ギャップを有するコードクローン検出のための頻出系列マイニング手法

宇田川　佳久*

Frequent Sequence Mining Techniques for Gapped Clone Detection

Yoshihisa Udagawa *

Abstract　Generally, code clones are altered by changing, adding and/or deleting statements after copying the original fragments of codes. Thus, the problem to find code clones essentially results in the detection of strings that partially match with gaps. We employed a maximal frequent sequence mining algorithm, and a modified LCS algorithm for computing the degrees of matches and gaps to identify Type-3 clones. The experimental results using *Java lang* package show that the elapsed times of our mining algorithm take approximately 4.8*N*t where N denotes the number of gaps, and t denotes the process time in the case of gaps 0.

はじめに

実用的な規模のソフトウェアには、コードクローンと呼ばれる類似したソースコード片が含まれている。ソフトウェアの開発では、類似したロジックが発生し、そのロジックを実装するために、類似したソースコードをコピーし、一部を修正するという開発手法が採用されているためである。コードクローンは、開発効率を高めるという点では、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は、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、関連するすべての類似ソースコードを検索し、修正する必要がある。コードクローンの存在は、ソフトウェアの脆弱性（vulnerability）の一因となっている[1]。これまでに、コードクローンを検索するために数多くの研究がなされてきた。

コードクローンは、構文上、以下の 3 種類に分類される[2]。

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

定義より、タイプ 2 はタイプ 1 を含み、タイプ 3 はタイプ 2 を含む。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ことから、実用的にもタイプ 3 クローンの検出が重要である。そのため、タイプ 3 クローンを検出する研究がここ 10 年で盛んになってきた。

Z. Li 氏らは[3]、ソースコードをトークン列に変換した後に、飽和頻出系列(Closed Frequent Sequence)を検出する CloSpan (Closed Sequential Pattern Mining)アルゴリズムを使用し、ギャップ（対応関係がない行数）が 1 までのコードクローンの検出を行っている。S.Ducasse 氏らは[4]、Gap を含む文字列一致において、6 種類の Code 正規化手法(変換）がクローン検索に与える影響について実験している。K. C. Roy 氏らは[5]、ソースコードを pretty-printing 手法でクリーニングした後に、最長共通部分列（LCS: Longest Common Subsequence)アルゴリズムを使ってタイプ 3 クローンの検出を行っている。H. Murakami 氏らは[6]、ソースコードをトークン列に変換した後に、Smith-Waterman アルゴリズムを用いてタイプ 3 クローンの検出を行っている。

本文では、Java のソースコードを対象としたコードクローンの検出手法について述べている。コードクローンを実用的な観点から解析するために、Java の文法に即した構文解析を行い、変数名の影響を除去し、制御文とメソッド呼出し文のシーケンス（列）から構成されるプログラム構造に変換する。変換されたプログラム構造に対し、独自に開発した頻出系列マイニングアルゴリズムを適用し、タイプ 3 クローンの検出を行っている。この研究の特徴は、制御文と変数名の影響を除去したメソッド呼出しから構成されるプログラム構造を対象としていること、および、極大頻出系列(Maximal Frequent Sequence) [7] 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抽出できることである。極大頻出系列を効率的に抽出するアルゴリズムとしては、P. Fournier-Viger 氏らの VMSP (Vertical mining of Maximal Sequential Patterns)がある[7]。VMSP は、バスケット分析への適用を考慮して“Item 集合の系列”を

* 東京工芸大学工学部コンピュータ応用学科教授
2015 年 9 月 24 日 受理

対象としているのに対し、本手法は“文字列の系列”を対象とし、簡潔な実装を実現している。また、与えられた部分系列を複数回含む系列も抽出しているという特徴がある。なお、ギャップを含む系列の一致数とギャップ数の計算は、最長共通部分列(LCS)アルゴリズムを採用した[8]。

以降、コードクローン抽出の処理の概要、独自に開発した頻出系列マイニングアルゴリズム、および、極大頻出系列を抽出するアルゴリズム、実験結果、実験で検出された注目すべきコードクローンの順に述べる。

