修士学位論文

題目

極大クリーク列挙による
ギャップを含むクローンセット検出手法の提案

指導教員

楠本 真二 教授

報告者

土居 真之

2020 年 2 月 5 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

令和元年度 修士学位論文

極大クリーク列挙による
ギャップを含むクローンセット検出手法の提案

土居 真之

内容梗概

これまでに多数のコードクローン検出手法が提案されている．検出手法には互いに類似するコード片の組であるクローンペア形式で出力する手法と互いに類似するコード片の集合であるクローンセット形式で出力する手法がある．どちらの形式で出力すべきかはコードクローン情報の用途によって異なる．例えばソースコードの重複度を算出したい場合や，ある箇所からバグが見つかりその類似箇所も確認したい場合はクローンペア形式の出力で十分だが，類似コード片の集約や類似コード片のライブラリ化にはクローンセット形式で出力する必要がある．しかしコードクローン間に文の挿入や削除といったギャップを含んでいる場合，クローンセットの検出が難しくなる．そのため多くの既存手法ではギャップを含むコードクローンはクローンペアの形式で出力する．さらにクローンセットを検出できる手法は検出の精度や，スケーラビリティの面で課題が残る．したがってクローンペア形式で出力されたコードクローンの情報をクローンセット形式に変換する手法が必要である．

そこで本研究では極大クリーク列挙を利用してクローンペア形式で出力されたコードクローンの情報からクローンセットを検出する手法を提案する．提案手法ではコード片を頂点，2 つのコード片がクローンペアであることを辺とみなしたグラフを作成したときに極大クリークを 1 つのクローンセットとみなす．実験では提案手法を実装したツールを，5 つのオープンソースソフトウェアに対して適用した．その結果，ギャップを含まないクローンセットよりも平均 2.4 倍も多くのギャップを含むクローンセットが得られた．さらにクローンセットの検出の実行時間も 131 ミリ秒以下で高速に検出できた．さらに検出されたクローンセットを用いてリファクタリングを行い，その結果をプルリクエストを送り，2 個のプルリクエストがマージされた．

主な用語

ギャップを含むクローン
クローンセット
極大クリーク列挙

目次

図目次

表目次

1 はじめに

コードクローンとは，ソースコード中に存在する互いに同一，あるいは類似したコード片である．コードクローンの主な発生要因はコピーアンドペーストである [1, 2]．コードクローンは保守性の低下 [3, 4, 5] やバグの伝播 [6, 7, 8] といったソフトウェアに悪影響を与えるとされている．例えば，あるコード片にバグが存在した場合，そのコード片のコードクローンに対しても同様のバグが存在する可能性がある．そのためバグの修正は元のコード片だけでなく，コードクローンになっているコード片に対しても修正を検討する必要がある．したがって，コードクローンの検出はリファクタリング [9, 10] やデバッグ [11, 12] といった様々な場面に利用されている．

これまでに多数のコードクローン検出器が開発されている [2]．コードクローンの検出器はその検出方法によってテキストベース [1, 13]，トークンベース [14, 15]，ツリーベース [16, 17]，PDG ベース [18, 19]，メトリックベース [20, 21]，ハイブリッド [22, 23] の 6 種類に分類される．これらの検出器は対象のソースコードから互いに類似するコード片の組であるクローンペアか互いに類似するコード片の集合であるクローンセットを出力する．どちらの形式で出力すべきかはコードクローン情報の用途によって異なる．例えばソースコードの重複度の算出や，ある箇所からバグが見つかりその類似箇所の確認 [6] を行う場合はクローンペア形式の出力で十分であるが，類似するコード片の集約 [10, 24] や，類似するコード片のライブラリへの抽出 [25] やコードの一貫性チェック [26] を行う場合にはクローンセット形式で出力する必要である．

コードクローンの中でも文の挿入や削除などの編集がなされたコードクローンは，ギャップを含むコードクローンと呼ばれている [27]．既存研究によるとギャップを含むクローンはソフトウェアに存在するクローンで最も多く存在すると言われている [28, 29]．したがってギャップを含むコードクローンは最も検出の必要があるコードクローンである．

ギャップを含むコードクローンを検出可能な検出器も多数開発されている．しかしこれらの検出器の多くはクローンペアの検出のみ可能であり，クローンセットの検出が可能な検出器は限られている．例えば，SourcererCC [15] は大規模なソフトウェアに対してギャップを含むクローンのペアを検出できるが，クローンセットを検出できない．一方で CCFinder [25] は suffix tree アルゴリズム [30] を用いてクローンセットを検出できるが，ギャップを含むクローンは検出できない．NiCAD [31] はギャップを含むクローンセットを検出できる検出器であるが，クローンペアで連結しているコードクローンを全て 1 つのクローンセットとみなす．そのためクローンセット内に全く類似しないコードクローンが存在する場合があり，最悪の場合検出された全てのコードクローンが 1 つのクローンセットとして検出される可能性がある [31]．iClones はギャップを含むクローンセットを generalized suffix tree アルゴリズムを用いて検出できる検出器である．しかし検出には膨大なメモリを必要とし，ギャップが大きいクロー

ンは検出できない．そのため任意の検出器で検出されたクローンペアからギャップを含むクローンに対してもクローンセットを得られる手法が必要である．

そこで本研究ではギャップを含むクローンペアからギャップを含むクローンセットを検出する手法を提案する．提案手法ではグラフ理論の極大クリーク列挙を利用する．クリークとは各頂点間に辺が存在する部分グラフである．コードクローンであるコード片を頂点，2 つのコード片がクローンペアであることを辺とみなしてグラフを生成する．こうして生成されたグラフに存在するクリーク内の頂点集合は互いに類似しているコードクローンである．提案手法ではこの頂点集合に含まれるコード片を 1 つのクローンセットとして出力する．

5 つのオープンソースソフトウェアに実験した結果，ギャップを含まないクローンセットと比較して 2.4 倍多くのギャップを含むクローンセットが存在した．さらに提案手法は NiCAD で検出された各プロジェクトのクローンペア情報からクローンセットを 131 ミリ秒以下で検出できた．また他のオープンソースソフトウェアに対して提案手法を適用し，提案手法で見つけたクローンセットを基に修正した．その修正を開発者に提案したところ 2 個の変更がマージされた．

以降，2 章では準備について述べる．3 章では，研究の動機について述べる．4 章では，提案手法の概要について述べる．5 章では，実装の詳細について述べる．6 章では，評価実験について，実験方法や結果について述べる．7 章では，妥当性への脅威について述べる．最後に，8 章では，本研究のまとめについて述べる．

