修士学位論文

題目

ギャップを含むコードクローンのフィルタリング手法の提案

指導教員
井上 克郎 教授

報告者
宮崎 宏海

平成 21 年 2 月 9 日

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ソフトウェア工学講座

平成 20 年度 修士学位論文

ギャップを含むコードクローンのフィルタリング手法の提案

宮崎 宏海

内容梗概

ソフトウェアの開発や保守を困難にする要因として，コードクローンが挙げられる．コードクローンとは，ソースコード中に存在するコード片で類似した (もしくは同一の) コード片を持つものを指し，主にコピーアンドペーストによる再利用や定型処理の記述によって生成される．もしあるコード片がコードクローンの場合，その対応する全てのコードクローンに対しても修正を検討する必要が生じる．

コードクローンの中でも不一致部分を含むものを Gapped クローンと呼ぶ．Gapped クローンを検出する手法に，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ものがある．この手法を用いることでソースコードの集合から効率的に Gapped クローンを検出できる．しかし，Gapped クローンとして調査の対象とする必要のないものが多く存在していた．

そこで，本研究では不要な Gapped クローンの基準を定義し，フィルタリングする手法を提案する．具体的には，グラフ構築時に特定のノード間にエッジを引かないことで，調査の必要がない Gapped クローンを生成しないようにした．調査の必要がない Gapped クローンをフィルタリングすることで，コードクローン分析の効率化が図れる．そして，手法を実装してフィルタリングが妥当であるかを確認した．

主な用語

コードクローン (Code Clone)
Gapped クローン (Gapped Clone)
AGM アルゴリズム (AGM Algorithm)
フィルタリング (Filtering)

目 次

1　まえがき

近年，ソフトウェア開発や保守の支援方法としてコードクローン検出技術が注目を浴びている．コードクローンとは，ソースコード中に存在するコード片で類似した (もしくは同一の) コード片を持つものを指し [2, 9]，主にコピーアンドペーストによる再利用や定型処理を記述することで生成される [10].

コードクローンによってソフトウェア開発や保守が困難になるといわれている．例えば，あるコード片に修正が生じた場合，そのコードクローン全てに対しても同一の修正を検討する必要があるが，大規模なソフトウェアに対してこのような修正を行うことは極めて多大なコストがかかる．そのため，コードクローンの検出やコードクローンを対象とした集約支援などの研究はソフトウェアの開発や支援に対して非常に有用であるといえる．

これまでにコードクローンに関して様々な研究が行われてきたが [1, 2, 5, 9]，それらの研究で対象とされているコードクローンの定義はそれぞれ異なっている．つまり，コードクローンには厳密で普遍的な定義はないが，Bellon はコードクローン間の違いの度合いにもとづいて，コードクローンをタイプ 1(Exact クローン)，タイプ 2(Parameterized クローン)，タイプ 3(Gapped クローン) という 3 つのタイプに分類した [3]．このうち，Exact クローンとは空白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し，Parameterized クローンとはユーザ定義名や一部の予約語が異なるコードクローンのことを指す．そして，Gapped クローンとは文単位の不一致部分を含むコードクローンのことを指す．コピーアンドペーストによって既存のコード片を再利用する場合，コード片をそのまま用いるよりも，ペースト先の文脈に合わせた修正が加えられることが多い．その結果，コピー元との不一致部分が生じ，その前後がそれぞれ異なるコードクローンとして検出されてしまう．一般に，短いコードクローンほどソフトウェア中に多く存在するため，不一致部分により分割されたコードクローンを検出しようとすると，不必要なコードクローンも多く検出してしまう．不必要なコードクローンが多く存在する場合，目的とするコードクローンを見つけるために要する分析者の労力や分析に要する時間が増加してしまう．

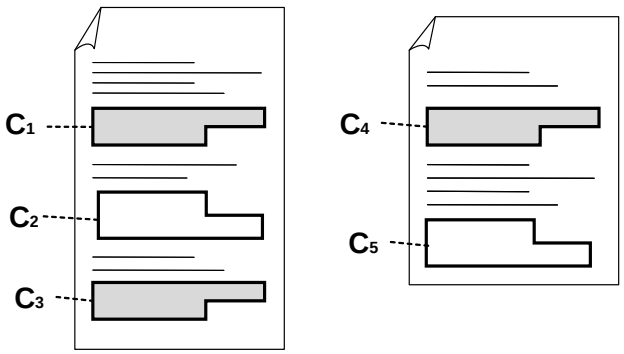
Gapped クローンを検出する手法として，藤野らが提案した AGM アルゴリズムを用いた手法がある [23]．AGM アルゴリズムとはグラフマイニングアルゴリズムの 1 つで，ラベル付きグラフから多頻度部分グラフパターンを効率良く取得することが出来る [20]．藤野らはこのアルゴリズムを用いて Gapped クローンを効率的に生成している．しかし，この手法で生成した Gapped クローンの中には生成要因がコピーアンドペーストではないものも多く含まれていた．

そこで，本研究では藤野らの手法を用いて生成した Gapped クローンを調査することで，不要な Gapped クローンの基準を定義した．定義した基準には生成要因がコピーアンドペー

ストではない Gapped クローンを表すものや，ソフトウェア保守の観点から考えて不要と考えられる Gapped クローンを表すものが含まれている．そして，グラフの構築時に特定のノード間にエッジを引かないことで，定義した基準を満たす Gapped クローンを効率的にフィルタリングする手法を提案する．

また，本手法を実装したシステムを作成し，複数の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．適用実験ではフィルタリングの妥当性を定量的に評価している．そして，フィルタリング後に得られた Gapped クローンについても調査を行った．

以降，2 章ではコードクローンや Gapped クローンについて説明し，3 章では AGM アルゴリズムと藤野らが用いた Gapped クローン検出法について述べる．4 章では藤野らの手法を用いて行った調査と手法の問題点を述べ，5 章では提案手法の説明を行う．6 章では提案手法を実装したシステムを用いて行った適用事例について，結果と考察を述べる．そして最後に 7 章でまとめと今後の課題を記す．



クローンペア	クローンセット
(C1, C3)	{C1, C3, C4}
(C1, C4)	{C2, C5}
(C3, C4)	
(C2, C5)	

図 1: クローンセットとクローンペア

2 準備

2.1 コードクローン

コードクローンとはソースコード中に存在するコード片で類似した (もしくは同一の) コード片を持つものを指す [19].

2.1.1 クローンセットとクローンペア

コードクローンを表現する単語としてクローンセットとクローンペアがある [9].

あるソースコード中に存在する 2 つのコード片 α，β が同一または類似しているとき，$C(\alpha, \beta)$ と書き，α は β とクローン関係をもつという．C は反射律，推移律，対称律が成り立つ同値関係である．クローン関係によって作られるコードクローンの同値類をクローンセットという．

任意の α，β に対して $C(\alpha, \beta)$ ならば，α の任意の部分系列 $\dot{\alpha}$ に対し，$C(\dot{\alpha}, \dot{\beta})$ となる β の部分系列 $\dot{\beta}$ が存在する．また，α，β をそれぞれ真に含む任意の系列 $\ddot{\alpha}$，$\ddot{\beta}$ (ただし $\ddot{\alpha} \neq \ddot{\beta}$) に対して $C(\alpha, \beta)$ かつ $\neg\, C(\ddot{\alpha}, \ddot{\beta})$ ならば，(α, β) をクローンペアという．

図 1 では 5 つのコード片 C_1〜C_5 が存在しており，同形のコード片はそれぞれ互いに同一もしくは類似している．この場合，2 つのクローンセット { C_1,C_3,C_4 } { C_2,C_5 } と，4 つのクローンペア $(C_1, C_3)(C_1, C_4)(C_3, C_4)(C_2, C_5)$ が存在する．

2.1.2 コードクローンの発生要因

コードクローンの発生要因として以下のものが挙げられる [21].

コピーアンドペーストによる既存のコードの再利用 正常に動作する既存のソースコードは信頼性が高いことから，流用して部分的に変更を加えるというコードの再利用は多く存在する．

定型処理 定義上簡単で頻繁に用いられる処理はコードクローンになる傾向がある．例えば，ファイルの入出力処理やデータ構造へのアクセスなどが挙げられる．

プログラミング言語の仕様 言語の仕様によって抽象データ型やローカル変数を用いることが出来ない場合，同じアルゴリズムを持つ処理を繰り返し書かなくてはならない場合がある．

パフォーマンスの改善 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記述することで，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 処理目的が同じである場合，コード生成ツールによって生成されたコードは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．

複数のプラットフォームへの対応 ソフトウェアを複数の OS(Windows，Macintosh，Linux，FreeBSD など) や CPU(i386 系，amd64 系など) に対応させる場合，各プラットフォーム用の処理の部分に類似コードが存在することが多い．

偶然 開発者が偶然類似コードを書いてしまうことがある．

2.1.3 コードクローンの分類

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，各手法によってコードクローンの定義が異なる．Bellon は，コードクローン間の違いの度合に基づいて，それらを以下の 3 つに分類した [3].

