論　文

グラフマイニングアルゴリズムを用いたギャップを含むコードクローン情報の生成

肥後　芳樹[†]　　宮崎　宏海[†]　　楠本　真二[†]　　井上　克郎[†]

Generating Gapped Code Clone Information Using Graph Mining Algorithm

Yoshiki HIGO[†], Hiromi MIYAZAKI[†], Shinji KUSUMOTO[†], and Katsuro INOUE[†]

あらまし　これまでに様々なコードクローン検出手法が提案されているが，ギャップ（不一致部分）を含むコードクローンを検出できる手法は少ない．本論文では，ギャップを含むコードクローンを検出できないコードクローン検出手法の出力結果に対して後処理を行うことで，ギャップを含むコードクローン情報を生成する手法を提案する．提案手法は，グラフマイニングアルゴリズムの一つである AGM アルゴリズムを用いており，効率的にギャップを含むコードクローン情報を生成することができる．提案手法を検出ツール CCFinder のポストプロセッサとして実装し，複数のオープンソースソフトウェアに対して適用したところ，多数の興味深いコードクローン情報を得ることができた．しかし，提示する必要がないと思われるコードクローンも生成してしまうことがあった．本論文では，この実験の結果について述べ，また，上記の問題に対する解決策についても考察する．

キーワード　コードクローン，グラフマイニングアルゴリズム，ソフトウェア保守

1. まえがき

近年，ソフトウェア開発及び保守を困難にしている要因の一つとしてコードクローンが注目されている．コードクローンとはソースコード中に存在する，ほかのコード片と同一または類似したコード片を指す言葉である [1]．コードクローンは，コピーアンドペーストや定型処理，パフォーマンス改善などの様々な理由によりソースコード中に作り込まれる [2]〜[4]．一般的に，コードクローンの存在により，ソフトウェア開発及び保守が困難になるといわれている．例えば，あるコード片にバグが含まれていた場合，そのコードクローンすべてに対して同様の修正の是非を検討する必要がある．このような作業は，特に大規模ソフトウェアでは非常に大きな手間を必要とする．したがって自動的にコードクローンを検出し，その情報を用いることは，ソフトウェア開発及び保守を行うにあたり，非常に有効であるといえる．

これまでに，様々なコードクローン検出手法が提案されているが，コードクローンの厳密で普遍的な定義は存在せず，各手法は独自に異なったコードクローンの定義をもつ [5], [6]．そのため，同一ソフトウェアから検出処理を行った場合でも，用いる検出ツールが異なれば検出結果も異なる．

Bellon は，コードクローン間の違いの度合に基づき，それらを以下の三つに分類している [7]．

Type 1：空白やタブの有無，括弧の位置などのコーディングスタイルを除いて，完全に一致するコードクローンを指す．

Type 2：変数名や関数名などのユーザ定義名，定数，または変数の型などの一部の予約語のみが異なるコードクローンを指す．

Type 3：コピーアンドペースト後に文の追加や削除，変更が行われたなどの結果により，ギャップ（不一致部分）を含むコードクローンを指す．

図 1 は，コピーアンドペーストによる再利用の流れを表している．単にコピーアンドペーストを行っただけの再利用であれば，コピー元とコピー先は完全一致クローンになる．コピーアンドペースト後に変数名などの付替えを行えば，名前変更クローンとなる．更に，コピーしたコード片を文単位で変更した場合は，ギャップを含むコードクローンとなる．

本論文では，Type 1 及び Type 2 のコードクローン

[†] 大阪大学大学院情報科学研究科，吹田市
Graduate School of Information Science and Technology, Osaka University, Suita-shi, 565–0871 Japan

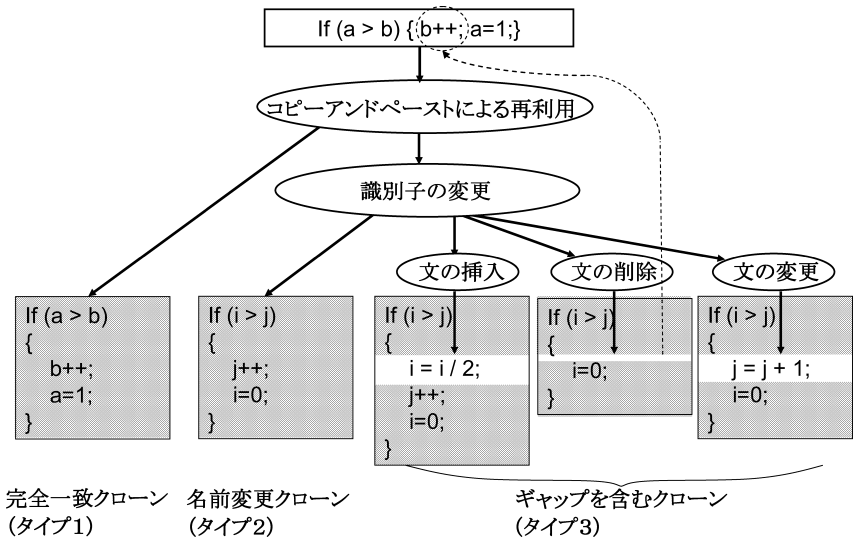


図 1　コピーアンドペーストによる再利用の流れ
Fig. 1　Reuse by copy and paste.

