Vol.2018-SE-199 No.14
2018/7/18

トークンの N-gram によるプログラム表現を用いたコードクローン検出手法

向井 達郎[1,a)]　小林 靖明[2]　紫藤 佑介[2]　山本 章博[2]　宮本 篤志[3]　松村 忠幸[3]　嶺 竜治[3]

概要：同一または類似したコード片の対はコードクローンと呼ばれ，ソフトウェアのメンテナンスを困難にさせる要因のひとつとして考えられている．このような背景の元で，数多くのコードクローン検出手法が提案されてきた．ソフトウェアの大規模化や GitHub などのソースコードリポジトリの発達に伴い，コードクローンの検出対象も拡大しており，高精度でスケーラブルなクローン検出手法の重要性が増してきている．本稿では，大規模ソフトウェア群を対象としたコードクローン検出手法である SourcererCC(Sajnani et al., ICSE 2016) に基づいて，ソースコードから得られるトークン列の N-gram を用いたコードクローン検出手法を提案する．この手法は SourcererCC の一般化として位置付けられ，SourcererCC の高速化方法が同様に利用できるため，大規模なソフトウェア群にも適用性がある．計算機実験の結果，N-gram の N を増やすことで，コードの構文レベルでの類似度があまり高くないコード対において，SourcererCC に比べて高いコードクローンの検出精度を実現できることを確認した．さらに，識別子やリテラルの正則化によって，どの程度検出精度に影響が出るのかを検証した．

キーワード：コードクローン，N-gram，大規模ソースコード

1. はじめに

コードクローンとは，ソースコード中の互いに一致あるいは類似したコード片の対のことであり，主にコード片のコピー&ペーストによって生じる．この際，単純なコピー&ペーストだけでなく，ペースト先のコーディングスタイルや必要としている機能に応じて，ペーストしたコード片が編集されることがある．コードクローンはバグの伝搬やソフトウェアの不必要な肥大化などを引き起こす可能性があるため，ソフトウェアの保守を困難にする要因のひとつとして考えられている．このようなコードクローンによる弊害への対策として，コードクローンを自動で検出する方法が研究されており，これまでにも様々なコードクローンの自動検出を行うツールが開発されてきた [19], [22]．これらのツールは，コードの剽窃や著作権侵害などの検出といった様々な用途にも利用することができる．

また GitHub や Bitbucket のようなソースコードリポジトリの発達やソフトウェアの大規模化に伴い，コードクローン検出ツールは大規模なプログラム群に対しても現実的な時間でコードクローンを検出することを求められるようになっている．このようなニーズに対して，大規模なソフトウェア群を対象としたコードクローンの検出ツールが開発されている [7], [13], [15]．それらの中でも，スケーラブルかつクローン検出精度の高いツールとして，Sajinani らによって提案された SourcererCC [21] がある．SourcererCC は，コードをトークンの多重集合と見なして，それらを多重集合間の距離基準を用いて比較することでクローン検出を行うツールである．このツールは従来のツールに比べて，高速にコードクローンを検出できる一方で，タイプ 3 クローン（詳細は次節）と呼ばれるコードクローンの検出精度には改善の余地を残している．そこで本研究では，SourcererCC のトークンの多重集合の比較という手段を踏襲することで高速な検出を可能にしつつ，タイプ 3 クローンへの検出精度を高めるために，ソースコードのトークンの多重集合のみを用いるだけでは捉えることのできない特徴量を考慮した手法を新たに提案する．

ここで，自然言語処理における言語モデルに着目する．自然言語処理においては，N-gram 言語モデルや N-gram 統計は古くから様々な場面で利用されている [4], [5]．これらがうまく働く要因のひとつは，単純にそれぞれの単語のみを用いるときに比べ，N を大きくすることでそれぞれの単語とその文脈を考慮することができるためである．プロ

[1] 京都大学大学院総合生存学館
[2] 京都大学大学院情報学研究科
[3] （株）日立製作所基礎研究センタ
[a)] tamukai@iip.ist.i.kyoto-u.ac.jp

グラム言語に対しても同様の考察があり，Hindle ら [10] は「理論的にはソースコードは非常に複雑な記述法を許すが，実際のソースコードは単純かつ慣用表現を多く含むため，統計的言語モデルによって統計的性質が予測可能」という仮説を提唱し，N-gram 言語モデルを用いて，その仮説の妥当性を実験的に示した．