タイプ 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．このようなコードクローンを以降 Exact クローンと呼称する．

タイプ 2 変数名や関数名などのユーザ定義名，また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．このようなコードクローンを以降 Parameterized クローンと呼称する．

タイプ 3 コピーアンドペースト後に，文の挿入や削除，修正が行われた結果，不一致部分 (Gap) を含んでいるコードクローンを指す．このようなコードクローンを以降 Gapped クローンと呼称する．

図 2 はコピーアンドペーストによってコードが再利用される流れを表したものである．単にコピーアンドペーストした場合は，コピー元とペースト先は Exact クローンとなる．コピーアンドペースト後に識別子の変更が行われた場合は，Parameterized クローンとなる．さらに，ペースト先に文単位で修正を加えた場合は Gapped クローンとなる．

2.1.4 コードクローンの検出

これまでに提案されているコードクローン検出技術は，以下の 5 種類に分類できる．

- 行単位の検出
- 字句単位の検出
- 抽象構文木を用いた検出
- プログラム依存グラフを用いた検出
- メトリクスやバースマークなど，その他の手法を用いた検出

行単位で検出する手法では，ソースコードを行単位で比較し，閾値以上の行数連続して一致している部分を重複コードとして検出する．この検出手法は他の検出技術に比べて高速であるという特徴を持つが，コーディングスタイルが異なる場合は同じ処理であってもコードクローンとして検出できないという欠点がある．Johnson や Ducasse らは，各行に含まれる空白やタブを取り除いた後にコードの比較を行っている [5, 6, 8].

字句単位で検出する手法では，まずソースコードを字句の列に変換し，閾値以上の字句が連続して一致していればその部分を重複コードとして検出する．この検出手法は，行単位で検出する手法と違いコーディングスタイルに依存せず，重複コードを検出できる．私が所属する研究グループでは，字句単位でコードクローンを検出するツール CCFinder を開発している [9]．CCFinder では検出処理を行う前にユーザ定義名に変化する処理を行っており，C や C++，Java，COBOL，FORTRAN など複数のプログラミング言語に対応している．

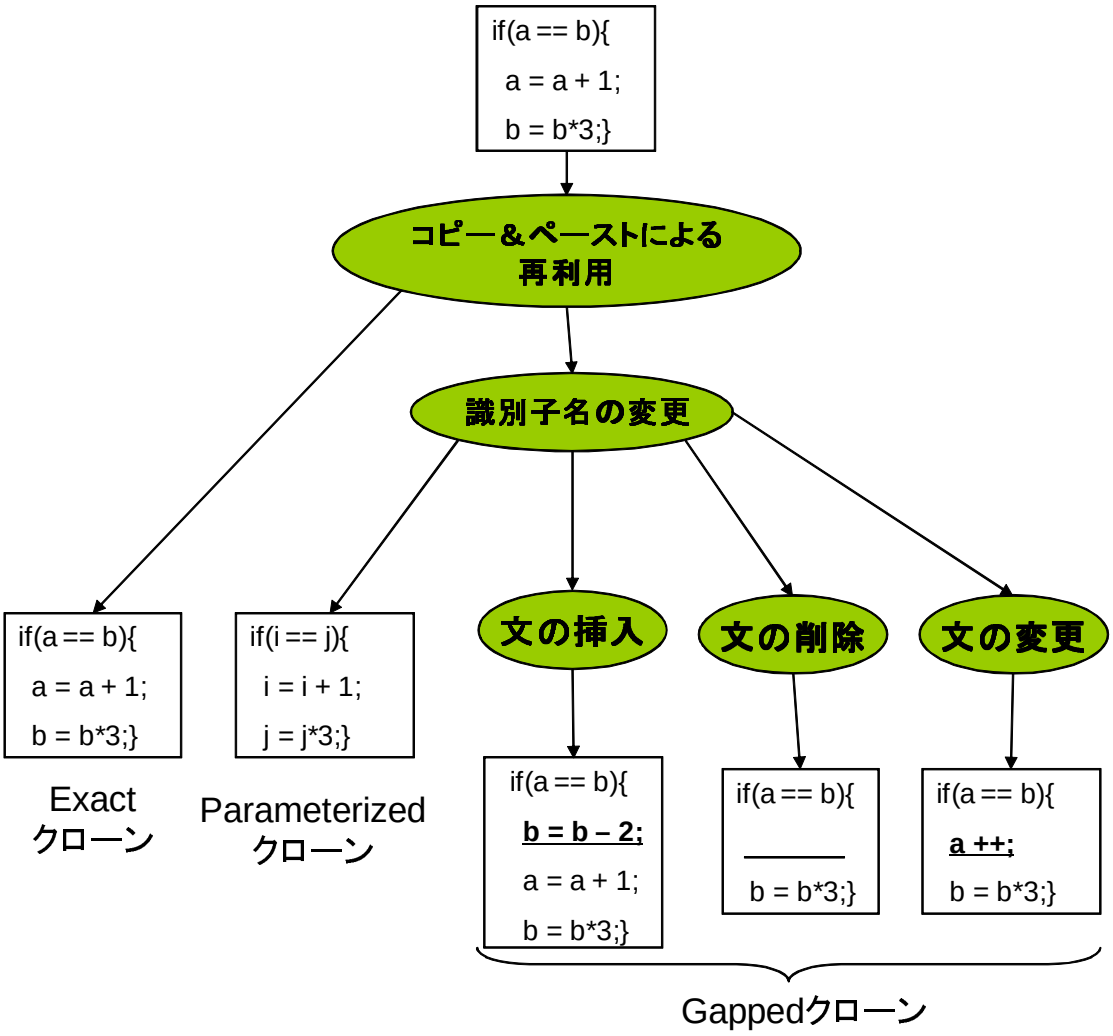


図 2: コピーアンドペーストによる再利用の流れ

抽象構文木を用いた検出手法では，検出の前処理として構文解析を行い，抽象構文木を構築する．そして抽象構文木上の同形の部分木がコードクローンとして検出される．この検出手法は，事前処理として抽象構文木を作成する必要があるため検出にかかるコストは高くなるが，プログラムの構造を無視したコードクローンは検出されないという特徴を持つ．Koschke らや Jiang らは抽象構文木を直列表現もしくは配列表現に変換した上でコードクローンを検出する手法を提案している [7, 14]．また，抽象構文木による検出手法を実装したツールとしては Baxter らの CloneDR [4] がある．

プログラム依存グラフを用いた検出手法では，検出の前処理としてソースコードの意味解析を行い，ソースコードの要素間 (文や式など) の依存関係が抽出され，要素を頂点，依存関係を有向辺とするグラフ (プログラム依存グラフ) を構築する．そして，グラフ上の同形の部分グラフをコードクローンとして検出する．この検出手法では，ソースコード上で順序が入れ替わっていても意味的に同一であるコード片をコードクローンとして検出できる (5.2 節の巻き付きコードクローンなど)．ただし，プログラム依存グラフの構築には高い計算コストがかかるため，大規模ソフトウェアに適用するのは現実的ではない．Krinke や Komondoor らはこの手法を用いてコードクローン検出を行っている [11, 15].

メトリクスを用いる手法では，プログラムのモジュール (ファイル，クラス，メソッドなど) に対してメトリクスを計測し，メトリクスが近似しているモジュールをコードクローンとして検出する．Mayrand らは関数に対してメトリクスを定義し，Kontogiannis は抽象構文木の部分木に対してメトリクスを定義している [12, 13, 17]．また，Ottenstein によってプログラムの固有の特徴を表すバースマークを用いた手法も行われている [18].

2.2 Gapped クローンの検出

既存のコードクローン検出手法のうち，CP-Miner など一部の手法では Gapped クローンを検出できるが [7, 16]，その他の多くの手法では Gapped クローンを検出することが出来ない．検出するコードクローンの大きさの閾値を下げることにより，1 つの Gapped クローンを複数の Exact クローンと Parameterized クローンとして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 1 つの Gapped 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．

そこで，藤野らは Exact クローンと Parameterized クローンの情報，および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て Gapped クローンを生成する手法を提案した [23]．この手法は，コードクローンの検出単位 (トークン単位や行単位など) に依存していないため，任意のコードクローン検出手法に適用可能である．コードクローン検出後の後処理としてこの手法を用いることで，分析者はより有益なコードクローン情報を得ることが可能になる．

次章では藤野らが提案した手法について説明する.

