特別研究報告

題目

グラフマイニングアルゴリズムを用いた
ギャップを含むクローン抽出手法の提案

指導教員

井上 克郎 教授

報告者

藤野陽平

平成 19 年 2 月 20 日

大阪大学 基礎工学部 情報科学科

平成 18 年度 特別研究報告

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クローン抽出手法の提案

藤野陽平

内容梗概

ソースコードの保守作業を困難にする要因の 1 つとして，ソフトウェア中にあるコードクローンの存在が指摘されている．コードクローンとは，ソースコード中に同一，または類似したコード片を持つようなコード片のことであり，”重複コード”とも呼ばれている．コードクローンは，既存コードの”コピーとペースト”による再利用や，意図的な同一処理の繰り返しなどによりソースコード中に作りこまれる．コードクローンの存在が保守作業を困難にする理由は，修正するコード片のコードクローンが存在した場合，それら全てに対して修正の是非を検討する必要があるからである．そのため，これまでに多くのコードクローン検出・分析手法が開発されている．我々の研究グループでもこれまでにコードクローン検出ツール CCFinder を開発してきている．CCFinder は大規模なソフトウェアから実用的な時間でコードクローンを検出することが可能である．

CCFinder はこれまでに国内外の個人・組織に配布され，運用されてきた．その結果，実利用を目指す上での課題が幾つか顕在化した．その課題の 1 つが，不一致部分を含んだコードクローンを発見できないことである．実際に，既存コードのコピーとペーストによる再利用の際など，ペーストされたコード片はそのまま利用されることは少なく，ペースト先の文脈に合わせた部分的な修正や変更が行われる可能性が高い．そのような場合，コピー元との不一致が生じるため，CCFinder を用いたクローン検出では，幾つかの短いクローンに分割され検出されてしまう．そこで本研究では，不一致部分を含んだコードクローンを，グラフマイニングアルゴリズムの 1 つである AGM アルゴリズムを用いて抽出する手法を提案する．またこの手法をツールとして実装し，Java や C 言語で記述されたオープンソースソフトウェア合計 4 個に対して，どのようなコードクローンが抽出されるか調査を行った．その結果，コピーとペーストによる再利用の際に生まれた不一致部分を含んだコードクローンを抽出することが確認された．これにより，より効率的にコードクローン分析を行うことができると期待される．

主な用語

コードクローン
ソフトウェア保守

目 次

1 まえがき

近年，ソフトウェアシステムの大規模化，複雑化に伴い，ソースコードの保守に要するコストが増加してきている．ソースコードの保守を困難にしている 1 つの要因としてコードクローンが指摘されている．

コードクローンとはソースコード中に存在する同一，または類似したコード片のことである．それらの多くは，既存システムに対する変更や拡張時における”コピーとペースト”による安易な再利用の際に発生する．もし，あるコード片にバグが含まれていた場合，そのコード片の全てのコードクローンについて修正を行うかどうかを検討しなければならない．コードクローンを含むソースコードの保守性を高めるため，クローンセット (コードクローンの同値類) を 1 つのサブルーチン等にまとめる方法が考えられる．そのためには，全てのコードクローンを検出することが必要である．

これまでに様々なコードクローン検出法が提案されている．我々の研究グループでもコードクローン検出ツール CCFinder[13] と分析環境 Gemini[19][20] を開発してきている．CCFinder の出力はテキストベースであり，実システムに対するコードクローン分析に直接 CCFinder の出力を用いるのは効率が悪い．このため，検出したコードクローン分析には Gemini を用いる．Gemini には種々のコードクローン視覚化手法が実装されており，ユーザはコードクローン情報を GUI を通して分析することができる．ユーザは CCFinder，Gemini を用いることにより，コードクローンの検出，分析，ソースコードの修正を容易に行うことができる [21].

我々はこれまでに CCFinder を様々なソフトウェア開発組織で運用してきた．その結果，実利用を目指す上での課題が幾つか顕在化した．その課題の 1 つが，不一致部分を含んだコードクローンを発見できないことである．実際に，既存コードのコピーとペーストによる再利用の際など，ペーストされたコード片はそのまま利用されることは少なく，ペースト先の文脈に合わせた部分的な修正や変更が行われる可能性が高い．そのような場合，コピー元との不一致が生じるため，CCFinder を用いたクローン検出では，幾つかの短いクローンに分割され検出されてしまう．一般に，短いコードクローンはソフトウェア中に膨大に存在するため，不一致部分のために分割された短いコードクローンを検出しようとすると，不必要な情報を多く検出してしまうことにより分析者の手間が増え，また検出のための計算コストも高くついてしまう．

そこで，本研究では，不一致部分を含んだコードクローンの抽出手法を提案する．本手法は，CCFinder によって検出された一致箇所の情報を利用して，連結することができる不一致箇所を特定し，その不一致箇所を含んだコードクローン (以下ギャップを含むクローンと呼ぶ) を抽出する．本手法では，グラフマイニングアルゴリズムである AGM アルゴリズム

を用いる．また，本手法をさまざまなオープンソフトウェアに適用し，本手法によって適切にギャップを含むクローンが抽出できているかどうか調査した．

以降 2 節では，コードクローンに対する諸定義，コードクローン検出ツール CCFinder，コードクローン分析環境 Gemini コードクローンの分類について説明する．3 節では，AGM アルゴリズムの説明をする．4 節では，AGM アルゴリズムを用いたギャップを含むクローンの検出手法の提案を行う．5 節では，4 節で提案した手法を用いて行った適用実験について説明する．最後に 6 節で本研究のまとめと今後の課題について述べる．

2　準備

2.1　コードクローンと既存のコードクローン検出法

コードクローンとは，ソースコード中に含まれる同一もしくは類似したコードのことであり，いわゆる”重複したコード”のことである．コードクローンがソフトウェアの中に作りこまれる，もしくは発生する原因として次のようなものがある [8][13].

既存コードのコピーとペーストによる再利用　近年のソフトウェア設計手法を利用すれば，構造化や再利用可能な設計が可能である．しかし，ゼロからコードを書くよりも既存コードをコピーして部分的な変更を加える方が信頼性が高いということもあり，実際には，コピーとペーストによる場当たり的な既存コードの再利用が多く存在する．

コーディングスタイル　規則的に必要なコードはスタイルとして同じように記述される場合がある．例えば，ユーザインターフェース処理を記述するコードなどである．

定型処理　定義上簡単で頻繁に用いられる処理，例えば，給与税の計算や，キューの挿入処理，データ構造アクセス処理などである．

プログラミング言語に適切な機能の欠如　抽象データ型や，ローカル変数を用いられない場合には，同じようなアルゴリズムを持った処理を繰り返し書かなくてはならないことがある．

パフォーマンス改善　リアルタイムシステムなど時間制約のあるシステムにおいて，インライン展開などの機能が提供されていない場合に，特定のコード片を意図的に繰り返し書くことによってパフォーマンスの改善を図ることがある．

コード生成ツールの生成コード　コード生成ツールにおいて，類似した処理を目的としたコードの生成には，識別子名等の違いはあろうとも，あらかじめ決められたコードをベースにして自動的に生成されるため，類似したコードが生成される．

偶然　単純に偶然一致してしまう場合もあるが，大きなコードクローンになる可能性は低い．

もしコードクローンが存在した場合には，一般的にコードの変更等が困難であるといわれ，保守容易性の低下の一因となっている．このようなコードクローンによる問題に対処する方法としては，

- コードクローン情報の文書化を行うこと変更の一貫性を保つ
- コードクローンを自動で検出する

の 2 つがある [21]．しかし，コードクローン情報の文書化には全てのコードクローンに対する情報を常に最新に保つことに非常に手間がかかるため，現実的に困難である．そこで，これまでにさまざまなコードクローン検出手法やツールが提案されている [1][2][3][4][5][6][7][8][10][13][15][16][17][18][24]．それぞれの手法やツールの特徴は次のようになっている (我々の開発した CCFinder は次節 2.2 参照)．

Covet 文献 [17] で定義された種々の特徴メトリクスの幾つかのメトリクス値を比較することによって，コードクローン検出を行う．検出対象言語は Java である．

CloneDR[8] 抽象構文木 (AST) の節点を比較することによってコードクローン (類似部分木) の検出を行う．また，部分的に異なっているコードクローンも検出することが可能であり，自動的に等価なサブルーチンやマクロに置き換えることも可能である．検出対象言語は，C，C++，COBOL，Java，Progress である．

Dup[2][3][4] ユーザ定義名のパラメータ化を行った後，行単位の比較によりコードクローンを検出する，マッチングアルゴリズムには，サフィックス木探索 [11] を用いているため線形時間で解析可能である．

Duploc[10] 前処理として，空白やコメント等を取り除いた後，行単位 (のハッシュ値) での表検索を用いた比較によってコードクローンを検出する．また，コードクローンの散布図等の GUI を備えたツールであり，ソースコードの参照支援を行う．検出対象言語は，C，COBOL，Python，Smalltalk である．

JPlag[18] ソースコードを字句解析し，トークン単位での比較を行う．プログラム盗用の検出を目的として開発され，プログラム間の類似率を検出する．検出対象言語は，C，C++，Java である．

Komondoor らの手法 [15] 関数等にまとめるのに適したコードクローンを抽出を目的として，プログラム依存グラフ (PDG) 上での各節点の比較を行うことでコードクローン (同型 (isomorphic) 部分グラフ) を検出する．文字列比較や抽象構文木等を用いた検出法では発見できなかった非連続コードクローンや，対応行の番号が異なるクローン，互いに絡み合ったクローン等を検出可能である．本研究との違いは，検出対象言語が C のみであること，Komondoor らの手法がグラフ上の各節点の比較を行うことでコードクローンを検出しているのに対して，本研究ではグラフ上で複数回現れたパスをコードクローンとしていることが挙げられる．

Krinke の手法 [16] AST や Traditional PDG に似た Fine-grained PDG というグラフ上での類似 (similar) 部分グラフ (同型部分グラフではない) を検出することで，コードクローンが存在すると思しき場所を検出する．試作ツールの検出対象言語は，C である．

SMC[5][6][7] まず特徴メトリクスによってソースコードをコードクローンと思しきメソッドに絞り込む．次に絞り込まれたメソッドのペアに対し，表検索を用いることでメソッド単位のコードクローンを検出する．特徴メトリクスによって絞り込まれているため，実用上ほぼ線形時間で解析可能である．また検出されたペアのメソッドは，特徴により 18 種類に分類される．さらにそれぞれの分類については共通メソッドへの書き換え指針が示されている．

MOSS[1] 検出アルゴリズムは公開されていない．JPlag 同様，プログラム盗用の検出を目的として開発された．検出対象言語は，Ada，C，C++，Java，Lisp，ML，Pascal，Scheme である．