本研究で提案する手法は，コード片をトークンの N-gram を用いて多重集合に変換し，多重集合間の距離基準を用いて，それらを比較することでコードクローンを抽出する手法である．ここで，ソースコードをトークンの 1-gram を用いて多重集合化した場合，提案する手法は既存手法の SourcererCC に一致する．提案手法では，自然言語処理において N-gram を用いた場合と同じように，各トークンにそのコード中の文脈の情報を付与することが意図されている．トークンのみを用いた場合よりもより複雑な特徴量をクローン検出に取り入れられることが期待される．計算機実験の結果，コードの構文レベルでの類似度が低いコードクローン検出においては，N を増やすことで検出精度が上昇することが確認できた．また，識別子やリテラルの正則化の効果を確認するため，いくつかのクローン検出ツールで用いられているトークンの正則化 [8], [12] を提案手法に組み込んだ実験も行った．さらに，提案するトークンの N-gram がコードの構造上の特徴を反映できているかを確認するため，木構造間の非類似度を測る指標である pq-gram 距離 [1] を用いて，コードから得られる AST(Abstract Syntax Tree) 間の距離とトークンの N-gram を用いた距離の関係とを調べた．その結果，それらは高い相関を示すことが実験的に確認された．これより，提案手法は単にトークン同士の比較ではあるが，AST 上での距離をある程度考慮できている可能性があることがわかった．

本稿の構成は次の通りである．2 節では研究の背景であるコードクローンやその他の用語の定義を行う．3 節では本研究の元となる先行研究について説明する．4 節では提案手法について説明する．5 節では提案手法のクローン検出精度について計算機実験による検証について示し，最後に本研究の結論と今後の課題について述べる．

2. 準備

本節では，以降で用いる用語の説明を行う．クローン検出の対象は各クローン検出のツールによって様々だが，ここではひとつの検出対象を**コード片** (code flagment) と呼ぶ．本研究の実験においては，ひとつのコード片はひとつの関数/メソッドに対応しているが，それらを異なる粒度に変更することは容易である．

2.1 コードクローンの分類

コード片のコピー&ペーストが行われていたとしても，元のコードと完全に同一であるとは限らない．例えば，コピー先のコーディングスタイルに合わせて，空白や改行が挿入または削除されたり，識別子名やリテラルが変更されたりすることは頻繁に起こり得る．また，いくつかの機能を追加・削除するためにコード片の一部を改変することも考えられる．コードクローンの定義を厳密に決めることは困難だが，本研究では，コードクローンの分類として標準的に用いられる以下の 4 つの分類 [19] を用いる．

タイプ 1 のコードクローン
: コメント及び空白や改行などのコーディングスタイルを除いて，内容が完全に一致するコード片．

タイプ 2 のコードクローン
: タイプ 1 に加えて，識別子名，リテラル，型名の変化を考慮して一致するようなコード片．

タイプ 3 のコードクローン
: タイプ 2 に加えて，いくつかの文の追加や削除を行って得られるコード片．

タイプ 4 のコードクローン
: 同じ処理を行うが，文法レベルで異なるコード片．

2.2 コードクローン検出アルゴリズムの分類

コードクローン検出アルゴリズムには様々なアプローチが知られているが，それらは文字列ベース [2], [20]，トークンベース [12], [17], 木ベース [3], [11]，意味ベース [16] などのように分類することができる．文字列ベースのアプローチはプログラムをテキストとして扱うのに対し，トークンベースのアプローチは字句解析によって得られた情報を利用し，木ベースのアプローチは構文解析を用いて得られる抽象構文木 (Abstract Syntax Tree, AST) を利用する．そして意味ベースのアプローチはプログラム依存グラフなどの高度な処理を用いる．文字列ベースやトークンベースの手法は他の手法に比べて，プログラムの持つ構造を陽には利用していないため，より高度な分析ができない反面，複雑な処理が不要なため，高速に動作するという利点がある．

3. 先行研究

Sajnani ら [21] によって大規模なソフトウェア群に適用可能なコードクローン検出手法である SourcererCC が提案されている．この手法はタイプ 3 のクローンの中でも構文レベルで高い類似度を持つようなクローンについては高い精度で検出可能である．さらに，一億行のソースコードにおいても約 1.5 日程度でのクローン検出を達成しており，現在最も高速なクローン検出ツールのひとつである．SourcererCC は，コード片をトークンの多重集合に変換し，集合同士を比較することでコードクローンを検出する手法である．

ここでふたつのコード片をトークン化して得られるトークンの多重集合をそれぞれ B_1 と B_2 とする．B_1 と B_2 の

類似度を計算する方法はいくつか考えられるが，ここでは *Jaccard* 係数[*1]を用いる．

定義 1. 多重集合 X と Y の *Jaccard* 係数を

$$J(X,Y) = \frac{|X \cap Y|}{|X \cup Y|}$$

と定義する．ただし，$\cap$ と $\cup$ はそれぞれ多重集合における共通集合と和集合の演算を表す．また，*Jaccard* 距離を $1 - J(X,Y)$ と定義する．

