

## プログラム依存グラフを用いた 増分的なコードクローン検出

肥 後 芳 樹<sup>†1</sup> 植 田 泰 士<sup>†1</sup>  
西 野 稔<sup>†1</sup> 楠 本 真 二<sup>†1</sup>

近年、増分的なコードクローン検出法が注目されている。増分的な検出法では、検出結果（とその中間生成物）はデータベースに保存され、次回以降の検出に利用される。このため、同じファイル群から何度もコードクローン検出を行う場合では、2回目以降の検出時間を大幅に短縮することができる。本論文では、プログラム依存グラフを用いた増分的な検出法を提案する。プログラム依存グラフを用いた検出は、行単位の検出などの他の検出法に比べて検出に長い時間を必要とするため、増分的な検出を行うメリットは大きい。また、提案手法に基づいてツールを開発し、適用実験を行った。実験の結果、提案手法は、従来のプログラム依存グラフを用いた検出法とほぼ同じコードクローンを検出するが、検出時間は大幅に短くなっていることを確認した。

### Incremental Code Clone Detection Using Program Dependency Graph

YOSHIKI HIGO,<sup>†1</sup> YASUSHI UEDA,<sup>†1</sup> MINORU NISHINO<sup>†1</sup>  
and SHINJI KUSUMOTO<sup>†1</sup>

Incremental code clone detection attracts much attention in the last few years. In incremental detections, code clone detection results persist by using databases, and it is used in next code clone detection. However, no incremental detection technique has been proposed for PDG-based detection, which requires much more time to detect code clones than line- or token-based detection. In this paper, we propose a PDG-based incremental code clone detection technique for improving practicality of PDG-based detection. A prototype tool has been developed based on the proposed method, and it has been applied to open source software. We confirmed that detection time extremely shortened and its detection result is almost the same as one of an existing PDG-based detection technique.

### 1. はじめに

コードクローンの存在はソフトウェアの保守作業に悪影響を与える要因の1つであるといわれている。コードクローンとは、ソースコード中に存在する重複コードであり、コピーアンドペーストなどのさまざまな理由により生成される<sup>7)</sup>。バグが顕在化した場合、ソースコード中の該当部分に対して修正が加えられる。しかし、そのコードクローンがシステム中に存在した場合は、それらに対しても同様の修正を検討する必要がある。修正を行った保守管理者が、コードクローンの存在に気づかなかつた場合、システムには潜在的なバグが残ってしまう可能性がある。

このような、コードクローンに起因する問題を解決・軽減するための1つの方法として、リファクタリングが挙げられる。リファクタリングを行うことにより、コードクローン部分は1つの関数やメソッドにまとめられるため、今後の修正作業により、その部分には不整合が起こらない。

これまでに、ソースコード中からコードクローンを自動的に検出するさまざまな手法が提案されている<sup>1)7)</sup>。既存の検出技術は、用いている技術により、行単位での検出、字句単位での検出、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、メトリクスやフィンガープリントなどを用いた検出に大別される。これらの中で、抽象構文木を用いた検出とプログラム依存グラフを用いた検出はリファクタリングに向いているといわれている<sup>14)</sup>。しかし、これらの検出技術は、他の検出技術に比べ検出に必要な計算量が高く、コードクローン情報が欲しい時に、瞬時にその情報を得ることが難しい。

また、近年、増分的なコードクローン検出<sup>\*1</sup>が注目を集めている。増分的なコードクローン検出とは、検出結果とその中間生成物をデータベースなどに保管し、次回以降の検出時に利用する検出手法である。検出対象ファイルが前回の検出時から更新されていない場合は、そのファイル自身は解析されず、ファイルに関する必要な情報はデータベースから取得される。このような枠組みを用いることによって、同じファイル集合から何度も検出を行う場合は、2回目以降の検出時間を大幅に短縮することができる。

増分的なコードクローン検出手法は、行単位での検出と字句単位での検出を行う手法がこれまでに提案されている<sup>5)9)</sup>。しかし、リファクタリングなどに有効な、抽象構文木を用い

<sup>†1</sup> 大阪大学 大学院情報科学研究科

Graduate School of Information Science and Technology, Osaka University

\*1 Incremental Code Clone Detection

## 2 プログラム依存グラフを用いた増分的なコードクローン検出

た検出やプログラム依存グラフを用いた検出はまだ提案されていない。また、これらの手法は、検出に必要な計算コストが高いため、増分的に検出を行うメリットは大きい。このような現在の状況を踏まえ、本論文では、プログラム依存グラフを用いた増分的なコードクローン検出手法を提案する。