研究の概要

(1) コード複製の検出処理の流れ

図 1 に本研究におけるコード複製の検出処理の流れを示す。「ソースコードの構造抽出」では、当該コードクローン検出処理が対象とするプログラム構造を抽出する。

本研究では、ギャップを含むコードクローンの抽出を行っている。ギャップを含む文字列の一致度合いは、最長共通部分列アルゴリズム[8]によって計算する。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、最長共通部分列アルゴリズムを有効に機能させるためには一つの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「抽出した構成要素を 32 進 3 文字に変換」である。

「ギャップを含む頻出系列の抽出」では、独自に開発した頻出系列マイニングアルゴリズムを、32 進 3 文字に変換された文字系列に適用する。「最大頻出系列の抽出」処理では、抽出された大量の頻出系列から、極大頻出系列(Maximal Frequent Sequence)を抽出する。なお、極大頻出系列とは、“頻出系列であり、かつ、その上位の系列が頻出系列でない系列”と定義される(A sequence is maximal frequent if none of its immediate super-sequence is frequent)[7]。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を示す。

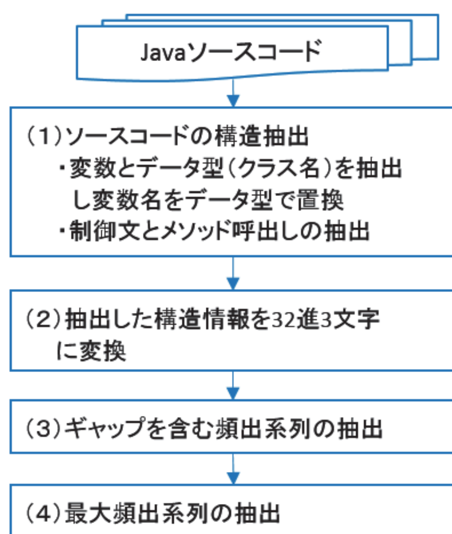

図 1 コード複製の検出処理の流れ

(2) ソースコードの構造抽出

ソースコードの構造抽出では、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[9]。

(a)段付けタブ、コメント、空白、空行など削除

(b)変数と変数の値域(クラス名またはデータ型)の対応を抽出し、変数名をクラス名またはデータ型に置き換える

(c)クラス名、メソッド名の抽出

(d)Java の制御文の抽出

(a)の処理により、タイプ 1 クローンに対応できる。(b)により変数の違いを除去し、タイプ 2 クローンへの対応を行っている。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)は、ソースコードの構造を抽出する主要な処理であり、クラス内に定義されたメソッドごとに、制御文とメソッドの呼び出しを抽出する。

クラス名、メソッド名と引数は、メソッドが定義されているクラス名とともに、下記の構文で抽出する。

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。(b)の処理は、変数名を該当する クラス名 または データ型 に置き替える。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、クラス名またはデータ型の違いは検索結果に反映される。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。

Java の制御文をネスト構造とともに抽出する。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抽出対象とする。

・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

(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 1.7.0.45* の *lang* 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:　210 個
クラス数:　301 個
メソッド数:　2,527 個
ソース総行数:　67,677 行

Java SDK 1.7.0.45 の *lang* パッケージの規模は、中規模システムと位置付けられるものである[10]。図 2 は抽出したソースコード構造の例であり、*StringCoding.java* ファイルの *encode(char[] ca, int off, int len)*　メソッドの構造である。