の位置情報から，Type 3 のコードクローンの位置情報を生成する手法を提案する．提案手法は，グラフマイニングアルゴリズムの一つである AGM アルゴリズムを用いている．この手法は特定のコードクローン検出手法に依存しておらず，任意の既存手法に対して適用可能である．提案手法をコードクローン検出の後処理として用いることにより，分析者はより有益なコードクローンの位置情報を得ることができる．

2. 研究の動機

これまでに提案されているコードクローン検出手法のうち，Ueda ら手法 [8] や Li らの手法 [9] など，一部の手法は Type 3 のコードクローンを検出できるが，その他の多くの手法は Type 3 のコードクローンを検出できない．しかし，コピーアンドペースト後に修正漏れが起こるという報告があり [10]，そのようなコードクローンは Type 3 になり得るため，タイプ 3 のコードクローンを検出することはソフトウェア保守の観点から重要である．検出するコードクローンの大きさのしきい値を下げることにより，一つのギャップを含むコードクローンを複数の完全一致クローンまたは名前変更クローンとしては検出することは可能であるが，分析者自らがそれら複数のコードクローンを一つのギャップを含むコードクローンであると認識しなければならず，効率的に分析作業ができるとはいえない．

Ueda の手法は，CCFinder が検出したコードクローンのペアのうち，ソースコード上で近くに存在するものを散布図上でつなぎ合わせることでギャップを含むコードクローンを分析する手法を提案している [8]．散布図上でギャップを含むコードクローンを認識することができるため，利用者が興味のある部分に存在するコードクローンを容易に分析することができる．しかし，ギャップを含むコードクローンをセット（集合）として検出することができない．コードクローンのセットに対してはこれまで有用なメトリックスが提案されており，それらに基づいて分析することで着目すべきコードクローンを容易に発見できるとの研究報告がある [11], [12]．よって，セットとしてコードクローンを認識することは重要である．

Li らは，シーケンシャルパターンマイニング(注1)の一つである CloSpan アルゴリズム [13] を利用して，ギャップを含むコードクローンを検出可能なツール CP-Miner を開発している [9]．CP-Miner はまず，対象プログラムに対して文単位でハッシュ化を行う．これにより，対象プログラムは，数値のシーケンスに変換される．このシーケンスに対して CloSpan アルゴリズムを適用することにより頻出するサブシーケンスを検出し，それを元のプログラムと対応させることにより，コードクローンを検出している．CloSpan

(注1)：シーケンシャルパターンマイニングとは，アイテムの発生順序を保った上で，頻出シーケンスを抽出する手法である．

は，サブシーケンスが連続していなくても検出可能なアルゴリズムである．例えば，シーケンス *agbch* にはサブシーケンス *abc* が一つ存在していると認識することができる．つまり，CloSpan アルゴリズムを利用することでギャップを含むコードクローンの検出が可能になっている．CloSpan のアルゴリズム自体は大きなギャップを含んだサブシーケンスも検出可能であるが，CP-Miner では高速に検出処理を完了するために，ギャップの大きさが 3 以上の場合はサブシーケンスの検出を行わない（つまり三つの文以上の大きさのギャップがある場合は，ギャップを含むコードクローンとして検出されない）．

以上の既存研究を踏まえ，下記のような特徴をもつギャップを含むコードクローンの検出手法が必要であると筆者は考えた．

- ギャップを含むコードクローンをペアとしてだけでなく，セットとして検出可能な手法．
- 短いギャップだけでなく，大きなギャップに対しても，それらを含むコードクローンをある程度高速に検出可能な手法．

この条件を満たす手法として，本論文では AGM アルゴリズムを用いてギャップを含むコードクローンを検出する手法を提案する．提案手法ではまず，既存の検出ツールを用いて，完全一致クローンと名前変更クローンを検出する．そして，それらを頂点とするグラフをソースファイルごとに構築し，そのグラフ中に存在する頻出パターンを抽出する．ギャップを含むコードクローンは，抽出された頻出パターンの各インスタンスであり，ペアではなく，セットとして検出される．また，グラフでは，大きなギャップも小さなギャップも一つの辺として同様に扱われているため，ギャップの大きさに依存せずにコードクローンが検出される．しかし，グラフの集合から頻出パターンを抽出することは NP 完全として知られる部分グラフ同型同型であるため，膨大な計算コストを必要とする．この問題を改善するため，提案手法では，抽出するパターンに制限を加えることにより（パターンの制限は，AGM アルゴリズムの文献 [14] にて提案されている方法で行う），少ない計算コストで検出処理を行う．