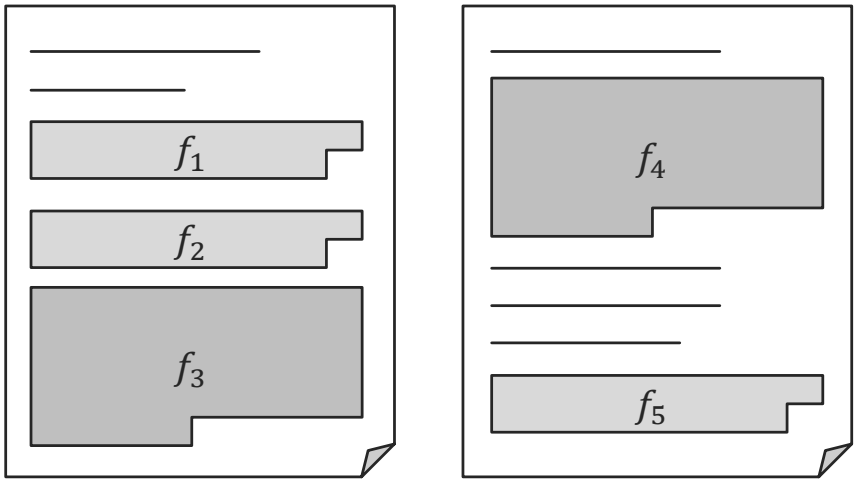

図 1　クローンペアとクローンセットの例

2　準備

2.1　コードクローン

2.1.1　定義

コードクローンとは，ソースコード中に存在する互いに同一，あるいは類似したコード片である．ソースコード中に存在する 2 つのコード片 α, β が類似しているとき，α と β は互いにコードクローンであるという．このとき，ペア (α, β) を**クローンペア**と呼ぶ．また，互いにコードクローンであるコード片を同値としたときの同値類を**クローンセット**と呼ぶ．ただし，どのような基準で類似していると判断するかは検出手法や検出器によって異なる．

クローンペアとクローンセットの例を図 1 に示す．コード片 f_1, f_2, f_5 は互いにコードクローンになっており，f_3 は f_4 とコードクローンになっている．このときクローンペアは $(f_1, f_2), (f_1, f_5), (f_2, f_5), (f_3, f_4)$ の 4 個存在し，クローンセットは $\{f_1, f_2, f_5\}, \{f_3, f_4\}$ の 2 個存在する．

2.1.2　コードクローンの種類

コードクローン間の類似度に基づき，以下の 3 つの種類に分類できる [32]．図 2 に例を示す．

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に依存する箇所を除いて，完全に一致するコードクローン．図 2 では元のコード片と完全に一致している．

Type-2：変数名や関数名などのユーザ定義名，また変数の型など一部の予約語のみが異なるコード

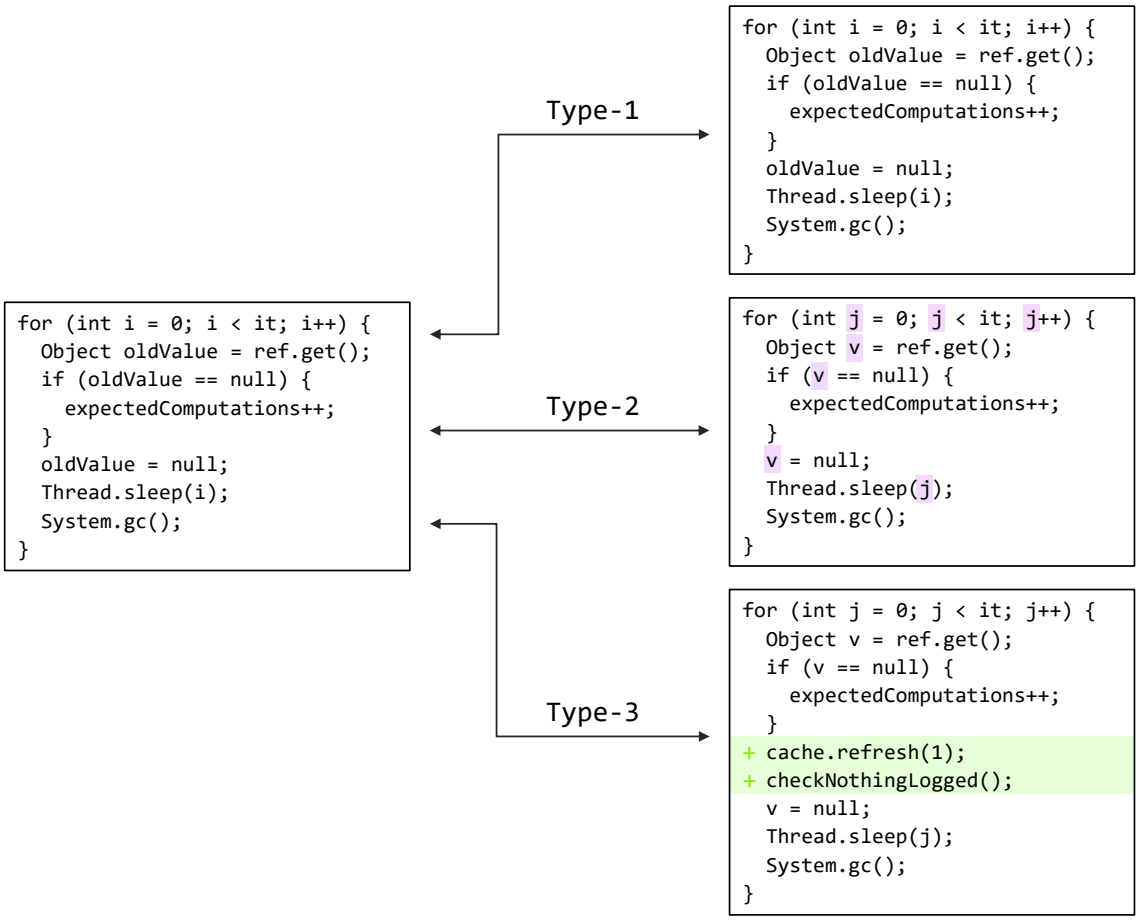

図 2　クローンタイプの例

クローン．図 2 では元のコード片と変数名が異なる．

Type-3 : Type-2 における変更に加えて，文の挿入や削除，変更が行われたコードクローン．**ギャップを含むクローン**とも呼ばれる．図 2 では Type-2 の違いに加えて 2 行の挿入が行われている．

2.1.3　発生の原因

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以下の原因が挙げられる [33, 34].

既存コードのコピーアンド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ることにより構造化や再利用可能な設計が可能である．しかし，コードの再利用が容易になったために，現実にはコピーアンド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行われるようになった．コピーアンドペーストによって生成されたコード片は，

コピー元のコード片とコードクローン関係になる．

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，所得税の計算や，キューの挿入処理，データ構造アクセス処理などである．

適切な機能の欠如

抽象データ型やローカル変数を用いれないプログラミング言語を開発に用いている場合，同じようなアルゴリズムを用いた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場合がある．

コード生成ツールの生成コード

コード生成ツールによって生成されるコードは，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．このため，類似した処理を目的としたコードを生成した場合，識別子名等の違いを除き，類似したコードが生成される．

複数のプラットフォームに対応したコード

複数の OS や CPU に対応したソフトウェアは，各プラットフォームを対象に生成されたコード部分に重複した処理が存在する傾向が強い．

偶然

偶然に，開発者が同一のコード片を書い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．

2.2 ギャップを含むクローンセットを検出できる検出器

2.2.1 NiCAD

NiCAD [1] は Roy らが開発した行単位でコードクローンを検出する検出器である．NiCAD はブロックもしくはメソッドごとに最長共通部分列 (LCS) を計算し，閾値以上の LCS を持つコード片の組をクローンペアとして検出する．NiCAD はクローンペアで連結されているコード片を全て同一クローンセットして検出する．