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

図 2 抽出したソースコードの構造情報の例

(4) 構造情報の 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、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、ソースコード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、類似するソースコードの検索においては同じ重みである必要がある。当該研究では、文字列の長さの影響を排除するために、前処理で抽出したソースコードを構成するすべての識別名を、3 桁の 32 進数（最大で 32,768）で表現している。この処理は、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 は、図 2 に対応する 32 進数で表記されたコードシーケンスである。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, {                          027, registerNatives()
002, super()                    028, System.getSecurityManager()
003, }                          029, ClassLoader.getClassLoader()
004, return                     02A, SecurityManager.checkPermission()
005, if{                        02B, checkMemberAccess()
006, ensureCapacityInternal()   02C, IllegalAccessException()
…                               …
```

図 3　3 桁の 32 進数と識別名のリスト

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

(A)処理前（図 2 に対応する系列）

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

(B)処理後

図 4　3 桁の 32 進数によるシーケンスの表現

頻出系列抽出アルゴリズムの概要

(1) 頻出系列抽出について

頻出するアイテム(item)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る。1994 年に Agrawal らが提起した頻出する集合を列挙すること（頻出集合列挙問題）と、それに対する効率のよい 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[11]。Apriori アルゴリズムが対象としたのは、スーパーマーケットでの購買品のパターンを見つけ、販売促進や店舗レイアウトに役立てようというバスケット分析を指向したものであるが、その枠組みが一般的なデータ解析に適用できる柔軟なものである。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 $D=\{t_1, t_2, \ldots, t_n\}$、アイテムの集合 t_i が $\{I_1, \ldots, I_m\}$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、与えられた最小サポート数（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する。Support(X)でデータベース D に出現する X の頻度とする。一般に、アイテム集合 X、Y が X⊂Y なら、support(X)≧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、この性質を利用している[11]。

図 5 に Apriori アルゴリズムの擬似コードを示す。Apriori アルゴリズムの入力は、データベース D と閾値 minSup (最小サポート数)であり、出力は minSup 以上出現するデータの集合である。初期値として(k=1)、要素数が 1 コである集合 F_1 をデータ集合から生成する。次に、apriori_gen()メソッドを使って、集合 F_1 の要素から 2 個の要素の組合せを生成し、それらの組み合わせを要素とする

集合 C_2（頻出データ集合の候補）を生成する。続いて、図5の for 文で、集合 C_2 のすべての要素に対し、その要素がデータ集合（T）に出現する回数をカウントする。For 文に続く if 文では、指定された最小サポート数（minSup）以上出現する要素を判別し、それに続く2行の実行文で集合 LS と集合 F_2 の要素として登録する。ここで集合 LS は、このアルゴリズムが停止するまでに抽出された要素の和集合であり、apriori アルゴリズムの処理結果である。この処理は F_k の要素数が0個になるまで繰り返される。なお、次候補を生成する apriori_gen()メソッドについては、文献11などに示されているため、詳細は省略する。

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

図5 Apriori アルゴリズム

Apriori アルゴリズムの原理は、アイテム系列（シーケンス）にも適用でき、これまでに、CM-SPADE、 PrefixSpan、CloSpan、 ClaSP、 VMSP を含む、数多くの頻出系列抽出アルゴリズムが実装されている[12]。

(2)間欠を含む頻出系列の抽出

筆者が独自に開発したアルゴリズムは、プログラムを構成する文の頻出系列を抽出するために開発されたものである。Apriori アルゴリズムと同様に、頻出であることを判別するための最小サポート数（minSup）と許容される最大の間欠（最大間欠）maxGap をパラメータとする。2つの系列の比較には、LCS アルゴリズムを採用して、一致する要素数と間欠数を計算する。なお、一つの系列が、与えられた部分系列を複数回含む場合にも対応しているという特徴がある。