CP-Miner[24] コピーとペーストによって生成されたコードの検出を目的として開発されたツールである．CloSpan(Closed Sequential Pattern Mining)[25] と呼ばれるマイニングアルゴリズムを用いることで，コピーとペーストによって生成されたコード片を検出する．また，コピーとペーストの後，数行の挿入や削除といった修正が加えられたコードも検出可能である．検出対象言語は，C，C++である．

いずれの手法，ツールにおいても提案者によってコードクローンの定義が異なっており，検出されるコードクローンが異なっている．つまり，コードクローンの定義とは検出アルゴリズムそのものによって定義される．Burd ら [9] も，CloneDR，Covet，JPlag，Moss，そして我々の開発した CCFinder を含めた 5 つのツールを用いて，それぞれ検出されるコードクローンの比較を行っているが，すべての面において他のツールより優れているツールはなく，使う場面に応じて，適切なツールを選ぶことが必要となってくると述べている．

2.2 コードクローン検出ツール CCFinder

2.2.1 概要

あるトークン列中に存在する 2 つの部分トークン列 α，β が等価であるとき，α は β のクローンであると定義する (その逆もクローンであるという)．また，(α，β) を**クローンペア** (図 1 参照) と呼ぶ．α，β それぞれを真に包含するどのようなトークン列も等価でないとき α，β を極大クローンという．また，クローンの同値類を**クローンセット** (図 1 参照) と呼び，ソースコード中でのクローンを特にコードクローンと呼ぶ [13]．

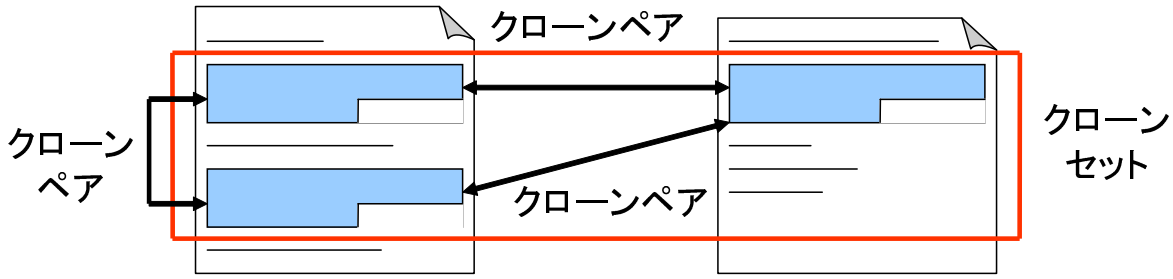

図 1: クローンペアとクローンセット

CCFinder は，単一または複数のファイルのソースコード中から全ての極大クローン検出し，それをクローンペアの位置情報として出力する．CCFinder の持つ主な特徴は次の通りである．

細粒度のコードクローンを検出　字句解析を行うことにより，トークン単位でのコードクローンを検出する．

大規模ソフトウェアを実用的な時間とメモリで解析可能　例えば 3MLOC のソースコードを 15 分 (実行環境 Pentium4 2GHz RAM 1GB) で解析可能である．

様々にプログラミング言語に対応可能　言語依存部分を取り替えることで，様々なプログラミング言語に対応できる．現在は，C，C++，Java，COBOL/COBOLS，Fortran，Emacs Lisp に対応している．またプレーンテキストに対しても，分かち書きされた文章として解析可能となっており，未対応の言語に対しても完全一致判定によるコードクローンは検出可能である．

実用的に意味の持たないコードクローンを取り除く　コードクローンは小さくなればなるほど偶然の一致の可能性が高くなるが，最小一致トークン数を指定することができるため，そのようなコードクローンの検出を防ぐことができる．

ある程度の違いは吸収可能

- ソースコード中に含まれるユーザ定義名，定数をパラメータ化することで，その違いを吸収できる．

- クラススコープや名前空間による複雑な名前の正規化を行うことで，その違いを吸収できる．
- その他，テーブル初期化コード，可視性キーワード (protected,public,private 等)，コンパウンド・ブロックの中括弧表記等の違いも吸収できる．

2.2.2 コードクローン検出処理手順

CCFinder のコードクローン検出処理は，以下の 4 ステップで構成されている．

ステップ 1: 字句解析 ソースコードをプログラミング言語の文法に沿ってトークン列に変換する．その際，空白とコメントは機能に影響しないので無視される．ファイルが複数の場合には，単一ファイルの解析と同じように処理できるよう，単一のトークン列に連結する．

ステップ 2: 変換処理 実用的に意味を持たないコードクローンを取り除くこと，及び，ある程度の違いを吸収することを目的とした変換ルールによりトークン列を変換する．例えば，変数名，関数名などは全て同一のユニークなトークンに置換される．

ステップ 3: 検出処理 トークン列の中から指定された長さ以上一致している部分をクローンペアとして全て検出する．

ステップ 4: 出力整形処理 検出されたクローンペアについて，元のソースコード上での位置情報を出力する．

2.2.3 検出例

実際に，CCFinder がどのようなコードクローンを検出するのか例を示す．最小一致トークン数を 5 トークンに定め，図 2 の Java ソースコードに対しコードクローン検出を行うと，図 2 中の A1(4 行目-6 行目) と A2(16 行目-17 行目)，B1(8 行目-10 行目) と B2(20 行目-22 行目)，そして C1(12 行目) と C2(25 行目) がそれぞれクローンペアとして検出される．それぞれのクローンペアの長さは順に 7,18,6 トークンとなっている．見ての通り，A1 と A2 の間，B1 と B2 の間には次のような幾らかの違いが含まれているがコードクローンとして検出可能となっている．

- 名前空間の違い (e.g. ”org.apache.regexp.RE” と ”RE”).
- 変数名の違い (e.g. ”pat” と ”exp”).

- 改行とインデントの違い

これらの違いは，2.2.1 節で述べた目的のため，CCFinder のトークン変換処理によって吸収されている．

```
    1.  static void foo() throws RESyntaxException
    2.  {
    3.    String a[] = new String [] {"123,400","abc"};
A1  4.    org.apache.regexp.RE pat =
A1  5.            new org.apache.regexp.RE("[0-9,]+");
A1  6.    int sum = 0;
    7.    for(int i=0; i<a.length; i++)
B1  8.    {
B1  9.      if(pat.match(a[i])){
B1 10.        sum += Sample.parseNumber(pat.getParen(0));}
    11.  }
C1  12.  System.out.println("sum ="+sum);
    13.}
    14. static void goo(String [] a) throws RESyntaxException
    15.{
A2  16.    RE exp = new RE("[0-9,]+");
A2  17.    int sum = 0;
    18.    int i = 0;
    19.    while(i<a.length)
B2  20.    {
B2  21.      if(exp.match(a[i]))
B2  22.          sum += parseNumber(exp.getParen(0));
    23.      i++;
    24.    }
C2  25.  System.out.println("sum ="+sum);
    26.}
```

図 2: コードクローン検出例

2.3 コードクローン分析環境 Gemini

Gemini は，CCFinder から得られた解析結果をグラフィカルにユーザに提供する．Gemini の解析はファイルベースの解析とクローンベースの解析に分かれている．ファイルベースの解析では，各ファイルにどの程度クローンが含まれているのかというような，ファイルを単位とした解析を行う．一方，クローンベースの解析では，特徴的なクローンがソフトウェアのどの部分に含まれているのかというような，クローンセットを単位とした解析を行う．

2.3.1 ファイルベースの解析

ファイルベースの解析では，ユーザは各ファイルにどの程度クローンが含まれているのか，また，どのファイルとどのファイルが多くのクローンを共有しているのかといったことを調べることができる．ファイルベースでは主に次のビューを用いる．

- スキャタープロット
- ディレクトリツリー
- ファイルリスト
- グループリスト

以降ではこの 4 つのビューについて説明する．

スキャタープロット　スキャタープロットはソフトウェアのどの部分とどの部分がクローンになっているのを俯瞰的に表示する．図 3 にスキャタープロットの簡単なモデルを示す．スキャタープロットでは，解析対象のソフトウェアのソースコード中のトークンが，その出現順に水平方向と，垂直方向に並べられている．そして，水平成分と垂直成分のトークンが等しいとき，点をプロットする．このように点をプロットしていくと，CCFinder が検出したコードクローンはスキャタープロット上では，ある一定以上の長さを持った対角線分として表示される．水平方向と垂直方向の格子はファイルの区切りを表している．また，スキャタープロットはディレクトリツリーと連動している．ディレクトリツリーでファイルが選択された場合，スキャタープロットは選択されたファイルの部分をハイライト表示する．

ディレクトリツリー　ディレクトリツリーは，対象ファイルをディレクトリの階層構造で表示する．水平方向ファイルと垂直方向ファイル用に 2 つ存在する．先にも述べたが，ディレクトリツリーでファイルが選択された場合，スキャタープロットは選択されたファイルの部分をハイライト表示する．

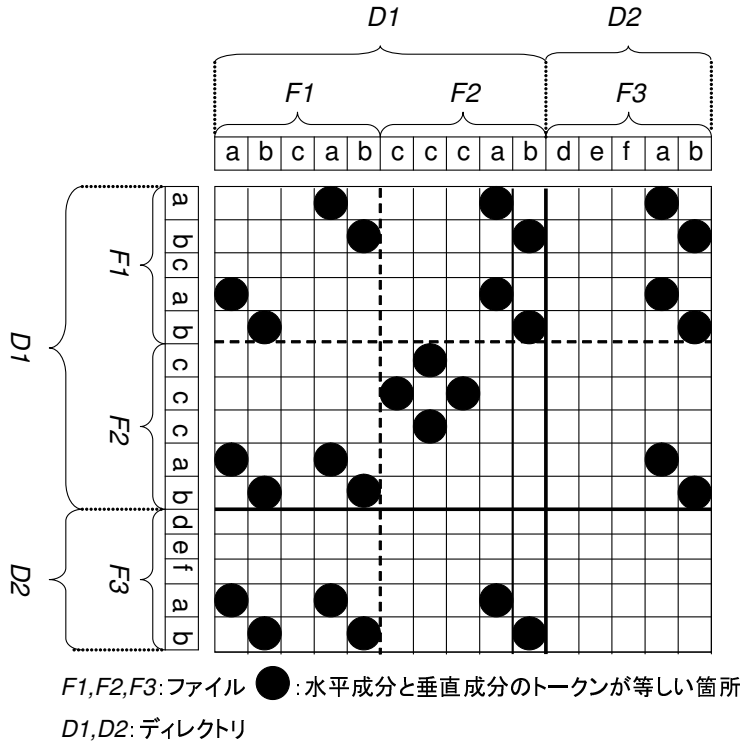

図 3: スキャタープロットの例

ファイルリスト　ファイルリストは，各ファイルの定量的な情報を提供している．ファイルリストでは，各ファイルの規模情報（行数，トークン数）と共に，そのファイルが含んでいるコードクローンの数，コードクローンがそのファイルに閉める割合などの情報が表示される．ユーザはファイルリストを用いることによって，どのファイルが多くクローンを含んでいるのか，どのファイルに重複部分が多く存在するのかといったことを知ることが出来る．また，ファイルリストには，フラグメントリストと関連ファイルリストというサブリストが存在す．るファイルリストでファイルを選択すると，そのファイル内に存在するコードクローンの情報がフラグメントリストに，選択されたファイルとコードクローンを共有しているファイル一覧が関連ファイルリストに表示される．