3. AGM アルゴリズム

AGM (Apriori-based Graph Mining) アルゴリズムは，複数のラベル付きグラフに含まれる多頻度グラフパターンを効率的に抽出するアルゴリズムである [14]．AGM アルゴリズムの概要を図 2 に示す．ここで G はグラフの集合，F_k は大きさ k の多頻度グラフの隣接行列の集合，C_k は大きさ k の多頻度グラフの候補隣接行列の集合を指す．大きさ k のグラフとは，k 個の頂点から構成されているグラフを意味する．また，$minsup$ は，多頻度とするために最低限必要な出現回数を表す．

```
1)  Main(G,minsup){
2)    C_1←{ 大きさ 1 の多頻度グラフの候補の隣接行列 };
3)    k←1;
4)    while(C_k≠∅){
5)      Count(G,C_k);
6)      F_k ← {c_k ∈ C_k | sup(G(c_k)) ≧ minsup};
7)      C_{k+1} ← Generate − Candidate(F_k);
8)      k ← k + 1;
9)    }
10)   return ⋃_k {f_k ∈ F_k | f_k is canonical};
11) }
```

図 2　AGM アルゴリズムの概要
Fig. 2　Overview of AGM algorithm.

AGM アルゴリズムは，大きさが 1 の多頻度グラフパターンから，順にサイズの大きなグラフパターンを抽出していく．以下に AGM アルゴリズム（図 2）の大まかな流れを示す．

2 行目：1×1 の隣接行列が頂点ラベルの数だけ生成され，C_1 に代入される．

5〜6 行目：大きさが k である多頻度グラフパターンの候補の支持度を求め(注2)，支持度が $minsup$ 以上であれば(注3)，F_k に加える．

7 行目：大きさ k の多頻度グラフパターンを組み合わせて，大きさ $k+1$ の多頻度グラフの候補を生成する．この操作は C_k が空集合になるまで続けられる．

10 行目：すべての多頻度グラフパターンが出力される．

大きさが k の多頻度グラフパターンから，大きさが $k+1$ の多頻度グラフパターンの候補を生成する際の条件を変えることにより，連結グラフパターン，順序木パターン，グラフのパスなど，様々なグラフパターンを取り出すことが可能である．この手法を B-AGM (Biased Apriori-based Graph Mining) と呼ぶ．詳細については文献 [14] を参照されたい．

提案手法では，グラフのパスを抽出する条件を用いて，グラフの頂点を Type 1 や Type 2 のコードク

(注2)：$Count(G, C_k)$ は，グラフ集合 G 内でのグラフパターンの集合 C_k に含まれる各パターンの出現数（支持度）を計測する関数である．
(注3)：$sup(G(c_k))$ は，パターン c_k の出現数（支持度）を返す関数である．

ローン，グラフの辺をコードクローン間のギャップとして，多頻度グラフパターンを抽出する．

4. 提案手法

提案手法の入力と出力は以下のとおりである．

入力：（既存のツールによって検出された）Type 1 と Type 2 のコードクローンの位置情報

出力：生成したギャップを含むコードクローンの位置情報

なお，本論文では，コードクローンの位置情報は，(1) そのコードクローンを含むファイル，(2) コードクローンの開始位置，(3) コードクローンの終了位置，の三つの情報からなる．

提案手法は，以下の手順でギャップを含むコードクローンの位置情報を生成する．

手順 1：各ソースファイルのグラフを構築

手順 2：多頻度サブグラフ（ギャップを含むコードクローン）の検出

手順 3：生成したコードクローンの位置情報の出力

手順 1 では，入力として与えられた Type 1 及び Type 2 のコードクローンの位置情報をもとに，AGM アルゴリズムの入力となるグラフ集合を構築する．手順 2 では，AGM アルゴリズムを用いて，手順 1 で構築したグラフ集合に現れる多頻度サブグラフを検出する．最後に手順 3 では，手順 2 で検出した多頻度サブグラフをギャップを含むコードクローンの情報として出力する．以下，各手順の詳細を述べる．

手順 1：各ソースファイルのグラフを構築

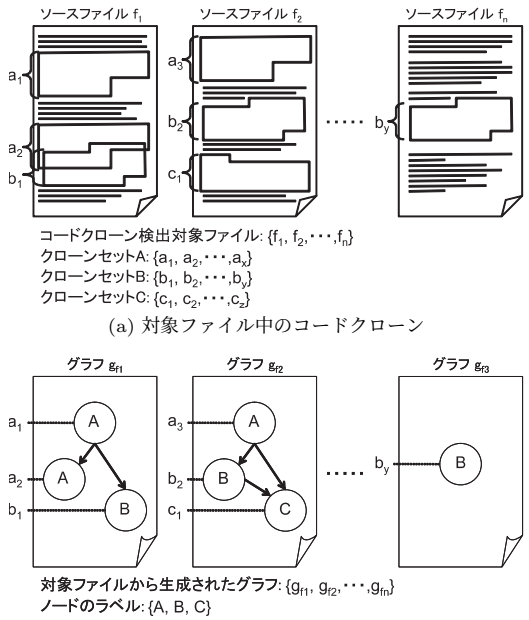
入力として与えられた Type 1 及び Type 2 のコードクローンの位置情報をもとに(注4)，それらを頂点とするグラフをソースファイルごとに作成する．同一クローンセット(注5) に含まれるコードクローンから生成された頂点は同一ラベルをもつ．ある二つのコードクローンがあり，それらがオーバラップしていない場合は，対応する頂点間に辺を引く．ファイル内における位置が上位から下位のコードクローンに向けて辺が引かれる．この操作をすべてのコードクローンの組合せに対して実行し，ソースファイルごとにグラフを構築する．