間欠を考慮した頻出系列抽出の概要

図6に頻出系列の例を示す。各系列の最初のアイテムは、Java プログラムのメソッド名に対応するものであり、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6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、プログラム要素の前後の繋がりを示す。

図 7 は、最大間欠 maxGap が 0 で、最小サポート数 minSup が 50%(出現数は 2 以上)の場合の頻出系列の抽出結果を示している。要素 005 は、6 回出現しており、出現する箇所は、MTHD1、MTHD2、MTHD3、MTHD4 であり、MTHD3 と MTHD4 では 2 箇所で出現している。要素の系列 005→003→は、3 回出現しており、出現する箇所は、MTHD1 と MTHD3 であり、MTHD3 では 2 箇所で出現している。

図 8 は、最大間欠 maxGap が 1 で、最小サポート数 minSup が 50%の場合の頻出系列の抽出結果を示している。要素 005 は、6 回出現しており、出現する箇所は、maxGap が 0 の場合と同様に、MTHD1、MTHD2、MTHD3、MTHD4 であり、MTHD3 と MTHD4 では 2 箇所で出現している。要素の系列 005→003→は、maxGap が 0 の場合と異なり、5 回出現している。出現する箇所は、MTHD1、MTHD 2 と MTHD4 で 1 箇所ずつ、MTHD3 では 2 箇所で出現している。間欠が 1 でマッチしているのは、MTHD 2 の 005→00A→003 と MTHD4 の 005→006→003 である。

図 9 は、最大間欠 maxGap が 2 で、最小サポート数 minSup が 50%の場合の頻出系列の抽出結果を示している。要素 005 は 6 回、要素の系列 005→003→は 5 回、要素の系列 005→006→は 2 回出現している。なお、要素の系列 005→006→は、間欠 2 で MTHD3 の 005→003→00F→006 とマッチし、MTHD4 の 005→006 とマッチしている。

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

図6 頻出系列データベースの例

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

図7 間欠を含まない場合の頻出系列の抽出例

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

図8 間欠が1 (Gap=1)の場合の頻出系列の抽出例

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

図9 間欠が2 (Gap=2)の場合の頻出系列の抽出例

本頻出系列抽出アルゴリズムの特徴

本研究で開発した頻出系列抽出アルゴリズムは、apriori アルゴリズムの原理に基づくものである。両アルゴリズムとも、最小サポート数 minSup を引数として取り、この最小サポート数以上に出現するデータを抽出する点が共通している。ただし、前者は集合、後者はシーケンスを扱うことから、以下の点で異なっている。

(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、頻出シーケンス抽出アルゴリズムの初期値は制御文である。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、注目すべきソースコードなら、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 準拠アルゴリズムでは、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、頻出シーケンス抽出アルゴリズムでは、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、ひとつ前の頻出シーケンス集合 T_{k-1} を基準として、処理対象とするデータベースを検索することで、頻出シーケンス集合の候補 C_k を生成することができる。ただし、1 つのソースコードのシーケンスに、複数の頻出（サブ）シーケンスが含まれることがある。例えば、A→B→A→C→A→B→D というソースコードのシーケンスには、A→B というシーケンスが 2 個含まれている。そのため、与えられた 1 つのソースコードのシーケンスから、一致する可能性のあるすべての部分シーケンスを生成する。例えば、A→B→A→C→A→B→D からは、A→B→A→C→A→B→D に加え、A→C→A→B→D という（サブ）シーケンスを切り出し、これも処理対象に含めている。

(C) 間欠(Gap)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行っている。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[13]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[14]がある。これらのアルゴリズムは、不一致（ミスマッチ）と間欠（ギャップ）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することから、本研究では、LCS アルゴリズムを採用した。

(3) 本研究で開発した頻出系列アルゴリズム

図 10 は、本研究で開発した頻出系列アルゴリズムの概要を示している。図 10 の変数 k は、検索の回数を示す。3 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 の制御文とした。これは、有意なプログラムなら、制御文が先行するという前提を反映したものである。5 行目の Retrieve_Cand()メソッドは、S_k に含まれる各要素（検索キー）に対し、系列の長さが 1 長い系列を検索するメソッドである。6 行目と 7 行目は次の検索に向けた変数の更新と初期化を行っている。8 行目から 15 行目までは、頻出系列を洗い出している。