グループリスト　グループリストでも定量的な情報が提供される．グループリストでは，グループ単位の定量的な情報が表示される．CCFinder でコードクローンを検出する際にオプションを設定することで，解析対象ファイルをグループに分割することが出来る．例えば，１つのモジュールを１つのグループにしてコードクローン検出を行うと，グループリストを用いることによってモジュール間の類似度を得ることが出来る．

2.3.2 クローンベースの解析

クローンベースの解析では，ユーザはコードクローンをその特徴に基に調べることができる．クローンベースの解析では主にメトリクスグラフを用いてコードクローンの絞込みを行う．図 4 にメトリクスグラフのスナップショットを示す． メトリクスグラフは多次元並行座標表現 [14] を用いている．Gemini のメトリクスグラフでは，RAD(S)，LEN(S)，RNR(S)，NIF(S)，POP(S)，DFL(S) の 6 つのメトリクスを用いている．以下に各メトリクスについての説明を行う．

RAD(S) クローンセット S の各要素であるコードクローンを含むファイルから共通の親ディレクトリまでの距離の最大値を表す．

LEN(S) クローンセット S に含まれるコードクローンの平均トークン数を表す．この値が大きいほど，コードクローンのサイズが大きいことになる．

RNR(S) クローンセット S に含まれるコードクローン内において，重複していない処理が存在する割合の平均値を表す．コードクローン C 中の総トークン数を $Tokens_{all}(C)$，繰り返し部分のトークン数を $Tokens_{repeated}(C)$ とすると，RNR(S) は式 (1) で表される．

$$RNR(S) = 1 - \frac{\sum_{C \in S} Tokens_{repeated}(C)}{\sum_{C \in S} Tokens_{all}(C)} \tag{1}$$

この値が大きければ大きいほど，繰り返し要素を多く含むことになる．このメトリクスを用いることによって，連続した変数宣言やメソッド呼び出しなどのコードクローンをフィルタリングすることが可能となり，それらを除外して，より効率的にコードクローン分析作業を行うことができるようになる．

NIF(S) クローンセット S に含まれるコードクローンを所有するファイルの数を表す．この値が大きいほど，コードクローンが多くのファイル中に含まれていることを表している．

POP(S) クローンセット S に含まれるコードクローンの数を表す．これは NIF とは違い，同じファイル中にクローンセット S に含まれるコードクローンが複数存在する場合には，その数の分カウントする．

DFL(S) クローンセット S に含まれるコードクローンをサブルーチンなどに集約した場合に，削減されるトークン数の予測値を表す．この値が大きいほど，プログラムサイズ面で

の集約の効果が大きいことを表す.

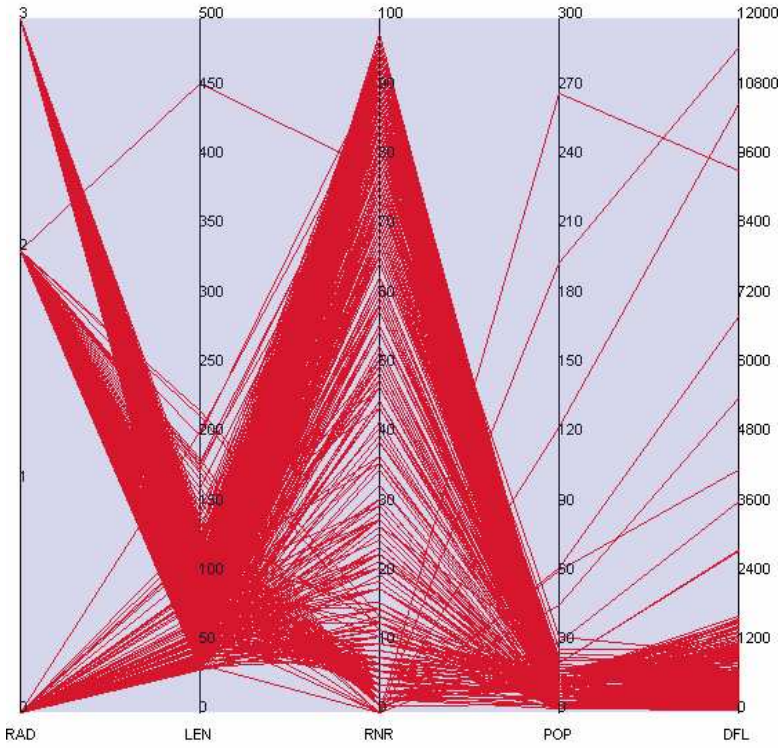

図 4: メトリクスグラフの例

メトリクスグラフでは，各メトリクスにつき，1 つの座標軸が用意されている．そして，各クローンセットのメトリクス値を元に各座標軸に点をプロットし，隣り合う点を直線で結ぶ．このようにすると，1 つのクローンセットは 1 本の折れ線として特徴を現すことが出来る．メトリクスグラフでは任意のメトリクスについて，上限と下限を変更することでクローンセットの絞込みが行えるようになっている．例として図 5(b) では RNR 値の下限を変更した状態を表している．この変更によって，図 5(a) では選択状態 (赤線) であったクローンセット S_2 が非選択状態 (破線) となっている．

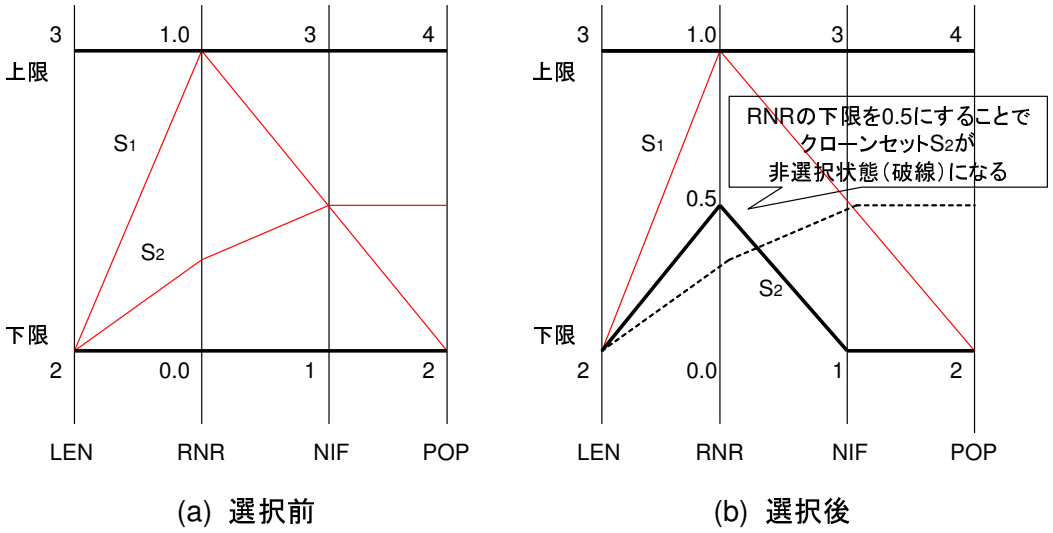

図 5: メトリクスグラフを用いたクローンセットの選択

2.4 コードクローンの分類

一般に，ある与えられたコード片 C に対するコードクローンは，以下の 3 種類に分類される．

Exact クローン (Exact clone): C に対し，プログラムテキストとして完全に一致したコード片．但し，文字列としての一致とは異なり，空白，改行，コメントなどの違いは考慮しない．

Parameterized クローン (Parametarized clone): C に対し，変数名，定数名，クラス名，メソッド名等のユーザ定義名の違いを除き一致しているコード片．つまり言語依存の予約語等の構文的な一致を指す．

Gapped クローン (Gapped clone): C に対し部分的に類似しているコード片．つまり，構文的にも一致しない不一致コード (ギャップ (Gap)) を，部分的に含む．

我々が検出対象にしているクローンの不一致部分 (ギャップ (Gap)) とは，コピーとペーストによるプログラミングにおいて，ペーストされたコード片に含まれる，新規追加されたコード，削除されたコード，または構文が変更されたようなコードを意味する．図 6 に，Exact クローン，Parameterized クローン，Gapped クローンの例を示す．

CCFinder で検出できるのは Exact クローンと Parameterized クローンである．しかし，既存コードのコピーとペーストによる再利用の際など，ペーストされたコード片はそのまま

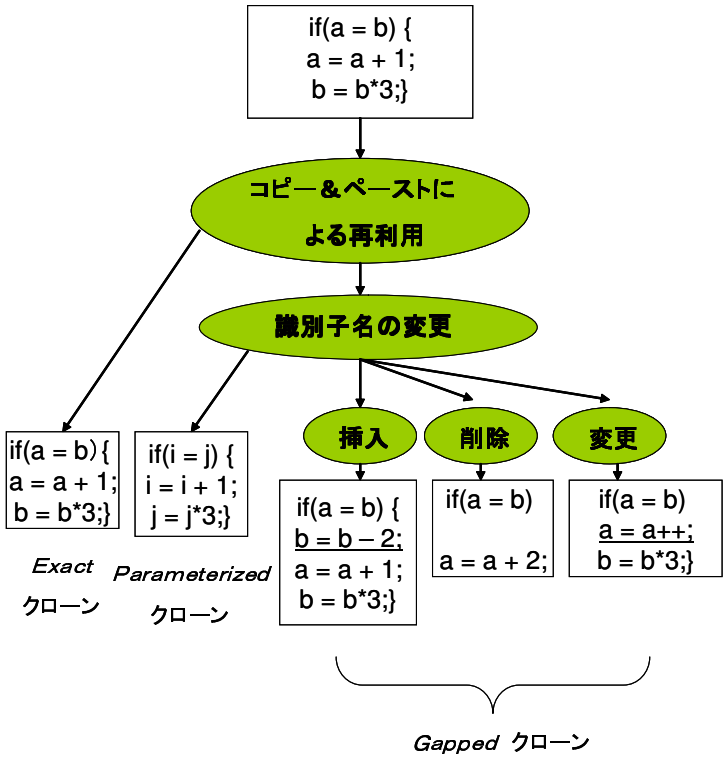

図 6: コードクローンの分類

利用されることは少なく，ペースト先の文脈に合わせた部分的な修正や変更が行われる可能性が高い．そのような場合，コピー元との不一致が生じるため，Gapped クローンも検出できることが望ましい．本研究では，グラフマイニングアルゴリズムの 1 つである AGM アルゴリズムを用いてギャップを含むクローンを抽出する手法を提案する．