3　AGM アルゴリズムを用いた Gapped クローン検出手法

本章では，藤野らが提案した AGM アルゴリズムを用いた Gapped クローン検出手法について説明する．

3.1　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 [20]．AGM アルゴリズムの概要を以下に示す．

1)　$Main(G, minsup)$ {
2)　　$C_1 \leftarrow$ { 大きさ 1 の多頻度グラフの候補の隣接行列 };
3)　　$k \leftarrow 1$;
4)　　$while(C_k \neq \emptyset)$ {
5)　　　$Count(G, C_k)$;
6)　　　$F_k \leftarrow \{\ c_k \in C_k | sup(G(c_k)) \geqq minsup\ \}$;
7)　　　$C_{k+1} \leftarrow Generate - Candidate(F_k)$;
8)　　　$k \leftarrow k + 1$;
9)　　}
10)　　$return \cup_k \{\ f_k \in F_k | f_k \quad is \quad canonical\ \}$;
11)　}

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 個のノード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．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 アルゴリズムのおおまかな流れを示す．

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，支持度が $minsup$ 以上であれば，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　全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，さまざま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．

藤野らが提案した手法では，グラフのパスを抽出する条件を用いて，グラフのノードを Exact クローンや Parameterized クローン，グラフのエッジ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している．

3.2 Gapped クローン検出手法

以下の手順で，Gapped クローンの情報を生成する．

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 (=Gapped クローン) の検出

手順 3 生成した Gapped クローン情報の出力

手順 1 では，入力として与えられた Exact クローンおよび Parameterized クローンの情報をもとに，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 Gapped クローンの情報として出力する．以下で，各手順の詳細を述べる．

3.2.1 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 Exact クローンおよび Parameterized クローンの情報から，ソースファイル毎にコードクローンをノードとするグラフを作成する．このノードにはラベルが付けられており，同一クローンセットに含まれるコードクローンを表すノードは同一ラベルを持つ．ノードに対応する 2 つのコードクローンがオーバーラップしていない場合，それらの間にエッジを引く．エッジはファイル内における位置が上位のコードクローンから下位のコードクローンに向けて引かれる．

以上の操作を全てのコードクローンの組み合わせに対して実行し，ソースファイル毎にグラフを構築する．もし，一部がオーバーラップしていたり，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 2 つのノード間にはエッジを引かない．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コードクローンである可能性が低いからである．また，コードクローン間の距離に閾値を設定し，2 つのコードク

ローン間の距離が閾値以上離れている場合もそれらのノード間にはエッジを引かない．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可能性が低いからである．

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,……,f_n から，3 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,……,a_x，クローンセット B には y 個のコード片 b_1,b_2,……,b_y，クローンセット C には z 個のコード片 c_1,c_2,……,c_z が含まれている．

図 3(a) は，ファイル f_1 と f_2 に含まれるコードクローンの様子を表している．ファイル f_1 中のコード片 a_1 と a_2 および a_1 と b_2，ファイル f_2 中のコード片 a_3 と b_2 および a_3 と c_1 は離れて存在しているので，対応するノード間にエッジを引く．それに対して，ファイル f_1 中のコード片 a_2 と b_1 は，一部が重複しているので，これらのノード間にはエッジを引かない (図 3(b) 参照)．

3.2.2 手順 2: 多頻度グラフパターンの検出

手順 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) 出現する部分グラフを Gapped 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，図 4 のようにファイル f_1 のパス a_1 b_1 とファイル f_2 のパス a_3 b_2 が Gapped クローンとして検出される．

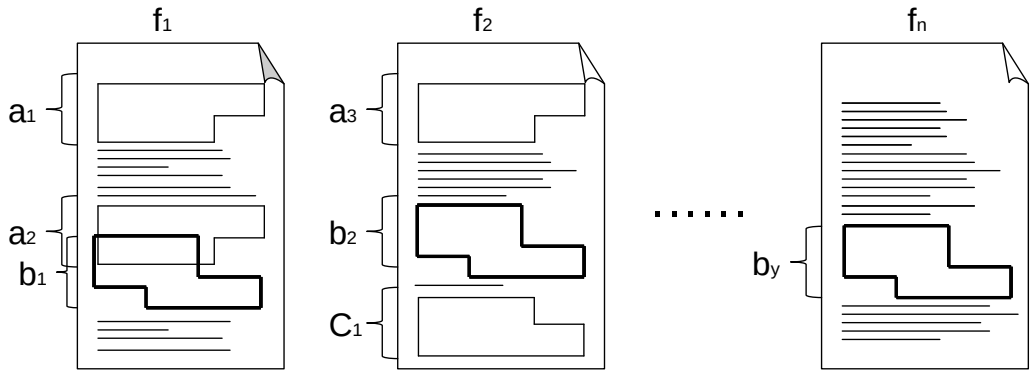
3.2.3 手順 3: 生成した Gapped クローン情報の出力

手順 2 で検出した Gapped クローン情報を，入力として与えられたコードクローン情報と同様のフォーマットで出力する．

3.3 手法の問題点

藤野らが提案した手法を用いることで効率的に Gapped クローンを検出することができる．しかし，検出した Gapped クローンの中には生成要因がコピーアンドペーストではないものや，ソフトウェア保守の観点から調査する必要がないものなど，分析を行う上で不要と考えられるものも多く含まれていた．

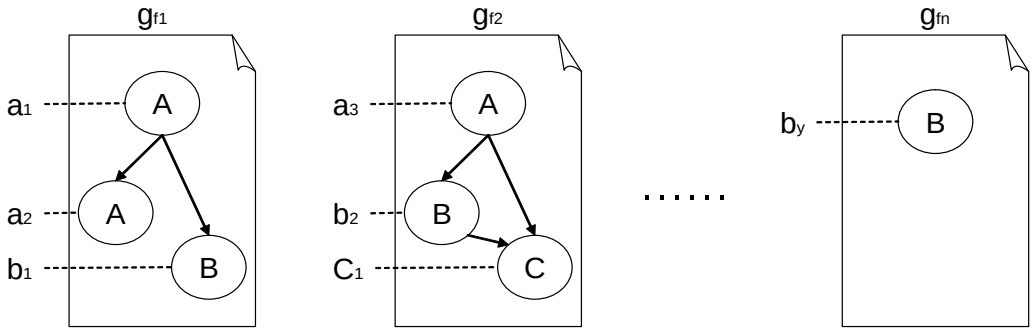
次章では，実際に手法を用いて検出した Gapped クローンの調査結果について述べる．



コードクローン検出対象ファイル: {f_1, f_2, ・・・, f_n}

クローンセットA: {a_1, a_2, ・・・, a_x}
クローンセットB: {b_1, b_2, ・・・, b_y}
クローンセットC: {c_1, c_2, ・・・, c_z}