2.2.2 iClones

iClones [35] は Göde らが開発した字句単位でコードクローンを検出する検出器である．iClones は suffix tree アルゴリズムを用いて Type-1 と Type-2 クローンを検出する．さらに Type-1 と Type-2 クローンを組み合わせてより大きなクローンにすることで iClones は Type-3 クローンの検出が可能である．具体的には Type-1 と Type-2 クローンを組み合わせたときの字句のギャップが特定の閾値を下回る場合には Baker らのアルゴリズム [36] を用いることで Type-1 と Type-2 のクローンを結合し Type-3 クローンを検出する．

2.3 クリーク

2.3.1 定義

グラフ G における頂点の部分集合 $C \subseteq V(G)$ のうち，C に属する任意の 2 頂点間を繋ぐ辺が存在するとき，C から誘導される部分グラフを**クリーク**と呼ぶ．n 個の頂点で構成されるクリークは K_n と記述される．クリークは密な構造を表す最も基礎的な構造である．

自身を除く G の任意のクリークの部分集合でないようなクリークを**極大クリーク**と呼ぶ．また，G に存在する全ての極大クリークを検出することを**極大クリーク列挙**と呼ぶ．極大クリーク列挙はデータマイニング [37, 38] や web マイニング [39] といった様々な研究に利用されている．

極大クリーク列挙の例を図 3 に示す．図 3 では 7 つの頂点で構成されているグラフから K_4, K_3, K_2 の 3 つのクリークが極大クリークとして検出される．

2.3.2 極大クリーク列挙のアルゴリズム

極大クリーク列挙は大規模なグラフが大規模に対して行うと実行に時間がかかりすぎる恐れがある [39]．そのため極大クリーク列挙のアルゴリズムは慎重に選択する必要がある．極大クリーク列挙の効率的なアルゴリズムとして主に 2 つのアルゴリズムが知られている．1 つは Makino-Uno アルゴリズム [40]，もう 1 つは Tomita アルゴリズム [41] である．Makino-Uno アルゴリズムは逆探索法に基

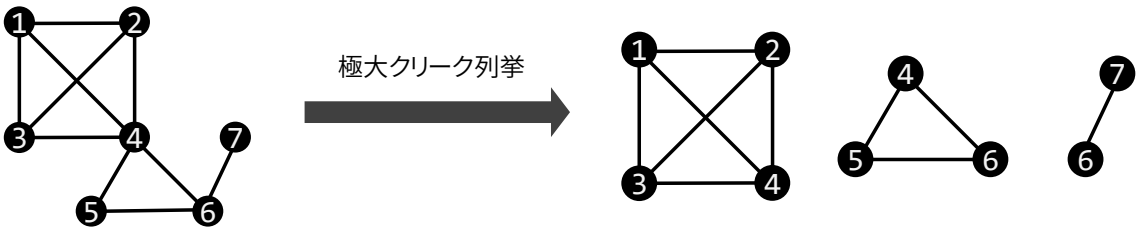

図 3　極大クリーク列挙の例

づいた多項式時間遅延アルゴリズムである．一方 Tomita アルゴリズムは枝刈り法に基づいたアルゴリズムである．Tomita らの計算実験によると Makino-Uno アルゴリズムは比較的疎なグラフに対して高速であり，Tomita アルゴリズムは比較的密なグラフに対して高速である [41]．なおデータマイニング [37, 38] や web マイニング [39] といった極大クリーク列挙を応用する研究において，比較的疎なグラフが多いことが知られている [40]．

3 研究の動機

ギャップを含むクローンセットを検出可能な検出器に NiCAD がある [1]．NiCAD は検出結果を xml ファイルに出力する．オープンソースソフトウェアである eclipse.jdt.core から NiCAD でクローン検出を行い，得られたクローンセットが記述されている xml ファイルの一部を簡略化したファイルを図 4 に示す．この xml ファイルは <class> と <source> の 2 つの要素で構成されている．<class> には検出されたクローンセットが 1 つが記述され，<class> の中にある <source> には検出されたコード片が 1 つ記述されている．すなわち <class> 内ににある <source> の集合が 1 つのクローンセットとして検出されたことを示す．しかし図 4 に記述されている 2 つの <source> 要素のコード片を見比べると全く類似していないことがわかる．NiCAD はこのような全く類似していないコード片を同一クローンセットとして検出してしまう場合がある．これは NiCAD がクローンペアで連結しているコード片を 1 つのクローンセットとして検出するためである．例えば 3 つのコード片 f_1, f_2, f_3 において $(f_1, f_2), (f_2, f_3)$ がそれぞれギャップを含むクローンペアとして検出されたとする．このとき f_1 と f_3 はクローンペアになっていないが，それぞれ f_2 とはクローンペアである．そのため f_1と f_3 間の類似度に関わらず $\{f_1, f_2, f_3\}$ は 1 つのクローンセットとして検出されてしまう．こうして得られたクローンセットは推移節が成り立たないので 2.1.1 で述べたクローンセットの定義には従わず，結果として全く異なるコード片がクローンセットに含まれる可能性がある．また理論上の最悪の場合，検出されたコードクローンが全て 1 つのクローンセットとして検出されてしまう [1]．

他にギャップを含むクローンセットを検出可能な検出器には iClones [35] が挙げられる．しかし iClones はクローン検出に膨大なメモリを必要とするため，大規模な対象からクローンを検出することはできない [15]．また対象が検出可能なサイズであっても，ギャップが大きくなると検出できないといった問題がある．このようにギャップを含むクローンセットを検出するのは困難である．そのため検出器で検出されたクローンペアに対してクローンセットを検出する手法が必要である．

```
<class classid="2" nclones="268">
<source file="DeclarationScanner.java">
    public void visitPackageDeclaration(PackageDeclaration d) {
        d.accept(pre);

        for(ClassDeclaration classDecl: d.getClasses()) {
            classDecl.accept(this);
        }

        for(InterfaceDeclaration interfaceDecl: d.getInterfaces()) {
            interfaceDecl.accept(this);
        }

        d.accept(post);
    }
</source>
<source file="Archive.java">
    public void close() {
        try {
            if (this.zipFile != null) {
                this.zipFile.close();
            }
            this.packagesCache = null;
        } catch (IOException e) {
            // ignore
        }
    }
</source>
<source> …
…
</class>
```

図 4　NiCAD により検出されたクローンセットの一部

4 提案手法

本研究では極大クリーク列挙を用いてクローンペアの情報からクローンセットを検出する手法を提案する．提案手法の概要を図 5 に示す．提案手法は入力としてクローンペアを読み込み，クローンペアから形成されるクローンセットを出力する．提案手法は前準備 (Step0) と以下の 3 つの処理で構成されている．