3 AGM アルゴリズム

AGM(Apriori-based Graph Mining) アルゴリズム [22][23] は，多くの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．AGM アルゴリズムでは，与えられるグラフが有向グラフでも無向グラフでも，多頻度グラフパターンを抽出できるが，本手法では有向グラフのみ扱うので，これ以降では有向グラフの場合のみを説明する．

3.1 グラフ構造データ

ラベル付きグラフ G は $G = (V, E, L_V, L_E, lb)$ で表される．ここで $V = \{v_1, v_2, \ldots, v_k\}$ は頂点の集合，$E = \{(v_i, v_j) \mid v_i, v_j \in V\}$ は辺の集合，$L_V = \{lb(v_i) \mid \forall v_i \in V\}$ は頂点ラベルの集合，$L_E = \{lb((v_i, v_j)) \mid \forall (v_i, v_j) \in E\}$ は辺ラベルの集合，lb はラベル付けの関数 $lb : (V \to L_V) \cup (E \to L_E)$ である．グラフ G の頂点，辺，頂点ラベル，辺ラベルの集合をそれぞれ $V(G)$, $E(G)$, $L_V(G)$, $L_E(G)$ と表す．グラフの頂点数をグラフのサイズと呼ぶ．

グラフは隣接行列を用いて表現できる．$num(lb(v_i))$ と $num(lb((v_i, v_j)))$ をそれぞれ頂点ラベルと辺ラベルに割り当てられた正の整数値とすると，サイズが k のグラフ G が与えられたとき，その隣接行列は $k \times k$ の整数行列 X_k であり，その (i, j)-要素 $x_{i,j}$ は

$$x_{i,j} = \begin{cases} num(lb((v_i, v_j))), & (v_i, v_j) \in E(G) \\ 0, & (v_i, v_j) \notin E(G) \end{cases}$$

となる．ここで，$i, j \in \{1, \ldots, k\}$ である．隣接行列の i 行 (i 列) に相当する頂点を第 i 頂点と呼ぶ．隣接行列 X_k のグラフ構造を $G(X_k)$ と表す．同一のグラフに対応する隣接行列は行 (列) の割り当て方で複数存在する．そこで，同一のグラフを一意に表現するためにある特定の条件を満たす隣接行列を正準形 (canonical form) と呼ぶ．正準形を定義するために，隣接行列のコードを以下のように定義する．

$$code(X_k) = c_1 c_2 c_3 c_4 \ldots c_n, (n = \frac{k(k-1)}{2})$$

ここで，$i < j$ として，

$$c_l = (\mid L_E \mid +1) x_{j,i} + x_{i,j}, (l = i + \frac{(j-1)(j-2)}{2})$$

である．関数 $code$ はグラフの構造と辺ラベルからなり，頂点ラベルの情報を含んでいない．そこで頂点ラベルを含めた関数 $CODE$ を

$$CODE(X_k) = num(X_k) code(X_k),$$

と定義する．ここで関数 $num(X_k)$ は，ラベルに割り当てられた整数値を第 1 頂点から順に並べたもので，

$$num(X_k) = num(lb(v_1)) \ldots num(lb(v_k))$$

である．あるグラフと一対一に対応する正準形は最大 (最小) の $CODE$ をもつ隣接行列と定義する．同型のグラフを表す隣接行列 X_k と Y_k が与えられたとき，以下のような $k \times k$ の変換行列 T_k を用いて互いに変換することができる．

$$t_{i,j} = \begin{cases} 1, & G(X_k) \text{ の第 } i \text{ 頂点が } G(Y_k) \text{ の第 } j \text{ 頂点に相当するとき} \\ 0, & \text{その他のとき} \end{cases}$$

Y_k は $Y_k = T_k^T X_k T_k$ と表される．

グラフ G と G_s が以下の条件を満たすような関数 $\phi : V(G_s) \rightarrow V(G)$ が存在するとき，G_s を G の部分グラフと呼び，$G_s \sqsubseteq G$ と表す．

$$\begin{aligned} &(1)\forall v \in V(G_s), && lb(v) = lb(\phi(v)) \\ &(2)\forall (v_i, v_j) \in E(G_s), && lb(v_i, v_j) = lb(\phi(v_i), \phi(v_j)) \end{aligned}$$

さらに以下を満たすとき，G_s を G の誘導部分グラフと呼び，$G_s \sqsubseteq_i G$ と表す．

$$(v_i, v_j) \in E(G_s) \Leftrightarrow (\phi(v_i), \phi(v_j)) \in E(G)$$

グラフ上の任意の 2 頂点間にパスが存在するとき，連結グラフと呼ぶ．

グラフデータの集合 GD が与えられたとき，グラフパターン G_s の支持度 $sup(G_s)$ を

$$sup(G_s) = \frac{\mid \{G \mid G \in GD, G_s \sqsubseteq_i G\} \mid}{\mid GD \mid}$$

と定義する．ユーザが指定した最小支持度以上の支持度を有するグラフパターンを多頻度グラフ (frequent sub-graph) と呼ぶ．

3.2 AGM アルゴリズムの概要

AGM アルゴリズムの概要を図 7 に示す．ここで GD はグラフの集合からなるデータベース，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の隣接行列の集合，$minsup$ は最小支持度を表している．

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

図 7: AGM アルゴリズムの概要

AGM アルゴリズムは，サイズが 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． 始めに 1 × 1 の隣接行列が頂点ラベルの種類だけ生成され，C_1 に代入される．次に，多頻度グラフパターンの候補の支持度を求め，多頻度グラフパターンを F_k に代入する．続いて，既に抽出されている大きさ 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，最後に全ての多頻度グラフパターンが出力される．

3.2.1 隣接行列結合

図 7 の関数 Generate-Candidate は隣接行列結合と部分グラフチェック，正準化の 3 つの部分からなる．隣接行列結合部では，大きさ k の多頻度グラフパターンの隣接行列を結合して，大きさ $k+1$ の多頻度グラフの候補を生成する．2 つの隣接行列 X_k と Y_k が与えられ，以下の条件を全て満たすとき，2 つの隣接行列を結合し，大きさ $k+1$ の隣接行列 Z_{k+1} を生成する．

条件 1 $V(G(X_k)) = \{x_1, \ldots, x_k\}, V(G(Y_k)) = \{y_1, \ldots, y_k\}$ とすると，X_k と Y_k が第 k 行，および第 k 列の要素以外の要素が全て等しいとき，すなわち

$$X_k = \begin{pmatrix} X_{k-1} & x_1 \\ x_2^T & 0 \end{pmatrix}, Y_k = \begin{pmatrix} X_{k-1} & y_1 \\ y_2^T & 0 \end{pmatrix}$$

であり，任意の $i = 1, \ldots, k-1$ に対して，$lb(x_i) = lb(y_i)$ が成り立つとき．ただし，$x_i \in V(G(X_k))$ であり，$y_i \in V(G(Y_k))$ である．x_i と y_i はそれぞれ $G(X_k)$ と $G(Y_k)$ の第 i 頂点である．

条件 2 X_k が正準形であるとき．

条件 3 $CODE(X_k) \geq CODE(Y_k)$ が満たされるとき．

もし，X_k と Y_k が条件 1-3 を満たし結合可能な場合，2 つの隣接行列を結合して，以下のように Z_{k+1} を生成する．

$$Z_{k+1} = \begin{pmatrix} X_{k-1} & x_1 & y_1 \\ x_2^T & 0 & z_{k,k+1} \\ y_2^T & z_{k+1,k} & 0 \end{pmatrix},$$

であり，任意の $i = 1, \ldots, k-1$ に対して，$lb(x_i) = lb(y_i)$，および $lb(x_k) = lb(y_k)$ が成り立つ．

X_k と Y_k は Z_{k+1} の第 1 生成行列と第 2 生成行列と呼ばれる．Z_{k+1} の 2 つの要素 $z_{k,k+1}$ と $z_{k+1,k}$ は X_k と Y_k の要素からは決めることが出来ない．それらの可能な値として，$G(Z_{k+1})$ の第 k 頂点，第 $k+1$ 頂点の間に辺が無い場合と，あるラベルをもつ辺が存在する場合があるので，Z_{k+1} は $(\mid L_E(E) \mid +1)$ 個の隣接行列が生成される．以上のようにして作られた隣接行列を正規形 (normal form) と呼ぶ．

条件 2 と 3 がなく条件 1 だけの場合でも，存在しうるグラフの正準形を生成することはできるが，条件 2 と 3 は正準形でない冗長な隣接行列の生成を減らすことで計算コストを減らすことが可能である．

3.2.2 部分グラフチェック

$G(Z_{k+1})$ が多頻度グラフであるための必要条件は，その全ての誘導部分グラフが多頻度グラフでなければならないことである．このチェックは部分グラフチェックで行われ，多頻度グラフパターンの候補数を減らすことができる．$G(Z_{k+1})$ の大きさ k の誘導部分グラフの隣接行列は，i 行，及び i 列の全ての要素を取り除くことで得られるが，それが必ずしも正規形とは限らない．AGM アルゴリズムは正規形の隣接行列しか探索，及び生成しないので，その隣接行列に相当するグラフが多頻度グラフであるかチェックするためには，正規形の隣接行列に変換する必要がある．

大きさ $k+1$ のグラフの候補を生成しているとき，大きさ k のグラフの隣接行列は図 8 に示すアルゴリズムで正規形に変換される．ただし全ての正規形の隣接行列は，正準形への変換行列を 1 つ記憶しているものとする．隣接行列 X_k の左上の $i \times i$ の部分行列を X_i，X_i の正準形への変換行列を P_i，$k \times k$ の単位行列を I_k とする．図の変換行列 P_k'，Q_k は以下のように生成される．図 8 の 4 行目は X_i を正準形に変換する操作で，7 行目は第 $i-1$ 頂点を第 k 頂点に，第 j 頂点 $(j = i, \ldots, k)$ を第 $j-1$ 頂点に変更する操作である．

$$P_k^{'} = \begin{pmatrix} P_i & 0 \\ 0 & I_{k-i} \end{pmatrix}, Q_k = \begin{pmatrix} I_{i-2} & 0 & 0 \\ 0 & 0 & 1 \\ 0 & I_{k-i+1} & 0 \end{pmatrix}.$$

0) Normalize(X_k){
1) i←1;
2) while($i \neq k+1$){
3) if(X_i が正規形，かつ第 1 生成行列となりうる){
4) $X_k \leftarrow P_k^{'T} X_k P_k^{'}$;
5) $i \leftarrow i+1$;
6) }else{
7) $X_k \leftarrow Q_k^T X_k Q_k$;
8) $i \leftarrow i-1$;
9) }
10) }
11) return X_k;
12) }