```
 1 GProve(String[] args){
 2     k= 1;
 3     LinkedList<String> Sk に探索の初期値(制御文)をセットする。
 4     do {
 5         Retrieve_Cand();  // 今回の検索キーよりも1個長い系列検索する
 6         k= k+1;
 7         Sk.clear();  //  ふるいに掛けた後の系列を記憶する
 8         while( Ck のすべての要素  e に対し){
 9             if ( Ck[e] の発生個数 >= minSup ){
10                     探索Key系列と発生個数＋発生箇所のリストを結果Fileに出力する
11                     while(Gapの範囲で探索Key系列に一致する系列をDBから検索する){
12                         Sk.add(Gap同義語);
13                     }
14               }
15         }
16     } while (Sk.size() > 0);
17 }
```

図 10 頻出系列の抽出アルゴリズム

図 11 は、間欠を考慮した系列候補を生成するアルゴリズムであり、図 10 の 5 行目の Retrieve_Cand()メソッドの実装である。7 行目の for 文は、検索の対象とする系列が、処理対象とするデータベース内の 1 個の系列に複数回発生することに対応する処理を行っている。記憶領域として使用されている Ck、 CkMD、 CkSyn は、Java の HashMap を使って実装している。8 行目では、2 つの系列 s、t 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 には検索のキーとする系列、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。従って、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|で計算される。9 行目から 13 行目では、系列と間欠の長さを満たすデータベース内の要素に関する情報を Ck、CkMD、 CkSyn に記憶している。

