修士学位論文

題目

プログラム依存グラフと抽象構文木を用いたコードクローン検出のための正規化

指導教員

楠本　真二 教授

報告者

佐飛　祐介

平成 29 年 2 月 7 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

平成 28 年度 修士学位論文

プログラム依存グラフと抽象構文木を用いたコードクローン検出のための正規化

佐飛　祐介

内容梗概

ソフトウェアの保守作業を困難にする要因の１つとして，コードクローンが指摘されている．コードクローンとは，ソースコード内に存在する一致または類似するコード片のことを指す．コードクローンの存在により，ソフトウェアの理解が困難になり，不具合に対する一貫した修正が困難になる．そのため，ソフトウェア開発において，ソースコード内に存在するコードクローンを把握することは重要であり，コードクローンを自動的に検出する手法が多数提案されている．しかし，ほとんどの既存のコードクローン検出手法ではプログラム文の順序が異なるコードクローンや，構文上の表記ゆれがあるコードクローンは検出対象の範囲外となっている．また，検出可能な手法であってもソースコードに対する意味解析や動的解析が必要なため，コードクローンの検出に長い時間を要する．そのようなコードクローンは検出対象となるソースコードに対して文の順序統一や構文の統一といった正規化を行うことで，意味解析や動的解析を行わない既存の検出手法でも検出可能になると考えられる．本研究では，上記の２種類の正規化を行うことによってコードクローン検出の結果がどのように変化するのかを調査した．文の順序統一には，プログラム文間に存在する依存関係を表したプログラム依存グラフを用い，構文の統一には，プログラムの構文情報を木構造で表した抽象構文木を用いた．調査ではオープンソースのソフトウェアに対して，それぞれの正規化前後におけるコードクローン検出の結果を比較した．その結果，文の順序統一を行うことでコードクローンの検出数は減少し，新たに検出されるコードクローンの中にプログラム文の順序が異なるコードクローンは存在しなかった．また，構文の統一を行うことでコードクローンの検出数は増加する傾向にあった．新しく検出されたコードクローンの中には，開発者が短く記述していたため，正規化前には検出されなかったコードクローンが存在していた．しかし，正規化後に新しく検出されたコードクローンの中に，プログラム文の順序が異なるコードクローンや，構文上の表記ゆれがあるコードクローンは存在しなかった．このことより，文の順序統一はコードクローン検出に有効でないと考えられる．また，構文の統一によって構文上の表記ゆれがあるコードクローンは検出されなかったが，新しく検出された

コードクローンが存在した．そのため，構文の統一はコードクローン検出に有効であると考えられる．

主な用語

コードクローン
プログラム依存グラフ
抽象構文木
正規化

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類似もしくは一致しているコード片のことである．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り [1]，コードクローンはソフトウェアの保守性を低下させる原因であると考えられている [2]．例えば，あるコード片に不具合が存在していた場合，その不具合を修正するだけでなく，そのコード片のコードクローンに対しても同様の修正を検討する必要がある．そのため，コードクローンがソフトウェア中のどこに存在しているか，またどの程度存在しているかを把握するのはソフトウェアの保守において重要であり，今までに多くのコードクローン検出手法が提案されている [3]．既存のコードクローン検出手法は，行単位 [4]，字句単位 [5], メソッド単位 [6] などそれぞれが異なるコードクローンの定義を持ち，同じソースコードに対しても検出されるコードクローンは異なる．

既存のコードクローン検出手法が用いているほとんどのアルゴリズムでは，プログラムの文の順序が異なるコードクローン (順序入れ替わりコードクローン) や構文上の表記ゆれがあるコードクローンは検出の対象範囲外となっている．これらのコードクローンを検出するにはソースコードに対して意味解析や動的解析を行う必要があり，それらを用いたコードクローン検出手法も提案されている [7][8]．しかし，これらの手法は計算コストが高く，コードクローン検出に長い時間を要する．

上記のコードクローンは，ソースコードに対してプログラム文の順序の統一，構文の統一といった正規化を行うことで意味解析や動的解析を行わない既存のコードクローン検出手法でも検出可能になると考えられる．しかし，今までにプログラム文の順序を統一するアルゴリズムや構文を統一するアルゴリズムを採用しているコードクローン検出手法は存在しない．そこで，本研究ではソースコードに対する正規化として文の順序統一や構文の統一を行うことにより，コードクローンの検出結果がどのように変化するかを調査した．

文の順序統一では，ソースコードをブロックに分割し，同じ変数を持つ 2 文の順序を保ちながら，そのブロック内の文を一定の規則に則って並べ替える．ソースコードをブロックに分割し，同じ変数を持つ 2 文を抽出するために，プログラム依存グラフを用いた．プログラム依存グラフはプログラムの文間に存在する依存関係を表したものである．しかし，従来のプログラム依存グラフの依存関係は同じ変数を持つ 2 文を抽出するのに十分でないため，本研究ではプログラム依存グラフにおける依存関係の定義を拡張した．

構文の統一では，プログラム構造における式 (Expression) に格納されている要素が変数名やリテラルなどの単純な要素でない場合，その式を変数に置換し，変数に式の代入を行う．このような形で式を単純化し，構文の統一を行う．構文の統一には，プログラムの構文情報を木構造で表した抽象構文木を用いた．

以上の 2 種類の正規化を用いることで，コードクローンの検出結果がどのように変化するかを調査するため，100 個の Java のオープンソースのプロジェクトに対して実験を行った．また，一部のプロジェクトに対して正規化前後での検出コードクローンを目視で分析した．その結果，文の順序統一を行うことでコードクローンの検出数は減少した．また，文の順序統一後のソースコードから検出されるコードクローンの中に順序入れ替わりコードクローンは存在しないことがわかった．一方，構文の統一を行うことでコードクローンの検出数は増加する傾向にあることがわかった．また，新しく検出されたコードクローンの中に，構文の表記ゆれがあるコードクローンは存在しなかったが，開発者が式の値を一時変数に代入せず短く記述していたため，正規化前では検出されなかったコードクローンが存在することがわかった．しかし，正規化により新しく検出されたコードクローンの中に，順序入れ替わりコードクローンや構文上の表記ゆれがあるコードクローンは存在しなかった．このことより，文の順序統一はコードクローンの検出に有効でないと考えられる．一方，構文の統一を行うことで，正規化前では検出されなかったクローンを検出することができたため，コードクローンの検出するためには有効であると考えられる．

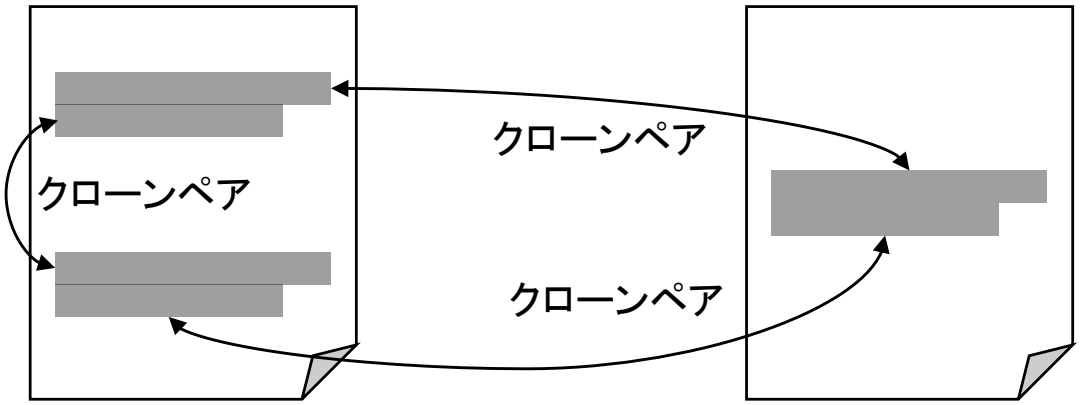


図 1: クローンペア

2 準備

2.1 コードクローン

コードクローン (以下，クローンと表記する) とはソースコード中に存在する一致または類似するコード片のことである．クローンはコピーアンドペースト操作による既存コード片の再利用など様々な要因によって発生する [9]．図 1 に示すように，ソースコード中の 2 つのコード片 α と β が類似しているとき，コード片 α と β は互いにクローンであるという．また，ペア (α,β) のことを “クローンペア” という [10]．

一般的に，クローンはその類似度に基づいて以下の 3 つのタイプに分類される [11]．

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，完全に一致するクローン．

Type-2: 変数名や関数名などのユーザ定義名，また変数の型などの一部の予約語のみが異なるクローン．

Type-3: Type-2 における変更に加えて，文の挿入や削除，変更が行われたクローン．

クローンはソフトウェアの保守性を低下させる要因と考えられている [2]．例えば，あるコード片に不具合が存在している場合，その不具合を修正するだけでなく，そのコード片の

クローンに対しても同様の修正を行うか否かを検討する必要がある．そのため，クローンがソースコード中のどこに存在しているか，どの程度存在しているかを把握することはソフトウェアの保守において重要である．

そのため，近年自動でクローンを検出する研究が盛んに行われており，多くのクローン検出ツールが提案されている [4] [6] [12] [13]．コード片同士が一致または類似しているかと判断する基準はそれぞれの検出手法や検出ツールによって異なる [10] [14]．