### 2. 関連研究

Hummelらは、各行の内容と位置情報を蓄積して、同内容の行を検索することでコードクローンを検出する手法を提案し、検出ツール ConQAT に実装している<sup>9)</sup>。この手法では、まずソースコード中の各行に対して、変数名などに対する置換を行った上で、ハッシュ値を計算する。そして、そのハッシュ値とファイル名や行番号を組にした情報を蓄積する。ある行に対してコードクローンとなる行は、この情報をその行のハッシュ値を用いて検索することで、取得することができる。検出結果として出力するのは複数行からなるコードクローンであるが、それについてはあるコード片の各行とコードクローンになる行の集合から容易に構築することができる。

Gödeらは、接尾辞木を改良し、情報の追加や削除が容易な汎用接尾辞木というデータ構造を定義して、それを用いた字句単位のコードクローン検出手法を提案している<sup>5)</sup>。接尾辞木とは、文字列に対して定義される文字数と等しい個数の葉を持つ木構造であり、根からそれぞれの葉  $i$  への経路が文字列の  $i$  文字目から最後の文字までの部分文字列(接尾部)に対応する。接尾辞木を用いることで、文字列中の繰り返し部分を高速に検出することができる。コードクローン検出においては、コードクローンがソースコード中の繰り返し部分であることから、プログラム全体を1つの文字列として接尾辞木を構築し、検出を行う手法が用いられている<sup>3),10)</sup>。汎用接尾辞木は、1つの文字列ではなく文字列の集合に対して定義され、根から葉への経路は各文字列における接尾部に対応する。各ファイルをそれぞれ1つの文字列として汎用接尾辞木を構築することにより、ファイルの追加や削除が容易になり、増分的な検出が可能になる一方、コードクローン検出の精度には影響を与えないことを示している。

### 3. プログラム依存グラフ

プログラム依存グラフ(Program Dependency Graph, 以降 PDG)とは、ソースコード中の要素(文や条件節の式)間の依存関係を表現した有向グラフである。PDGのノードは要素、エッジは要素間の依存関係を表す。一般的なPDGでは、依存関係は制御依存とデータ依存の2種類が存在する。

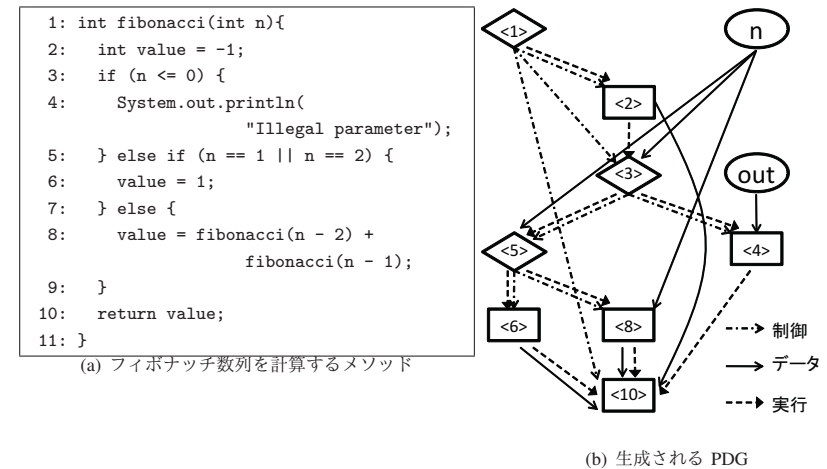


図 1 PDG の例  
Fig. 1 PDG Example

以下の全ての条件を満たすとき、要素  $s_1$  と  $s_2$  の間には制御依存関係が存在する。

- $s_1$  は条件節の式である
- $s_1$  の結果により、 $s_2$  が実行されるか否かが決定する

また、以下の全ての条件を満たすとき、要素  $s_3$  と  $s_4$  の間にはデータ依存関係が存在する。

- $s_3$  では、変数  $v$  を定義している(変数  $v$  に対して代入処理を行っている)
- $s_4$  では、変数  $v$  を参照している
- $s_3$  から  $s_4$  への経路の中に、変数  $v$  を再定義しないものが存在する