Step1 隣接リストの生成

Step2 極大クリーク列挙

Step3 変換

以降それぞれのステップについて説明する．

4.1 Step0: クローン検出

このステップでは提案手法の入力であるギャップを含むクローンペアを検出する．クローンペアの検出には既存の検出器を用いる．

4.2 Step1: 隣接リストの生成

多くの検出器は検出されたクローンペアをテキストファイルに出力する．提案手法ではそのテキストファイルを読み込んで得られるクローンペアを基に隣接リストを生成する．隣接リストとはグラフを表すデータ構造の 1 つで，各頂点について 1 つの辺でその頂点とつながっている全頂点のリストである．このステップでは入力されたクローンペアから隣接リストを生成する．隣接リストにおける頂点はクローンペアに含まれている全てのコード片とし，辺は 2 つの頂点間にクローンペアの関係が成り立つとき引かれる．

4.3 Step2: 極大クリーク列挙

このステップでは Step1 で生成された隣接リストを基に極大クリーク列挙を行う．2.3 節で述べたように，クリークに存在する任意の頂点間に辺が存在する．提案手法における辺はクローンペアであることを示すため，クリーク内に存在するコード片は同じクリーク内に存在する他のコード片全てとクローンペアとなっている．したがって極大クリーク列挙により得られるクリークは，互いに類似したコード片の集合であるクローンセットとなる．

2.3.2 で述べたように，極大クリーク列挙のアルゴリズムは複数ある．本研究では Makino-Uno アルゴリズム [40] を用いる．

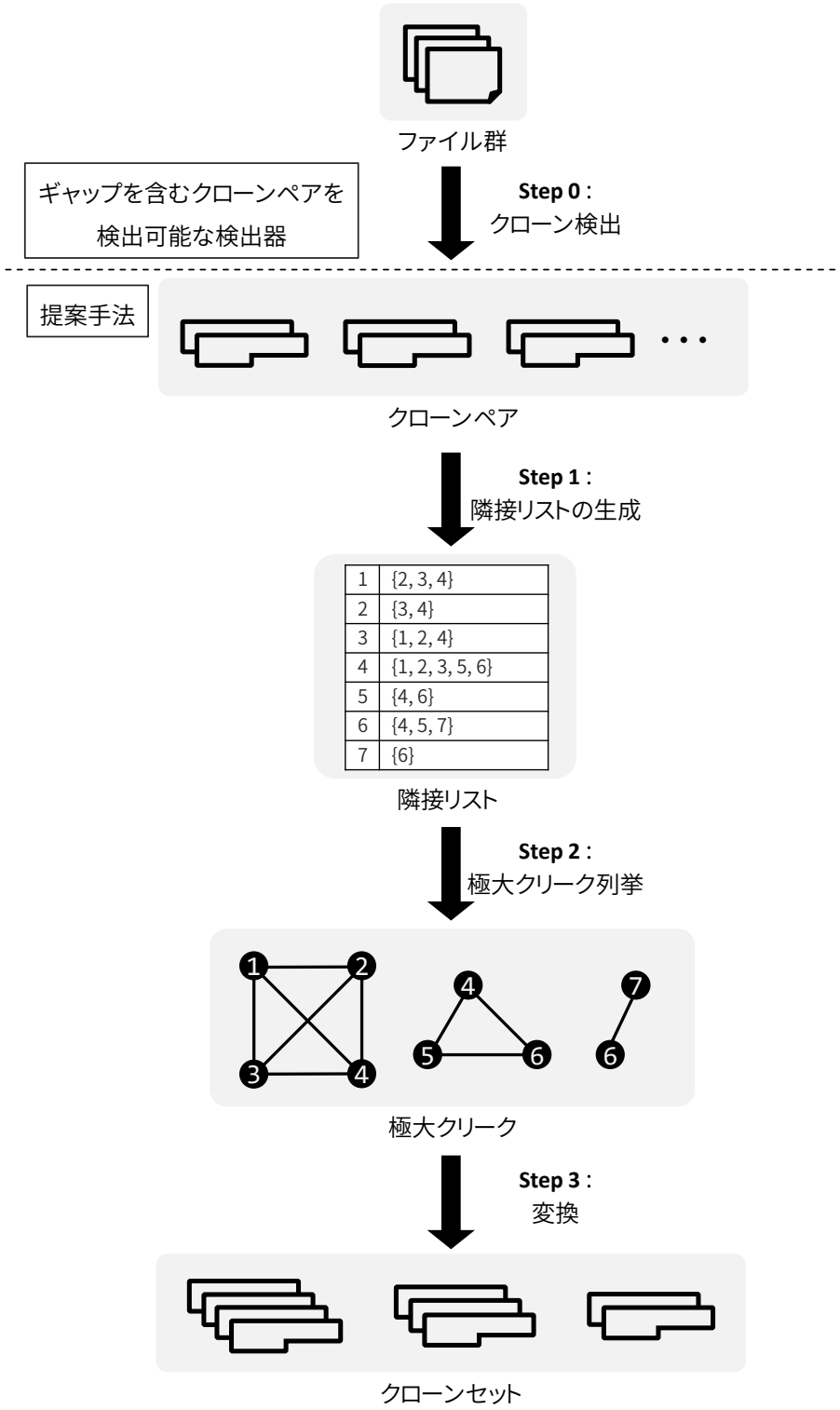
ファイル群
ギャップを含むクローンペアを
検出可能な検出器
Step 0 :
クローン検出
提案手法
クローンペア
Step 1 :
隣接リストの生成
1 {2, 3, 4}
2 {3, 4}
3 {1, 2, 4}
4 {1, 2, 3, 5, 6}
5 {4, 6}
6 {4, 5, 7}
7 {6}
隣接リスト
Step 2 :
極大クリーク列挙
極大クリーク
Step 3 :
変換
クローンセット

図 5　提案手法の概要

4.4 Step3: 変換

極大クリーク列挙により得られたクリークからクローンセットに変換する．極大クリークとクローンセットは一対一に対応しており，図 5 の場合 3 つのクローンセットが得られる．

5 実装

提案手法の実装について述べる．

5.1 検出結果の読み込み

提案手法ではクローンペアの情報を入力として受け取るために，コードクローンの検出器が出力したクローンペアの検出結果ファイルを読み込む必要がある．本研究では NiCAD [31] の検出結果を読み込めるように実装した．NiCAD 以外の検出器でも検出結果を読み込む処理を実装することで任意の検出器に提案手法を適用できる．

5.2 ID 付け

極大クリーク列挙を行うためにはコード片と頂点番号との紐づけが必要である．そのため検出結果に含まれる全てのコード片に ID をつける．NiCAD の検出結果に含まれる全てのコード片に pcid と呼ばれる ID が既に割り振られているため，本研究ではこの pcid を頂点番号とした．

5.3 極大クリーク列挙

極大クリーク列挙は大規模なグラフが大規模に対して行うと実行に時間がかかりすぎる恐れがある [39]．そのため極大クリーク列挙のアルゴリズムは慎重に選択する必要がある．本研究では Makino らが提案した極大クリーク列挙のアルゴリズム [40] を実装しているツールの MACE [42] を用いた．Makino らが提案した極大クリーク列挙のアルゴリズムは逆探索に基づいた多項式時間遅延アルゴリズムであり，入力されたグラフの最大次数を Δ としたときに極大クリーク 1 個当たり $O(\Delta^4)$ の計算時間で検出できる．