一部がオーバラップしている場合や，あるコードクローンが他のコードクローンを完全に包含している場合には，これら二つの頂点の間には辺を引かない．このような部分は，コピーアンドペースト後の修正により発生したギャップを含むコードクローンであるとは考えにくいからである．また，二つのコードクローンがあまりに遠く離れている場合も，それらの頂点間には辺を引かない．この場合も，コピーアンドペースト後の修正により発生したコードクローンであるとは考えにくいからである．

図 3 は，グラフ構築の例を表している．この例では，n 個のコードクローン検出対象ファイル $\{f_1,f_2,\cdots,f_n\}$ から，三つのクローンセット A,B,C が検出されている．クローンセット A には x 個のコード片 $\{a_1,a_2,\cdots,a_x\}$，クローンセット B には y 個のコード片 $\{b_1,b_2,\cdots,b_y\}$，クローンセット C には z 個のコード片 $\{c_1,c_2,\cdots,c_z\}$ が含まれている．図 3 (b) は，生成されたグラフを表している．ファイル f_1 では，三つのコード片 a_1，a_2，b_1 が存在しているが，a_2 と b_1 はオーバラップしているため，それらの頂点間には辺が引かれていない．

手順 2：多頻度サブグラフの検出

手順 1 で構築したグラフ集合に対して AGM アル



(a) 対象ファイル中のコードクローン

(b) 生成されたグラフ

図 3 グラフ構造の生成

Fig. 3 Generating graphs from source code.

(注4)：手順 1 では，既存のツールによって検出されたすべてのコードクローンが用いられる．Type 1 のみを検出可能なツールを用いた場合は Type 1 のコードクローンの位置情報のみが用いられ，Type 1 と Type 2 の両方を検出可能なツールを用いた場合は，Type 1 と Type 2 のコードクローンの位置情報が用いられる．

(注5)：クローンセットとは，互いに類似したコード片をすべて含んだ集合である．

ゴリズムを適用し，そのグラフ集合に現れる多頻度グラフパターンを抽出する．具体的には，グラフ集合に2回以上（AGMアルゴリズムにおける最小支持度以上）出現するパスをギャップを含むコードクローンのパターンとする．多頻度グラフパターンを求めた後，各パターンのインスタンスがグラフのどの部分に存在するのかを求める．図3の場合では，ファイル f_1 のパス a_1b_1 とファイル f_2 のパス a_3b_2 がギャップを含むコードクローンとして検出される．

手順3：生成したコードクローンの位置情報の出力

手順2で検出したギャップを含むコードクローンの位置情報を，入力として与えられたコードクローンの位置情報と同様のフォーマットで出力する．

5. 実　　装

提案手法をコードクローン検出ツールCCFinder [3]のポストプロセッサとして実装した．CCFinderは，検出したコードクローンの位置情報をテキスト形式で出力するツールであり，分析作業は付属の可視化ツールGemini [15]を用いて行う．CCFinderは，ソースコードの字句解析を行い，分割した字句単位で比較を行うことによりコードクローンを検出する．CCFinderはType 1とType 2のコードクローンを検出可能である．CCFinderを用いた従来の（つまりType 1とType 2のコードクローンの）検出及び分析手順は以下のとおりである．

（1） CCFinderを用いて（Type 1とType 2の）コードクローンを検出

（2） Geminiを用いて検出されたコードクローンの分析

一方，実装したポストプロセッサを用いた（Type 3のコードクローンの）検出及び分析手順は以下のとおりである．

（1） CCFinderを用いて（Type 1とType 2の）コードクローンを検出

（2） ポストプロセッサを用いてType 3のコードクローン情報を生成

（3） Geminiを用いて生成されたコードクローンの分析

このように，実装したポストプロセッサを用いることによって，Type 1とType 2に加えてType 3，つまりギャップを含むコードクローンの分析もGeminiを用いて行えるようになる．

6. 適 用 実 験

6.1 概　　要

提案手法を用いることにより，どのようなコードクローンが検出されるかを調査するために適用実験を行った．この実験には，前章で述べたポストプロセッサを用いた．対象ソフトウェアは，C言語で記述されたCanna 3.6とhttpd 2.2.4，及びJava言語で記述されたAnt 1.6.0とJHotDraw 7.0.9の四つである．この実験では，CCFinderは字句以上のコードクローンを検出するように設定し，コードクローン間のギャップが30字句以内の場合に，それらに対応する頂点間に辺を引くことにした．表1は，対象ソフトウェアの規模を表している．なお，ギャップを含むコードクローン情報の生成に要した時間は，CCFinderの実行も含めて数分から10数分であった．