(a) 対象ファイル中のコードクローン

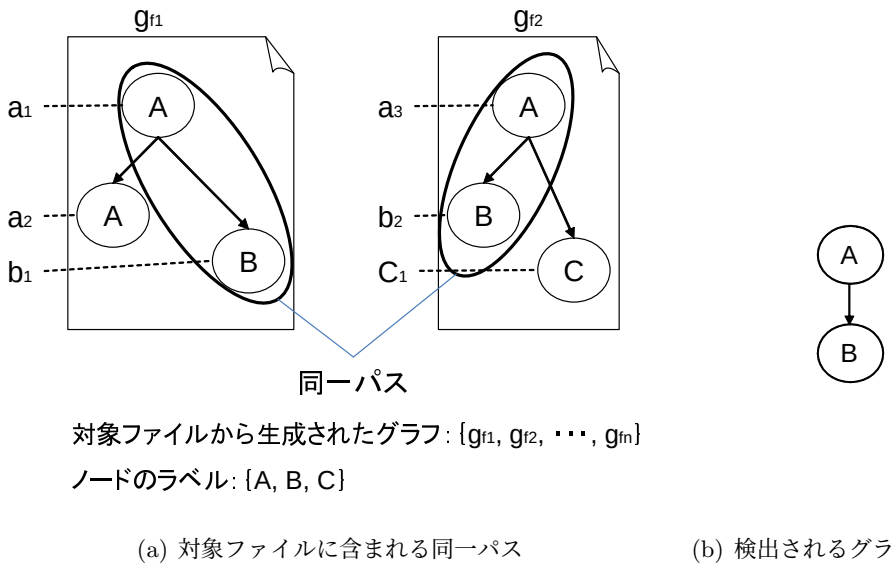


対象ファイルから生成されたグラフ: {g_{f1}, g_{f2}, ・・・, g_{fn}}

ノードのラベル: {A, B, C}

(b) 生成されるグラフ

図 3: グラフ構造の生成



(a) 対象ファイルに含まれる同一パス　　(b) 検出されるグラフパターン

図 4: 多頻度グラフパターンの検出

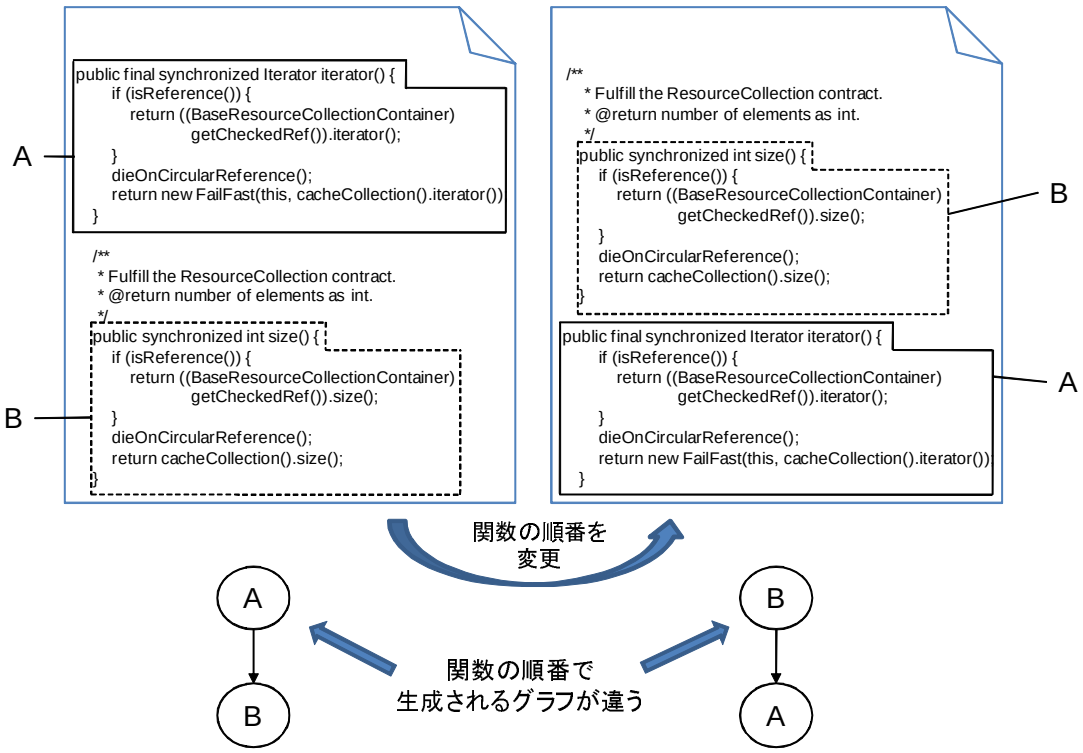


図 5: 複数関数にまたがる Gapped クローン

4 藤野らが提案した手法により取得した Gapped クローンの調査

本章では，藤野らが提案した手法を複数のオープンソースソフトウェアに対して適用し，生成した Gapped クローンについて調査を行った．

4.1 実験概要

実験を行った環境，使用したツール，対象としたソフトウェアについて説明する．
本実験は以下の環境で行った．

- CPU: Pentium4 3.20GHz
- メモリ: 2GB
- OS: Windows Vista

また，本実験ではコードクローンの検出に CCFinder を用いた [9]．CCFinder で検出するコードクローンの最小トークン数は 30 トークンとし，Gapped クローンはコードクローン間が 30 トークン以上離れていないものを取得した．

対象とするソフトウェアは Canna，http，Ant，JHotDraw，JDK の 5 つのオープンソースソフトウェアである．Canna はクライアントサーバ方式の日本語入力システム，httpd は Unix 上で動作する Web アプリケーションである．また，Ant はメイクツール，JHotDraw は 2 次元のグラフィックフレームワーク，JDK はプログラミングツール群である．各ソフトウェアのファイル数や総行数は表 4.1 の通りである．

4.2 評価の基準

調査の際，Gapped クローンを以下の 4 つの基準のいずれかに当てはめてその個数を調べた．なお，複数の基準を満たすものも存在する．

基準 1 Gapped クローンを構成するコード片が複数の関数やメソッドにまたがっているもの

基準 2 同じ Gapped クローンセットに属するインスタンス同士がオーバーラップしているもの

基準 3 ソフトウェア開発や保守の観点から調査の必要がないもの

基準 4 基準 1〜3 のいずれにも属さないもの

上記の基準についてそれぞれ説明する．

基準 1 については，Gapped クローンが 1 つの関数内で閉じている場合，関数内で処理が異なっている部分がギャップとなっており，関数内の処理が違う部分を検出することは有益であると考えられる．一方，Gapped クローンが複数の関数にまたがっている場合，関数の外部でギャップになっている部分があることを意味している．複数の関数をまとめてコピーアンドペーストして修正を加えた場合にこのような Gapped クローンが生成されるが，図 5 に示すように関数は記述の順序によって処理に影響はないため，複数関数にまたがる Gapped クローンを検出する必要はないと考えられる．

表 1: 対象ソフトウェア

ソフトウェア	バージョン	記述言語	ファイル数	総行数
Canna	3.6	C	136	100,146
httpd	2.2.11	C	803	333,258
Ant	1.7.1	Java	787	200,401
JHotDraw	7.0.9	Java	471	90,214
JDK	1.5.0	Java	6,557	1,887,008

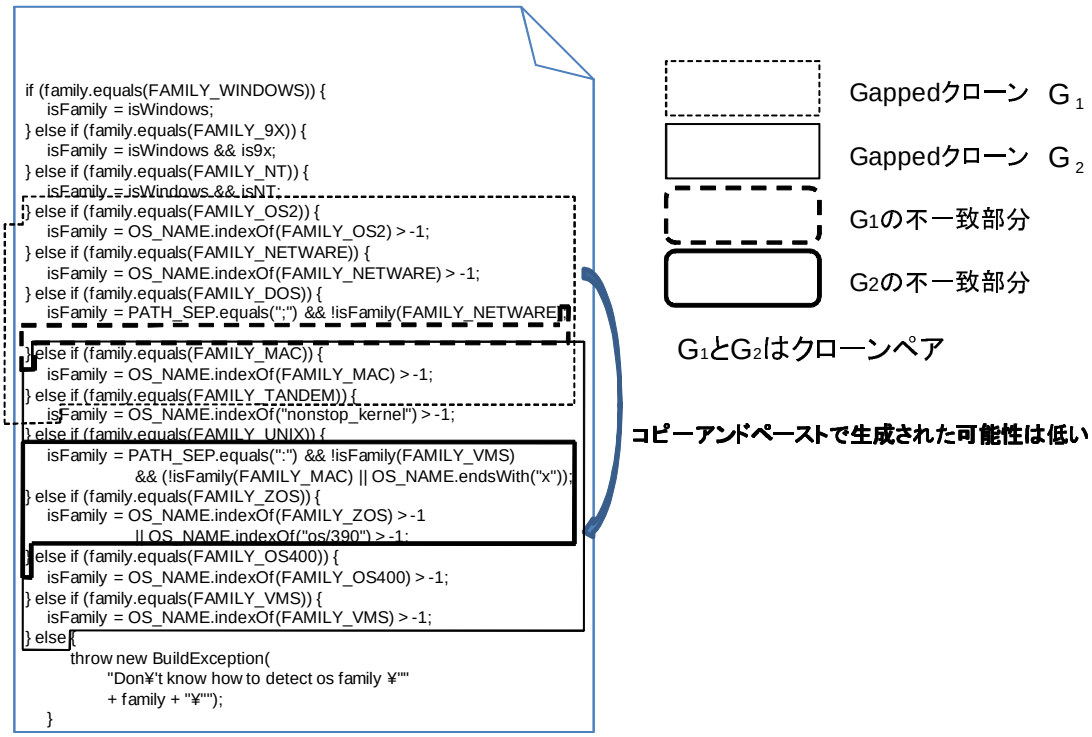
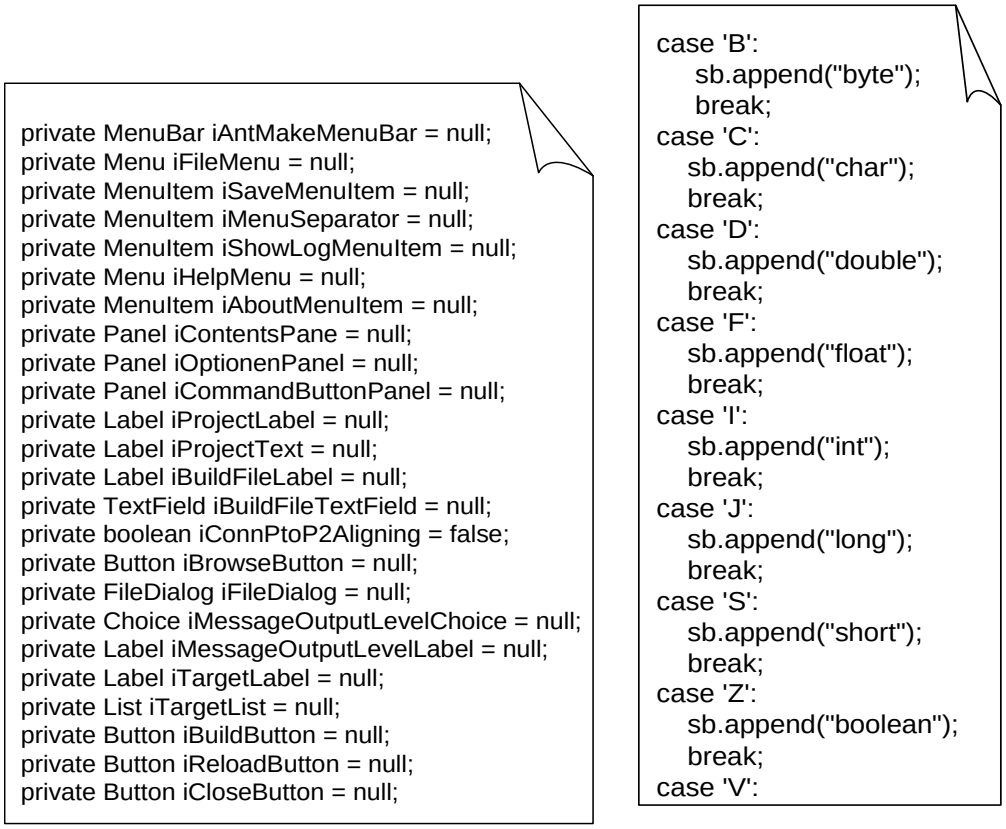