5.3.1 MACE

MACE [42] はグラフ $G(V, E)$ の隣接リストを入力に受け取ることで，$G(V, E)$ に存在する極大クリークを出力する．MACE が実装している Makino-Uno アルゴリズム [40] の疑似コードを Algorithm 1 に示す．

Algorithm 1 で呼び出されている関数の詳細を図 6 のグラフ $G(V, E)$ を用いながら述べる．8 行目の $X^\star(S, i)$ はクリーク S に対して，

- S に加えても S がクリークであるような頂点の中で最小 ID な頂点を，ID が i 以下であれば，S に加える．

Algorithm 1 Makino-Uno のアルゴリズム [40]

Input: $G(V,E)$ の隣接リスト

Output: $G(V,E)$ 内に存在する全ての極大クリーク ($MaximalCliques$)

1: **Procedure** $Enum\_Maximal\_Clique\_Main(G(V,E))$
2: $MaximalCliques = \phi$
3: **for** $i = 1$ to $|V|$ **do**
4: $\quad Enum\_Maximal\_Clique\_Iter(\{v_i\}, i)$
5: **end for**
6: **return** $MaximalCliques$
7: **Procedure** $Enum\_Maximal\_Clique\_Iter(S,i)$
8: $S = X^{\star}(|V|, S), J = J(S,i)$
9: $MaximalCliques = \{MaximalCliques \cup S\}$
10: $v_j = l(J), v_h = l(S)$
11: **if** $(j, h \leq i)$ **then return**
12: **if** $(h > j)$ **then**
13: $\quad S = S \backslash \{v_h\}$
14: **else**
15: $\quad J = J \backslash \{v_j\}, S' = \bar{X}(S, v_{i+1})$
16: $\quad$ **if** $(N(S') = \phi)$ **then**
17: $\quad\quad S' := X^{\star}(i, S' \backslash \{v_j\})$
18: $\quad\quad$ **if** $(S' = S)$ **then**
19: $\quad\quad\quad Enum\_Maximal\_Clique\_Iter(S', j)$
20: $\quad\quad$ **end if**
21: $\quad$ **end if**
22: **end if**
23: **goto** 10:

という作業を S が極大になるまで繰り返して得られる頂点集合である．例えば $X^{\star}(\{1\}, 6) = \{1,2,3\}$ である．また $J(S,i)$ は S から $X^{\star}(|V|, S)$ を得るまでに通過する反復で追加する頂点 v_k のうち，頂点 $v_j \in X^{\star}(|V|, S), j > k$ が隣接する頂点を ID 順に並べたリストである．したがって図 6 において $J(\{1\}, 1) = 2$ である．10 行目の $l(L)$ はリスト L の末尾の頂点である．ただし L が空の場合は $l(L)$ には v_{-1} という架空の頂点を入れる．15 行目の $\bar{X}(S,v)$ は S に v を加え, v に隣接する頂点をすべて取り除いて得られる頂点集合である．例えば $\bar{X}(\{v_1, v_2\}, v_3) = \{v_3\}$ である．最後に 16 行目の頂点集合

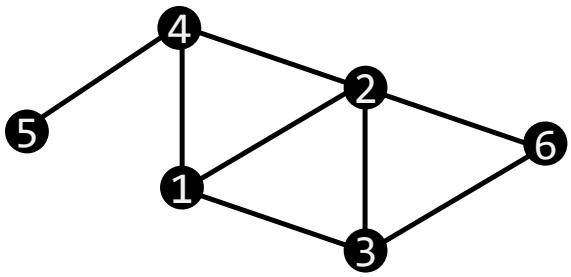

図 6　Algorithm 1 の説明に用いるグラフ $G(V, E)$

S に対し, S の全ての頂点と隣接する頂点の集合を $N(S)$ とする．例えば $N(\{v_1, v_2\}) = \{v_3, v_4, v_6\}$ である．

表 1 に図 6 のグラフ $G(V, E)$ から作られる隣接リストを MACE の入力として与え，Algorithm 1 の 10 行目が呼び出された後の $i, (v_j, v_h), MaximalCliques$ を示している．

表 1　図 6 に対して Algorithm 1 の 10 行目実行後の $i, (v_j, v_h), MaximalCliques$

i	(v_j, v_h)	$MaximalCliques$
1	(v_2, v_3)	{{1,2,3}}
1	(v_2, v_2)	{{1,2,3}}
1	(v_{-1}, v_2)	{{1,2,3}}
1	(v_{-1}, v_1)	{{1,2,3}}
2	(v_1, v_3)	{{1,2,3}}
2	(v_1, v_2)	{{1,2,3}}
3	(v_1, v_3)	{{1,2,3}}
4	(v_1, v_4)	{{1,2,3},{1,2,4}}
5	(v_{-1}, v_5)	{{1,2,3},{1,2,4},{4,5}}
6	(v_2, v_6)	{{1,2,3},{1,2,4},{4,5},{2,3,6}}

6 実験

6.1 調査項目

提案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，以下の調査項目で提案手法を評価した．

提案手法により新たに見つかるクローンセットはどの程度あるか 提案手法により新たに検出できるようになったギャップを含むクローンセットの数を調査した．

実行時間 クローンペアからクローンセットを得るのにどの程度の時間が必要か調査した．

検出結果の有用性 GitHub のオープンソースソフトウェアに対して提案手法でクローンセットを検出し，検出結果からリファクタリング可能なクローンセットについてリファクタリングを行い，その修正のプルリクエストがマージされるかを調査した．

6.2 実験環境

本実験では計算機として Ubuntu 18.04 を用いた．CPU は Intel Xeon CPU 2.10GHz であり，メモリサイズは 128.0GB である．

6.3 検出器

提案手法の入力であるクローンペアの検出には NiCAD [31] を用いた．検出単位はメソッド単位とし，類似度の閾値はデフォルトの 0.30 とした．

6.4 対象プロジェクト

実験には以下の 5 つのプロジェクトを用いた．実験対象のプロジェクトからテストコードを削除してからクローンペアを検出した．

- eclipse.jdt.core
- guava
- JFreeChart
- RxJava
- tomcat

6.5 ギャップを含むクローンセットの割合調査

本節の実験では提案手法により新たに検出が可能になったギャップを含むクローンセットの数を調査した．実験の結果を表 2 に示す．最もギャップを含むクローンセットが多く存在したのは JFreeChart であった．この要因は演算子の違いによりギャップと判定されるクローンペアが多数存在したためと考えられる．JFreeChart はグラフ描画のプロジェクトであり，座標計算のような演算を行うメソッドが多数存在する．JFreeChart ではその演算ごとにメソッドを分けているため，演算子だけが異なるメソッドが多い．逆に最もギャップを含むクローンセットが少なかったのは guava であった．guava は android パッケージと android 以外のパッケージとで同一の処理が多いため，ギャップを含まないクローンペアが多数存在していたのだと考えられる．