```
 1 Retrieve_Cand(){
 2     Ck.clear();   // 探索キー系列の発生個数を記憶する領域
 3     CkMD.clear();  // 探索キー系列の発生位置を記憶する領域
 4     CkSyn.clear();  // 探索キー系列に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データ t に対し) {
 7             for ( t で  s の先頭の要素と一致する箇所 p に対し){
 8                 LongestCommonSubsequence(s, t);
 9                 if (一致した要素数 >= k && Gap数  <= maxGap ){
10                     Ck.put(探索キー系列, 発生個数);
11                     CkMD.put(探索キー系列, 発生した行の番号);
12                     t からサブシーケンスを切り出す(= 検索キー同義語系列 )
13                     CkSyn.put(検索キー同義語系列, 探索キー系列);
14                 }
15             }
16         }
17     }
18 }
```

図 11 系列候補の生成アルゴリズム

(4) 極大頻出系列の抽出

一般に、頻出系列は大量に抽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抽出する技法が研究されてきた [7][12]。極大頻出系列 (Maximal Frequent Sequence)は、頻出系列であり、かつ、その上位の系列が頻出系列でない系列と定義されている。

本研究で開発した頻出系列アルゴリズムでは、間欠を考慮するために検索で使うキー系列だけでは、極大性を判定することができない。検索キー系列に対し与えられた最小サポート数 minSup と最大間欠 maxGap の範囲でマッチする系列を検索キーの同義語系列と呼ぶ(図 12)。Match(x,y)を系列 x, y の要素が一致する数、Gap (x,y)を系列 x, y の要素の間欠とするとき、検索キーS_k の同義語系列 $SynS_k$ は以下の式で定義される。

$$SynS_k = \{\ x \mid Match(x, S_k) \geqq k\ \&\ Gap(x, S_k) \leqq maxGap\}$$

ギャップを含む極大頻出系列は、同義語系列を使って次のように定義される。

定義：与えられた検索キー系列 S_k に対応する同義語系列 $SynS_k$ のすべてに対し、要素が 1 個多い頻出系列が存在しないとき、S_k は極大頻出系列の要素である。

例えば、図 7 に示した頻出系列に対応する極大頻出系列は、{005→003→, 005→006→}である。

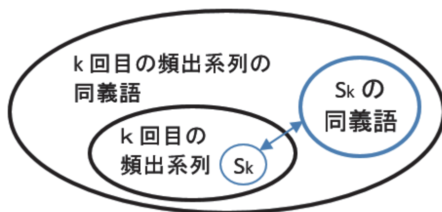

図 12 頻出系列、系列の同義語と極大頻出系列

実験結果

(1) 実験対象データの概要

Java SDK 1.7.0.45 の *lang* パッケージから抽出したプログラム構造の主な指標は以下の通り。

- 1 行以上の制御文またはメソッド呼出しを含む有意なメソッドは 2,522 個である。
- 識別子の種類は 1,286 個である。
- 識別子の総数は 18,205 個である。
- 抽出されたメソッドの最大の系列の長さは、*java.lang.invoke.MethodHandleNatives.java* ファイル内の *Constants* クラスの *isCallerSensitiveMethod()* メソッドの 127 である。
- 抽出されたメソッドの最大のネストの深さは、*java.lang.Class.java* の *getEnclosingMethod()* メソッドの 7 である。

(2) 間欠と検出された系列の最大長さ

本研究で開発したアルゴリズムは、間欠を含めた系列の検索を行うことを特徴としている。図 13 は、最小サポート数が 2 から 10 について、間欠と検出された系列の最大長さをグラフ表示したものである。最小サポート数 minSup が小さくなるに連れて、検索された系列の最大長さが長くなる。minSup が 2 で、最大間欠 maxGap が 0 と 1 のときは、系列の最大長さは 84 と 91 であるが、間欠が 2、3、4 のときは、系列の最大長さは 120 であった。

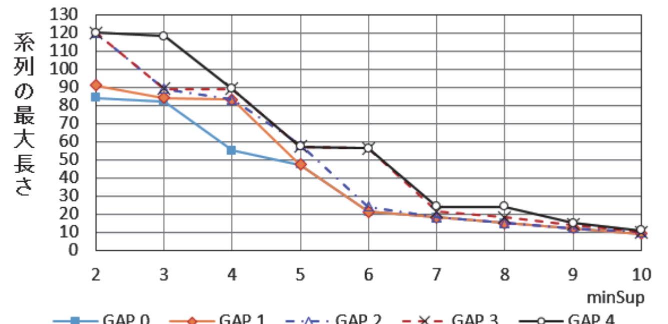

図 13 間欠と検出された系列の最大長さ

(3) 頻出系列と極大頻出系列