SourcererCC では，ある閾値 $\theta(0 \leq \theta \leq 1)$ を用いて，$J(B_1, B_2) \geq \theta$ ならば B_1 と B_2 をクローンと判定する．この手法はコード片のトークン化のみを必要とするためにシンプルで，次節で述べる高速化を用いることで巨大なコード群に対するクローン検出に成功している．注目すべきは，クローン検出においてトークンの多重集合のみを必要とするため，多くのプログラミング言語に適用可能で，構文解析のような前処理をも不要とする点である．

3.1 SourcererCC における高速化

SourcererCC の高速化には，複数の集合間における類似度計算 (set similarity join) で用いられているヒューリスティックス [6], [26] が使われている．n 個のコード片が与えられたとき，全てのコード片の間の類似度を計算するためには，$\Theta(n^2)$ 対に関して類似度を計算する必要がある．しかしながら，クローン検出においては閾値が θ を超えるかのみを判別すれば良いため，以下に説明するようなフィルター型のヒューリスティクスを適用して，類似度計算を行う対の数を削減する．

Jaccard 係数を用いる場合，クローンと判定するための必要十分条件はコード片の多重集合 B_1, B_2 において $J(B_1, B_2) \geq \theta$ を満たすときである．よって，$|B_1 \cap B_2| \geq \theta \cdot |B_1 \cup B_2|$ を満たす．簡単のために，$t = |B_1| = |B_2|$ とすると，$t \leq |B_1 \cup B_2|$ より，クローンとして判定されるための自明な必要条件は，$|B_1 \cap B_2| \geq \theta \cdot t$ となる．これは B_1(または B_2) において少なくとも θ の割合以上 B_2(または B_1) と共通部分を持たなければならないことを意味する．つまり，B_1 の任意の $s = \lceil (1-\theta)t \rceil$ よりも大きい任意の部分集合は B_2 と共通部分を持つ．そのため，大きさ $s+1$ の部分集合を用いてインデックスを作成し，そのインデックスが共通部分を持つ場合にのみ類似度を計算する．さらに，このインデックスを効果的に働かせるために，データセットに現れる全てのトークンをその出現頻度順で昇順ソートし，各トークンの多重集合は，できるだけ出現頻度が小さいトークンのみを使ってインデックスを作成するようにする．これは，ソースコード中には高頻度で現れる識別子 (例えば，`i`, `name`, `result` など) やプログラミング言語のキーワード (`if`, `for` など) は，インデックス内で共通しやすいため，それらがインデックスに含まれることを避けるための工夫である．また，ふたつのインデックスで共通するトークン数 c とインデックスに含まれなかったトークン数 $t - s - 1$ をインデックス以外で共通するトークン数の上界として用いて，クローン検出のための単純な必要条件 $(c + t - s - 1)/t \geq \theta$ を満たさなければそれらを比較しないようにすることで，さらなる高速化も行っている．Sajnani らは，これらのヒューリスティクスを適用することで，比較するコード片の対がコード片の数に対して線形になることを実験的に示した．

[*1] [21] では，Jaccard 係数とはわずかに異なる類似度を用いているが，本質的な違いはなく，高速化の手法なども同じく適用可能である．

4. 提案手法

SourcererCC は，コード片をトークンの多重集合と見なし，それらを比較することでクローンの検出を行う．前節で述べたように多重集合の比較とすることで，集合間の類似度計算の高速化手法が利用できるため，非常に高速に動作する．しかしながら，SourcererCC ではトークンの並びから得られる情報は考慮されていない．そのため，タイプ 3 のクローンにおいて，特にコード片に文の追加や削除がより多く行われているクローンの検出は困難になると考えられる．実際に，[21] の実験においては，「構文の類似度」(詳細は 5 節) が低いタイプ 3 クローンの検出に関しては検出精度がそれほど高くはない．このようなクローンを検出するためには単純なトークンの多重集合比較のみではなく，コード片が持つさらなる特徴量に着目する必要があると考えられる．

そこで本研究では，SourcererCC のトークンの多重集合の比較という高速化の性質は踏襲しつつ，SourcererCC ではうまく捉えられない特徴量を考慮した N-gram に基づく新たな手法を提案する．SourcererCC がソースコードをトークンの多重集合と表現していたのに対し，提案手法ではトークンの N-gram の多重集合として表現する．トークンの N-gram とは，コードから得られるトークン列の N 個の連接のことであり，1-gram は単にトークンそれ自身を表す．図 2 は図 1 のメソッドの本文を 4-gram のトークンで集合化したものである．