実験の結果としては提案手法により検出されたクローンセットのうち，平均 70.4% がギャップを含むクローンセットであった．

次に検出されたクローンセットから 3 個以上のコード片で構成されているクローンセットに対象を

表 2　クローンセット数

プロジェクト名	ギャップを含まないクローンセット数 (%)	ギャップを含むクローンセット数 (%)
eclipse.jdt.core	360 (22.42%)	1,246 (77.58%)
guava	954 (73.05%)	352 (26.95%)
JFreeChart	72 (2.68%)	2,613 (97.32%)
RxJava	111 (23.22%)	367 (76.78%)
tomcat	127 (26.62%)	350 (73.38%)

表 3　クローンペアを除いたクローンセット数

プロジェクト名	コード片の数が 3 個以上のクローンセット数 (%)	コード片の数が 2 個のクローンセット数 (%)
eclipse.jdt.core	555 (34.6%)	1,051 (65.4%)
guava	328(25.1%)	978 (74.9%)
JFreeChart	2,502 (93.2%)	183 (6.8%)
RxJava	234 (49.0%)	244 (51.0%)
tomcat	131 (27.5%)	346 (72.5%)

絞ってギャップを含むクローンセットの数を調査した．これは 2 個のコード片で構成されているクローンセットはクローンペアと同じであり，提案手法の入力として与えたクローンペアに含まれているためである．検出されたクローンセットのうち，コード片の数が 3 個以上であるクローンセットの数を表 3 に示す．

表 3 のコード片の数が 3 個以上のクローンセットに対しギャップを含むクローンセットの数を調査した．結果を表 4 に示す．実験の結果，クローンペアを検出結果から除くとほとんどのクローンセットはギャップを含んだクローンセットであった．

6.6 実行時間の調査

2.3 節で述べたように極大クリーク列挙は NP 困難であることが知られている．そのためクローンセットを得るために膨大な実行時間が必要となる可能性がある．本実験ではクローンペアを読み込んでから極大クリーク列挙によりクローンセットを得るまでの実行時間を計測した．実験の結果を表 5 に示す．5 つのプロジェクトで最も時間がかかったプロジェクトは 6.5 節の実験で最もクローンセット数の多かった JFreeChart であり， その実行時間も 131 ミリ秒と短時間でクローンセットを得られた．さらにクローンセットを実際に検出する場合の多くはまずクローンペアの検出を行い，その結果に対してクローンセットを検出する．また NiCAD による JFreeChart のクローンペア検出には 11 秒かかった．そのため実際にはクローンセットの検出にかかる時間はクローンペアの検出と比較してほぼ無視できると考えられる．さらに 5 つのプロジェクト全てからクローンペアを検出した場合のクローンセットの検出時間を計測したが，この場合も 219 ms で終わった．これより対象プロジェクトが大規模になっても

表 4　クローンペアを除いたギャップを含むクローンセット数

プロジェクト名	ギャップを含む クローンセット数	ギャップを含まない クローンセット数	クローンセット数
eclipse.jdt.core	551	4	555
guava	325	3	328
JFreeChart	2,500	2	2,502
RxJava	233	1	234
tomcat	131	0	132

表 5　極大クリーク列挙の実行時間

プロジェクト名	eclipse.jdt.core	guava	JFreeChart	RxJava	tomcat	All project
実行時間	98 ms	62 ms	131 ms	73 ms	44 ms	219 ms

検出可能であると考えられる．

6.7 ケーススタディ

本節では，提案手法の有効性を確認するためのケーススタディを行った結果について示す．

本ケーススタディでは GitHub に存在するオープンソースソフトウェアを題材に提案手法を適用し，得られたクローンセットを基にリファクタリングやバグ修正を行いプルリクエストを送った．マージされた 2 個のプルリクエストを基に提案手法の活用シナリオを述べる．

対象プロジェクトは十分な保守が行われているアクティブなプロジェクトを選ぶために以下の条件で検索し，スター数が多いプロジェクトから実験した．

- スター数 $\geq$ 1,000
- 最新更新日時 $\geq$ 2019:12:01

実験の結果 2 つのプルリクエストがマージされた．それぞれの結果について述べる．

メソッド抽出によるリファクタリング

メソッド抽出によるリファクタリングではソースコード中の処理の一部を別のメソッドとして切り出す．このリファクタリングによって，重複した処理を再利用でき，意味のある処理の単位を 1 つのメソッドとしてまとめて名前をつけるためソースコードの可読性が向上するという利点があげられる．近年ギャップを含むコードクローンもラムダ式を用いることでリファクタリング可能となるコードクローンが増えたことが報告されている [10]．

そこで実験では提案手法により見つかったクローンセットに対してメソッド抽出によるリファクタリングを行いプルリクエストを送った．その結果 apache/camel に送ったプルリクエストがマージされた[*1]．そのプルリクエストにおけるソースコードの差分を図 7 に示す．

apache/camel では 3 つのメソッドがギャップを含むクローンセットとして検出された．それぞれの違いは変数 exchange に代入する式だけである．それぞれ receive(endpointUri)，receive(endpointUri, timeout)，receiveNoWait(endpointUri) となっていた．このメソッドの引数の数が異なるのでこのクローンセットは既存のギャップを含まないクローンセットとしては検出できず，提案手法により新しく検出可能となったクローンセットである．そのためプルリクエストではメソッド抽出によるリファクタリングを行った．

[*1] https://github.com/apache/camel/pull/3486

```
  @Override
  public Object receiveBody(String endpointUri) {
-   Object answer;
-   Exchange exchange = receive(endpointUri);
-   try {
-     answer = extractResultBody(exchange);
-   } finally {
-     doneUoW(exchange);
-   }
-   return answer;
+   return receiveBody(receive(endpointUri));
  }
```

```
  @Override
  public Object receiveBody(String endpointUri, long timeout) {
-   Object answer;
-   Exchange exchange = receive(endpointUri, timeout);
-   try {
-     answer = extractResultBody(exchange);
-   } finally {
-     doneUoW(exchange);
-   }
-   return answer;
+   return receiveBody(receive(endpointUri, timeout));
  }
```

```
  @Override
  public Object receiveBodyNoWait(String endpointUri) {
-   Object answer;
-   Exchange exchange = receiveNoWait(endpointUri);
-   try {
-     answer = extractResultBody(exchange);
-   } finally {
-     doneUoW(exchange);
-   }
-   return answer;
+   return receiveBody(receiveNoWait(endpointUri));
  }
```

```
+ private Object receiveBody(Exchange exchange) {
+   Object answer;
+   try {
+     answer = extractResultBody(exchange);
+   } finally {
+     doneUoW(exchange);
+   }
+   return answer;
+ }
```

図 7　メソッド抽出を行いマージされたプルリクエストの差分

位置に一貫性のない変数宣言の検出

プログラムの記述方法の一貫性は本質的な振る舞いには影響を及ぼさないが，開発者に対するコードの視認性や可読性に強い影響を与えるという報告がある [43].