図 6: インスタンス同士がオーバーラップしている Gapped クローン



```
private MenuBar iAntMakeMenuBar = null;
private Menu iFileMenu = null;
private MenuItem iSaveMenuItem = null;
private MenuItem iMenuSeparator = null;
private MenuItem iShowLogMenuItem = null;
private Menu iHelpMenu = null;
private MenuItem iAboutMenuItem = null;
private Panel iContentsPane = null;
private Panel iOptionenPanel = null;
private Panel iCommandButtonPanel = null;
private Label iProjectLabel = null;
private Label iProjectText = null;
private Label iBuildFileLabel = null;
private TextField iBuildFileTextField = null;
private boolean iConnPtoP2Aligning = false;
private Button iBrowseButton = null;
private FileDialog iFileDialog = null;
private Choice iMessageOutputLevelChoice = null;
private Label iMessageOutputLevelLabel = null;
private Label iTargetLabel = null;
private List iTargetList = null;
private Button iBuildButton = null;
private Button iReloadButton = null;
private Button iCloseButton = null;
```

(a) 連続した変数宣言

```
case 'B':
    sb.append("byte");
    break;
case 'C':
   sb.append("char");
   break;
case 'D':
   sb.append("double");
   break;
case 'F':
   sb.append("float");
   break;
case 'I':
   sb.append("int");
   break;
case 'J':
   sb.append("long");
   break;
case 'S':
   sb.append("short");
   break;
case 'Z':
   sb.append("boolean");
   break;
case 'V':
```

(b) 連続した case 文

図 7: 繰り返し構造を持つコード片

基準 2 については，Gapped クローンのインスタンス同士がオーバーラップしている場合，一方がもう一方をコピーアンドペーストして作成された可能性は低い (図 6 参照)．そのため，このような Gapped クローンを検出する意味はないと考えられる．

基準 3 については，コードクローンの中にはソフトウェア開発や保守を行う視点から調査の対象にする必要がないものも存在する [22]．例えば，図 7 連続した変数宣言や switch 文の連続した case 文などで，これらは繰り返し構造を持っているものが多い．

本研究では，基準 1〜基準 3 のいずれにも属さない基準 4 の Gapped クローンをソフトウェアの開発や保守をする上で有用な Gapped クローンとした．

4.3 実験内容

3 つのオープンソースソフトウェア，Ant，JHotDraw，JDK に対して手法を適用し，4.2 節に記した基準に当てはまる Gapped クローンの個数を調べた．実験の結果を表 2 に示す．

例えば，表 2 に示す通り Ant からは 650 個の Gapped クローンセットを検出した．そのうち複数関数にまたがっているもの (基準 1) は 340 個，インスタンス同士がオーバーラップしているもの (基準 2) は 445 個，調査の必要がないもの (基準 3) は 380 個それぞれ検出した．そして，基準 1〜基準 3 のいずれにも属さないものは全体の 8.3 %である 54 個であった．

同様にその他のソフトウェアについても調査を行った結果，基準 1〜基準 3 に属する Gapped クローンによって有用である基準 4 に属する Gapped クローンの割合が大きく下げられていることが分かった．そのため，基準 1〜基準 3 に属する Gapped クローンをフィルタリングする手法を次章で提案する．

表 2: 検出された Gapped クローンの内訳

ソフトウェア	合計	基準 1	基準 2	基準 3	基準 4
Canna	1,318	339(25.7 %)	910(69.0 %)	1,035(78.5 %)	106(8.0 %)
httpd	4,965	893(18.0 %)	4,393(88.5 %)	4,366(87.9 %)	173(3.5 %)
Ant	664	369(55.6 %)	455(68.5 %)	393(59.2 %)	54(8.1 %)
JHotDraw	599	281(46.9 %)	427(71.3 %)	400(66.8 %)	61(10.2 %)
JDK	15,433	8,204(53.2 %)	11,444(74.2 %)	11,764(76.2 %)	929(6.0 %)

5　提案手法

本章では 4 章で紹介した基準 1〜基準 3 に該当する Gapped クローンをフィルタリングする手法を紹介する．

5.1　複数の関数にまたがる Gapped クローンのフィルタリング

Exact クローンと Parameterized クローンが含まれる関数を事前に求めておき，グラフ作成時に含まれる関数が異なるノード間にはエッジを引かなければ，複数の関数にまたがった Gapped クローンは生成されない．図 8 はフィルタリングの様子を表したものである．

図 8(a) では，コード片 1〜3 が関数 A に，コード片 4 が関数 B にそれぞれ含まれている．フィルタリングを行わない場合，図 8(a) にあるグラフが生成される．一方，フィルタリングを行った場合は関数 A に含まれるコード片 1〜3 からコード片 4 へのエッジが引かれなくなり，図 8(b) に示すグラフが生成される．

5.2　インスタンス同士がオーバーラップする Gapped クローンのフィルタリング

同一クローンセットに属するコード片を表すノード間にエッジを引かないようにすることで，同じ Gapped クローンセットに属するインスタンス同士のオーバーラップをある程度フィルタリングできる．図 9 はフィルタリングの様子を表したものである．

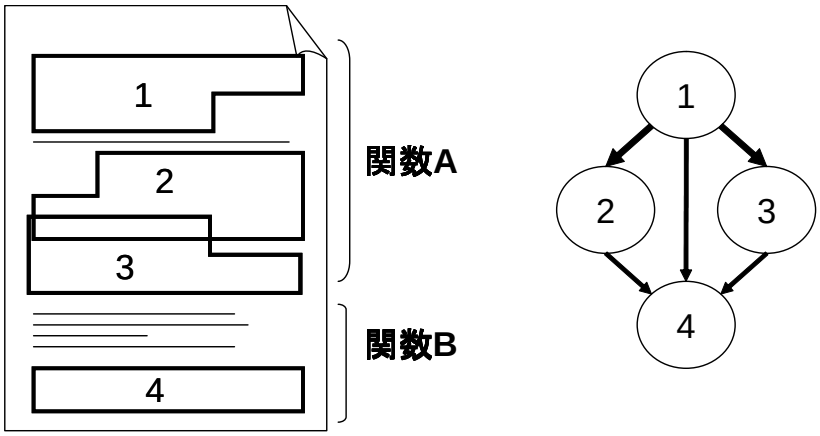
図 9(a) では，クローンセット 1 に属するコード片が 3 つ，クローンセット 2 に属するコード片が 1 つ含まれている．フィルタリングを行わない場合，図 9(a) にあるグラフが生成される．一方，フィルタリングを行った場合は，図 9(b) のようなグラフが生成される．

この手法の場合，同一クローンセットに属するコード片が連続していないがオーバーラップしている Gapped クローンをフィルタリングすることができない．しかし，このような Gapped クローンは巻き付きコードクローンの一種であると考えられるため，実際に不要であるかどうかは検討の必要がある．巻き付きコードクローンとは，図 10 に示すααとββの部分のように互いに巻き付き合っているコードクローンのことをいう [11]．

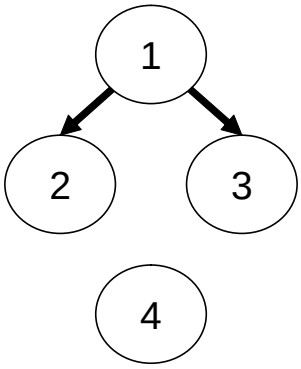
5.3　調査の必要がない Gapped クローンのフィルタリング

図 7(a) や図 7(b) のようにソフトウェア保守の観点から調査の必要がないコードクローンは繰り返し構造をしている傾向が強いことが分かっており，繰り返し構造の度合を表すためのメトリクスとして RNR(5.3.1 項参照) が定義されている [22]．

そこで，RNR が 0.5 未満であるコードクローンをグラフのノードとして用いないことで調査の必要がない Gapped クローンのフィルタリングを行う．

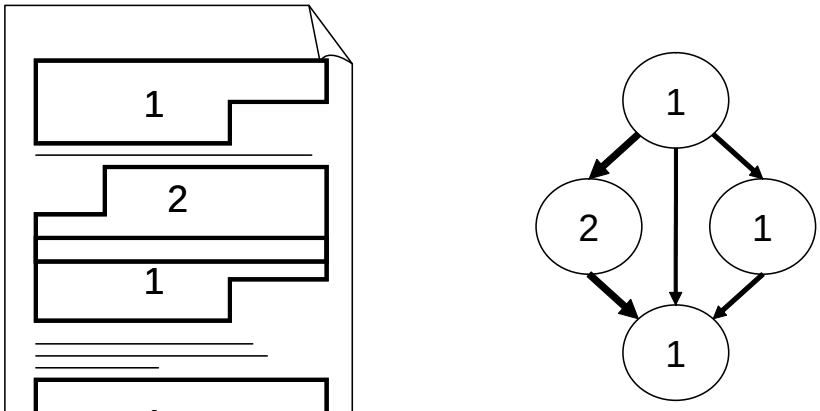


(a) 対象ファイル中の関数およびコードクローンと生成されるグラフ

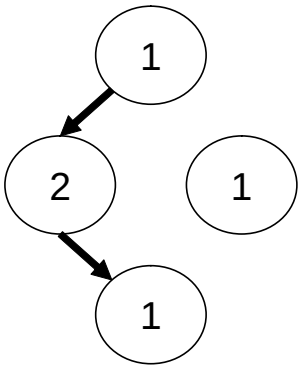


(b) フィルタリング後に取得するグラフ

図 8: 複数の関数にまたがる Gapped クローンのフィルタリング



(a) 対象ファイル中のコードクローンと生成されるグラフ



(b) フィルタリング後に取得するグラフ

図 9: インスタンス同士がオーバーラップする Gapped クローンのフィルタリング

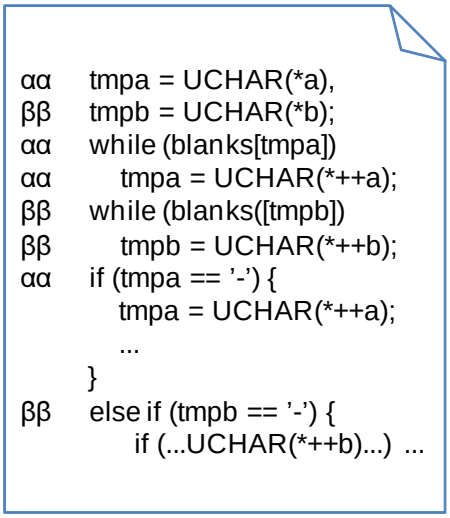


図 10: 巻き付きコードクローン

5.3.1 メトリクス RNR

RNR(Ratio of Non-Repeated elements) とは以下の式 (S はクローンセット) で求められる値で，値が小さいコードクローンは調査の必要がないコードクローンであるといえる [22].