提案手法は，SourcererCC のコードクローン検出手法に対して，自然言語処理の N-gram を適用し，より一般化させたものと解釈することができる．自然言語処理においては，文字列の N-gram から得られるヒストグラムが文章の剽窃検出 [4] や文書分類 [5] に利用されたり，単純に文字列の編集距離に対する近似 [14], [25] として利用されたりしている．これらの結果は，単純に単語の出現回数のみを見るのではなく，文脈の中でどのように出現するかを考慮することが重要であるという点を示唆している．プログラムコードについても同様のことが成り立つならば，コードク

```
public double M(int a)
{  String s = "abc";
   if(a == 1){
      System.out.println(s);
   }
   return ((double) a);
}
```

図 1: Java 言語で記述されたサンプルコード

```
{ {Strings= , Strings="abc" , s="abc"; , ="abc";if ,
 "abc";if( , ;if(a , if(a== , (a==1 , a==1) , ==1{ ,
 1){System , ){System. , {System.out ,
 System.out. , .out.println , out.println( ,
 .println(s , println(s) , (s); , s);} , );}return ,
 ;}return( , }return(( , return((double ,
 ((double) , (double)a , double)a) , )a); ,
 a);} , );} , ;} , }    }
```

図 2: 図 1 の本文をトークンの 4-gram で集合化した集合

ローンの検出においても，単純にトークンの出現回数のみを見るのではなく，どういった文脈においてトークンが出現するかを考慮することが必要になると考えられる．本研究の意味するところは，タイプ 3 などのより検出の難しいクローン対に対して，自然言語処理系の技術を参考に，これまで考慮しきれていなかった文脈や構造の情報を取り込むことにある．

タイプ 2 やタイプ 3 のコードクローンの検出精度を向上させるテクニックとして，トークンの正則化 (例えば [8], [12] など) がある．本研究では，トークンの N-gram に加えて，このトークンの正則化のテクニックを合わせて適用する．具体的には，数値リテラル，文字リテラル，文字列リテラル，識別子を，それぞれの特殊なトークンに置換することでソースコードを正則化して，正則化されたトークン列からトークンの N-gram 多重集合を得て，Jaccard 係数で比較を行う．トークンの正則化を行うことで，タイプ 2 のクローンで現れるような識別子やリテラルの書き換えに対しては強いロバスト性を持つことが期待できるだけでなく，異なるトークンの N-gram の数が正則化を行わない場合と比較して減少するため，高速化も望める．その一方で，識別子の文字列が持つ意味（例えば `max` という変数名はなんらかの最大値を表すなど）については，失われる可能性があるという点には注意が必要である．

5. 実験

本研究の実験においては，コードクローン検出のベンチマークのひとつである BigCloneBench [23] を用いた．

5.1 BigCloneBench

BigCloneBench は，複数のオープンソース Java プロジェクトで構成されているソフトウェアリポジトリ IJaDataset 2.0 [9] から特定の機能の関数をマイニングすることによって作られたベンチマークである．BigCloneBench には，約 620 万件のクローン対と約 26 万件の偽クローン対が登録されており，それぞれの対には前節で述べたような正則化を施した後の行単位での共有率が「構文の類似度 (syntactic similarity)」として 0 から 1 までの値で登録されている (類似度が 1 であることは，正則化を行った後のふたつのコード片が行単位で完全一致することを意味する)．620 万件のクローン対における構文の類似度の分布を図 3 に示す．クローンタイプの定義より，タイプ 1 とタイプ 2 は構文の類似度が 1 となるため，数多くのクローン対がタイプ 3 またはタイプ 4 であり，そのうちの多くが類似度の低いクローン対であることが確認できる．[23] では，類似度が $[0.7, 1)$ の半区間に属するクローン対を強タイプ 3(Strongly Type-3)，$[0.5, 0.7)$ に属するものを中タイプ 3(Moderately Type-3)，$[0.0, 0.5)$ に属するものを弱タイプ 3+4(Weakly Type-3+4) と定義している．そのため，BigCloneBench に登録されているほとんどのクローン対は弱タイプ 3+4 に分類される．