そこで提案手法により見つかったクローンセットに対して一貫性のない変数宣言に一貫性を持たせる修正した．その結果 apache/kylin に送ったプルリクエストがマージされた[*2]．そのプルリクエストにおけるソースコードの差分を図 7 に示す．

apache/kylin では図 7 の 4 つのメソッド (postRequest(), getRequest(), putRequest(), deleteRequest()) がギャップを含むクローンセットとして検出された．このクローンセットのギャップは変数 url の宣言位置である．postRequest() は call メソッドの中で宣言しているが，他の 3 個のメソッドでは call メソッドの外で宣言している．そのため postRequest() に対して他の 3 個のメソッドと url の宣言位置をそろえる修正を行い，プルリクエストを送ったところマージされた．

[*2] https://github.com/apache/kylin/pull/1062

```
    private Object postRequest(final String path, final String requestContent) throws IOException {
+       final String url = getBaseUrl() + path;
        CoordinatorResponse response = retryCaller.call(new CoordinatorRetryCallable() {
            @Override
            public CoordinatorResponse call() throws Exception {
-               String url = getBaseUrl() + path;
                String msg = restService.postRequest(url, requestContent);
                return JsonUtil.readValue(msg, CoordinatorResponse.class);
            }
        });
        return response.getData();
    }

    private Object getRequest(String path) throws IOException {
        final String url = getBaseUrl() + path;
        CoordinatorResponse response = retryCaller.call(new CoordinatorRetryCallable() {
            @Override
            public CoordinatorResponse call() throws Exception {
                String msg = restService.getRequest(url);
                return JsonUtil.readValue(msg, CoordinatorResponse.class);
            }
        });
        return response.getData();
    }

    private Object putRequest(String path) throws IOException {
        final String url = getBaseUrl() + path;
        CoordinatorResponse response = retryCaller.call(new CoordinatorRetryCallable() {
            @Override
            public CoordinatorResponse call() throws Exception {
                String msg = restService.putRequest(url);
                return JsonUtil.readValue(msg, CoordinatorResponse.class);
            }
        });
        return response.getData();
    }

    private Object deleteRequest(String path) throws IOException {
        final String url = getBaseUrl() + path;
        CoordinatorResponse response = retryCaller.call(new CoordinatorRetryCallable() {
            @Override
            public CoordinatorResponse call() throws Exception {
                String msg = restService.deleteRequest(url);
                return JsonUtil.readValue(msg, CoordinatorResponse.class);
            }
        });
        return response.getData();
    }
```

図 8　変数宣言の位置を揃えてマージされたプルリクエストの差分の例 (4 つのメソッドがクローンセットとして検出された)

7　妥当性への脅威

7.1　実験対象

実験の対象には 5 つのプロジェクトに対して実験を行った．この 5 つのプロジェクトは検出されるクローンの評価によく用いられるプロジェクトであるが，他のオープンソースソフトウェアや企業のプロジェクトに対して同様の結果になるとは限らない．

7.2　検出器

実験では NiCAD によって検出されたクローンペアに対して評価した．しかし NiCAD 以外の検出器で得られたクローンペアに対しても同様のクローンセットが得られるとは限らない．

8　あとがき

本研究では，極大クリーク列挙を用いることでクローンペアからクローンセットを検出する手法を提案した．実験では複数のオープンソースソフトウェアに適用した．ギャップを含まないクローンセットに比べ 2.4 倍多くギャップを含むクローンセットを検出した．また各プロジェクトのクローンペアに対して 131 ミリ秒以下でクローンセットを検出できた．さらに GitHub にあるオープンソースソフトウェアに適用し，提案手法に有用性を示した．今後の課題としては NiCAD 以外のコードクローン検出器への適用があげられる．

謝辞

本研究を行うにあたり，理解あるご指導を賜り，常に励ましていただきました楠本真二教授に，心より感謝申し上げます．

本研究の全過程を通して，研究に対する考え方や方向性など，丁寧かつ熱心なご指導を賜りました肥後芳樹准教授に，深く感謝申し上げます．

本研究に関して，議論の中で貴重なご助言をいただきました柗本真佑助教に，心より感謝申し上げます．

本研究を進めるにあたり，様々な形で励まし，ご助言を頂きましたその他の楠本研究室の皆様のご協力に深く感謝致します．

最後に，本研究に至るまでに，講義，演習，実験等でお世話になりました大阪大学大学院情報科学研究科の諸先生方に，この場を借りて心から御礼申し上げます．

参考文献

[1] Chanchal K Roy, James R Cordy, and Rainer Koschke. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Science of computer programming*, Vol. 74, No. 7, pp. 470–495, 2009.

[2] Dhavleesh Rattan, Rajesh Bhatia, and Maninder Singh. Software clone detection: A systematic review. *Information and Software Technology*, Vol. 55, No. 7, pp. 1165–1199, 2013.

[3] Angela Lozano and Michel Wermelinger. Assessing the effect of clones on changeability. In *2008 IEEE International Conference on Software Maintenance*, pp. 227–236. IEEE, 2008.

[4] Manishankar Mondal, Md Saidur Rahman, Ripon K Saha, Chanchal K Roy, Jens Krinke, and Kevin A Schneider. An empirical study of the impacts of clones in software maintenance. In *2011 IEEE 19th International Conference on Program Comprehension*, pp. 242–245. IEEE, 2011.

[5] Suresh Thummalapenta, Luigi Cerulo, Lerina Aversano, and Massimiliano Di Penta. An empirical study on the maintenance of source code clones. *Empirical Software Engineering*, Vol. 15, No. 1, pp. 1–34, 2010.

[6] Heejung Kim, Yungbum Jung, Sunghun Kim, and Kwankeun Yi. MeCC: memory comparison-based clone detector. In *2011 33rd International Conference on Software Engineering*, pp. 301–310. IEEE, 2011.

[7] Foyzur Rahman, Christian Bird, and Premkumar Devanbu. Clones: What is that smell? *Empirical Software Engineering*, Vol. 17, No. 4-5, pp. 503–530, 2012.

[8] Tianyi Zhang and Miryung Kim. Automated transplantation and differential testing for clones. In *Proceedings of the 39th International Conference on Software Engineering*, pp. 665–676. IEEE, 2017.

[9] Shane McIntosh, Martin Poehlmann, Elmar Juergens, Audris Mockus, Bram Adams, Ahmed E Hassan, Brigitte Haupt, and Christian Wagner. Collecting and leveraging a benchmark of build system clones to aid in quality assessments. In *Companion proceedings of the 36th international conference on software engineering*, pp. 145–154. ACM, 2014.

[10] Nikolaos Tsantalis, Davood Mazinanian, and Shahriar Rostami. Clone refactoring with lambda expressions. In *Proceedings of the 39th International Conference on Software Engineering*, pp. 60–70. IEEE, 2017.

[11] Lingxiao Jiang, Zhendong Su, and Edwin Chiu. Context-based detection of clone-related bugs.

In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pp. 55–64. ACM, 2007.

[12] Elmar Juergens, Florian Deissenboeck, Benjamin Hummel, and Stefan Wagner. Do code clones matter? In *2009 IEEE 31st International Conference on Software Engineering*, pp. 485–495. IEEE, 2009.