2.1.1 順序入れ替わりクローン

順序入れ替わりクローンとは，図 2 に示すようなブログラム文の順序が異なるクローンのことである．図 2 では，クローン片 A の 81–88 行目とクローン片 B の 96–103 行目が順序入れ替わりクローンとなっている．クローン片 A ではクローン片の最終行である 88 行目にて変数 *result* が宣言されている．一方，クローン片 B ではクローンの 1 行目である 96 行目にて同様の変数 *result* が宣言されており，プログラム文の順序が異なることがわかる．

順序入れ替わりクローンはコード片の字句や構文は異なるが，意味的には同一であるため，クローンの一種として考えられる．このようなクローンを検出するためには，ソースコードに対して意味解析を行う必要がある．そのため，字句解析や構文解析のみを行っているクローン検出手法では検出対象範囲外となっている．

Komondoor らの手法では順序入れ替わりクローンの検出は可能である [7]．Komondoor らの手法ではまず，検出対象となるソースコードに対して意味解析を行い，プログラム依存グラフを構築する．次に，構築したプログラム依存グラフから同型部分グラフを検出し，それに対応するコード片をクローンとして検出している．しかし，同型部分グラフを検出する処理は長い時間を要する処理であり，そのためクローンの検出に長い時間を要する．また，Komondoor らの手法ではプログラム文がソースコード上で連続していない非連続クローンが検出可能であるが，そのようなクローンの中には，開発者にとって有益でないクローンが多く含まれるとの報告もある [15]．

2.1.2 構文上の表記ゆれがあるクローン

構文上の表記ゆれがあるクローンとは図 3 に示すようなクローンである．図 3 の例では，クローン片 C の 54–56 行目とクローン片 D の 31–32 行目が構文上の表記ゆれがあるクローンとなっている．クローン片 C では，55 行目にて変数 *multiplied* に x * y の値を一時的に格納し，56 行目にてその変数を関数の引数として与えている．一方，クローン片 D のでは，32 行目にて直接 x * y の値を関数の引数として与えている．これらのコード片の違いは，引数として与えている値を一時的に変数に与えているか否かという構文上の表記のみの違いで

```
                          ⋮
81:   Date startDate = null;
82:   if(sinceDays > 0) {
83:       Calendar cal = Calendar.getInstance();
84:       cal.setTime(new Date());
85:       cal.add(Calendar.DATE, -1 * sinceDays);
86:       startDate = cal.getTime();
87:   }
88:   List<SubscriptionEntry> results = new ……
                          ⋮
```

クローン片A

```
                          ⋮
96:   List<WeblogEntryWrapper> results = new ……
97:   Date startDate = null;
98:   if(sinceDays > 0) {
99:       Calendar cal = Calendar.getInstance();
100:      cal.setTime(new Date());
101:      cal.add(Calendar.DATE, -1 * sinceDays);
102:      startDate = cal.getTime();
103:  }
                          ⋮
```

クローン片B

図 2: 順序入れ替わりクローンの例

```
                          ⋮
54:   int discordant = num - x - y + xy - ……;
55:   int multiplied = x * y;
56:   return discordant / Math.sqrt(multiplied);
                          ⋮
```

クローン片C

```
                          ⋮
31:   int discordant = num - x - y + xy - ……;
32:   return discordant / Math.sqrt(x * y);
                          ⋮
```

クローン片D

図 3: 構文上の表記ゆれがあるクローンの例

ある．

このような構文上の表記ゆれがあるクローンはコード片の字句や構文は異なるが，意味的には同一であるため，クローンの一種として考えられる．このようなクローンを検出するためには，ソースコードに対して意味解析や，動的解析を行う必要がある．そのため，字句解析や構文解析のみを行っているクローン検出手法では検出対象範囲外となっている．

Jiang らの手法では構文上の表記ゆれがあるクローンを検出することが可能である [8]．Jiang らの手法ではコード片に対して複数のテストを生成し，それらの出力が同じであったコード片同士をクローンとして検出している．そのため，図 3 に示すような構文上の表記ゆれがあるクローンだけでなく，バブルソートやクイックソートといったアルゴリズムが異なるクローンも検出可能である．しかし，コード片に対してテストを実行する必要があるため，クローンの検出に膨大な時間がかかる．

2.2 プログラム依存グラフ

プログラム依存グラフ (Program Dependency Graph，以下 PDG と表記する) とはプログラムの文間に存在する依存関係を表した有向グラフである．PDG にはノードと有効辺 (エッジ) が存在し，ノードはプログラムの文を表し，ノード間に引かれるエッジはそれらのノードが表すプログラム文の間に存在する依存関係を表す．

PDG で用いられている 2 種類の依存関係を以下に示す [16]．

データ依存

- 変数 v が文 s で定義されている．
- 変数 v が文 t で参照されている．
- s から t の間に v を再定義しない経路が存在する．

以上の 3 つの条件を満たすとき，文 s と文 t 間にデータ依存が存在する．

制御依存

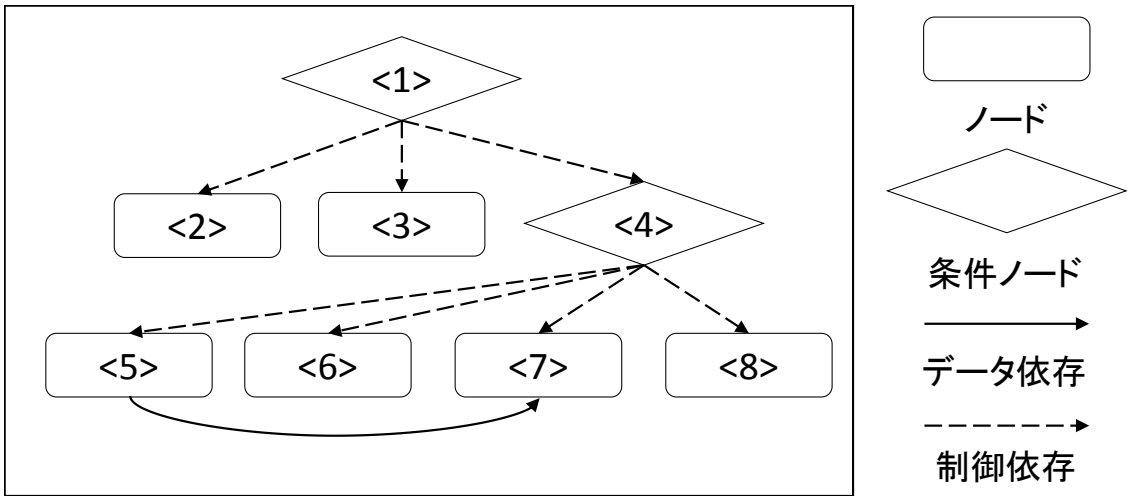
- 文 s が条件文である．
- 文 t は文 s の後に実行される可能性がある．
- s の実行結果によって t が実行されるかが決まる．

以上の 3 つの条件を満たすとき，文 s と文 t 間に制御依存が存在する．

図 4(b) に簡単な PDG の例を示す．図 4(b) は図 4(a) のソースコードを PDG で表したも

```
1: if (flag1){
2:     int value1 = 1;
3:     int value2 = 2;
4:     if (flag2){
5:       value1 = 3;
6:       value2 = 4;
7:       value1 = value1 * x;
8:       return;
       }
   }
```

(a) ソースコード



(b) PDG

図 4: PDG の例

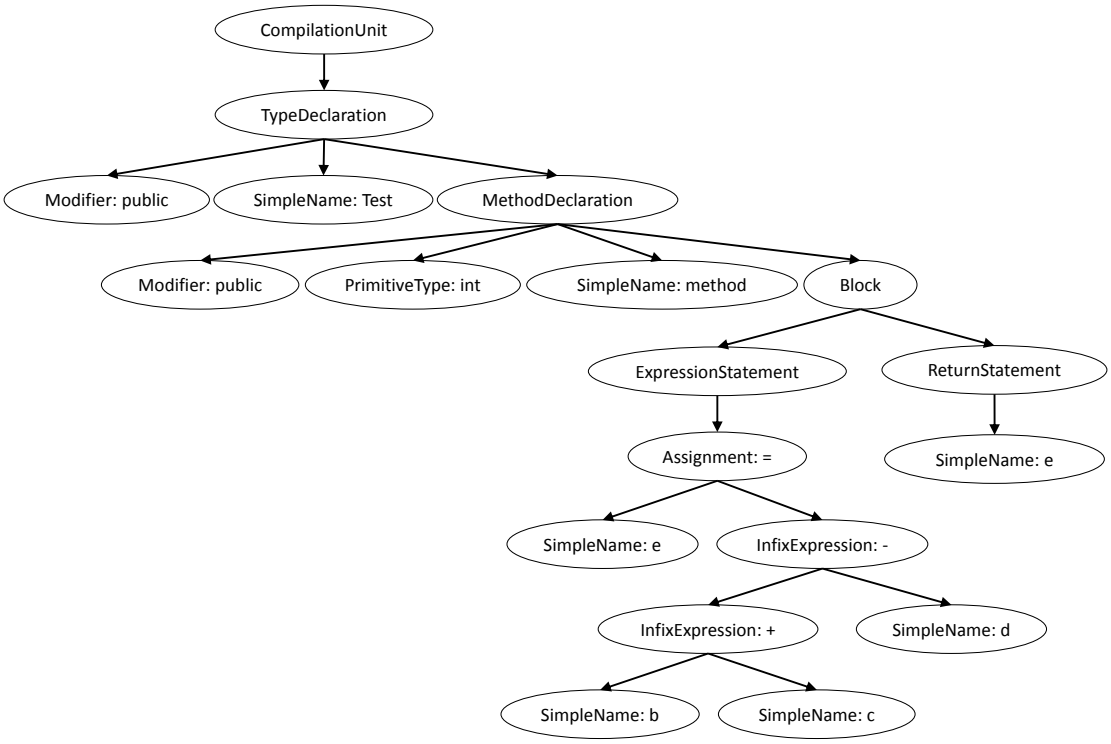
のである．変数 $value1$ が定義されている 5 行目から同じ変数が参照されている 7 行目へとデータ依存が存在していることがわかる．また，1 行目の if 文の条件文からその内部に存在する 2–4 行目の文に制御依存が存在しており，同様に 4 行目の if 文の条件文からその内部に存在する 5–8 行目の文に制御依存が存在していることがわかる．