$$RNR(S) = 1 - \frac{S \text{ の繰り返し部分の総トークン数}}{S \text{ の総トークン数}}$$

例えば，以下のようなトークンの並びを持つファイル F_1，F_2 があるとする．a,b,c,d,e はトークンを表しており，'*' は 1 つ前のトークンと同じトークンであることを表す.

F_1 a a^* a^* b c

F_2 a a^* d e

F_n の i 番目から j 番目のトークンで構成されるコード片を $C(F_n, i, j)$ とすると，この例では，$C(F_1, 1, 2)$，$C(F_1, 2, 3)$，$C(F_2, 1, 2)$ がそれぞれ aa となっており，互いにコードクローンになっている.

これらのコードクローンが属するクローンセットを S とすると，S のインスタンスである $C(F_1, 1, 2)$ および $C(F_2, 1, 2)$ には繰り返し部分であるトークンが 1 つ含まれており，$C(F_1, 2, 3)$ には 2 つ含まれている．これを上記の RNR の式に当てはめると，RNR(S) $=$ 0.33 となる.

6 実験

本章では，前述の提案手法を実装して行った実験について説明する．

6.1 目的

提案手法によってフィルタリングされる Gapped クローンを調査して，不要なものの数を調べることで手法の妥当性を確認する．以降，この実験を実験 1 と称する．また，フィルタリングを行う場合と行わない場合を比較し，フィルタリング後に取得した Gapped クローンについて調査を行うことで手法の有効性を確認する．以降，この実験を実験 2 と称する．

6.2 実験対象

実験対象にはオープンソースソフトウェアである Ant，JHotDraw，JDK を用いた．ツールがプロトタイプであり，Java にしか対応していないため，4 章の実験で用いた Canna と httpd は実験の対象外とした．それぞれのソフトウェアの概要を表 6.2 に記す．また，各ソフトウェアから得られたコードクローンの個数，フィルタリングせずに検出した Gapped クローンの個数，Gapped クローンの検出時間を表 4 に記す．

6.3 実験環境

本実験は 4 章の実験と同様に，以下の環境で行った．

- CPU: Pentium4 3.20GHz
- メモリ: 2GB
- OS: Windows Vista

また，コードクローン検出に用いるツールについても同様に CCFinder を用いた．それに加えて，コードクローンが含まれる関数を知る必要があるため，メトリクス計測プラグイン

表 3: 対象ソフトウェア

ソフトウェア	バージョン	記述言語	ファイル数	総行数
Ant	1.7.1	Java	787	200,401
JHotDraw	7.0.9	Java	471	90,214
JDK	1.5.0	Java	6,557	1,887,008

プラットフォーム MASU [24] を用いた．MASU を使うことで関数の位置情報を知ることができるため，コードクローンがどの関数に含まれるか知ることができる．

6.4 実験 1

6.4.1 評価方法

4.2 節で記述した通り，関数は記述の順序によって処理に影響はないため，複数関数にまたがる Gapped クローンは不要な Gapped クローンであるといえる．また，RNR が低いコードクローンはソフトウェア保守の観点で対象とする必要がないことから，RNR によるフィルタリングも妥当であるといえる．

しかし，同一クローンセットのラベルを持つノード間にエッジを引かない場合，インスタンス同士がオーバーラップしていない Gapped クローンもフィルタリングしてしまう．そのため，同一クローンセットのインスタンスで構成される Gapped クローンが不要であるかを調査する必要がある．この実験では，対象のソフトウェアに対して同じラベルを持つノード間にのみエッジを引いたグラフを生成し，各 Gapped クローンが有用であるかどうかを調べた．

有用であるかの判断は以下の基準で行う．基準 1〜基準 4 を満たしていない場合は有用な Gapped クローンであると判断した．

基準 1 複数の関数にまたがっている

基準 2 インスタンス同士が互いにオーバーラップしている

基準 3 メトリクス RNR が 0.5 未満である

基準 4 不一致部分に変更が加わっていない

基準 5 基準 1〜基準 4 のいずれも満たしていない

基準 1〜3 については，4 章で記述した通り，これらの条件を満たしている場合，Gapped クローンとして不要であると判断した．

表 4: 各ソフトウェアのコードクローン情報

ソフトウェア名	コードクローン数	Gapped クローン数	検出時間 (s)
Ant	998	664	20
JHotDraw	723	599	12
JDK	5,436	15,433	11,221

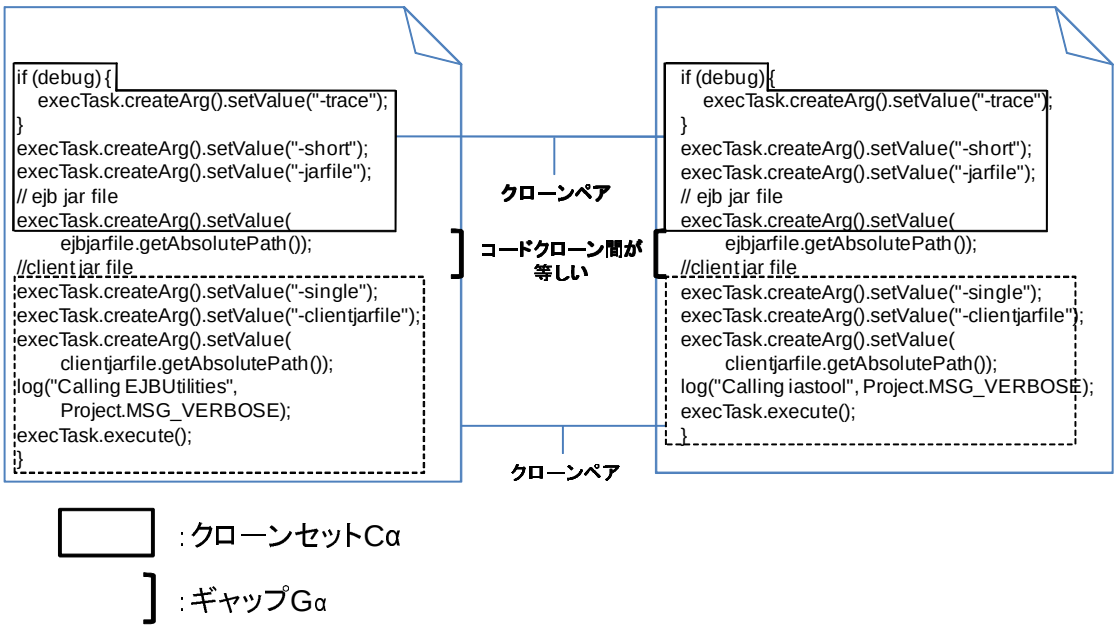


図 11: 不一致部分が同じコードクローン

基準 4 については，図 11 のように，コードクローン間の部分が完全に一致してしまう Gapped クローンを取得してしまうことがある．この Gapped クローンは，クローンセット C_{α} に属するコード片 c_n が他の箇所にも存在しており，かつ c_n に続くコードが G_{α} と一致または類似していない際に生成される．この Gapped クローンは 1 つのコードクローンとして検出することができるため，Gapped クローンとして不要であると判断した．

ソフトウェア毎に Gapped クローンがどの基準に当てはまるかを判断し，その数を調べた．なお，複数の基準を満たす Gapped クローンも存在する．

6.4.2 実験結果

同一クローンセットのみで構成された Gapped クローンの数と各基準に当てはまる Gapped クローンの数は表 5 のようになった．

基準を満たす Gapped クローンとして，インスタンス同士がオーバーラップする Gapped クローンは図 12 に挙げるものが存在した．また，有用な Gapped クローンのとして図 13 のコードが挙げられる．