提案手法を定量的に評価するために，以下の基準を設け，各基準に該当するギャップを含むクローンセットの数を調査した．

(1) 複数関数：一つのコードクローンが複数の関数やメソッドにまたがっているクローンセットを指す．一つの関数内に閉じたコードクローンは，ギャップ部分（処理内容が異なる部分）が関数内にあり，そのような処理の違いを含むコードクローンを検出することは有益である．一方，複数の関数にまたがって存在しているコードクローンは，関数の外側にギャップが存在するため，そのようなコードクローンを検出する価値はないと筆者は考える．

(2) オーバラップ：同一クローンセットに含まれるコードクローンがオーバラップして存在していることを指す．オーバラップしているので，コピーアンドペーストにより生成されたとは考えにくい．そのため，このようなコードクローンを検出する価値はないと著者は考える．

(3) 調査の必要がない：コードクローンの中には，ソフトウェア開発・保守を行う視点でそれらを扱う場合

表1　対象ソフトウェア
Table 1　Target software.

ソフトウェア			規模	
名前	バージョン	記述言語	ファイル数	総行数
Canna	3.6	C言語	96	90,326
httpd	2.2.4	C言語	760	409,489
Ant	1.6.0	Java言語	627	180,844
JHotDraw	7.0.9	Java言語	471	90,214

表 2　検出されたギャップを含むクローンセットの内訳
Table 2　Classification of detected gapped code clones.

ソフトウェア	ギャップを含むクローンセットの数				
	合計	(1) 複数関数	(2) オーバラップ	(3) 調査の必要がない	(4) 有益
Canna	208	130 (62.5%)	21 (10.1%)	20 (9.6%)	75 (36.1%)
httpd	275	161 (58.5%)	29 (10.5%)	21 (7.6%)	98 (35.6%)
Ant	113	73 (64.6%)	23 (20.4%)	13 (11.5%)	30 (26.5%)
JHotDraw	257	155 (60.3%)	22 (8.6%)	22 (8.6%)	87 (33.9%)

```
    int i;

    d->nbytes = 0;
    yc->kouhoCount = 0;

    if (yc->ys < yc->kEndp || yc->ye != yc->kEndp) {
        i = yc->curbun; /* とっとく */
        if (chikuji_subst_yomi(d) == -1) {
            makeGLineMessageFromString(d, jrKanjiError);
            return TanMuhenkan(d);
        }
        if (RkwGoTo(yc->context, i) == -1) {
            (void)makeRkError(d, "文字列1", "文字列2");
            /* 文節の移動に失敗しました */
            return TanMuhenkan(d);
        }
        yc->curbun = i;
    }
    if ((yc->nbunsetsu = RkwEnlarge(yc->context)) <= 0) {
        (void)makeRkError(d, "文字列1", "文字列2");
        /* 文節の拡大に失敗しました */
        return TanMuhenkan(d);
    }
    if (chikuji_restore_yomi(d) == NG) {
        return TanMuhenkan(d);
    }
    yc->status |= CHIKUJI_OVERWRAP;
    makeKanjiStatusReturn(d, yc);
    return d->nbytes;
```

(a) コード片 1：変換対象領域の拡大処理

```
    int i;

    d->nbytes = 0;
    yc->kouhoCount = 0;
    if (yc->ys < yc->kEndp || yc->ye != yc->kEndp) {
        i = yc->curbun;
        if (chikuji_subst_yomi(d) == -1) {
            makeGLineMessageFromString(d, jrKanjiError);
            return TanMuhenkan(d);
        }
        if (RkwGoTo(yc->context, i) == -1) {
            (void)makeRkError(d, "文字列1", "文字列2");
            /* 文節の縮小に失敗しました */
            return TanMuhenkan(d);
        }
        yc->curbun = i;
    }

G   if (RkwGetStat(yc->context, &stat) < 0 || stat.ylen == 1) {
G       /* これ以上短くできるかどうか確認。要る？ */
G       return NothingForGLine(d);
G   }
    yc->nbunsetsu = RkwShorten(yc->context);
    if (yc->nbunsetsu <= 0) { /* 0 ってことあんのかなあ？ */
        (void)makeRkError(d, "文字列1", "文字列2");
        /* 文節の縮小に失敗しました */
        return TanMuhenkan(d);
    }
    if (chikuji_restore_yomi(d) == NG) {
        return TanMuhenkan(d);
    }
    yc->status |= CHIKUJI_OVERWRAP;
    makeKanjiStatusReturn(d, yc);
    return d->nbytes;
```

(b) コード片 2：変換対象領域の縮小処理

図 4　Canna から検出されたギャップを含むコードクローン
Fig. 4　A clone pair detected from Canna.