[13] Howard J Johnson. Substring Matching for Clone Detection and Change Tracking. In *ICSM*, Vol. 94, pp. 120–126, 1994.

[14] Xiao Cheng, Lingxiao Jiang, Hao Zhong, Haibo Yu, and Jianjun Zhao. On the feasibility of detecting cross-platform code clones via identifier similarity. In *Proceedings of the 5th International Workshop on Software Mining*, pp. 39–42, 2016.

[15] Hitesh Sajnani, Vaibhav Saini, Jeffrey Svajlenko, Chanchal K Roy, and Cristina V Lopes. SourcererCC: Scaling code clone detection to big-code. In *Proceedings of the 38th International Conference on Software Engineering*, pp. 1157–1168, 2016.

[16] Lingxiao Jiang, Ghassan Misherghi, Zhendong Su, and Stephane Glondu. Deckard: Scalable and accurate tree-based detection of code clones. In *29th International Conference on Software Engineering*, pp. 96–105. IEEE, 2007.

[17] Martin White, Michele Tufano, Christopher Vendome, and Denys Poshyvanyk. Deep learning code fragments for code clone detection. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, pp. 87–98. IEEE, 2016.

[18] Raghavan Komondoor and Susan Horwitz. Using slicing to identify duplication in source code. In *International static analysis symposium*, pp. 40–56. Springer, 2001.

[19] Jens Krinke. Identifying similar code with program dependence graphs. In *Proceedings Eighth Working Conference on Reverse Engineering*, pp. 301–309. IEEE, 2001.

[20] Jean Mayrand, Claude Leblanc, and Ettore Merlo. Experiment on the Automatic Detection of Function Clones in a Software System Using Metrics. In *icsm*, Vol. 96, p. 244, 1996.

[21] A Perumal, S Kanmani, and E Kodhai. Extracting the similarity in detected software clones using metrics. In *International Conference on Computer and Communication Technology*, pp. 575–579. IEEE, 2010.

[22] Yingnong Dang, Dongmei Zhang, Song Ge, Chengyun Chu, Yingjun Qiu, and Tao Xie. XIAO: tuning code clones at hands of engineers in practice. In *Proceedings of the 28th Annual Computer Security Applications Conference*, pp. 369–378, 2012.

[23] Yue Jia, David Binkley, Mark Harman, Jens Krinke, and Makoto Matsushita. KClone: a proposed approach to fast precise code clone detection. In *Third International Workshop on Detection of Software Clones (IWSC)*, Vol. 1, 2009.

[24] Norihiro Yoshida, Takuya Ishizu, Bufurod Edwards III, and Katsuro Inoue. How slim will my system be? estimating refactored code size by merging clones. In *Proceedings of the 26th Conference on Program Comprehension*, pp. 352–360, 2018.

[25] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. CCFinder: a multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering*, Vol. 28, No. 7, pp. 654–670, 2002.

[26] Jens Krinke. A study of consistent and inconsistent changes to code clones. In *Proceedings of the 14th working conference on reverse engineering*, pp. 170–178. IEEE, 2007.

[27] Yasushi Ueda, Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. On detection of gapped code clones using gap locations. In *Ninth Asia-Pacific Software Engineering Conference, 2002.*, pp. 327–336. IEEE, 2002.

[28] Chanchal K Roy and James R Cordy. Near-miss function clones in open source software: an empirical study. *Journal of Software Maintenance and Evolution: Research and Practice*, Vol. 22, No. 3, pp. 165–189, 2010.

[29] Jeffrey Svajlenko, Judith F Islam, Iman Keivanloo, Chanchal K Roy, and Mohammad Mamun Mia. Towards a big data curated benchmark of inter-project code clones. In *2014 IEEE International Conference on Software Maintenance and Evolution*, pp. 476–480. IEEE, 2014.

[30] Dan Gusfield. Algorithms on stings, trees, and sequences: Computer science and computational biology. *Acm Sigact News*, Vol. 28, No. 4, pp. 41–60, 1997.

[31] James R Cordy and Chanchal K Roy. The NiCad clone detector. In *2011 IEEE 19th International Conference on Program Comprehension*, pp. 219–220. IEEE, 2011.

[32] Stefan Bellon, Rainer Koschke, Giulio Antoniol, Jens Krinke, and Ettore Merlo. Comparison and evaluation of clone detection tools. *IEEE Transactions on software engineering*, Vol. 33, No. 9, pp. 577–591, 2007.

[33] Ira D Baxter, Andrew Yahin, Leonardo Moura, Marcelo Sant'Anna, and Lorraine Bier. Clone detection using abstract syntax trees. In *Proceedings. International Conference on Software Maintenance (Cat. No. 98CB36272)*, pp. 368–377. IEEE, 1998.

[34] Shinji Uchida, Akito Monden, Naoki Ohsugi, Toshihiro Kamiya, Ken-Ichi Matsumoto, and Hideo Kudo. Software analysis by code clones in open source software. *Journal of Computer*

Information Systems, Vol. 45, No. 3, pp. 1–11, 2005.

[35] Nils Göde and Rainer Koschke. Incremental clone detection. In *2009 13th European Conference on Software Maintenance and Reengineering*, pp. 219–228. IEEE, 2009.

[36] Brenda S Baker and Raffaele Giancarlo. Sparse dynamic programming for longest common subsequence from fragments. *Journal of algorithms*, Vol. 42, No. 2, pp. 231–254, 2002.

[37] Rakesh Agrawal, Ramakrishnan Srikant, et al. Fast algorithms for mining association rules. In *Proc. 20th int. conf. very large data bases, VLDB*, Vol. 1215, pp. 487–499, 1994.

[38] Rakesh Agrawal, Heikki Mannila, Ramakrishnan Srikant, Hannu Toivonen, A Inkeri Verkamo, et al. Fast discovery of association rules. *Advances in knowledge discovery and data mining*, Vol. 12, No. 1, pp. 307–328, 1996.

[39] Ravi Kumar, Prabhakar Raghavan, Sridhar Rajagopalan, and Andrew Tomkins. Trawling the web for emerging cyber-communities. *Computer networks*, Vol. 31, No. 11-16, pp. 1481–1493, 1999.

[40] Kazuhisa Makino and Takeaki Uno. New algorithms for enumerating all maximal cliques. In *Scandinavian workshop on algorithm theory*, pp. 260–272. Springer, 2004.

[41] Etsuji Tomita, Akira Tanaka, and Haruhisa Takahashi. The worst-case time complexity for generating all maximal cliques and computational experiments. *Theoretical computer science*, Vol. 363, No. 1, pp. 28–42, 2006.

[42] Uno Takeaki. Implementation issues of clique enumeration algorithm. *Special issue: Theoretical computer science and discrete mathematics, Progress in Informatics*, Vol. 9, pp. 25–30, 2012.

[43] Paul W Oman and Curtis R Cook. A taxonomy for programming style. In *Proceedings of the 1990 ACM annual conference on Cooperation*, pp. 244–250, 1990.