図 8: 正規化アルゴリズムの概要

3.2.3 正準化

候補となるグラフの隣接行列を生成したあと，データベースにアクセスしてそれらの支持度を求める．しかし，正規形の中にも同一のグラフを表す隣接行列が複数個存在するため，正規形の隣接行列の中でどの行列が正準形であるか求める必要がある．X_k から i 行，及び i 列の要素を除いた隣接行列を X_{k-1}^i とする．X_{k-1}^i を正規系に変換する行列を T_{k-1}^i とする．正規化された $(T_{k-1}^i)^T X_k^i T_{k-1}^i$ を正準形に変換する行列を S_{k-1}^i とする．X_k の変換行列 S_k^i と T_k^i は S_{k-1}^i と T_{k-1}^i から，以下のように生成される．

$$S_k^i = \begin{pmatrix} S_{k-1}^i & 0 \\ 0 & 1 \end{pmatrix}, T_k^i = \begin{pmatrix} I_{i-1} & 0 & 0 \\ 0 & 0 & 1 \\ 0 & I_{k-i} & 0 \end{pmatrix} \begin{pmatrix} T_{k-1}^i & 0 \\ 0 & 1 \end{pmatrix}.$$

X_k の正準形の候補は

$$X_c k = \arg\max CODE((T_k^i S_k^i)^T X_k (T_k^i S_k^i))(i = 1, \ldots, k) \tag{2}$$

で与えられる．X_k をその正準形の候補へ変換する変換行列は式 (2) を最大化する行列 $T_k^i S_k^i$ で与えられる．式 (2) の途中で正準形への変換行列 $S_k^{'}$ が既知である隣接行列 $(T_k^i S_k^i)^T X_k (T_k^i S_k^i)$ となった場合には，X_k の正準形は $S_k^{'T} (T_k^i S_k^i)^T X_k (T_k^i S_k^i) S_k^{'}$ で与えられ，全ての i について式 (2) を計算する必要はない．しかしながら，正規形の隣接行列は正準形への変換行列を 1 つしか記憶していないので，式で必ずしも正準形を見つけられるわけではない．このような場合は，正準形に変換したい行列の第 1 生成行列と正準形の第 1 生成行列が等しい場合に起こりうる．このような場合には，X_k の頂点の順列によって正準形が求められる．

3.2.4 頻度計算

全ての正準形が求められたあと，グラフの集合からなるデータベースにアクセスして多頻度グラフパターンの候補の支持度を求める．

例えば大きさが 3 のグラフの多頻度グラフの候補の支持度を計算するとき，大きさが 1 の隣接行列からはじめて，他の隣接行列と組み合わせることにより，順次サイズの大きな誘導部分グラフの隣接行列を生成していく．このとき多頻度グラフに含まれない隣接行列は，他のどの隣接行列とも組み合わせを行わない．最後に大きさが 3 の隣接行列が組み合わせによって作られれば，その正準形のカウンタを 1 つ増やす．

3.3 追加のバイアスを適用した AGM アルゴリズム

文献 [23] では，今まで述べた AGM アルゴリズムに追加のバイアスを適用することで，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を可能にしている．この手法を B-AGM(Biased Apriori-based Graph Mining) と呼ぶ．抽出したいグラフパターンを指定するためのバイアスは，正準形の定義，生成行列の結合条件からなる．本研究ではその中でも，グラフのパス抽出バイアスを加えることにした．本研究では，ファイル中のコードクローンの関係をグラフで表わし，ギャップを含むクローンを，グラフ上で 2 回以上現れるパスと定義しているためである．次節ではパス抽出バイアスの説明をする．

3.3.1 パス抽出バイアス

このバイアスは，閉路や枝分かれを持たないグラフパターンであるパスを抽出するためのバイアスである．

正準形　隣接行列 X_k の左上側 $i \times i$ の部分行列を $X_i(1 \leq i \leq k)$ とする．同一のグラフ G_1 を表す隣接行列の集合 $\Gamma_1(G_1)$ を

$$\Gamma_1(G_1) = \{X_k \mid G(X_i) : \text{連結}, \forall i = 1, \ldots, k-1, G_1 \equiv G(X_k)\}$$

と定義すると，グラフ G_1 の正準形 C_k は $\Gamma_1(G_1)$ の中で $CODE$ が最大になる隣接行列である．

$$C_k \quad s.t. \quad CODE(C_k) = \max_{X_k \in \Gamma_1(G_1)} CODE(X_k)$$

生成行列の結合条件　条件 1，2 は従来の AGM と同様で，新たに条件 3，4，5 が加えられる．

条件 3 $CODE(X_k) \geq CODE(Y_k)$，あるいは $G(Y_k)$ が非連結グラフであるとき．
条件 4 $G(X_k)$ が連結グラフであるとき．
条件 5 $G(Y_k)$ が連結である場合は，Z_{k+1} の要素 $z_{k+1,k}$ と $z_{k,k+1}$ は 0 となるとき．

4 AGM アルゴリズムを用いたギャップを含むクローン抽出手法

本研究では，グラフマイニングアルゴリズムの 1 つである AGM アルゴリズムを利用して，ギャップを含んだクローンを抽出する手法の提案をする．

4.1 提案手法の説明

本手法では，CCFinder が出力する，Exact クローンと Parameterized クローン（以下，この 2 つを併せて Ng クローン (Non-gapped クローン) と呼ぶことにする）の位置情報が書かれたファイルを入力とし，ギャップを含むクローンの位置情報が書かれたファイルを出力とする．CCFinder が出力する，Ng クローンの位置情報が書かれたファイルの例を図 9 に示す．

図 9 の#begin{set} から#end{set} の間までが，1 つのクローンセットに含まれるコードクローンの位置情報を表している．コードクローンの位置情報は，”グループ ID. ファイル ID　開始行, 開始カラム, 開始トークン　終了行, 終了カラム, 終了トークン　繰り返し処理ではないトークン数”というフォーマットで書かれている．

```
#version: ccfinder 7.3.2
#format: classwise
#langspec: JAVA
#option: -b 30
#option: -e char
#option: -k 30
#option: -r abcdfikmnop-rsuv
#option: -c wfg
#option: -y
#begin{file description}
0.0     1404    2309    C:¥Documents and Settings¥Administrator¥My Documents¥f2j-0.7¥f2j-0.7¥goto_trans¥byte.c
0.1     948     2048    C:¥Documents and Settings¥Administrator¥My Documents¥f2j-0.7¥f2j-0.7¥goto_trans¥class.c
0.2     292     598     C:¥Documents and Settings¥Administrator¥My Documents¥f2j-0.7¥f2j-0.7¥goto_trans¥dump.c
0.3     306     617     C:¥Documents and Settings¥Administrator¥My Documents¥f2j-0.7¥f2j-0.7¥goto_trans¥main.c
0.4     130     292     C:¥Documents and Settings¥Administrator¥My Documents¥f2j-0.7¥f2j-0.7¥goto_trans¥symtab.c
0.5     117     230     C:¥Documents and Settings¥Administrator¥My Documents¥f2j-0.7¥f2j-0.7¥libbytecode¥dlist.c
0.6     438     56      C:¥Documents and Settings¥Administrator¥My Documents¥f2j-0.7¥f2j-0.7¥libbytecode¥globals.c
0.7     115     224     C:¥Documents and Settings¥Administrator¥My Documents¥f2j-0.7¥f2j-0.7¥src¥dlist.c
0.8     170     236     C:¥Documents and Settings¥Administrator¥My Documents¥f2j-0.7¥f2j-0.7¥src¥f2jmem.c
0.9     104     209     C:¥Documents and Settings¥Administrator¥My Documents¥f2j-0.7¥f2j-0.7¥src¥getopt.c
0.10    939     240     C:¥Documents and Settings¥Administrator¥My Documents¥f2j-0.7¥f2j-0.7¥src¥globals.c
0.11    1429    2806    C:¥Documents and Settings¥Administrator¥My Documents¥f2j-0.7¥f2j-0.7¥src¥optimize.c
0.12    291     444     C:¥Documents and Settings¥Administrator¥My Documents¥f2j-0.7¥f2j-0.7¥src¥symtab.c
0.13    1472    3074    C:¥Documents and Settings¥Administrator¥My Documents¥f2j-0.7¥f2j-0.7¥src¥typecheck.c
0.14    958     2004    C:¥Documents and Settings¥Administrator¥My Documents¥f2j-0.7¥f2j-0.7¥src¥vcg_emitter.c
#end{file description}
#begin{syntax error}
#end{syntax error}
#begin{clone}
#begin{set}
0.1     73,3,124        142,59,392      170
0.1     71,3,114        142,59,392      181
#end{set}
```

フォーマット
グループID.ファイルID 開始行,開始カラム,開始トークン 終了行,終了カラム,終了トークン 繰り返し処理でないトークン数
0 . 1 73 , 3 , 124 142 , 59 , 392 170

図 9: クローンの位置情報が書かれたファイルの例

4.1.1 以降ではギャップを含むクローンの抽出手順について説明する．

4.1.1 抽出手順

本研究で提案するギャップを含むクローン抽出手法は，次の 3 つのステップで構成されている．

ステップ 1:各ソースファイルのグラフの構築

ステップ 2:構築したグラフに現れる多頻度グラフパターン (=ギャップを含むクローン) の検出

ステップ 3:検出したギャップを含むクローンの位置情報の出力

ステップ 1 においては，入力として与えられた，CCFinder が出力する Ng クローンの位置情報を元に，AGM アルゴリズムの入力であるグラフ集合を構築する．ステップ 2 においては，AGM アルゴリズムを用いて，ステップ 1 で構築したグラフ集合に現れる多頻度グラフパターンを抽出する．ステップ 3 においては．ステップ 2 で抽出したギャップを含むクローンの位置情報を出力し，Gemini で分析できるようにする．

以下，各ステップの詳細を説明する．

ステップ 1(各ソースファイルのグラフの構築): 入力として与えられた Ng クローンの位置情報を元に，各クローンのコード片をノードとするグラフをソースファイル毎に作成する．ある 2 つのクローンがあり，その 2 つのクローンの間に不一致部分 (Gap) があれば，それらのクローンに対応するノードにエッジを引く．不一致部分があるかどうかの判断は，あるクローンの終了位置と別のクローンの開始位置が離れていれば，不一致部分があると判断する．これを全てのクローンに対して実行し，ソースファイル毎にグラフを構築する．構築されるグラフの例を図 10 に示す．

グラフを構築する際，2 つのクローンの位置がオーバーラップしている場合や，1 つのクローンが他のクローンを包含している場合には，これら 2 つのノードの間にはエッジを引かない．なぜなら，オーバーラップしているということは，部分的な修正や変更が行われた結果，不一致部分が生まれた訳ではないと考えられるからである．