に，対象とする必要がないものが存在する．これらのコードクローンの多くは，連続した変数宣言や import 文，switch 文の連続した case エントリなど，繰返し構造をもつことが分かっている [12]．この実験では，文献 [12] で紹介されている，繰返し構造をもつコードクローンを，調査の必要がないコードクローンとした．

(4) 有益：この実験では，関数内にギャップをもち，同一クローンセットに属するほかのすべてのコードクローンともオーバラップしておらず，文献 [12] で紹介されている繰返し構造をもっていない場合に，そのギャップを含むコードクローンは検出を行う価値があると考え，有益とした．

6.2　調 査 結 果

対象ソフトウェアから検出されたギャップを含むコードクローンのソースコードをすべて閲覧し，各基準に該当するクローンセットの数を計測した．表 2 に検出されたギャップを含むクローンセットの総数及び各基準に該当した個数を示す．なおこの調査は本論文の筆者が行い，作業には約 40 人時を要した．この表から，各ソフトウェアから検出されたソフトウェアのうち，およそ 25〜35%のギャップを含むクローンセットが有益であると判定されたことが分かる．

有益と判定されたギャップを含むコードクローンの例を示す．図 4 は Canna から検出されたコードクローンである(注6)．コード片 1（図 4 (a)）は日本語の変換処理対象領域を拡大する処理を実装しており，コード片 2（図 4 (b)）は，その領域の縮小処理を実装している．縮小処理のコード片には，更に縮小可能かを判定する処理が含まれており，この部分が拡大処理とのギャップになっている（図 4 (b) の行頭が G の部分）．ソースコード中のコメントから，開発者はこの処理の正しさについて確信がないことが分かる．ギャップを含むコードクローンを検出することにより，このよう

(注6)：コード片 1 及びコード片 2 の文字列リテラル “文字列 1” 及び “文字列 2” は，実際にはもっと長い文字列であるが，ソースコードをより分かりやすく掲載するために置換している．

な調査を行う必要がある部分を発見することができた．

すべての対象ソフトウェアからはかなりの数の,「(1) 複数関数」,「(2) オーバラップ」,「(3) 調査の必要がない」，の基準に該当するクローンセットが検出されており,「(4) 有益」に該当するクローンセットの全体に対する割合を下げていることが分かる．なお，一つのコードクローンには，二つ以上のギャップが含まれていることがあり，このような場合は複数の基準に該当することがある．例えば，二つのギャップを含んでいるコードクローンがあり，そのうちの一つが関数の外側，残りの一つが関数の内側にあった場合は，そのコードクローンを含むクローンセットは,「(1) 複数関数」と「(4) 有益」の両方の基準に該当する．

7. 考　　察

本章では,「(1) 複数関数」,「(2) オーバラップ」,「(3) 調査の必要がない」に該当するコードクローンが検出された原因を述べ，更にそれらの検出を抑える対策について考察する．また，提案手法と，**2.** で紹介した CP-Miner [9] との間における検出されるコードクローンの違いについても考察を行う．

7.1 「(1) 複数関数」への対策

実験では，有益と判定されたクローンセットは，全体の約 25〜35%であり，高い割合とはいえない．その大きな原因は,「(1) 複数関数」に該当するクローンセットが非常に多く検出されたことである．いずれの対象ソフトウェアも，処理内容が類似した関数が多く定義されており，それらは連続した位置にあった．それらの開始部分と終了部分が共通であり（完全一致クローン若しくは名前変更クローンとなっており），中央のみが異なっているために，ある関数の終了部分から次の関数の開始部分までが一つのギャップを含むコードクローンとして検出されていた．

複数の関数にまたがるコードクローンの検出を防ぐには，各関数の開始位置及び終了位置の情報を取得し，ギャップを含むコードクローンがそれをまたぐかどうかを判定しなければならない．しかし，コードクローン検出結果には対象ソースコードに含まれる関数の開始位置及び終了位置の情報は含まれていないために，ポストプロセッサ内で再度ソースコード解析を行わなければならない．

ポストプロセッサでソースコード解析を行わない利点は，用いるコードクローン検出ツールが対応しているプログラミング言語であれば，どの言語であってもポストプロセッサを適用できることである．つまり，ポストプロセッサ内でソースコード解析を行うことによって,「(1) 複数関数」に該当するクローンセットの検出を防ぐことは可能であるが，検出対象ソフトウェアのソースコード解析器を構築する必要があり，複数のプログラミング言語を対象とするような場合には，ポストプロセッサの実装に高いコストを必要とする．ctags [16] などの既存のツールを用いて比較的容易に実装できるとも思われる．

7.2 「(2) オーバラップ」への対策

すべての対象ソフトウェアにおいて，検出されたクローンセットのうち約 10%はオーバラップしたコードクローンをもっていた．すべての対象ソフトウェアには，独自の頻出する処理が存在しており，この部分からオーバラップしたコードクローンが検出された．頻出する処理は，連続した if-else 文の各条件分岐先に存在しており，比較的互いに近い位置に存在していたため，ギャップを含むコードクローンとして認識されてしまっていた．

提案手法では，一つのファイル内に同じクローンセットに含まれるコードクローンが三つ存在した場合，1 番目と 2 番目をつないだコードクローンと 2 番目と 3 番目をつないだコードクローンが一つのギャップを含むクローンセットを構成し，それらはともに 2 番目のコード片を含んでいるため，オーバラップしている．このようにして，オーバラップしたコードクローンをもつクローンセットが多数検出された．

オーバラップしたコードクローンの検出を防ぐ手段は二つ考えられる．

対策 1：ギャップを含むコードクローンを生成した後，それらの位置からオーバラップの判定を行う．ギャップを含むコードクローンを構成している完全一致クローン及び名前変更クローンの位置情報は，コードクローン検出ツールの出力結果から得られるため，この処理は容易に行うことが可能である．