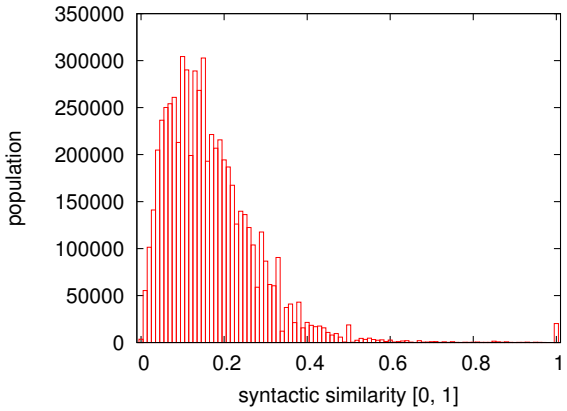

図 3: BigCloneBench のクローン対データにおける構文の類似度の分布．横軸は構文の類似度で縦軸はその類似度を持つようなクローン対の数を表す．

5.2 タイプ 3 クローンの検出精度の検証

タイプ 3 クローン対に対する検出精度を確かめる計算機実験とその結果について示す．データセットについては以下のような方法で BigCloneBench から抽出を行った．5.1 節で述べたように，各クローン対には構文上の類似度が設定されている．単純な無作為抽出をすると，類似度が低い対が数多く抽出されてしまうため，類似度に基づいてある程度均一にクローン対が抽出されるように以下のような方法で抽出を行った．構文の類似度が 1 未満のクローン対を類似度が 0.1 刻みの半区間 $[0.1, 0.2), [0.2, 0.3), \ldots, [0.9, 1.0)$ に分割し，各半区間からコード片の行数が 6 行以上のものを最大で 3 千件ずつ無作為に抽出した．コードの行数に下限を設けることは，検出精度を比較する上でいくつかの手法において行われている標準的な方法である [21]．類似度

l	0.9	0.8	0.7	0.6	0.5	0.4	0.3	0.2	0.1
AUC($N=1$)	∗0.9991	0.9996	0.9989	0.9965	0.9926	0.9862	0.9742	0.9540	0.9175
AUC($N=2$)	0.9990	∗0.9996	∗0.9991	0.9974	0.9949	0.9910	0.9830	0.9673	0.9356
AUC($N=3$)	0.9990	0.9996	0.9991	∗0.9974	∗0.9951	0.9918	0.9850	0.9710	0.9424
AUC($N=4$)	0.9990	0.9995	0.9991	0.9973	0.9950	∗0.9919	∗0.9854	0.9723	0.9451
AUC($N=5$)	0.9990	0.9995	0.9990	0.9972	0.9950	0.9917	0.9854	∗0.9726	∗0.9462
AUC($N=6$)	0.9990	0.9995	0.9989	0.9970	0.9946	0.9914	0.9850	0.9723	0.9462
AUC($N=7$)	0.9990	0.9994	0.9988	0.9968	0.9944	0.9911	0.9846	0.9719	0.9461

表 1: 構文の類似度が $[l, 1.0)$ にあるようなデータに対して N を 1 から 7 へ変化させたときの AUC の変化．∗ を含むセルは，各列の最大値を意味する．

l	0.9	0.8	0.7	0.6	0.5	0.4	0.3	0.2	0.1
AUC($N=1$)	0.9986	0.9985	0.9953	0.9894	0.9812	0.9694	0.9524	0.9288	0.8905
AUC($N=2$)	0.9989	0.9994	0.9982	0.9953	0.9903	0.9823	0.9690	0.9484	0.9125
AUC($N=3$)	0.9990	0.9995	0.9989	0.9968	0.9932	0.9871	0.9762	0.9582	0.9255
AUC($N=4$)	0.9990	0.9995	0.9990	0.9972	0.9941	0.9888	0.9789	0.9622	0.9317
AUC($N=5$)	0.9990	0.9996	0.9991	0.9973	0.9945	0.9896	0.9801	0.9640	0.9349
AUC($N=6$)	0.9990	0.9996	0.9991	0.9974	0.9947	0.9900	0.9807	0.9638	0.9365
AUC($N=7$)	0.9990	0.9996	0.9991	0.9975	0.9949	0.9902	0.9810	0.9651	0.9373

表 2: 構文の類似度が $[l, 1.0)$ にあるようなデータに対して，トークンの正則化を適用したときの $N = 1, 3, 6$ における AUC.