2.3 抽象構文木

抽象構文木 (Abstract Syntax Tree，以下 AST と表記する) とは，木構造を用いてソースコードの構文情報を表したものである．AST は順序木であり，子ノードの数に制限はない．例として，簡単な Java ソースコードとそれに対応する AST を図 5 に示す．図 5 の AST はプログラムに対応する 19 のノードを持つ．それぞれのノードは ID とプログラムの構造に

```
public class Test{
    public int method() {
        e = b + c - d;
        return e;
    }
}
```

(a) ソースコード



(b) AST

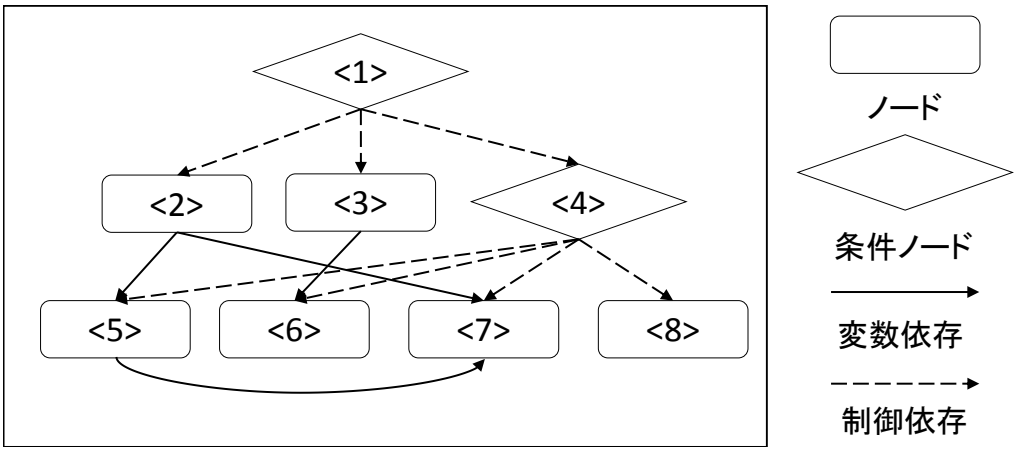
図 5: AST の例

対応するラベルとソースコード中のトークンに対応する値を持つ．例えば，図 5(b) において,「SimpleName: e」は SimpleName がラベル，e が変数名を表す．AST 上のノードはソースコードにおける構文情報を表しており，エッジで結ばれた子ノードは詳細情報を表す．例えば，MethodDeclaration の子ノードを辿ることで，返り値の型が int 型であることや，メソッド名が *method* であることなどがわかる．

AST は構文情報を表した木であるため，AST においてノードの追加，削除，移動, 置換を行い，ソースコードへと復元することで，ソースコードを改変することができる．そのため，本研究では構文の統一に AST を利用する．

```
1: if (flag1){
2:     int value1 = 1;
3:     int value2 = 2;
4:     if (flag2){
5:       value1 = 3;
6:       value2 = 4;
7:       value1 = value1 * x;
8:       return;
       }
   }
```

(a) ソースコード



(b) 拡張 PDG

図 6: 拡張 PDG の例

3 文の順序統一

3.1 アプローチ

文の順序統一では，以下の 2 つの条件を満たすように文を並べ替える．

1. ソースコードをブロックに分割し，そのブロック内の文の順序を統一する．

2. 同じ変数を使用している 2 文の順序は保つ．

これらの条件を満たしながら文の並べ替えを行うため，拡張 PDG を用いた．
以下，拡張 PDG について説明する

3.1.1 拡張 PDG

本研究では従来の PDG に対して依存関係の拡張を行っている．以降，本研究で用いる拡張を行った PDG のことを “拡張 PDG” と表記する．本研究で用いる拡張 PDG は 2 つの依存関係を持つ．1 つ目の依存関係は従来の PDG と同じ定義の制御依存である．2 つ目の依存関係は従来の PDG におけるデータ依存を拡張した変数依存である．変数依存の定義を以下に示す．

図 6(b) に拡張 PDG の例を示す．図 6(b) は図 6(a) のソースコードを拡張 PDG で表したものである．変数 $value1$ が定義されている 5 行目から同じ変数が参照されている 7 行目へと変数依存が存在するだけでなく，変数 $value1$ が定義されている 2 行目から同じ変数が再定義されている 5 行目へも変数依存が存在することがわかる．

変数依存

- 変数 v が文 s で定義もしくは参照されている．
- 変数 v が文 t で定義もしくは参照されている．
- s から t の間に v を再定義しない経路が存在する．

以上の 3 つの条件を満たすとき，文 s と文 t 間に変数依存が存在する．

ソースコードをブロックに分割するためには制御依存を用いる．同じノードからの制御依存を持つノードをグループに分割することで同じブロック内に存在するプログラム文と対応したノードのグループを得る．また，拡張 PDG において依存関係が存在する 2 文を順序を保つ文としている．ここで，拡張 PDG ではなく，通常の PDG を用いた場合，同じ変数を使用している 2 文の順序が保たれない可能性がある．例えば，図 7 のソースコードの 6，7 行目には 2 つのメソッド呼び出し文 “append()” が存在する．これらの文が異なる順序で実行された場合，変数が指しているオブジェクトの中身は異なるものとなる．そのため，この 2 文の順序を並べ替えてはいけない．従来の PDG で用いられているデータ依存を用いた場合，6，7 行目間に依存関係が存在せず，順序が保たれない可能性がある．一方，変数依存を使用した場合，6，7 行目間に依存関係が存在するため，2 文の順序は保たれる．このように，同じ変数を用いた文の順序を保つために変数依存を用いている．

3.2 手順

文の順序統一は以下の 4 つの Step で行われる．入力と出力は以下のとおりである．

- 入力…Java ソースコード

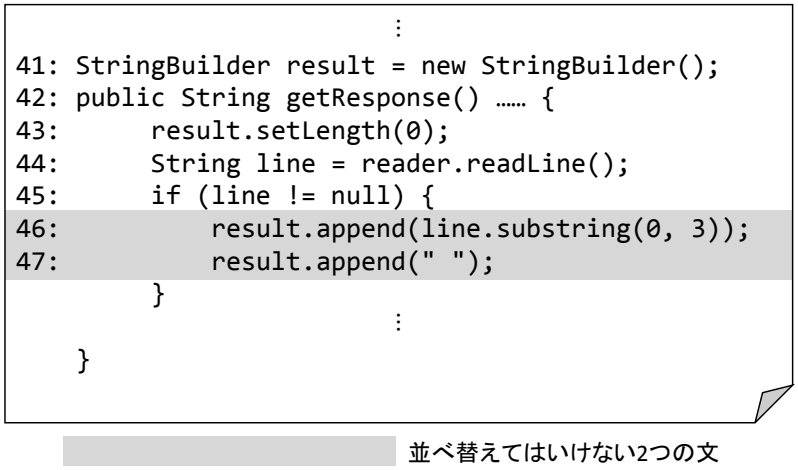


図 7: 順序を並べ替えていけない文の例

- 出力…文の順序統一が行われた Java ソースコード

Step-1: 入力されたソースコードから拡張 PDG を生成.

Step-2: 同じブロックに存在するノードのグループを取得.

Step-3: 依存関係を変えないノードの並び順を選択.

Step-4: 拡張 PDG からソースコードを復元.