対策 2：提案手法の手順 2 において，同一クローンセットに含まれる頂点間には辺を引かないことにする．この処理も容易に行うことが可能である．

対策 1 は，すべてのオーバラップしたコードクローンの検出を抑える．それに対して，対策 2 は，開始コード片と終了コード片が異なるクローンセットに含まれている場合は，ギャップを含むコードクローンとして出力を行う．このようなコードクローンは，巻き付きコードクローン [17] の一種であると考えられ，一

概に検出する価値がないとはいえない．どちらの対策の方が有効であるかを調査するためには，更なる実験を行う必要がある．

7.3 「(3) 調査の必要がない」への対策

オーバラップしたクローンセットと同様に，調査の必要がないクローンセットも全体の約 10%を占めていた．本論文では，調査の必要がないコードクローンとは，「ソフトウェア開発・保守を行う視点でそれらを扱う場合に特に対象とする必要がないコードクローン」と定義しており，具体的には，連続した変数宣言やメソッド呼出し，switch 文の連続した case エントリなどの処理を行っているコードクローンである．筆者らの過去の調査により，調査の必要がないコードクローンは，繰返し構造をもつ傾向であることが分かっている [12]．このようなコードクローンは，どのソフトウェアにも存在するため，ある程度検出されてしまうのは当然であるといえる．

このようなコードクローンを自動的に除去するためには，ソースコードの構造を調査しなければならない．例えば，どの部分が連続した変数の定義であるか，どの部分が switch 文の連続した case エントリであるか，ということが分からなければこのようなコードクローンの検出を抑えることはできない．しかし，コードクローン検出ツールの出力結果には，ソースコードの構造情報は含まれていないため，ポストプロセッサでソースコード解析を行う必要がある．**7.1** でも述べたが，ポストプロセッサ内でソースコード解析を行うことによって，「(1) 調査の必要がない」に該当するクローンセットの検出を防ぐことは可能であるが，検出対象ソフトウェアのソースコード解析器を構築する必要があり，適用にはより高いコストを必要とする．

7.4 CP-Miner との比較

提案手法では，ギャップの前後が Type 1 クローン若しくは Type 2 クローンとして検出されなければ，その部分をギャップを含むコードクローンとして検出できない．つまり，ギャップを含むコードクローンとして検出されるためには，ギャップの前後にある程度以上の重複コードが存在していなければならない．それに対して，CP-Miner はギャップの前後が 1 文でも重複していれば，ギャップを含むコードクローンとして検出可能である．

CP-Miner はギャップの大きさが 1 文若しくは 2 文の場合のみギャップを含むコードクローンとして検出可能である．CP-Miner が採用している CloSpan アルゴリズム自体は大きなギャップを含むサブシーケンスも検出可能であるが，そのような設定でギャップを含むコードクローンの検出を行うと，検出に必要な計算量が膨大になってしまい，実規模ソフトウェアに対しては適用が難しい．それに対して提案手法では，小さなギャップも大きなギャップもグラフ上では同様に扱われるため，大きなギャップを含むコードクローンも検出可能である．また，頻出パターンの検出の際に制限を設けることによって高速に検出を行っているため，実規模ソフトウェアに対しても適用可能である．

最後に検出速度に関して比較する．CP-Miner は Linux カーネルから約 20 分でコードクローン検出を行うことができる [9]．一方，提案手法は，四つのソフトウェアに対して数分から数十分で検出処理を行った．対象ソフトウェアの規模の違いを考慮すると CP-Miner の方が高速にコードクローン検出を行っているといえるが，提案手法も実際に使うには問題のない速度だと筆者は考える．

8. むすび

本論文では，完全一致クローンと名前変更クローンのコードクローン情報からギャップを含むコードクローンの生成手法を提案した．提案手法は，コードクローン検出ツールの出力結果のみを用いるため，コードクローン検出対象ソースファイルを直接解析しない．よって，利用するコードクローン検出ツールが対応しているプログラミング言語であれば，どの言語であっても適用することが可能である．

実際に提案手法を検出ツール CCFinder のポストプロセッサとして実装し，どのようなギャップを含むコードクローン情報が生成されるのかを調査を行った．C 言語あるいは Java 言語で記述された四つのソフトウェアについて調査を行ったところ，生成結果の約 25〜35%が有益なコードクローン情報であった．残りのコードクローンは，何らかの理由により，検出する価値がないコードクローンであると思われた．

また，これらの検出する価値がないと思われるコードクローンの除去方法について考察した．今後は，考察した除去方法の実装と，CCFinder 以外のコードクローン検出ツールを使ってギャップを含むコードクローン検出を行うことを予定している．

謝辞 本研究を行うにあたり御助力頂いた藤野陽平氏（現 JR 東日本）に感謝する．本研究は一部，文部科学省「次世代 IT 基盤構築のための研究開発」（研

究開発領域名：ソフトウェア構築状況の可視化技術の開発普及）の委託に基づいて行われた．また，日本学術振興会科学研究費補助金基盤研究 (A)（課題番号：21240002），及び文部科学省科学研究費補助金基盤研究 (C)（課題番号：20500033）の助成を得た．