一般に、頻出系列は大量に抽出されることが知られている。本研究では、頻出系列と極大頻出系列の両方を抽出し、抽出された個数を計測した。

図 14 は、最小サポート数 minSup が 2 から 10 について、検出された頻出系列の数をグラフ表示したものである。最大間欠 maxGap が 0 以外の場合に比べて、maxGap が 0 の場合の抽出された個数（正方形のマーカーの実線）は 1/7 から 1/60 であることから、図 14 の右側の軸で表示している。

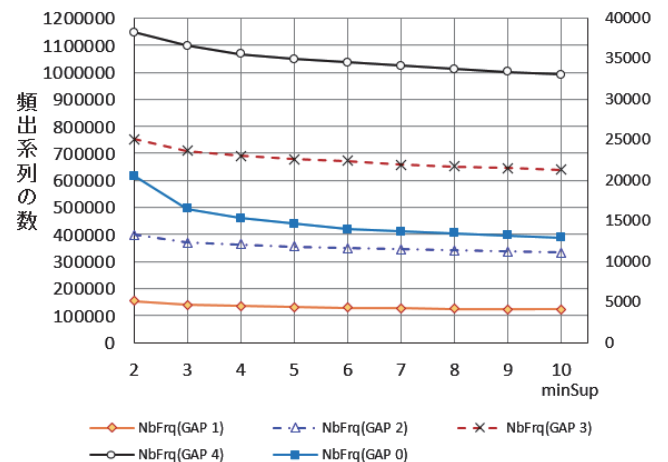

図 14 検出された頻出系列の個数

図 15 は、minSup が 2 から 10 について、検出された極大頻出系列の数をグラフ表示したものである。図 14 と同様に、maxGap が 0 の場合（正方形のマーカーの実線）の個数は、右側の軸で表示している。

この実験から、抽出された極大頻出系列の数は、頻出系列の数に比べて、1/30 から 1/100 であった。また、間欠の

数が大きくなるほど比率は小さくなり、maxGap が 4 で minSup が 2 の場合では、頻出系列の数は 1,148,613 個、極大頻出系列の数は 11,456 個であり、比率は約 1/100 であった。

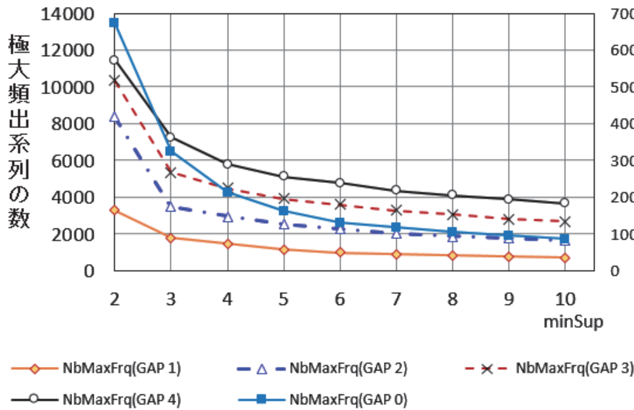

図 15 検出された極大頻出系列の個数

(4) 処理時間

図 16 は、最小サポート数が 2 から 10 について、頻出系列を抽出するのに要した時間をグラフ表示したものである。maxGap が 0 以外の場合に比べて、maxGap が 0 の場合の処理時間(正方形のマーカーの実線)は 1/5 から 1/18 であることから、右側の軸で表示している。この結果から、maxGap が 0 の場合の処理時間に比べ、maxGap が 1 では 4.9 倍、maxGap が 2 では 9.7 倍、maxGap が 3 では 15.0 倍、maxGap が 4 では 18.5 倍の処理時間を要していることが判明した。近似値は、4.8*N である。

図 17 は、頻出系列から極大頻出系列を抽出するのに要した時間をグラフ表示したものである。図 16 と同様に、maxGap が 0 の場合（正方形のマーカーの実線）の個数は、右側の軸で表示している。

計測で用いた PC 環境は、以下の通りである。

CPU: Intel Core i3-540 3.07 GHz

主メモリ: 8 GB

OS: Windows 7 64 Bit

Java のバージョン: Java 1.7.0

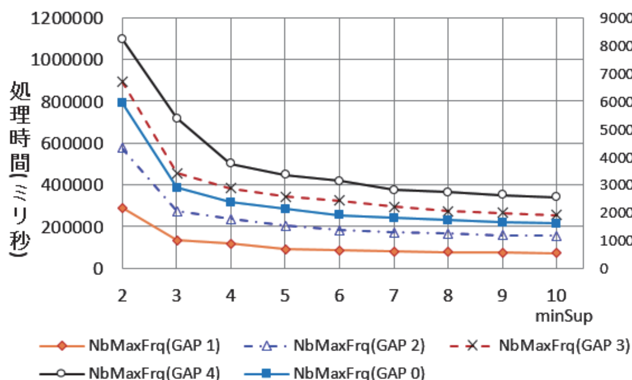

図 16 頻出系列を抽出するのに要した処理時間

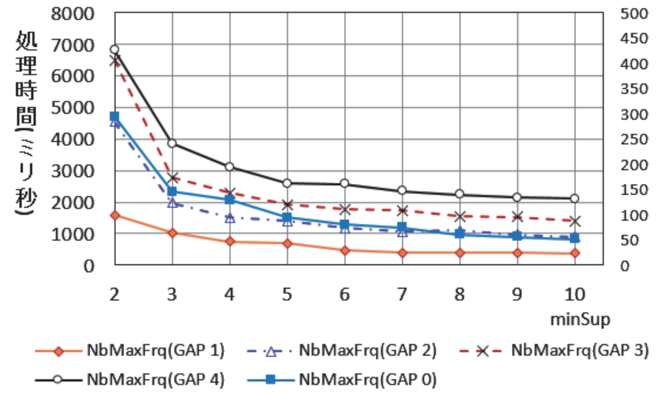

図 17 極大頻出系列を抽出するのに要した時間

検索されたソースコードの例

一般に、間欠が大きくなると検索結果の一致の度合いの精度が悪くなる傾向があることから、ここでは maxGap が 1 の事例を示す。なお、紙面の制約から、短い系列の長さの事例を示す。

図 18 は、005→004→003→00C→141→142→00V→ という系列で、間欠 1 でマッチした 4 個のメソッドの一覧である。メソッド名と引数から、これらは類似した処理を行っていることが推測される。

005→004→003→00C→141→142→00V→ という系列と完全に一致するメソッドは 3 個あった。図 18 の 3 番目のメソッド(図 1 に示した *StringEncoder::encode(char[] ca, int off, int len)*メソッド）は、*CharsetEncoder.reset()*文(32 進数表現で 14F)が挿入されていることが分かる。このコーディングの違いは、プログラマに確認する必要があるものと思われる。

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

図 18 間欠が 1 で一致するメソッド

まとめと今後の研究方針

本文では、間欠を含む極大頻出系列マイニングアルゴリズムを使ったコードクローン検索の試みについて述べた。本研究で開発したアルゴリズムを *Java SDK 1.7.0.45 lang* のソースコードに適用した結果、間欠が 0 の場合の処理時間を t としたとき、間欠が N の場合の処理時間は約 4.8*N*t を要していることを確認した。処理時間はデータの性質に依存するものである。他のソースコードに対する実験を行って、どのような性質があるときに、間欠に比例する処理時間が必要になるか検証する予定である。