が $[0.0, 0.1)$ の区間については，類似度が低すぎるため実験からは除外した．また，類似度が $[0.9, 1.0)$ に入るクローン対については，3 千件に満たずに約 1800 件であった．偽クローン対については同じく行数の下限を設けて 5 万件を無作為抽出した．最終的にこれらより，各 $l = 0.1, 0.2, \ldots, 0.9$ について類似度が $[l, 1.0)$ に入る抽出されたクローン対と 5 万件の無作為抽出された偽クローン対をひとつのデータセットとして実験を行った．このようなデータセットの偏りを持たせた理由としては，産業用のソースコードには最大で 20%程度のクローンが含まれるという結果 [18] を考慮したためである．クローン検出の評価には，ハイパーパラメータである閾値 θ の存在と正データの偏りを考慮するため AUC を用いた．

N の値を 1 から 7 へ変化させたときの AUC の値の結果を表 1 に示す．ただし，$N = k$ においては，トークン列から $1 \leq n \leq k$ を満たす n-gram 全てを用いて計算している．表 1 からわかるように，構文の類似度が高い場合は N が小さいほど AUC が高く，類似度が低いほど N が大きくなるにつれて AUC が大きくなることがわかる．類似度が高い強タイプ 3 クローンの検出については $N = 1$ でも十分な精度であるが，中タイプ 3 クローンを含むデータ・セットについては，N = 2, 3，さらに弱タイプ 3+4 クローンを含めると，N = 4, 5 で最も検出精度が高かった．これらの実験から，類似度が低いクローンの検出には，トークンの多重集合から得られる特徴量だけでは高い検出精度を得ることができないが，トークンの N-gram を用いた本手法は，それ以外の特徴量をいくらか掴んでいるため，検出精度が向上したと解釈することができる．

また，トークンの正則化を行った場合について，N の値を 1 から 7 へ変化させた時の AUC の値の結果を表 2 に示す．トークンの正則化を行わない場合と同様に，構文の類似度が低い場合には，N を大きくすることで AUC が向上していることがわかる．しかし一方で，表 1 と表 2 を見比べると，トークンの正則化を行わない場合に比べて AUC が全体的に低くなる結果になった．これは，今回用いた BigCloneBench においては，タイプ 2 のような識別子やリテラルの変更を吸収することの効果よりも，正則化を行うことで失われる情報の方がクローン検出精度に対して大きな影響を与えていたためと考えられる．字句そのものについてのロバスト性とのトレードオフを考慮した上で，提案手法に適した正則化については，今後も考えていく必要がある．

5.3 トークンの N-gram 同士の Jaccard 距離と AST 間の距離

次に，N-gram を用いた距離がコードの構造上の特徴をどのくらい反映できているのかを確認するため，トークンの N-gram 同士の Jaccard 距離と AST(Abstract Syntax Tree) 間の距離の相関について調べた結果を示す．ここで，AST はラベル付き根付き順序木である．それらの非類似度を定義する指標として最も有名なもののひとつに木の編集距離 [24] が知られている．木の編集距離は文字列の編集距離の一般化としても知られており，編集操作は文字列の場合と同様にノードの挿入，削除，置換からなる．編集距離は片方の木にいくつかの編集操作を適用し，もう片方の木を得るための最小コストとして定義される．木の編集

N	1	2	3	4	5	6	7
$p=1,q=2$	0.8571	0.8992	0.9050	0.9011	0.8950	0.8883	0.8824
$p=2,q=3$	0.7780	0.8389	0.8576	0.8625	0.8619	0.8588	0.8551
$p=3,q=3$	0.7359	0.7995	0.8203	0.8269	0.8275	0.8256	0.8229

表 3: pq-gram 距離とトークンの N-gram における Jaccard 距離の相関.

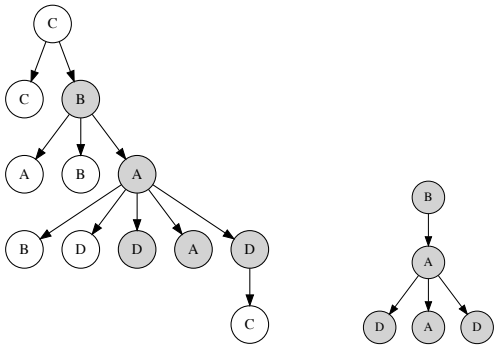

図 4: ラベル付き根付き順序木 (左図) と $p=2,q=3$ における pq-gram(右図) の例．右図において子供を 3 つ持つノードを中心と呼ぶ.

距離は多項式時間で計算することはできるが，その計算量は大規模なデータセットに適用できるほど小さくはない．クローン検出に木の編集距離を用いる手法も提案されている [11] が，それらでは距離の計算に近似的な方法が用いられている．

木の編集距離を求める際に最もよく用いられる単位コストモデルは，木のノード間の「重要性」を考慮しないため，多くのアプリケーションでは不適である．そこで，Augsten ら [1] は，その「重要性」をノードが持つ子供の数に比例すると考え，fanout 重み編集距離 (fanout weighted tree edit distance) を定義した．AST においても，子供の多いノードを編集することは，プログラムの構造を大きく変える可能性が高いため，単位コストよりも fanout 重み編集距離を用いることにある程度の妥当性があると考えられる．fanout 重み編集距離を計算することは，通常の編集距離と同様に高コストであるが，Augsten らは近似計算方法として pq-gram 距離を提案している．ラベル付き根付き順序木の *pq-gram* とは図 4 のような部分木のことである．図 4 の右図における子供を 3 つ持つノードをこの pq-gram の中心と呼び，そこから根方向に p 個のノード，その子として連続した q 個のノードからなる部分木を pq-gram と呼ぶ．木の全てのノードが少なくとも一回は中心となるようにダミーのノードを追加することで木を拡張し，そこから得られる全ての pq-gram の多重集合同士の Jaccard 距離を pq-gram 距離と呼ぶ (詳細は [1]).