文　献

[1] 井上克郎，“エンピリカルソフトウェア工学の研究と実践—コードクローンを例に，” EASE プロジェクトニュースレター，vol.4, pp.1–4, Dec. 2005.
[2] I. Baxter, A. Yahin, L. Moura, M. Anna, and L. Bier, “Clone detection using abstract syntax trees,” Proc. International Conference on Software Maintenance 98, pp.368–377, March 1998.
[3] T. Kamiya, S. Kusumoto, and K. Inoue, “CCFinder: A multilinguistic token-based code clone detection system for large scale source code,” IEEE Trans. Softw. Eng., vol.28, no.7, pp.654–670, July 2002.
[4] S. Uchida, A. Monden, N. Ohsugi, T. Kamiya, K. Ichi Matsumoto, and H. Kudo, “Software analysis by code clones in open source software,” J. Computer Information Systems, vol.XLV, no.3, pp.1–11, April 2005.
[5] “Clone detection literature,” http://www.cis.uab.edu/tairasr/clones/literature/
[6] 肥後芳樹，楠本真二，井上克郎，“コードクローン検出とその関連技術，” 信学論（D），vol.J91-D, no.6, pp.1465–1481, June 2008.
[7] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. Merlo, “Comparison and evaluation of clone detection tools,” IEEE Trans. Softw. Eng., vol.31, no.10, pp.804–818, Oct. 2007.
[8] Y. Ueda, T. Kamiya, S. Kusumoto, and K. Inoue, “On detection of gapped code clones using gap locations,” Proc. 9th Asia-Pacific Software Engineering Conference, pp.327–336, Dec. 2002.
[9] Z. Li, S. Lu, S. Myagmar, and Y. Zhou, “CP-miner: Finding copy-paste and related bugs in large-scale software code,” IEEE Trans. Softw. Eng., vol.32, no.3, pp.176–192, March 2006.
[10] M. Balint, T. Girba, and R. Marinescu, “How developers copy,” Proc. 14th IEEE International Conference on Program Conprehension, pp.56–68, June 2006.
[11] Y. Ueda, T. Kamiya, S. Kusumoto, and K. Inoue, “Gemini: Maintenance support environment based on code clone analysis,” Proc. 8th International Symposium on Software Matrics, pp.67–76, June 2002.
[12] 肥後芳樹，吉田則裕，楠本真二，井上克郎，“産学連携に基づいたコードクローン可視化手法の改良と実装，” 情処学論，vol.48, no.2, pp.811–822, Feb. 2007.
[13] X. Yan, J. Han, and R. Afshar, “CloSpan: Mining closed sequential patterns in large datasets,” Proc. 2003 SIAM International Conference on Data Mining, pp.166–177, May 2003.
[14] 猪口明博，鷲尾　隆，元田　浩，“多頻度グラフマイニング手法の一般化，” 人工知能誌，vol.19, no.5, pp.368–378, May 2004.
[15] 植田泰士，神谷年洋，楠本真二，井上克郎，“開発保守支援を目指したコードクローン分析環境，” 信学論（D-I），vol.86-D-I, no.12, pp.863–871, Dec. 2003.
[16] “ctags,” http://ctags.sourceforge.net/
[17] R. Komondoor and S. Horwitz, “Using slicing to identify duplication in source code,” Proc. 8th International Symposium on Static Analysis, pp.40–56, July 2001.

（平成 21 年 11 月 16 日受付，22 年 3 月 11 日再受付）

肥後　芳樹　（正員）

平 14 阪大・基礎工・情報中退．平 18 同大大学院博士後期課程了，平 19 阪大・情報・コンピュータサイエンス・助教．博士（情報科学）．コードクローン分析・リファクタリングに関する研究に従事．情報処理学会，IEEE 各会員．

宮崎　宏海

平 19 阪大・基礎工・情報卒．平 21 同大大学院博士前期課程了．在学時，コードクローンに関する研究に従事．

楠本　真二　（正員）

昭 63 阪大・基礎工・情報卒．平 3 同大大学院博士課程中退．同年同大・基礎工・情報・助手．平 8 同大講師．平 11 同大助教授．平 14 阪大・情報・コンピュータサイエンス・助教授．平 17 同学科教授．博士（工学）．ソフトウェアの生産性や品質の定量的評価，プロジェクト管理に関する研究に従事．情報処理学会，IEEE，JFPUG 各会員．

井上　克郎　（正員：フェロー）

昭 54 阪大・基礎工・情報卒．昭 59 同大大学院博士課程了．同年同大・基礎工・情報・助手．昭 59〜61 ハワイ大マノア校・情報工学科・助教授．平 1 阪大・基礎工・情報・講師．平 3 同学科・助教授．平 7 同学科・教授．博士（工学）．平 14 阪大・情報・コンピュータサイエンス・教授．ソフトウェア工学の研究に従事．情報処理学会，日本ソフトウェア科学会，IEEE，ACM 各会員．