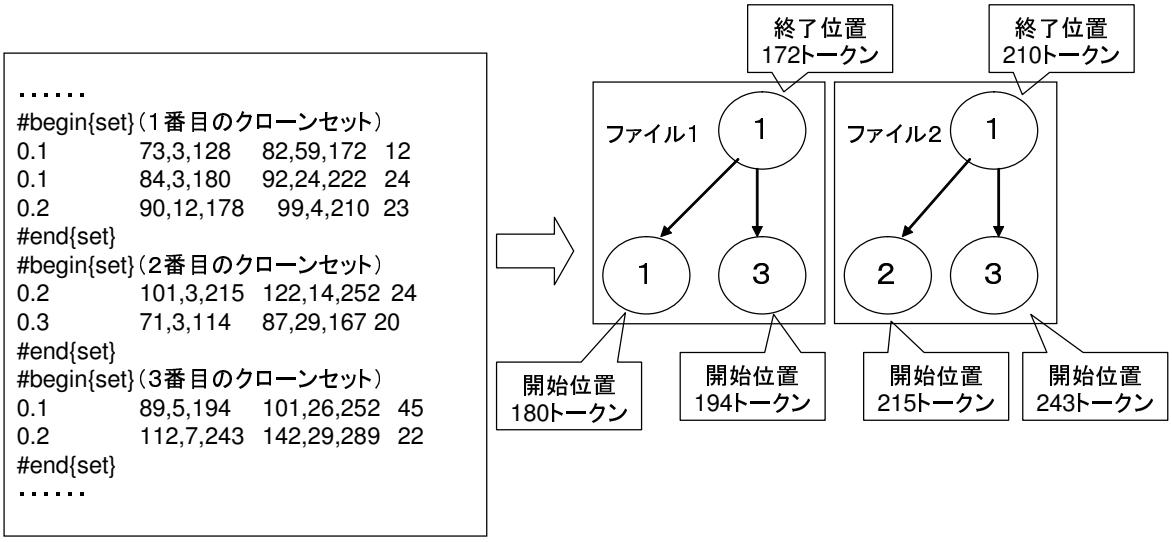

図 10: 構築されるグラフの例

ステップ 2(多頻度グラフパターン (=ギャップを含むクローン) の抽出): ステップ 1 で構築したグラフ集合に対して AGM アルゴリズムを適用し，そのグラフ集合に現れる多頻度グラフパターンを抽出し，そのパスをギャップを含むクローンとする．具体的には，グラフ集合の中で 2 回以上 (AGM アルゴリズムにおける最小支持度) 現れたパスをギャップを含むクローンとする．図 11 の場合では，ファイル 1 とファイル 2 に含まれる $1 \rightarrow 3 \rightarrow 4$ というパスが，ギャップを含むクローンとして検出される．

ステップ 3(ギャップを含むクローンの位置情報の出力): ステップ 2 で抽出した多頻度グラフパターンを使って，各グラフからその多頻度グラフパターンに対応するグラフのパスを取得し，ノードが保持するクローンの位置情報からギャップを含むクローンの位置情報を出力し，Gemini で分析ができるようにする．ギャップを含むクローンの位置情報の例を図 12 に示す.

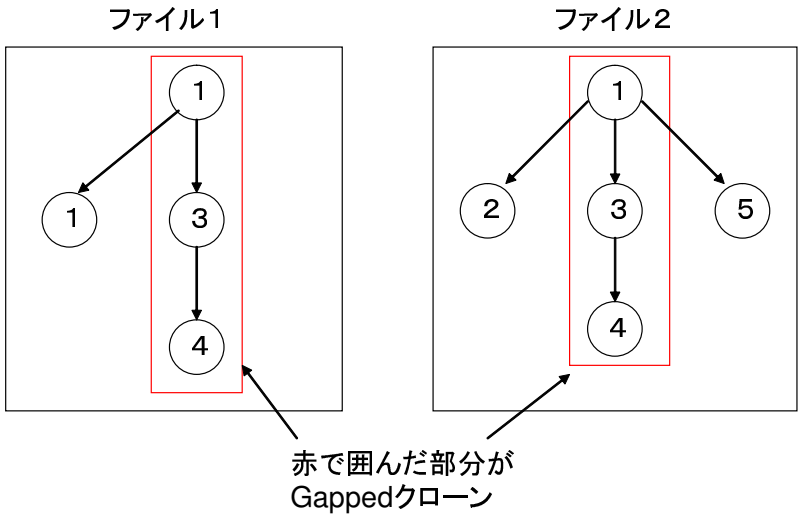

図 11: ギャップを含むクローンの抽出例

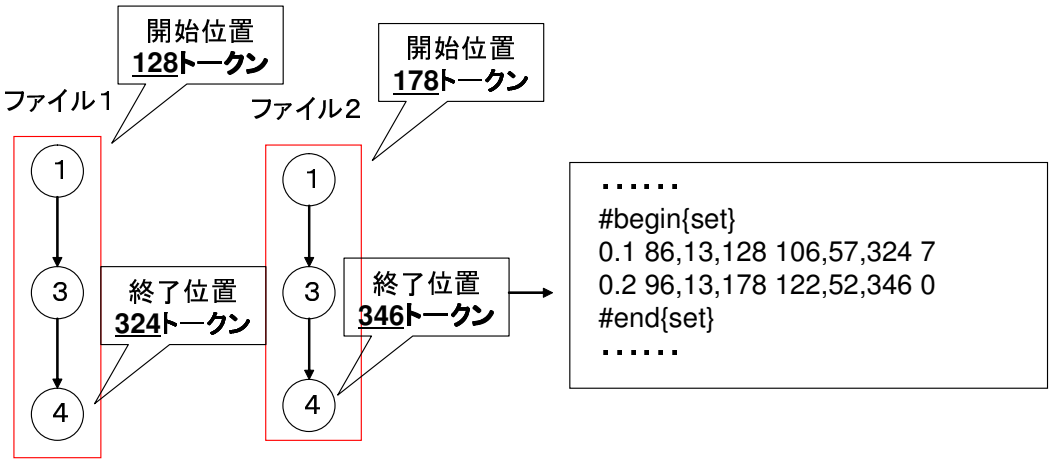

図 12: ギャップを含むクローン位置情報の出力例

5 適用実験

本節では，前章で述べた提案手法によって，どのようなコードクローンが抽出されるかを調査するために行った実験について説明する．本実験では，CCFinder の検出する最小のコードクローンの大きさは 30 トークンとした．30 トークンとは，我々がこれまでに CCFinder を用いて行ってきた経験から導かれた値である．また，不一致部分の最大トークン数を 100 トークンとした．実験を行ったオープンソースソフトウェアは，オープンソース・ソフトウェア開発サイト SourceForge.net[12] から入手した．それらのデータサイズを表 1 に示す．

この適用実験では，2.3.2 節で述べたメトリクス値である RNR を用いて，グラフのノードとして用いるクローンをフィルタリングすることにした．閾値としては 0.5 を用いることにした．フィルタリングをした理由として，printf 文が連続している部分や数十，数百の定数定義が続いている部分などのように，単純な構文が連続している部分はそもそもクローンとして検出する必要がなく，実際にユーザにとっても中身を検討する価値がないことが挙げられる．

5.1 実行時間

実行時間の計測結果は次の表 2 の通りである．

ソフトウェア名	プログラミング言語	クローンセット数	ファイル数	コードサイズ
EJE	Java	82	57	約 6,000 行
f2j	C	90	15	約 10,000 行
jasmin	Java	149	103	約 14,000 行
javadjvu	Java	215	66	約 37,000 行

表 1: 調査対象データサイズ

ソフトウェア名	クローンセット数	ファイル数	ギャップを含むクローンセット数	実行時間 (s)
EJE	82	57	5	3.995
f2j	90	15	33	16.73
jasmin	149	104	17	94.165
javadjvu	215	66	13	1518.604

表 2: 実行時間調査結果

以上の結果から，クローンセット数が少なくてもファイル数が多ければ，実行時間が多くなっていることがわかった．これは，各ファイル毎にグラフを構築するので，ファイル数が多ければグラフの構築に時間がかかってしまうためであると考えられる．

また，1 つのファイルにクローンが集中していると，実行時間が格段にかかってしまうこともわかった．これは，グラフのサイズが大きくなると，AGM アルゴリズム中の隣接行列結合や，頻度計算の部分で時間がかかってしまうためである．

5.2 抽出したギャップを含むクローンの調査

抽出したギャップを含むクローンにどのようなギャップが含まれるか調査を行った．その理由は，本研究の目的が，ペースト先の文脈に合わせて部分的な修正や変更が行われたために，コピー元との不一致が生じてしまったクローンを発見することだからである．調査結果は以下の表 3 のようになる．ソースコードを見るだけでは，どちらのファイルのコードを挿入・削除したか判断できないので，挿入・削除が加えられたクローンセットはまとめてある．また，挿入・削除と変更が同時に起こっているものもあったので，それらの合計数と，編集が加えられた数は，必ずしも一致するわけではない．

この中で，f2j で抽出されたクローンのうち，編集が加えられていなかったものの 1 つを図 13 に示す．このクローンセットは optimize という同じファイル中のほぼ同じ行のところで見つかったものである．このようなクローンは，コピーとペーストによって生じたクローンを編集したものではないと考えられる．他のソフトウェアでも，クローンがオーバーラップした例が見つかったので，これらをフィルタリング出来るようにすれば，編集が加えられたクローンセットの割合が増え，更に良い結果になると考えられる．

	EJE	f2j	jasmin	javadjvu
編集が加えられたクローンセット数	4	7	7	4
挿入・削除が加えられたクローンセット数	2	7	7	4
変更が加えられたクローンセット数	3	4	3	1
編集が加えられていないクローンセット数	1	26	10	9
編集が加えられたクローンセット数の割合	80%	21%	41%	31%