ここでは fanout 重み編集距離が AST 間の距離を測る指標として妥当であると仮定し，それらを近似する距離である pq-gram 距離とトークンの N-gram 距離の相関がどれだけあるかを観測する．p と q の具体的な値については Augsten らが道路のマッチングへの応用で用いた値 $(p,q) \in \{(1,2),(2,3),(3,3)\}$ を用いた．また，実験には BigCloneBench のクローン対と偽クローン対それぞれから無作為に 500 ずつ抽出して得られる計 1 千対のコード片を用いた．その実験結果を表 3 に示す．

表 3 より，$N=1$ に比べて，N がある程度大きいときに pq-gram 距離との相関が最も高くなることがわかる．ここで，一般的には AST 内でひとつの pq-gram 内にトークンの N-gram が出現するわけではないため，必ずしも N と $p+q$ の近いことが相関の高さに影響を与えているわけではないことに注意する．この実験より，トークンの N-gram を用いた手法はある程度の構文の構造情報を考慮しているとも考えることができる．また，pq-gram で必要となる AST の構築は，トークンの N-gram を用いる提案手法では不要なため，pq-gram を用いてクローン検出を行う方法と比べても本手法は高速であることが期待できる．

6. おわりに

本研究では，プログラムのトークンの N-gram を用いたコードクローン検出手法を提案した．提案した手法は大規模なソフトウェア群を対象とした Sajnani ら [21] の手法の拡張として位置付けられる．Sajnani らが用いた高速化手法は本手法にもそのまま適用できるため，同じく大規模なソフトウェア群に適用可能であることが期待できる．

計算機実験の結果，構文の類似度が低いコードクローンに対しては，N をある程度大きくすることでクローン検出精度の改善が見られた．これは，自然言語処理で用いられる N-gram のテクニックが，プログラムコードにおいても効果的となり得ることを示している．また，識別子名やリテラルの書き換えを吸収するための正則化と本手法を組み合わせた手法についても実験を行ったが，正則化を行わなかった場合に比べて検出精度は下がる傾向にあった．識別子名やリテラルなどを全て書き換える正則化では，本来必要となる意味的な情報を消失させすぎてしまう可能性がある．N-gram と組み合わせる上においても，より適切な正則化については今後も考察していく必要がある．

さらに N-gram の効果を確認するために，木構造間の距離基準として知られる pq-gram を用いて，コード片の AST 間の pq-gram 距離とトークンの N-gram における Jaccard 距離との比較実験も行った．その結果，$N=3,4$ 程度で pq-gram 距離との相関が高くなることが実験的に確認で

きた．*pq*-gram 距離はノードの重要性を考慮した木の編集距離をある程度近似しているため N を大きくすることで AST の構造的な情報を捉えれられるという仮説の証拠となる可能性がある．

今後の課題としては，実際のパフォーマンスが $N = 1$ の場合と比べてどれだけ悪くなるかを計測することが挙げられる．SourcererCC の高速化手法を $N > 1$ においても適用した場合，多重集合の相異なる要素数が増加し，Jaccard 係数の計算時間の増加や，高い閾値を用いたインデックスによる比較が難しくなることも予想される．そのため，これらの高速化手法を盛り込んだ上でパフォーマンスの計測を行い，然るべき高速化を施す必要がある．また，SourcererCC ではコード片を多重集合として考え，複数の集合間における類似度計算のヒューリスティックスを用いて高速化を行ったが，これは，コード片を距離空間に埋め込んだとも考えることができるため，クラスタリングや LSH の利用 [27] によってさらなる高速化を実現することも期待できる．