以下，それぞれの Step について図 8 の例を用いて説明する．図 8 では，図 4(a) のソースコードを入力として与えている．

3.2.1 Step-1: 入力されたソースコードから拡張 PDG を生成

この Step では，入力ソースコードから拡張 PDG を生成する．

3.2.2 Step-2: 同じブロックに存在するノードのグループを取得

この Step では，ノードで構成されるグループを取得する．それぞれのグループは同じノードからの制御依存を持つノードで構成される．ここで，グループ内のノードはソースコードにおいて同じブロックに存在するプログラム文に対応する．この Step ではブロックの最後に return 文，break 文，continue 文がある場合，それと対応するノードをグループから除外するというヒューリスティックを用いた．

図 8 の例では，ノード <1> はノード <2>，<3>，<4> への制御依存エッジを持っており，またノード <4> はノード <5>，<6>，<7>，<8> への制御依存エッジを持っている．その

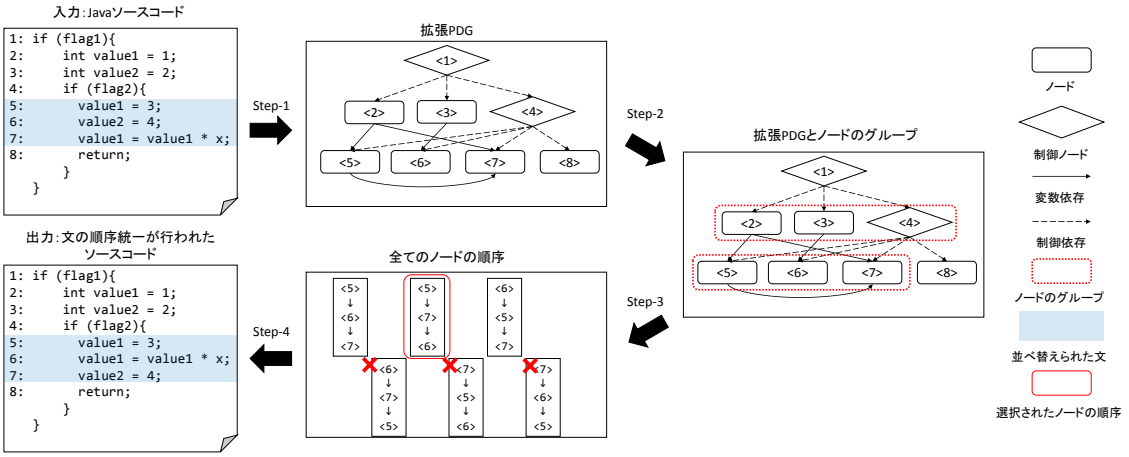


図 8: 文の順序統一の例

ため，ノード <2>，<3>，<4> で構成されるグループとノード <5>，<6>，<7>，<8> で構成されるグループがそれぞれ存在する．次に，ノード <8> は return 文であるため，2 つ目のグループから除外する．ここで，条件ノードはそのブロックに含まれる全てのノードとみなす．つまり，ノード <4> は，ノード <4>，<5>，<6>，<7>，<8> とみなす．以上より，図 8 の例ではノード <2>，<3>，<4> からなるグループと，ノード <5>，<6>，<7> からなるグループが取得できる．

3.2.3 Step-3: 依存関係を変えないノードの並び順を選択

この Step では，Step-2 で取得したグループそれぞれに対して，依存エッジの始点と終点となっているノードの組の順序が並べ替えられていないノードの順序を選択する．はじめに，ノードの数を n としたとき，$n!$ 通りのノードの順序を候補として列挙する．次に，依存エッジの始点と終点となっているノードの組の順序が 1 つでも並べ替えられている順序を候補から除外する．その次に，候補として残っている順序に対して，ノードに含まれる変数名の正規化を行う．最後に，変数名の正規化を行った文をもとにして，残っている順序をアルファベット順に並べ替え，最初のものを選択する．

図 9 と図 10 の例を用いて説明する．図 9 はノード <5>，<6>，<7> で構成されるグループについて，$3! = 6$ 通りのノードの順序をソースコードで表している．それぞれの順序を (a)–(f) とする．この図において，行番号はノードの ID と対応している．図 8 に示すとおり，ノード <5> とノード <7> の間には変数依存が存在する．(d)，(e)，(f) ではそれらのノードの順序が並べ変わっているため，候補から除外する．次に，残っている順序 ((a)，(b)，(c)) に対して変数名の正規化を行う．変数名の正規化はソースコード単位ではなく，文単位

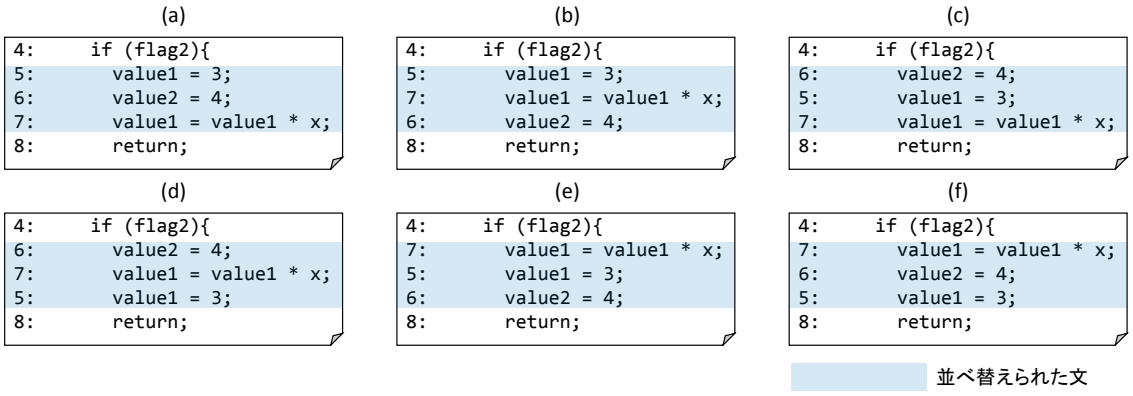


図 9: ノードの順序をそれぞれソースコードで表したもの

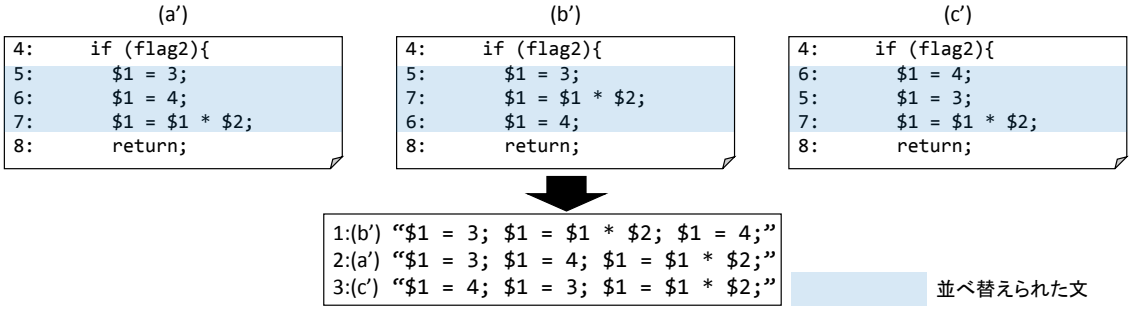


図 10: 変数名の正規化とアルファベット順のソート

で行う．つまり，1 文に 2 種類以上の変数が出現する場合，それぞれの変数を出現した順番に $1, $2 という変数で置換する．正規化を行ったソースコード ((a')，(b')，(c')) を図 10 に示す．次に，それぞれのソースコードを 1 行で表した文字列を取得する．それらの文字列をアルファベット順に並べ，最初のものを文の順序統一後の順序として選択する．この場合，(b') の文字列がアルファベット順で最初となるため，(b) の順序を並べ替え後の順序として選択する．つまり，ノード <5>，<6>，<7> はノード <5>，<7>，<6> という順序に並べ替えられる．また，ノード <2>，<3>，<4> で構成されるグループについては，並べ替えは行われない．

3.2.4 Step-4: 拡張 PDG からソースコードを復元

ここまでで，それぞれのグループに対してノードの順序が得られた．この Step では，Step-3 の結果をもとにして拡張 PDG からソースコードを生成する．

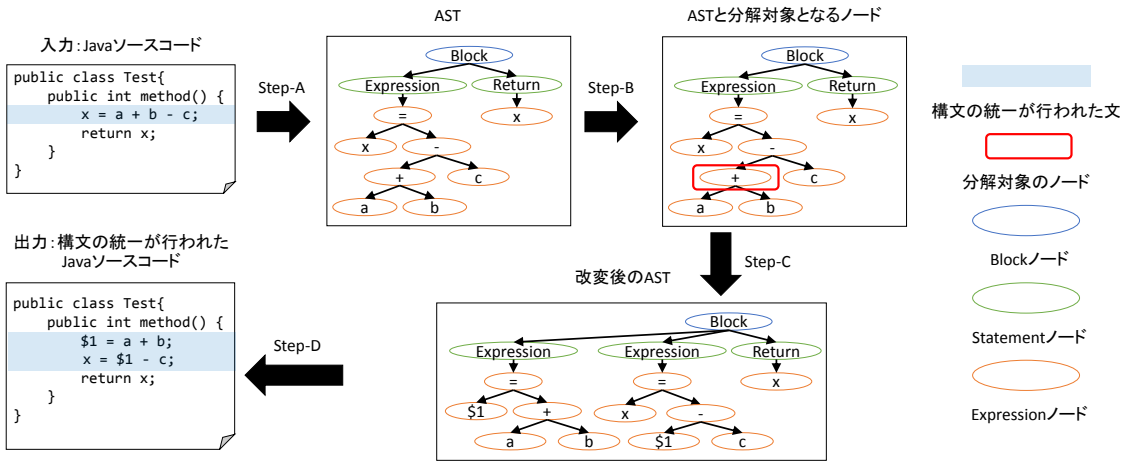


図 11: 構文の統一の例

4　構文の統一

4.1　アプローチ

構文の統一では，構文における式に格納されている要素が変数名やリテラルなどの単純な要素でない場合，その式を変数に置換し，変数に式の代入を行う．このような形で式を単純化し，構文の統一を行う．