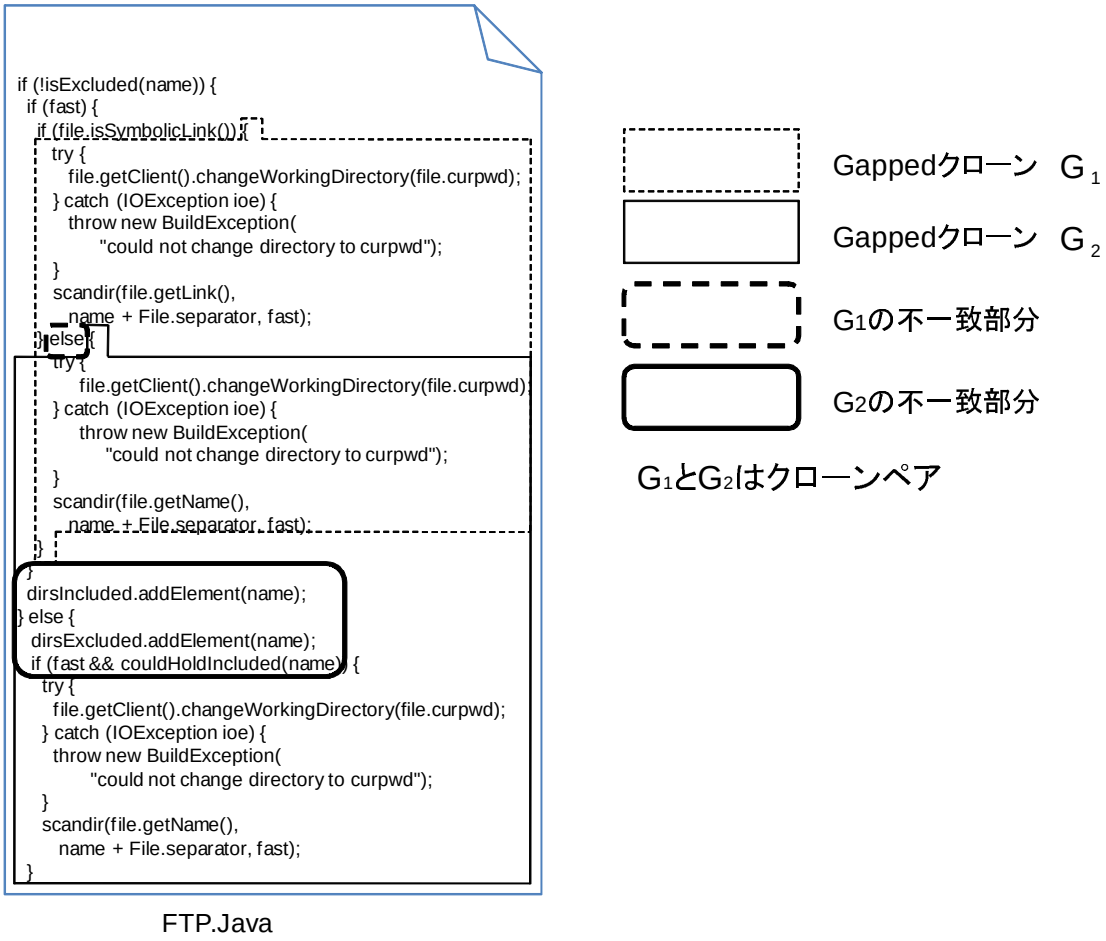


図 12: インスタンス同士がオーバーラップしている Gapped クローン (Ant)

表 5: 同一クローンセットのみで構成された Gapped クローン

ソフトウェア	総 Gapped クローン数	基準 1	基準 2	基準 3	基準 4	基準 5
Ant	39	27	34	17	1	1
JHotDraw	91	22	58	41	25	2
JDK	378	91	265	226	52	12

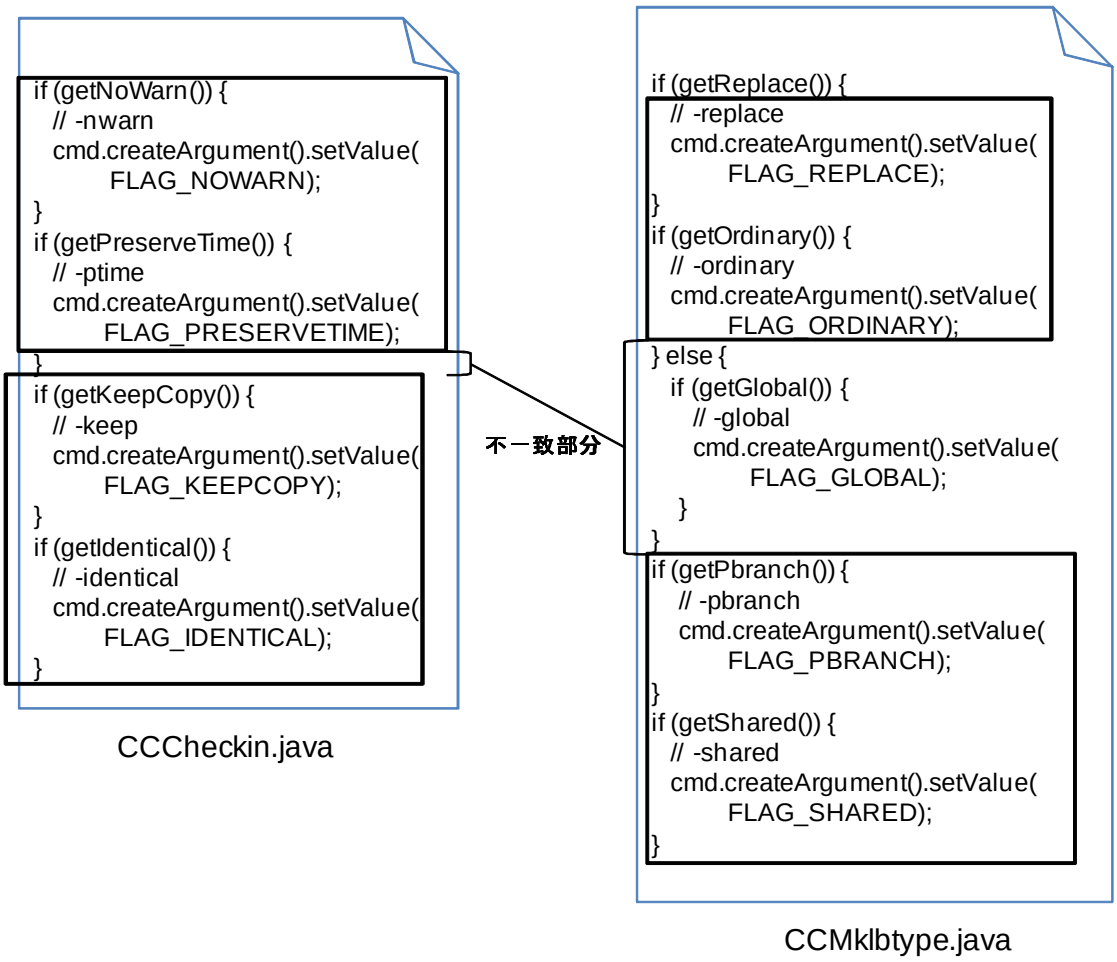


図 13: 有用な Gapped クローン (Ant)

6.4.3 考察

実験の結果から，同一クローンセットのみで構成された Gapped クローンには有用なものがほとんどないことが分かった．そのため，同じクローンセットのラベルを持つノード間にエッジを引かないフィルタリング手法は有効であるといえる．

しかし図 13 のコードのように不要ではない Gapped クローンも一部存在したため，本手法にはまだ改善の余地があると考えられる．

6.5 実験 2

6.5.1 評価方法

検出する Gapped クローンの数や検出にかかる時間について，フィルタリングを行う場合と行わない場合を比較する．また，フィルタリング後に取得した Gapped クローンの中から有用なものを挙げる．

6.5.2 実験結果

提案手法の 3 つのフィルタリングを行った後に取得した Gapped クローンの数と検出にかかった時間は表 6 の通りになった．

次に，フィルタリング後に取得した Gapped クローンの一部を紹介する．

図 14 は Ant 内のファイルである”Copy.java”と”Move.java”に含まれる Gapped クローンである．不一致部分は Move.java に含まれる File インスタンスの作成部分と if 文に該当する．Move 処理は Copy 処理と異なりコピー元のファイルを削除しなければならないため，コピー処理に加えてコピー元のファイルを削除する処理が加わっている．不一致部分が元のファイルを削除する処理に該当するので，この Gapped クローンはプログラム理解に有用であるといえる．

表 6: 検出した Gapped クローン数と検出時間

	Gapped クローン数		検出時間 (s)	
ソフトウェア名	フィルタリング無	フィルタリング有	フィルタリング無	フィルタリング有
Ant	664	78	20	53
JHotDraw	599	91	12	33
JDK	15,433	1,020	11,221	887

