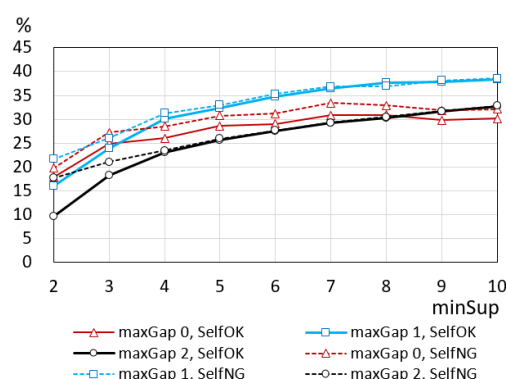


図 17 極大頻出系列と頻出系列の検出数の比率

図 18 は、頻出系列から極大頻出系列を検出するのに要した処理時間を表している。処理時間は最大でも 20 秒以下であり、頻出系列の検出処理時間の 1/200 程度である。処理時間に比べて、検出される系列の数を概ね 1/3 に減少できることから、極大頻出系列を検出することは、コードクローン検出という応用の観点からも有効なものと考えられる。

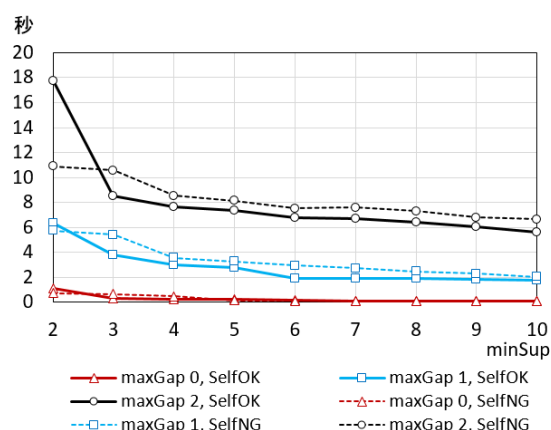


図 18 極大頻出系列を検出するのに要した時間

## 4.3 検索されたソースコードの考察

表 1 は、002→05E→005→//K→002→11M→015→11D→0D7→005→005→0AG→11D→005→ という系列と maxGap が 1 で検出した 5 個のメソッドの一覧である。なお、表 1 の No.1 のプログラム構造は図 2 に示したもので

ある．表 1 の No.4 のメソッドは，上記の系列を完全に含んでいるが，No.4 以外のメソッドは →11K という識別子を含んでいないため間欠が 1 で検出された．表 1 では，→11K を含まない間欠の場所を全角空白で表記したため...→005 →002...となっている．なお，メソッドの最初の "{" (001) と最後の "}" (005) は検索処理の対象外である．11K は，getRootPane()メソッドの呼び出しに対応しており，No.4 のメソッドは，RootPane に対して状態のチェックを可能にしているものと推測できる．クラス名，メソッド名とメソッドの構造（およびソースコード）から，表 1 の 5 個のメソッドはコードクローンであると考えられる．

表 1 間欠 1 で一致した 5 個のメソッドの一覧

No.	メソッドの構造
1	JApplet::setRootPane(JRootPane root) →001→002→05E→005 →002→11M→015→11D→0D7→005→005→0AG→11D→005→005
2	JDialog::setRootPane(JRootPane root) →001→002→05E→005 →002→11M→015→11D→0D7→005→005→0AG→11D→005→005
3	JFrame::setRootPane(JRootPane root) →001→002→05E→005 →002→11M→015→11D→0D7→005→005→0AG→11D→005→005
4	FocusPropertyChangeListener::setRootPane(JRootPane root) →001→002→05E→005 →11K→002→11M→015→11D→0D7→005→005→0AG→11D→005→005
5	JWindow::setRootPane(JRootPane root) →001→002→05E→005 →002→11M→015→11D→0D7→005→005→0AG→11D→005→005

表 2 は，minSup が 2 で初めて検出した，すなわち minSup が 3 以上では検出されない，54 個の識別子から構成される系列である．該当するメソッドは，javax/swing/AbstractButton.java ファイル内の getAfterIndex(int part, int index) メソッドと javax/swing/JLabel.java ファイル内の getAfterIndex(int part, int index) メソッドである．

表 2 54 個の識別子から構成される系列の一覧

No.	メソッドの構造
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

これら 2 つのメソッドの系列は間欠が 0 で一致しており，コピーであると推定される．図 19 は，

javax/swing/AbstractButton.java ファイル内の getAfterIndex(int part, int index)メソッドの全ソースコードである．このソースコードと javax/swing/JLabel.java ファイル内の getAfterIndex(int part, int index)メソッドを比較すると，プログラムとしてのコードはもちろんのこと，コメントまでも一致しており，コードクローンと考えられる（後者のソースコードは省略）．

```
/**
 * 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;
    }
}
```

図 19 getAfterIndex(int part, int index)メソッドのソースコード



## 5. まとめと今後の研究方針

本文では、コードクローンの検出のために開発した頻出系列検出アルゴリズムと Java SDK 1.8.0.101 の SWING パッケージのソースコードに適用した結果について述べた。