4.2　手順

構文の統一における，入力と出力は以下のとおりである．

- 入力…Java ソースコード
- 出力…構文の統一が行われた Java ソースコード

構文の統一は以下の 4 つの Step で行われる．

Step-A: Java ソースコードから AST を構築

Step-B: 置換する対象となるノードを抽出

Step-C: AST の改変

Step-D: ソースコードへの復元

以下，それぞれの Step について図 11 の例を用いて説明する．図 11 の AST では一部ラベル等の表記を省略している．

4.2.1 Step-A: Java ソースコードから AST を構築

この Step では入力された Java ソースコードから AST を構築する．AST の構築には Eclipse Foundation によってオープンソースソフトウェアとして提供されている Java Development Tools (JDT) [17] を利用した．

4.2.2 Step-B: 置換する対象となるノードを抽出

この Step では Step-A で構築された AST を探索し，置換する対象となるノードを抽出する．探索する順序は帰りがけ順である．置換する対象となるノードは以下の 4 つの条件を満たすノードである．

1. ノードが式 (Expression) を表している．
2. ノードの型が単純でない．
3. 直近の親ノードが Assignment でない．
4. Assignment でない．

ここで，ノードの型が単純でないとは，ノードの型が以下のもの以外であるということである．

- BooleanLiteral
- CharacterLiteral
- NullLiteral
- NumberLiteral
- SimpleName
- StringLiteral
- ThisExpression
- TypeLiteral

式に格納されているノードの型が上記の条件を満たす場合，そのノードを変数に置き換える．また，代入文を生成し左辺に変数を，右辺に式を与える．

図 11 の例において，以上の条件を満たすノードは二項演算子 “-” の左辺である二項演算子 “+” である．

4.2.3 Step-C: AST の置換

この Step では Step-B で抽出されたノードに対して，ノードの置換，代入文の生成などを行い，AST を改変する．AST の改変は以下の手順で行われる．

1. Step-B で抽出されたノードを一時変数に置換する．
2. 代入文を表す AST を作成し，左辺に置換した変数を，右辺に対象となる式を与える．
3. 対象となる式を持つ文の直前に代入文を挿入する．

4.2.4 Step-D: ソースコードへの復元

ここまでで，改変を行った AST が得られた．この Step では Step-C までの結果で得られた AST をソースコードへ復元し，構文の統一を行ったソースコードを得る．

5　実験

本実験では，前述した 2 種類の正規化をソースコードに対して行うことで，クローン検出手法の検出結果がどのように変化するのかを調査する．

5.1　実験対象

GitHub [18] より 2017/1/23 時点で人気上位 100 個の Java プロジェクトを実験対象とした．表**??**に各プロジェクトの詳細を載せる．いくつかのプロジェクトにはテストケースやソースコード自動生成ツールによって生成されたソースコードが存在する．このようなソースコードからもクローンは検出されるが，このようなクローンは検出結果に悪影響を及ぼすと考えられる．既存の研究においてもテストケースやソースコード自動生成ツールによって生成されたソースコードは検出対象から除外している [19] [20] [21]．そのため，本実験ではプロジェクトからテストケース，自動生成ファイルを除いたものを実験対象とした．テストケース，自動生成ファイルの除去には Java ファイルに対して “test”,“generated” の文字列で検索を行い，該当したファイルからテストケース，自動生成ファイルとみられるものを目視で除去した．

全ての実験対象の Java ファイル数と LOC を図 3 に示す．

5.2　実験方法

本実験は以下の手順で行う．

1. 元のソースコードに対して 2 種類の正規化を行う．
2. 元のソースコードと正規化を行った 2 種類のソースコードそれぞれに対してクローンを検出する．

クローンの検出には CCFinderX を用いた [22]．CCFinderX は一般的なクローン検出ツール CCFinder [23] のバージョンアップ版であり，Type-2 までのクローンを検出可能な手法である．CCFinderX および CCFinder は既存の研究 [24][25][26][27] において一般的なクローン検出ツールとして用いられている．そのため，本研究でも一般的なクローン検出ツールとし

表 1: 実験対象プロジェクト

合計 Java ファイル数	合計 LOC	合計 Java ファイルサイズ
41,452	6,319,089	2.8Mbyte

て CCFinderX を用いた．CCFinderX のパラメータである最小一致トークン数は 50，最小トークン種類数は 12 とした．これは CCFinderX のデフォルトの設定である．

5.3 実験結果

5.3.1 文の順序統一

文の順序統一の正規化を行い実験を行った結果を図 12 に載せる．横軸に正規化を行わない元のソースコードから検出されたクローンペア数，縦軸に文の順序統一の正規化を行ったソースコードから検出されたクローンペア数をとっている．また，縦軸，横軸ともに対数軸である．クローンペア数が減少したプロジェクトを四角，変化しなかったプロジェクトを三角，増加したプロジェクトを丸のマーカーで表している．また，表 2 に文の順序統一によりクローンペア数が増加した，変化しなかった，減少したプロジェクトの数を示す．

正規化前後の検出クローンペア数の 2 群に対して，ウィルコスソンの符号順位検定を行ったところ，有意水準 1%の基で有意差が認められた．また，表 2 より，クローンペア数が増加しているプロジェクトよりも減少しているプロジェクトの方が多い．このことより，文の順序統一の正規化を行うことで，プロジェクトにもよるが，検出されるクローンペア数が減少する傾向にあることがわかる．

5.3.2 構文の統一

構文の統一の正規化を行い実験を行った結果を図 14 に載せる．図 12 と同様に，横軸に正規化を行わない元のソースコードから検出されたクローンペア数，縦軸に構文の統一の正規化を行ったソースコードから検出されたクローンペア数をとっている．また，縦軸，横軸ともに対数軸である．クローンペア数が減少したプロジェクトを四角，変化しなかったプロジェクトを三角，増加したプロジェクトを丸のマーカーで表している．また，表 2 に構文の統一によりクローンペア数が増加した，変化しなかった，減少したプロジェクトの数を示す．

図 14 より構文の統一により，クローンペアの数が増加しているプロジェクトが多いことがわかる．検出クローンペア数が増加したプロジェクトの中には検出されるクローンペア数が 2 倍に増加したものもあった．また，同様に正規化前後の検出クローンペア数の 2 群に対