参考文献

[1] B. Augsten, M. Böhlen, J. Gamper: The *pq*-gram distance between ordered labeled trees. ACM Transactions on Database Systems 35(1), 4:1–4:36 (2010)
[2] B. S. Baker: On finding duplication and near-duplication in large software systems. In Proc. of WCRE 1995, pp. 86–95 (1995)
[3] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, L. Bier: Clone Detextion Using Abstract Syntax Trees. In Proc. of ICSM 1998, pp. 368–378 (1998)
[4] A. Barrón-Cedeño, P. Rosso: On Automatic Plagiarism Detection Based on *n*-Grams Comparison. In Proc. of ECIR 2009, pp. 696–700 (2009)
[5] W. B. Cavnar, J. M. Trenkle: *N*-gram based text categorization. In Proc. of SDAIR 1994, pp. 161–175 (1994)
[6] S. Chaudhuri, V. Ganti, R. Haushik: A primitive operator for similarity joins in data cleaning. In Proc. of ICDE 2006, pp. 5–16 (2006)
[7] K. Chen, P. Liu, Y. Zhang: Achieving accuracy and scalability simultaneously in detecting application clones on android markets. In Proc. of ICSE 2014, pp. 175–186 (2014)
[8] S. Ducasse, O. Nierstrasz, M. Rieger: On the effectiveness of clone detection by string matching. Journal of Software: Evolution and Process 18(1), 37–58 (2006)
[9] Ambient Software Evolution Group: IJaDataset 2.0. `http://secold.org/projects/seclone` (2018 年 5 月 23 日)
[10] A. Hindle, E.T. Barr, M. Gabel, Z. Su, P. Devanbu: On the naturalness of software. Communucations of the ACM vol. 56(5), 122–131 (2016)
[11] L. Jiang, G. Misherghi, Z. Su, S. Glondu: DECKARD: Scalable and Accurate Tree-Based Detection of Code Clones. In Proc. of ICSE 2007, pp. 96–105 (2007)
[12] T. Kamiya, S. Kusumoto, K. Inoue: CCFinder: a multilinguistic token-based code clone detection system for large scale source code. IEEE Transactions on Software Engineering 28(7), 654–670 (2002)
[13] I. Keivanloo, J. Rilling, P. Charland: Internet-scale real-time code clone search via multi-level indexing. In Proc. of WCRE 2011, pp. 23–27 (2011)
[14] J. Y. Kim, J. Shawe-Taylor: An approximate string-matching algorithm. Theoretical Computer Science 92(1), 107–117 (1992)
[15] R. Koschke: Large-scale inter-system clone detection using suffix trees. In Proc. of CSMR 2012, pp. 309–318 (2012)
[16] J. Krinke: Identifying Similar Code with Program Dependence Graphs. In Proc. of WCRE 2001, pp. 301–309 (2001)
[17] Z. Li, S. Lu, S. Myagmar, Y. Zhou: CP-Miner: Finding Copy-Paste and Related Bugs in Large-Scale Software Code. IEEE Transactions on Software Engineering 32(3), 176–192 (2006)
[18] J. Mayrand, C. Leblanc, E. Merlo: Experiment on the Automatic Detection of Function Clones in a Software System Using Metrics. In Proc. of ICSM 1996, pp. 244–253 (1996)
[19] C. K. Roy, J. R. Cordy: A survey on software clone detection research. Queen's Technical Report 541 (2007)
[20] C. K. Roy, J. R. Cordy: NICAD: Accurate Detection of Near-Miss Intentional Clones Using Flexible Pretty-Printing and Code Normalization. In Proc. of ICPC 2008, pp. 172–181 (2008)
[21] H. Sajinani, V. Saini, J. Svajlenko, C. K. Roy, C. V. Lopes: SourcererCC: Scaling Code Clone Detection to Big-Code. In Proc. of ICSE 2016, pp. 1157–1168 (2016)
[22] A. Sheneamer, J. Kalita: A Survey of Software Clone Detection Techniques. International Journal of Computer Applications 137(10), 1–21 (2016)
[23] J. Svajlenko, J. F. Islam, I. Keivanloo, C. K. Roy, M. M. Mia: Towards a Big Data Curated Benchmark of Inter-Project Code Clones. In Proc. of ICSME 2014, pp. 476–480 (2014)
[24] K.-C. Tai: The tree-to-tree correction problem. Journal of the ACM 26, 422–433 (1976)
[25] E. Ukkonen: Approximate string-matching with *q*-grams and maximal matches. Theoretical Computer Science 92(1), 191–211 (1992)
[26] C. Xiao, W. Wang, X. Lin, J. X. Yu: Efficient similarity joins for near duplicate detection. In Proc. of WWW 2008, pp. 131–140 (2008)
[27] 徳井翔悟，吉田則裕，崔恩瀞，井上克郎: 局所性鋭敏型ハッシュを用いたコードクローン検出のためのパラメータ決定手法. 電子情報通信学会技術研究報告 117(477), 57-62 (2018)