この研究の特徴は、間欠を含みかつ自己参照を含むコードクローンを検出できること、極大頻出系列を検出することにより最もコンパクトなコードクローンの集合を検出できること、ソースコードの特性を反映した刈り込み処理により大規模なソースコードにも適用できる性能を有することである。

一般に、Apriori 準拠アルゴリズムでは大量の頻出系列が検出される。検出される系列数を低減する方法として極大頻出系列が提案されているが、従来の極大頻出系列は、間欠を含まない系列に対して定義されている。本研究では、間欠を含む系列に対する極大頻出系列の定義を行い、実験によって間欠を含む系列の検出数の低減に関する効果を検証した。

検出される系列数において、極大頻出系列は頻出系列を大幅に減少させるものであるが、検出される極大頻出系列は1万件に及ぶことから、コードクローンの検出という観点からは、更なる絞込みが必要である。今後は解析対象とするソースコードの特性を反映した絞込みの技法を開発し、有効性を評価する予定である。

## 謝辞

本研究は、JSPS 科研費 基盤研究(C)一般 JP16K00161 の助成を受けたものです。

## 参考文献

- 1) Agrawal, R. and Srikant, R.: Mining sequential patterns, Proc. 11th IEEE International Conference on Data Engineering(ICDE), pp.3-14 (1995).
- 2) Pei, J., Han, J., Mortazavi-Asl, B., Wang, J. Pinto, H., Chen, Q., Dayal, U., and Hsu, M.-C.: Mining Sequential Patterns by Pattern-Growth: The PrefixSpan Approach, IEEE Transactions on Knowledge and Data Engineering, Vol.16, No.11, pp.1424-1440 (2004).
- 3) Yan, X., Han, J., and Afshar, R.: CloSpan: Mining closed sequential patterns in large datasets, Proc. 3rd SIAM International Conference on Data Mining, pp.166-177 (2003).
- 4) 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.
- 5) Fournier-Viger, P. An Open-Source Data Mining Library, Version 2.18, <http://www.philippe-fournier-viger.com/spmf/index.php>, (2017).
- 6) Li, Z., Lu, S., Myagmar, S., and Zhou, Y.: CP-Miner: A Tool for Finding Copy-paste and Related Bugs in Operating System Code. Proc. 6th Symposium on Operating System Design and Implementation, pp.289-302 (2004).
- 7) 石尾隆, 伊達浩典, 三宅達也, 井上克郎: シーケンシャルパターンマイニングを用いたコーディングパターン抽出, 情報処理学会論文誌, Vol.50, No.2, pp.860-871 (2009).
- 8) El-Matarawy, A., El-Matarawy, M., El-Ramly, and Bahgat, R.: Code Clone Detection using Sequential Pattern Mining, International Journal of Computer Applications, Vol.127, Issue.2, pp.10-18 (2015).
- 9) Roy, C. K. and Cordy, J. R.: A survey on software clone detection research, Queen's Technical Report: 541. Queen's University at Kingston, Ontario, Canada, pp.1-115 (2007).
- 10) Tan P-N., Steinbach M., and Kumar V.: Introduction to Data Mining, Addison-Wesley, March 2006.
- 11) 有村博紀: データベースからの頻出アイテム集合発見 - アソシエーションルール発見 -, <http://www-ikn.ist.hokudai.ac.jp/ikn-tokuron/itemset.pdf>, (2005).
- 12) Longest common subsequence, [http://rosettacode.org/wiki/Longest\\_common\\_subsequence](http://rosettacode.org/wiki/Longest_common_subsequence), (2016).
- 13) Udagawa, Y., Kitamura, M.: Sequence Data Mining Approach for Detecting Type-3 Clones, Proc. 2nd International Conference on Advances and Trends in Software Engineering(IARIA SOFTENG 2016), pp.82-88 (2016).
- 14) 情報処理推進機構 (IPA): ソフトウェア開発データ白書 2014-2015, 日経 BP 社, (2014).
- 15) Ding, B., Lo, D., Han, J., and Khoo, S.-C.: Efficient Mining of Closed Repetitive Gapped Subsequences from a Sequence Database, Proc. 25th IEEE International Conference on Data Engineering, pp.1024-1035 (2009).
- 16) Needleman-Wunsch algorithm, [https://en.wikipedia.org/wiki/Needleman%E2%80%93Wunsch\\_algorithm](https://en.wikipedia.org/wiki/Needleman%E2%80%93Wunsch_algorithm), (2016).
- 17) Smith-Waterman algorithm, [https://en.wikipedia.org/wiki/Smith%E2%80%93Waterman\\_algorithm](https://en.wikipedia.org/wiki/Smith%E2%80%93Waterman_algorithm), (2016).
- 18) Hamilton, H.: Apriori Implementation, [http://www2.cs.uregina.ca/~dbd/cs831/notes/itemsets/itemset\\_prog1.html](http://www2.cs.uregina.ca/~dbd/cs831/notes/itemsets/itemset_prog1.html), (2012).