表 2: 検出クローンペア数の変化

	増加	変化なし	減少
文の順序統一	7 個	62 個	31 個
構文の統一	91 個	6 個	3 個

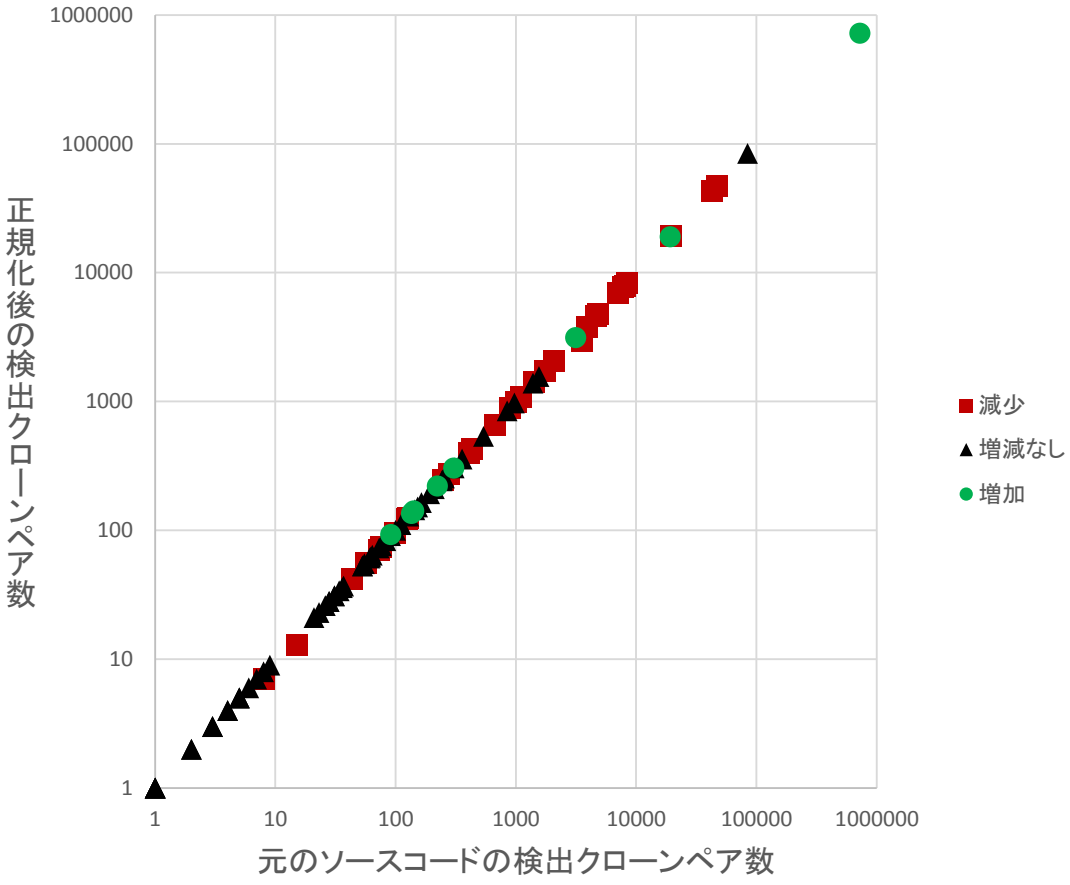


図 12: 文の順序統一の実験結果

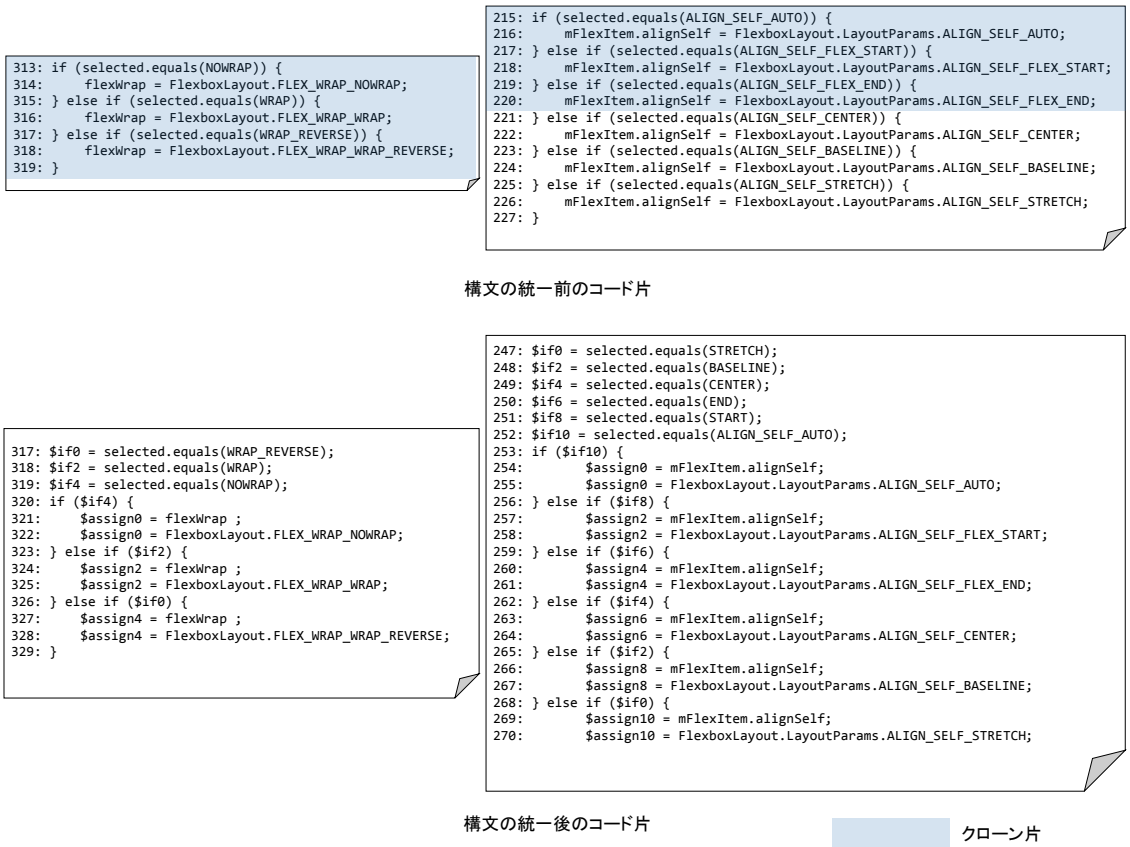


図 13: 構文の統一により検出されなくなったクローンペアの例

して，ウィルコクソンの符号順位検定を行ったところ，有意水準 1%の基で有意差が認められた．また，表 2 より，クローンペア数が増加しているプロジェクトがほとんどである．検出されたクローン数が減少しているプロジェクトでは図 13 のように if-else 文の条件式が分解され，if ブロックのトークン数が減少し，CCFinderX の閾値を超えなくなったため，検出されなくなった例がほとんどであった．

このことより，構文の統一の正規化を行うことで，検出されるクローンペア数の数が増加することがわかる．

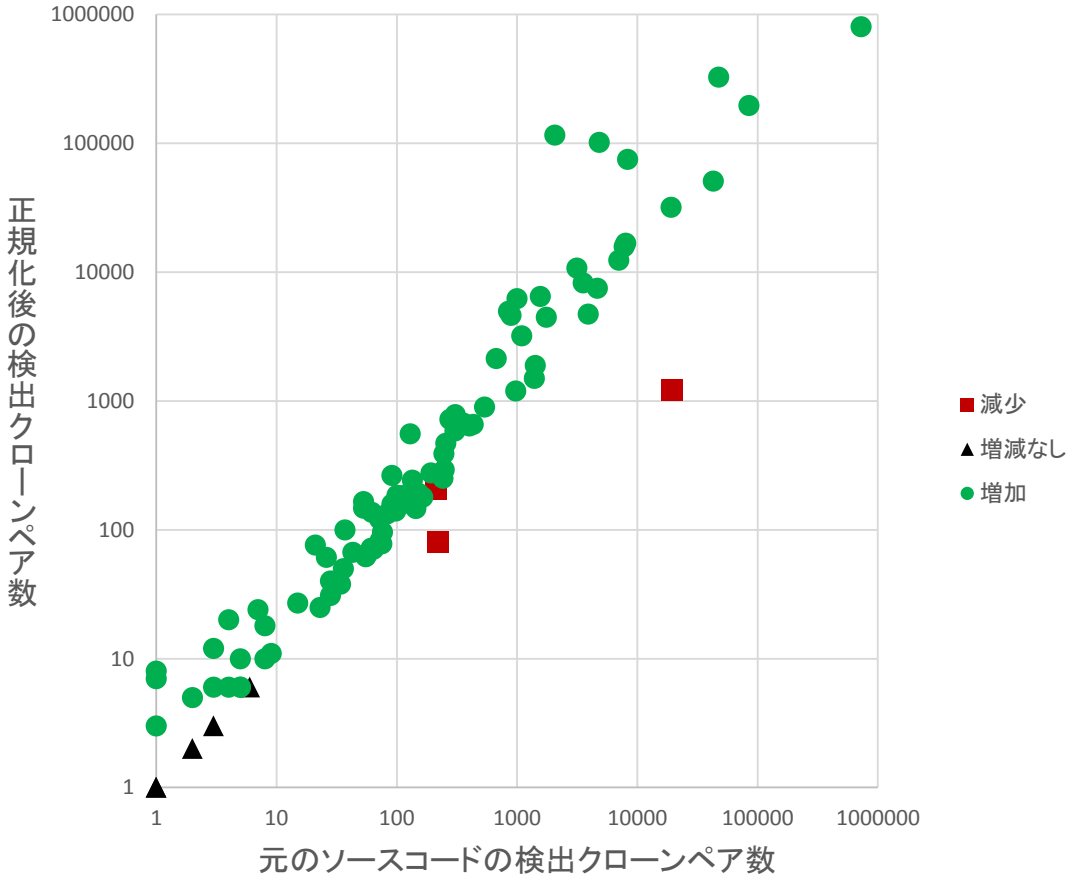


図 14: 構文の統一の実験結果

図 15: 文の順序統一により新しく検出されたクローンペアの例

6 考察

5 章で得られた実験結果に対する考察を行う．今回，実験を行った 100 個のプロジェクトの内，どちらの正規化でも検出クローン数が増加しているプロジェクトを 1 つ抽出し，それぞれの正規化後に検出されたクローンの分析を行った．分析を行ったプロジェクトの詳細を表 3 に示す．

表 3: 抽出したプロジェクト

プロジェクト名	検出クローンペア数			Java ファイル数	LOC
	正規化なし	文の順序統一	構文の統一		
ActionBarSherlock	304	305	586	157	41,980

6.1 文の順序統一

文の順序統一後のソースコードから検出された 305 組のクローンペアを目視で分析したところ，文の順序統一により新しく検出されたクローンペアは 1 組のみであった．そのクローンペアを図 15 に示す．

図 15 のクローンペアでは，文の順序統一前では 51 行目の文が挿入されているため，Type-3 のクローンとなっている．そのため，Type-2 までのクローンしか検出できない CCFinderX では，これらのコード片をクローンとして検出することができない．文の順序統一の正規化を行うことで，51 行目の挿入されていた文が 47 行目へと並べ替えられ，左のコード片の 32–39 行目と右のコード片の 46–55 行目が Type-2 のクローンとして検出された．

このように，文の順序統一により，順序入れ替わりクローンの他にも Type-3 のクローンが検出可能になることが考えられる．今回実験に使用した検出手法が CCFinderX であったため，正規化を行わなければこのような Type-3 のクローンは検出できなかった．しかし，Type-3 クローンは他の手法を使うことで検出可能であるため，正規化に新しく検出されても有益なクローンではない．また，抽出したプロジェクトから検出されたクローンの中に順序入れ替わりクローンは存在しなかった．

6.2 構文の統一

表 3 に示すとおり，構文の統一後のソースコードから検出された 586 組のクローンペアを目視で分析したところ，そのほとんどのクローンが構文の統一により字句数が CCFinderX の閾値である 50 トークンを超え，検出されたクローンであった．そのクローンペアの例を図 16 に示す．