表 3: ギャップを含むクローンの調査結果

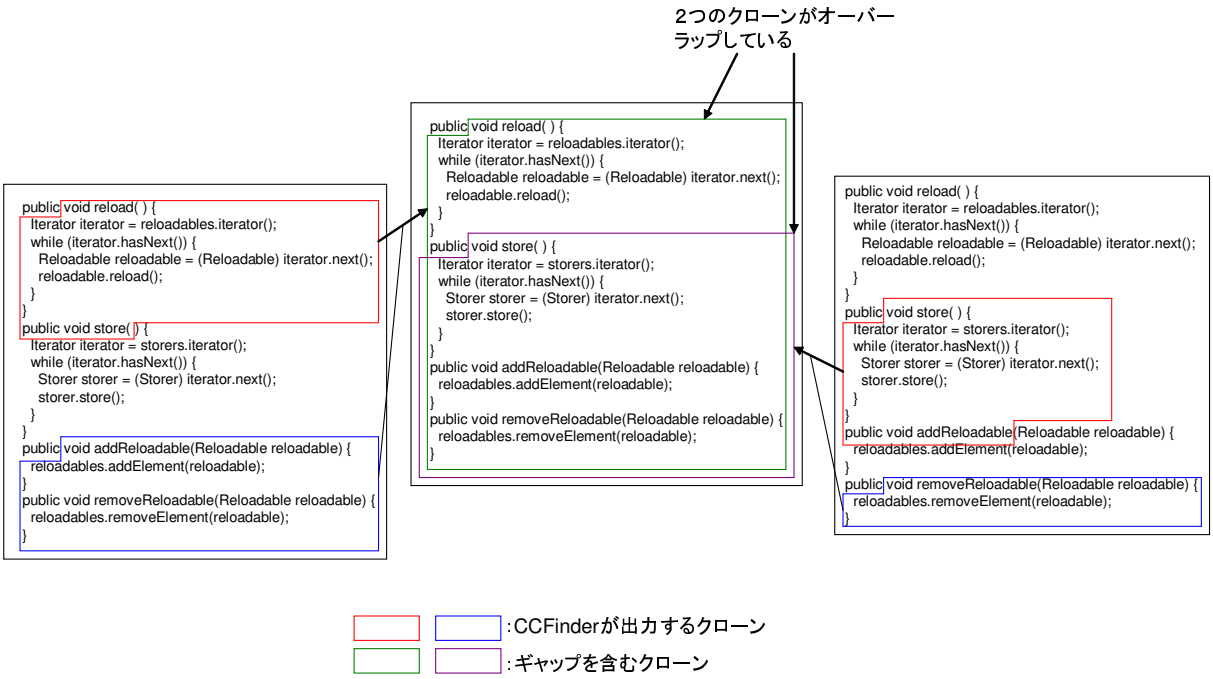

図 13: 編集が加えられていないクローンセットの例

編集が加えられたクローンとして，前述した挿入や削除，変更の他に，不一致部分にコメントしか含まないものも挙げられる．図 14 にその例を示す．本実験では、このような例は見られなかった．

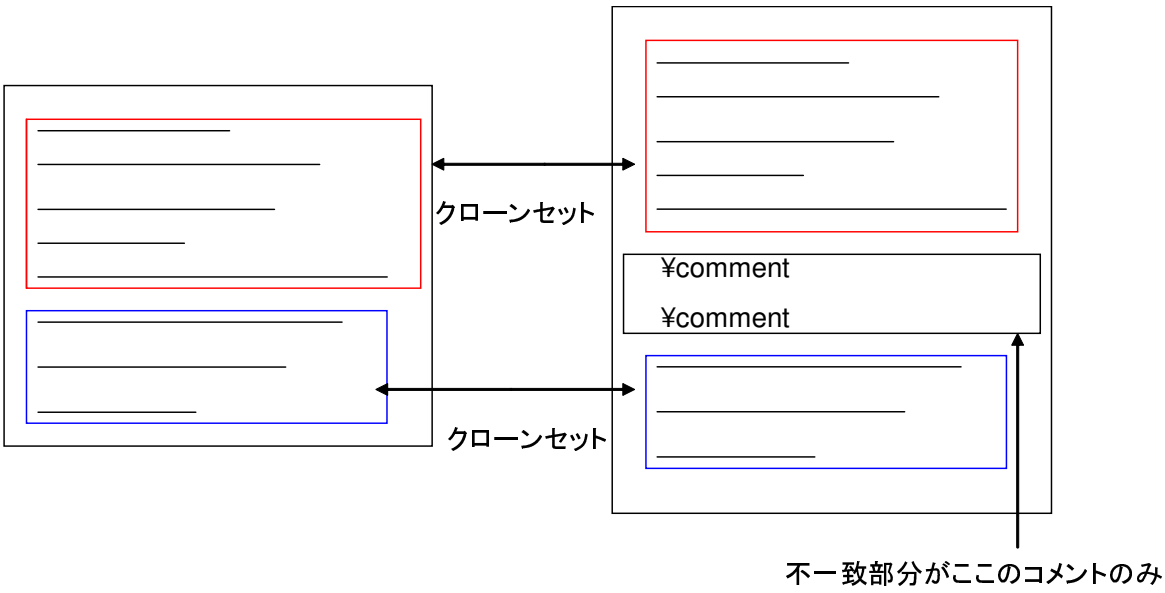

図 14: 不一致部分にコメントしか含まれないクローンセットの例

図 15 に，jasmin で見つかったギャップを含むクローンの中で，編集が加えられていたクローンセットの例を示す．jasmin は Java アセンブラプログラムで，図 15 の左半分が hworld.java ファイルの main メソッド，右半分が all.java ファイルの main メソッド中のコードを表している．メソッド呼び出しの際の引数が違っていたり，新しいオブジェクト doit を生成しているなど，太字で示した処理が追加されただけで，本質的にはこれら 2 つのメソッドは同様のことを行っており，これらのコード片の間には依然として強い類似性があると考えられる．残りのクローンセットについては主なものを付録に載せている．

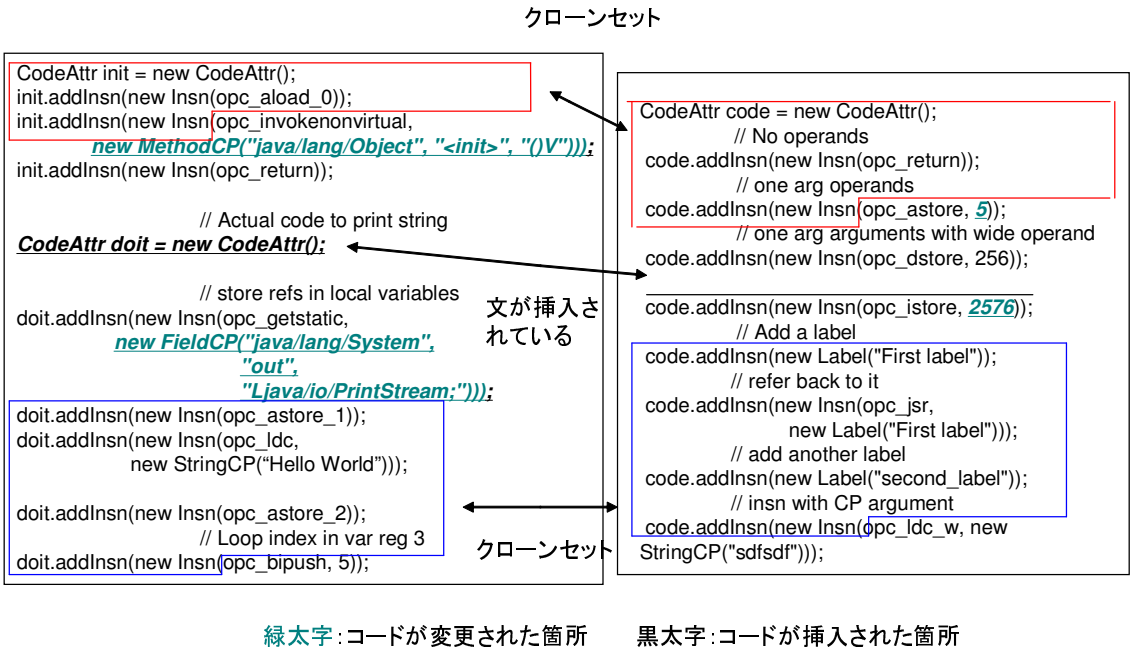

図 15: 編集が加えられていたクローンセットの例 1

6　まとめと今後の課題

本研究では，グラフマイニングアルゴリズムである AGM アルゴリズムを利用して，ギャップを含むクローンの抽出手法の提案を行った．具体的には，CCFinder が出力するコードクローンの位置情報を元に，各ソースファイル毎にグラフを作成し，そのグラフに 2 回以上現れるパスをギャップを含むクローンとして抽出する．そして抽出したギャップを含むクローンの位置情報を出力し，Gemini で分析できるようにする．

そしてこの手法を Java を用いて実装し，オープンソースソフトウェアに対して適用実験を行った．その結果，どのオープンソフトウェアに対してもギャップを含むクローンを適切に抽出できることが確認された．

ゆえに，本手法を用いることで，CCFinder が検出できなかった不一致部分を含むクローンを抽出することが出来るようになり，より効率的にコードクローン分析を行うことができると期待される．

今回提案した手法では，実験で用いたソフトウェアによって実行時間に大きな差が出てしまったので，実装したプログラムに更に改良を加え，実行時間を更に短くする必要がある．また，有効性を更に高めるために，ギャップを含むクローンの中でも，編集されていないクローンをフィルタリング出来るようにする必要がある．更に，ギャップの中にバグが見つかれば，バグ検出手法に応用が可能だが，本研究ではバグの調査が出来なかったので，バグの調査をしてバグ検出手法を評価をする必要がある．以上より，今後の課題としては，実行時間の短縮，編集されていないクローンのフィルタリング，バグ検出手法としての有益性評価が挙げられる．

謝辞

本研究を通して，常に適切な御指導，御助言を賜りました大阪大学大学院情報科学研究科コンピュータサイエンス専攻井上克郎教授に心より深く感謝致します．

本研究を通して，常に適切な御指導，御助言を賜りました大阪大学大学院情報科学研究科コンピュータサイエンス専攻楠本真二教授に心から感謝致します．

本研究を通して，常に適切な御指導，御助言を賜りました大阪大学大学院情報科学研究科コンピュータサイエンス専攻松下誠助教授に心から感謝致します．

本研究を通して，逐次適切な御指導，御助言を頂きました大阪大学大学院情報科学研究科コンピュータサイエンス専攻肥後芳樹氏に深く感謝致します．

最後に，その他様々な御指導，御助言等を頂いた大阪大学大学院情報科学研究科コンピュータサイエンス専攻井上研究室の皆様に深く感謝致します．

参考文献

[1] A. Aiken, " A System for Detecting Software Plagiarism (Moss Hompage) ", http://www.cs.berkley.edu/~aiken/moss.html

[2] B. S. Baker, "A Program for Identifying Duplicated Code", *Computing Science and Statistics*, 24:pp.49-57, 1992.

[3] B. S. Baker, " On Finding Duplication and Near-Duplication in Large Software Systems ", *Proceedings of the 2nd Working Conference on Reverse Engineering*, pp.86-95, 1995.

[4] B. S. Baker, " Parameterized Duplication in Strings: Algorithms and an Application to Software Maintenance ", *SIAM Journal on Computing*, 26(5):pp.1343-1362, 1997.

[5] M. Balazinska, E. Merlo, M. Dagenais, B. Lag¨ue, and K. Kontoginannis, " Advanced Clone-Analysis to Support Object-Oriented System Refactoring ", *Proceedings of the 7th Working Conference on Reverse Engineering*, pp.98-107, 2000.