一般に、頻出系列の抽出では、大量の系列が抽出される。この研究では、抽出された系列の絞込みを、極大頻出系列を使って行った。しかしながら、抽出された極大頻出系列は数千件に及ぶものであり、更なる絞込みの技法を開発する必要があることも判明した。

参考文献

1) M. F. Zibran, R. K. Saha, C. K. Roy, and Kevin A. Schneider: Genealogical insights into the facts and fictions of clone removal, ACM SIGAPP Applied Computing Review, Volume 13 Issue 4, (Dec. 2013), pp.30-42.
2) 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.
3) Z. Li, S.Lu, S. Myagmar, and Y. Zhou: CP-Miner: A Tool for Finding Copy-paste and Related Bugs in Operating System Code. Proceedings of the 6th Symposium on Operat-ing System Design and Implementation (Dec, 2004), pp.289-302.
4) S. Ducasse, O. Nierstrasz, and M. Rieger: On the effectiveness of clone detection by string matching, J. Softw. Maint. Evol. Res. Pract. (2006), pp.37–58.
5) C. K. Roy and J. R. Cordy: NICAD: Accurate Detection of Near-Miss Intentional Clons Using Flexible Pretty-Printing and Code Normalization. Proceedings of the 16th IEEE Inter-national Conference on Program Comprehension (June 2008), pp.172-181.
6) H. Murakami, K. Hotta, Y. Higo, H. Igaki, and S. Kusumoto: Gapped Code Clone Detection with Lightweight Source Code Analysis, (May 2013), pp.93-102.
7) P. Fournier-Viger, C.-W. Wu, A. Gomariz, and V. S. Tseng: VMSP: Efficient Vertical Mining of Maximal Sequential Pat-terns. Proc. 27th Canadian Conference on Artificial Intelligence (May 2014), Springer, LNAI, pp. 83-94.
8) Longest common subsequence, http://rosettacode.org /wiki/Longest_common_subsequence (Sept. 2014).
9) Y. Udagawa: A Novel Technique for Retrieving Source Code Duplication, Proc. 9th International Conference on Systems (ICONS 2014), Vol. 9, pp. 172-177, Feb. 2014.
10) 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).
11) R.Agrawal and R.Srikant: Fast algorithms for mining associa-tion rules, In Proceedings of 20th Int. Conf. Very Large Data Bases, VLDB (1994), pp.487-499
12) An Open-Source Data Mining Library: http://www.philippe-fournier-viger.com/spmf/index.php (August 2015).
13) Needleman–Wunsch: http://bi.biopapyrus.net/seq/needleman%E2%80%93wunsch.html (Sept. 2014).
14) Smith-Waterman: http://bi.biopapyrus.net/seq/smith-waterman.html (Sept. 2014).