図 16 の例では，構文の統一前におけるコード片はそれぞれトークン数が 50 を超えていないため，CCFinderX ではクローンとして検出されていない．構文を統一の正規化を行うことにより，複数の代入文が挿入され，コード片のトークン数が 50 を超えたため，501–509 行目のコード片と 217–225 行目のコード片がクローンとして検出された．

このことより，構文の統一によって，開発者が短く記述していたため，正規化を行わない場合では検出できなかったクローンが検出できることがわかった．しかし，抽出したプロジェクトから検出されたクローンの中に構文上の表記ゆれがあるクローンは存在しなかった．

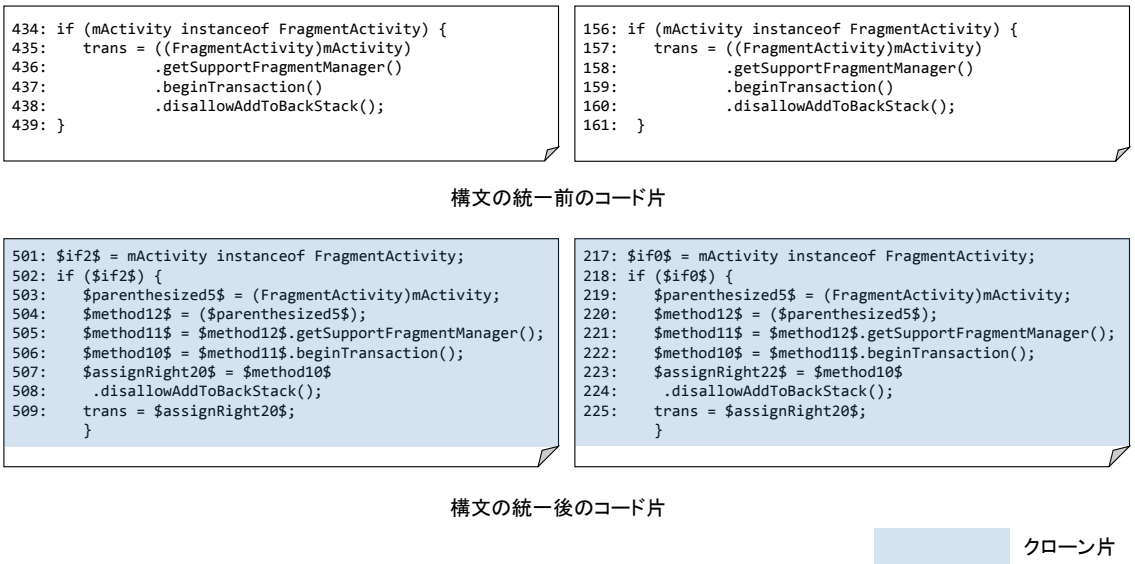


図 16: 構文の統一により新しく検出されたクローンペアの例

7 追加実験

前章までで，それぞれの正規化を行った効果を実験により調査した．このような正規化は独立して行うことができるが，2 種類の正規化を組み合わせることによって更なるクローンの検出が期待できる．本章では 2 種類の正規化を組み合わせることによってクローンの検出結果がどのように変化するかを調査する．クローン検出手法，検出対象は 5 章と同じである．

7.1 実験方法

追加実験では以下の手順で実験を行う．

1. 元のソースコードに対して文の順序統一を行う．
2. 文の順序統一を行ったソースコードに対して構文の統一を行う．
3. 元のソースコードと 2 種類の正規化を行ったソースコードそれぞれに対してクローンを検出する．

構文の統一を先に行った場合，文の順序統一で生成される拡張 PDG のノード数が増加し，正規化の時間が大幅に増加すると考えられる．そのため，今回の追加実験では文の順序統一を行った後に，構文の統一を行った．

7.2 実験結果

以上の手順で実験を行った結果を図 17 に載せる．5 章と同様に，横軸に正規化を行わない元のソースコードから検出されたクローンペア数，縦軸に文の順序統一の正規化を行ったソースコードから検出されたクローンペア数をとっている．また，縦軸，横軸ともに対数軸である．クローンペア数が減少したプロジェクトを四角，変化しなかったプロジェクトを三角，増加したプロジェクトを丸のマーカーで表している．表 4 に文の順序統一によりクローンペア数が増加した，変化しなかった，減少したプロジェクトの数を示す．

また，5 章と同様に正規化前後の検出クローンペア数の 2 群に対して，ウィルコクソンの符号順位検定を行ったところ，有意水準 1%の基で有意差が認められた．また，表 4 より増加しているプロジェクトがほとんどである．このことより，2 種類の正規化を組み合わせることによって検出されるクローンペア数が増加することがわかった．

7.3 考察

本実験においても，6 章と同じプロジェクトに対して，目視でのクローンの分析を行った．抽出したプロジェクトにおいて 2 種類の正規化後に検出されたクローンペア数は 587 組である．それらのクローンペアに対して目視で分析したところ，増えたクローンの内 1 組は 15 と同様のクローンペアであり，それ以外のクローンは構文の統一によって新しく検出されたクローンと同様のクローンであった．

このことより，検出されたクローンの中に 2 種類の正規化を組み合わせないと検出できないようなクローンは存在しなかった．

表 4: 検出クローンペア数の変化 (追加実験)

	増加	変化なし	減少
正規化 2 種	90 個	6 個	4 個

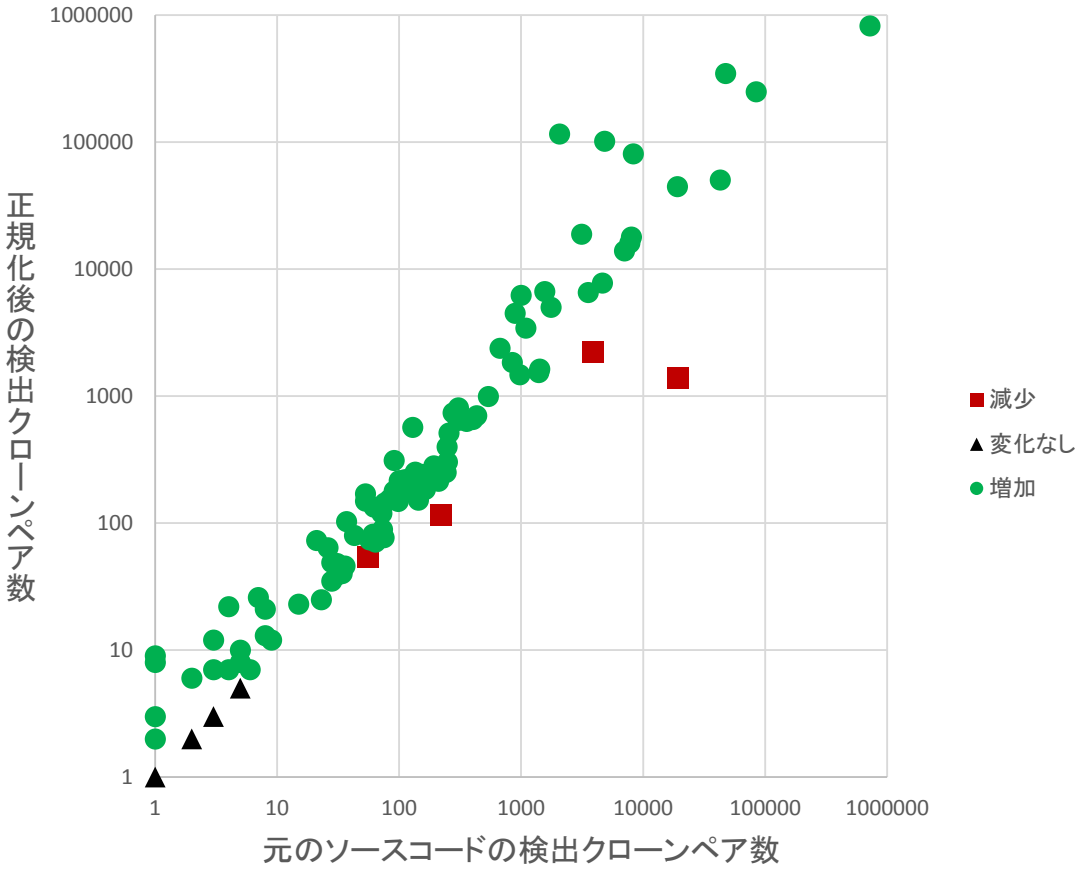


図 17: 追加実験の実験結果

8　実験の妥当性について考慮すべき点

本実験における妥当性について考慮すべき点として，以下が挙げられる．

8.1　対象言語

本研究の文の順序統一，構文の統一では対象となる言語は Java のみである．一部 Java でのみ適用可能であるヒューリスティックを用いているが，PDG や AST の概念はプログラミング言語に依存するものではなく，他の言語でも実現可能である．他の言語にもこの正規化を適用し，実験を行うことで，得られる結果が異なる可能性がある．

8.2　クローン検出手法

本研究ではクローン検出手法として Type-2 までのクローンを検出可能であり，一般的なクローン検出手法である CCFinderX を用いた．異なるクローン検出手法を用いた場合，検出されるクローンが異なるため，得られる結果が異なる可能性が高い．