[6] M. Balazinska, E. Merlo, M. Dagenais, B. Lag¨ue, and K. Kontoginannis, " Measuring Clone Based Reengineering Opportunities ", *Proceedings of the 6th IEEE International Symposium on Software Metrics*, pp.292-303, 1999.

[7] M. Balazinska, E. Merlo, M. Dagenais, B. Lag¨ue, and K. Kontoginannis, " Partial Redesign of Java Software Systems Based on Clone Analysis ", *Proceedings of the 6th IEEE International Working Conference on Reverse Engineering*, pp.326-336, 1999.

[8] I.D. Baxter, A. Yahin, L. Moura, M. Sant ' Anna, and L. Bier, " Clone Detection Using Abstract Syntax Trees", *Proceedings of the 14th IEEE International Conference on Software Maintenace-1998*, pp.368-377, 1998.

[9] E.Burd, and J. Bailey, "Evaluating Clone Detection Tools for Use during Preventative Maintenance", *Proceedings of the 2nd IEEE International Workshop on Source Code Analysis and Manipulation*, pp.36-43, 2002.

[10] S. Ducasse, M. Rieger, and S. Demeyer, "A Language Independent Approach for Detecting Duplicated Code ", *Proceedings of the 15th IEEE International Conference on Software Maintenance-1999*, pp.109-118, 1999.

[11] D. Gusfield，*Algorithms on Strings，Trees，And Sequence*，Cambridge University Press，1997.

[12] SorceForge.net，http://sourceforge.net/.

[13] T. Kamiya，S. Kusumoto，and K. Inoue，" CCFinder: A multi-linguistic token-based code clone detection system for large scale source code "，*IEEE Transactions on Software Engineering*，28(7):pp.654-670，2002.

[14] 加藤博己，" データベースのビジュアルな検索と分析 (OLAP) "，IPSJ Magazine Vol.41 No.4 pp.363-368，2000.

[15] R. Komondoor，and S. Hirwitz，" Using Slicing to Identify Duplication in Source Code "，*Proceedings of the 8th International Symposium on Static Analysis*，2001.

[16] J. Krinke，" Identifying Similar Code with Program Dependece Graphs "，*Proceedings of the 8th Working Conference on Reverse Enngineering*，pp.562-584，2001.

[17] J. Mayland，C. Leblanc，and E. Merlo，" Experiment on the Automatic Detection of Function Clones in a Software System Using Metrics "，*Proceedings of the 12th IEEE International Conference on Software Maintenace-1996*，pp.244-253，1996.

[18] L. Prechelt，G. Malpohl，and M. Philippsen，"Finding plagiarisms among a set of programswith JPlag "，*resubmitted to Journal of Universal Computer Science*，2001. http://www.ipd.uka.de/?prechelt/Biblio/#jplag

[19] Y. Ueda，T. Kamiya，S. Kusumoto，and K. Inoue，" Gemini: Maintenance Support Environment Based on Code Clone Analysis "，*Proceedings of the 8th International Symposium on Software Metrics*，pp.67-76，2002.

[20] 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"，電子情報通信学会論文誌 D-I，vol.86-D-I，no.12，pp.863-871，2003.

[21] Y. Ueda，T. Kamiya，S. Kusumoto，and K. Inoue，"On Detection of Gapped Code Clones using Gap Locations "，*Proceedings of the 9th Asia-Pacific Software Engineering Conference*，pp.327-336，2002.

[22] 猪口明博，鷲尾隆，元田浩，熊澤公平，荒井尚英，" 多頻度グラフパターンの完全な高速マイニング手法"，人工知能学会誌，vol.15，no.6，pp.1052-1063，2000.

[23] 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”，人工知能学会論文誌，vol.19，no.5-B，pp.368-378，2004.

[24] Z. Li，S. Lu，S. Myagmar，and Y. Zhou，”CP-Miner: Finding Copy-Paste and Related Bugs in Large-Scale Software Code ”，*IEEE Transactions on Software Engineering*，32(3):pp.176-192，2006.

[25] X. Yan, J. Han, and R. Afshar, ”CloSpan: Mining Closed Sequential Patterns in Large Datasets,”*Proc. SIAM Int’l Conf.Data Mining*, May 2003.

付録

A. 適用実験によって抽出したギャップを含むクローンの例

A. 適用実験によって抽出したギャップを含むクローンの例

ここでは，本研究の適用実験によって抽出したギャップを含むクローンのうち，編集が加えられたギャップを含むクローンの例を提示する．

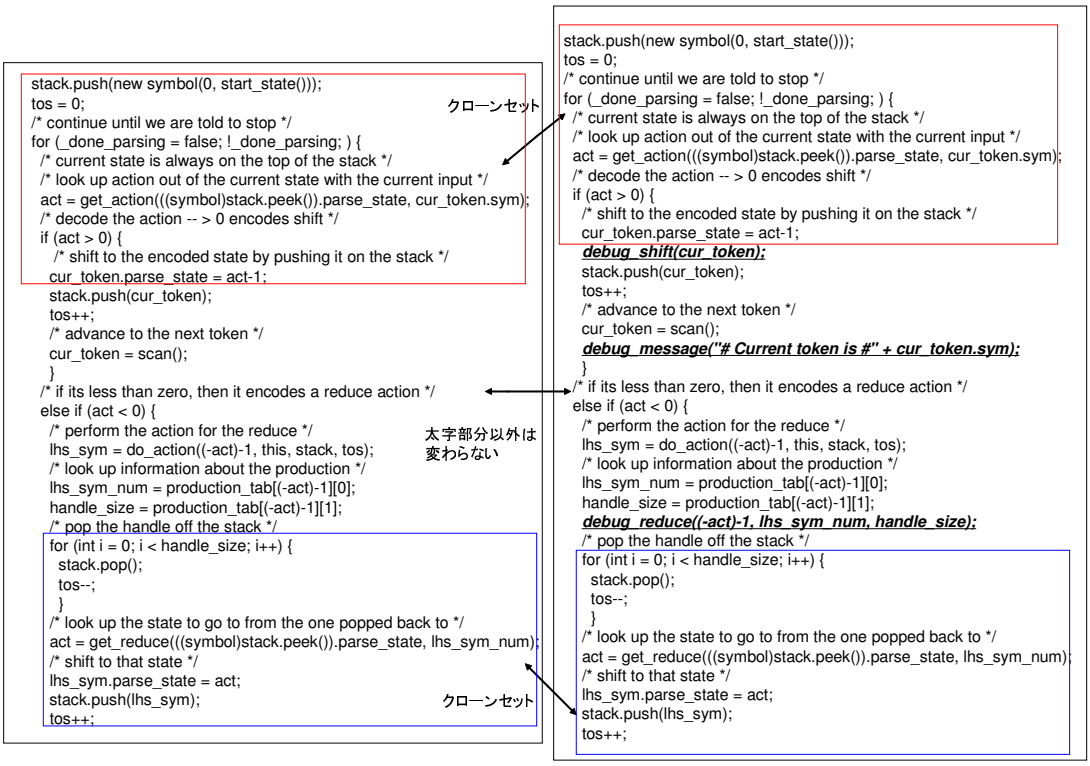

図 16: 編集が加えられていたクローンセットの例 2

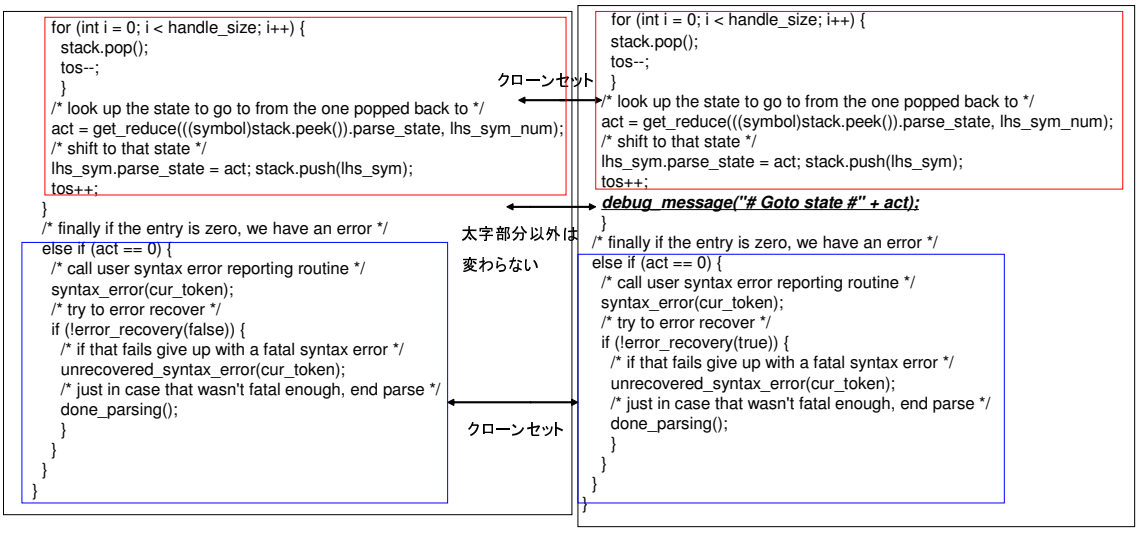

図 17: 編集が加えられていたクローンセットの例 3

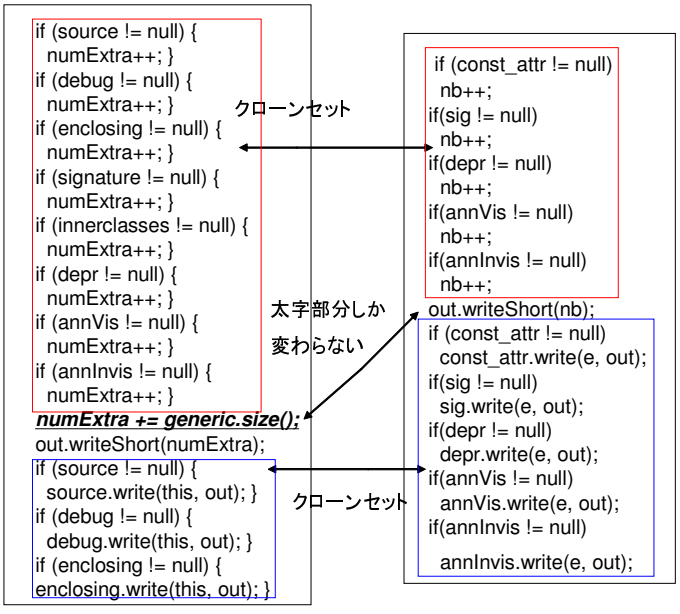

図 18: 編集が加えられていたクローンセットの例 4