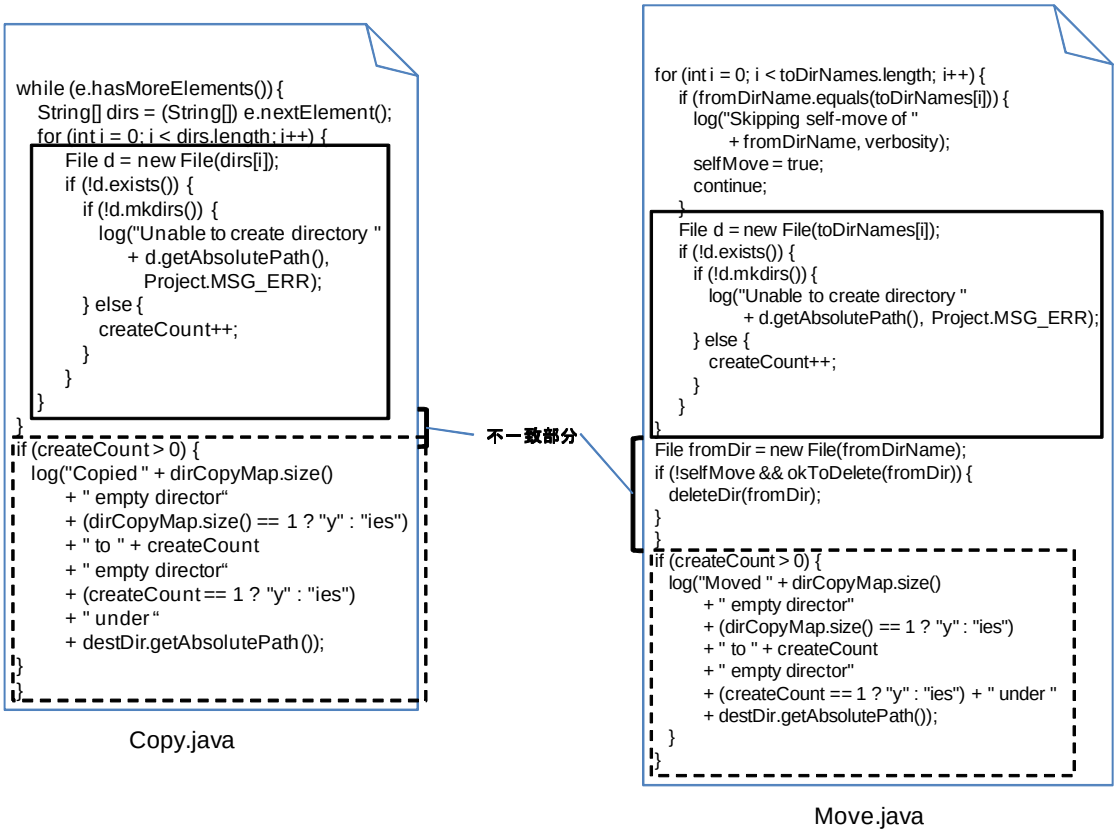


図 14: 不一致部分がメソッドの処理の違いを表す Gapped クローン (Ant)

6.5.3 考察

表 6 から非常に多くの Gapped クローンがフィルタリングできていることが分かる．検出時間についても，関数の位置情報を調べるための前処理のため Ant と JHotDraw ではフィルタリングしない場合よりも時間がかかっているが，JDK では大幅に時間短縮できている．このことから，サイズが大きい対象に対しては，検出時間の面でも有効であることが分かる．また，フィルタリング後の Gapped クローンに有用なものも存在していた．

しかし，フィルタリング後の Gapped クローンの中に，不一致部分が等しいものが含まれていた．この Gapped クローンのフィルタリングは今後の課題である．

7　まとめと今後の課題

本稿では，AGM アルゴリズムを用いて得られた Gapped クローンについて基準を定めて有用か不要かを判断し，不要な Gapped クローンについては以下の 3 つの手法を用いてフィルタリングを行った．

手法 1 複数関数にまたがっているノード間にはエッジを引かない

手法 2 同じラベルを持つノード間にはエッジを引かない

手法 3 RNR が 0.5 未満であるコードクローンをラベルに持つノードをグラフに含めない

そして，手法 2 は不要ではない Gapped クローンを削除してしまう可能性があるため，同じラベルを持つノードのみで構成される Gapped クローンを調査した．その結果，大半が不要なクローンであったため手法の妥当性を示すことが出来た．また，フィルタリングを行う場合と行わない場合を比較したところ，多くの Gapped クローンを取り除くことができ，サイズが大きい対象に対しては検出時間を大幅に短縮することができた．

今後の課題として，不一致部分が等しい Gapped クローンのフィルタリングが挙げられる．不一致部分が等しい場合，Gapped クローンではなく 1 つのクローンとして検出されるべきであるため，このフィルタリングを実現することでさらに不要な Gapped クローンを除外することができると考えられる．また，AGM アルゴリズムを用いた検出手法は様々なコードクローン検出手法に適用可能であるため，CCFinder 以外に対しても適用して調査を行う予定である．

謝辞

本研究の全過程において，常に適切な御指導及び御助言を賜わ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.

本研究の全過程において，随時適切な御指導及び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 松下誠 准教授に深く感謝致します.

本研究の作成において，随時適切な御指導及び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 石尾隆 助教に深く感謝致します.

本研究の全過程において，終始適切な御指導及び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 助教に心より深く感謝致します.

本論文の全過程において，適切な御指導及び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 吉田則裕 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 井上研究室の皆様に深く感謝致します.

参考文献

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

[2] I. Baxter, A.Yahin, L.Moura, M.Anna, and L.Bier. Clone Detection Using Abstract Syntax Trees. In *Proc. of International Conference on Software Maintenance 98*, pp. 368–377, 1998.

[3] S. Bellon, R.Koschke, G.Antoniol, J.Krinke, and E.Merlo. Comparison and Evaluation of Clone Detection Tools. In *IEEE Transaction on Software Engineering*, Vol. 31, pp. 804–818, 2007.

[4] CloneDR. `http://www.semdesigns.com/Products/Clone/`.

[5] S. Ducasse, M.Rieger, and S.Demeyer. A Language Independent Approach for Detecting Duplicated Code. In *Proc. of International Conference on Software Maintenance 99*, pp. 109–118, 1999.

[6] S. Ducasse, O.Nierstrasz, and M.Rieger. On the Effectiveness of Clone Detection by String Matching. *Journal of Software Maintenance and Evolution: Research and Practice*, 18(1):37–58, 2006.

[7] L. Jiang, G.Misheeghi, Z.Su, and S.Glondu. DECKARD : Scalable and Accurate Tree-Based Detection of Code Clones. In *Proc. of the 29th Intenational Conference on Software Engineering*, Vol. 18, pp. 96–105, 2007.

[8] H. Johnson, J. Substring Matching for Clone Detection and Change Tracking. In *Proc. of International Conference on Software Maintenance 94*, pp. 120–126, 1994.

[9] T. Kamiya, S.Kusumoto, and K.Inoue. CCFinder: A Multi-Linguistic Token-Based Code Clone Detection System for Large Scale Source Code. In *IEEE Transactions on Software Engineering*, Vol. 28, pp. 654–670, 2002.

[10] M. Kim, L.Bergman, T.Lau, and D.Nortkin. An Ethnographic Study of Copy and Paste Programming Practices in OOPL. In *Proc. of ISESE 2004*, pp. 83–92, 2004.

[11] R. Komondoor and S.Horwitz. Using Slicing to Identify Duplication in Source Code. In *Proc. of the 8th International Symposium on Static Analysis*, pp. 40–56, 2001.

[12] K. Kontogiannis. Evaluation Experiments on the Detection of Programming Patterns Using Software Metrics. In *Proc. of 4th Working Conference on Reverse Engineering*, pp. 44–55, 1997.

[13] K. Kontogiannis, R.Demori, E.Merlo, M.Galler, and M.Bernstein. Pattern Matching for Clone and Concept Detection. *Automated Software Engineering*, 3(1-2):77–108, 1996.

[14] R. Koschke, R.Falke, and P.Frenzel. Clone Detection Using Abstract Syntax Suffix Trees. In *Proc. of the 13th Working Conference on Reverse Engineering*, pp. 253–262, 2006.

[15] J. Krinke. Identifying Similar Code with Program Dependence Graphs. In *Proc. of the 8th Working Conference on Reverse Engineering*, pp. 301–309, 2001.

[16] Z. Li, S.Lu, S.Myagmar, and Y.Zhou. CP-Miner : Finding Copy-Paste and Related Bugs in Large-Scale Software Code. In *IEEE Transaction on Software Engineering*, Vol. 32, pp. 176–192, 2006.

[17] J. Mayrand, C.Leblanc, and E.Merlo. Experiment on the Automatic Detection of Function Clones in a Software System Using Metrics. In *Proc. of the International Conference on Software Maintenance 96*, pp. 244–253, 1996.

[18] J. Ottenstein, K. An Algorithmic Approach to the Detection and Prevention of Plagiarism. *ACM SIGCSE Bulletin*, 8(4):30–41, 1976.

[19] 井上克郎. エンピリカルソフトウェア工学の研究と実践 ーコードクローンを例にー. EASE プロジェクトニュースレター, pp. 1–4, 2005.

[20] 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, 19:811–822, 2004.

[21] 肥後芳樹, 楠本真二, 井上克郎. コードクローン検出と関連技術. 電子情報通信学会論文誌 D, 91-D(6):1465–1481, 2008.

[22] 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. 情報処理学会論文誌, 48(2):811–822, 2007.

[23] 藤野陽平. 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. Technical report, 大阪大学 卒業研究, 2007.

[24] 三宅達也, 肥後芳樹, 井上克郎. メトリクス計測プラグインプラットフォーム masu の開発. ソフトウェアエンジニアリング最前線 2008, pp. 63–70, 2008.