8.3　並べ替え基準

本研究の文の順序統一においては，順序統一後の順序を選択する基準としてアルファベット順を用いた．文の順序統一時の条件を満たす複数のノードの順序から 1 つの順序を選択するために基準を設けるため，文の順序統一によってクローン片外の文がクローン片内に並べ替わってしまう可能性は排除できないが，異なる基準を用いて並べ替え後の順序を選択した場合，異なる結果が得られる可能性がある．

9 関連研究

Choi らはクローン検出手法の入力となるソースコードに対して，様々な種類の正規化を行うことで，クローン検出手法の実行時間がどのように変化するかを調査した [24]. その結果，一部の正規化をソースコードに対して行うことにより，クローン検出手法の実行時間が短くなることを確認している．Choi らの研究はソースコードに対して正規化を行うという点では共通しているが，本研究では実行時間ではなく，クローン検出結果がどのように変化しているかを調査するという点で異なる．

村上らは検出対象となるソースコードに対して繰り返し部分のたたみこみを行い，オーバーラップするクローンの検出を削減するクローン検出手法 FRISC を提案している [28]. 村上らは，データセットに対して実験を行い，適合率が 50%上昇したことを確認している．村上らの研究はソースコードに対して正規化を行うという点では共通しているが，本研究では適合率ではなく，検出されるクローンがどのように変化しているかに着目している点で異なる．

Komondoor らは PDG を用いたクローン検出手法を提案している [7]．Komondoor らの手法では順序入れ替わりクローンや，同一ファイルに存在し，片方のクローン片の終了行がもう片方のクローン片の開始行より大きいクローンである巻き付きクローンなど特殊なクローンが検出可能である．しかし，PDG を生成し，同型部分グラフを検出するため検出に長い時間を要する．また，検出されたクローンには開発者にとって有益でないクローンが多く含まれるとの報告もある [15].

10　おわりに

本研究ではクローン検出における正規化として，文の順序統一と構文の統一を行うことで，クローンの検出結果がどのように変化するのかを調査した．文の順序統一では，本研究のために依存関係の拡張を行った拡張 PDG を用い，同じ変数を持つ 2 文の順序が入れ替わらないように文の順序を統一した．構文の統一では，AST を用い，構文の単位である式 (Expression) の要素が単純でない場合，変数を宣言し，式をその変数に代入することで文を単純化し，ソースコードの構文を統一した．

上記の 2 種類の正規化を行うことで，クローンの検出結果がどのように変化するかを 100 個の Java プロジェクトに対して調査した．その結果，文の順序統一を行うことでクローン検出数は減少した．一方，構文の統一を行うことでクローン検出数は増加し，新しく検出されたクローンには開発者が短く記述していたため，正規化前には検出されなかったクローンが存在した．しかし，正規化後に新しく検出されたクローンの中に，順序入れ替わりクローンや構文上の表記ゆれがあるクローンは存在しなかった．このことより，クローンを検出するために，構文の統一は有効であると考えられるが，文の順序統一は有効でないと考えられる．

今後の課題としては，まず順序入れ替わりクローンや構文の表記ゆれがあるクローンに対する詳細な調査が挙げられる．具体的には，ソースコード中にこれらのクローンは存在しているが，正規化を用いても検出することができないのか，それともソースコード中にこれらのクローンが存在しないのかを調査する必要があると考えている．また，他のプログラミング言語に対応して実験を行い，本研究で得られた結果が一般的であるかを確かめる必要があると考えている．

謝辞

本研究を行うにあたり，理解あるご指導と日頃より大いな励ましを頂きました楠本 真二教授に心より感謝の念を申し上げます.

本研究の全過程を通して，熱心なご指導と的確なご助言を頂きました，肥後 芳樹准教授に深く感謝の念を申し上げます.

本研究に関して，日頃から多くのご助言と適切なご指摘を頂きました，柗本 真佑助教授に心より感謝いたします.

本研究に加えて，日頃から様々なご協力と心強いお力添えを頂きました，楠本研究室の皆様に深く感謝いたします.

最後に，本研究に至るまでに，講義，演習，実験等様々な場面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，この場を借りて心より御礼申し上げます.

参考文献

[1] C. K. Roy, J. R. Cordy, and R. 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] X. Wang, Y. Dang, L. Zhang, D. Zhang, E. Lan, and H. Mei. Can I Clone This Piece of Code Here? In *Proc. of the 27th International Conference on Automated Software Engineering*, pp. 170–179, 2012.

[3] 肥後芳樹, 楠本真二, 井上克郎. コードクローン検出とその関連技術. 電子情報通信学会論文誌. D, Vol. 91-D-1, No. 6, pp. 1465–1481, 2008.

[4] B. S. Baker. Parameterized Duplication in Strings: Algorithms and an Application to Software Maintenance. *SIAM Journal on Computing*, Vol. 26, No. 5, pp. 1343–1362, 1997.

[5] Z. Li, S. Lu, S.a Myagmar, and Y. Zhou. CP-Miner: Finding copy-paste and related bugs in large-scale software code. *IEEE Transactions on software Engineering*, Vol. 32, No. 3, pp. 176–192, 2006.

[6] C. K. Roy and J. R. Cordy. NICAD: Accurate Detection of Near-Miss Intentional Clones Using Flexible Pretty-Printing and Code Normalization. In *Proc. of the 16th International Conference on Program Comprehension*, pp. 172–181, 2008.

[7] 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.

[8] L. Jiang and Z. Su. Automatic mining of functionally equivalent code fragments via random testing. In *Proc. of the 18th International Symposium on Software Testing and Analysis*, pp. 81–92, 2009.

[9] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier. Clone detection using abstract syntax trees. In *Proc. of the 6th International Conference on Software Maintenance*, pp. 368–377, 1998.

[10] 井上克郎, 神谷年洋, 楠本真二. コードクローン検出法. コンピュータ ソフトウェア, Vol. 18, No. 5, pp. 529–536, 2001.

[11] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. Merlo. Comparison and Evaluation of Clone Detection Tools. *IEEE Transactions on Software Engineering*, Vol. 33, No. 9, pp. 577–591, 2007.

[12] H. Murakami, K. Hotta, Y. Higo, H. Igaki, and S. Kusumoto. Gapped code clone detection with lightweight source code analysis. In *Proc. of the 38th International Conference on Program Comprehension*, pp. 93–102, 2013.

[13] H. Sajnani, V. Saini, J. Svajlenko, C. K. Roy, and C. V. Lopes. SourcererCC: Scaling Code Clone Detection to Big-code. In *Proc. of the 38th International Conference on Software Engineering*, pp. 1157–1168, 2016.

[14] 肥後芳樹, 楠本真二, 井上克郎. コードクローン検出とその関連技術. 電子情報通信学会論文誌 D, Vol. 91, No. 6, pp. 1465–1481, 2008.

[15] 肥後芳樹, 楠本真二. プログラム依存グラフを用いたコードクローン検出法の改善と評価. 情報処理学会論文誌, Vol. 51, No. 12, pp. 2149–2168, 2010.

[16] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The Program Dependence Graph and Its Use in Optimization. *ACM Transactions on Programming Languages and Systems*, Vol. 9, No. 3, pp. 319–349, 1987.

[17] Java development tools. http://eclipse.org/jdt/.

[18] GitHub. https://github.com/.

[19] 石原知也, 堀田圭佑, 肥後芳樹, 井垣宏, 楠本真二. 大規模なソフトウェア群を対象とするメソッド単位でのコードクローン検出. 情報処理学会論文誌, Vol. 54, No. 2, pp. 835–844, 2013.

[20] J. Harder and N. Göde. Cloned code: stable code. *Journal of Software: Evolution and Process*, Vol. 25, No. 10, pp. 1063–1088, 2013.

[21] L. Barbour, F. Khomh, and Y. Zou. Late propagation in software clones. In *Proc. of the 27th International Conference on Software Maintenance*, pp. 273–282, 2011.

[22] CCFinderX. http://www.ccfinder.net/ccfinderxos-j.html.

[23] T. Kamiya, S. Kusumoto, and K. 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.

[24] E. Choi, H. Yoshida, Y. Higo, and K. Inoue. Proposing and Evaluating Clone Detection Approaches with Preprocessing Input Source Files. *IEICE Transactions on Information and Systems*, Vol. E98.D, No. 2, pp. 325–333, 2015.

[25] L. Barbour, F. Khomh, and Y. Zou. An empirical study of faults in late propagation clone genealogies. *Journal of Software: Evolution and Process*, Vol. 25, No. 11, pp. 1139–1165, 2013.

[26] M. Kim, V. Sazawal, D. Notkin, and G. Murphy. An empirical study of code clone genealogies. In *Proc. of the 10th European Software Engineering Conference*, Vol. 30, pp. 187–196, 2005.

[27] P. Weissgerber and S. Diehl. Identifying refactorings from source-code changes. In *Proc. of the 21st Automated Software Engineering*, pp. 231–240, 2006.

[28] 村上寛明, 堀田圭佑, 肥後芳樹, 井垣宏, 楠本真二. ソースコード中の繰返し部分に着目したコードクローン検出ツールの実装と評価. 情報処理学会論文誌, Vol. 54, No. 2, pp. 845–856, 2013.