本研究では、上記の依存関係に加えて、実行依存も用いる。実行依存とは実行の順序を表す依存関係であり、要素  $s_5$  が実行された直後に要素  $s_6$  が実行される可能性がある場合、 $s_5$  から  $s_6$  へ実行依存エッジが引かれる。実行依存を PDG に追加することにより、コードクローンの検出能力が高まることが確認されている<sup>6)</sup>。

図 1 は、フィボナッチ数列の計算を行なうプログラムとその PDG である。PDG のノードの数字は、その要素のソースコード中での位置(行番号)を表している。なお、<1>のノードは、メソッドの入り口ノードを表し、そのノードは便宜上、条件節の式とみなされる。また、 $n$  や  $out$  は、それぞれ仮引数や共有変数を表すノードである。

### 3 プログラム依存グラフを用いた増分的なコードクローン検出

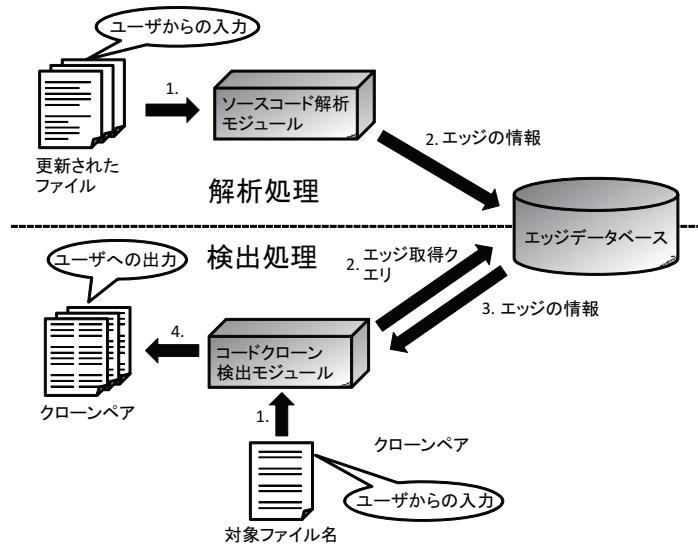


図2 提案手法の概観  
Fig. 2 Overview of the Proposed Method

#### 4. 提案手法

図2は提案手法の概観を表している。提案手法は解析処理と検出処理の2つから成る。解析処理では、ソースコードからPDGが構築され、その情報がデータベースに保存される。検出処理では、データベースからPDGの情報を取り出し、コードクローンの検出を行う。

以降、本章では、4.1節で提案手法で利用する種々の定義を行い、4.2節では解析処理について説明する。その後、4.3節で検出処理について説明する。

##### 4.1 定義

まず、プログラム依存グラフを定義する。

**定義1 (プログラム依存グラフ)** 本研究で用いるプログラム依存グラフは連結グラフであるため、そのグラフ ( $g$  とする) を構成するエッジ集合として定義することができる。

$$g := (e_1, e_2, \dots, e_n)$$

また、本論文では、エッジを下記のように定義する。

**定義2 (エッジ)** エッジ  $e$  の出発ノードを  $v_1$ 、到達ノードを  $v_2$ 、エッジの種類 (データ・

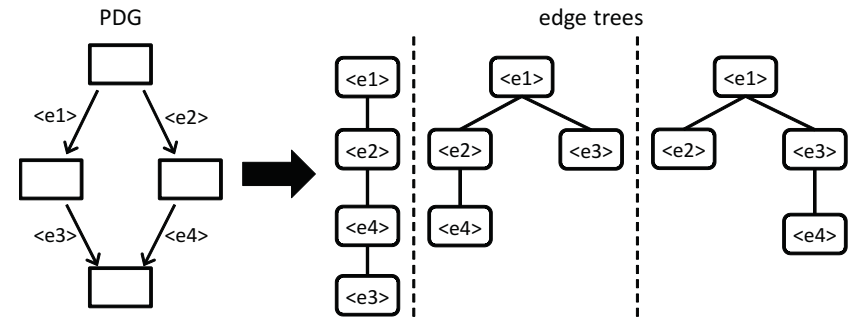


図3 1つのPDGから生成されるエッジツリー  
Fig. 3 Edge trees generated from a PDG

制御・実行) を  $t$  とすると、エッジ  $e$  は下記の式で表される。

$$e = (v_1, v_2, t)$$

次に、エッジの隣接関係を定義する。

**定義3 (隣接関係)** エッジ  $e_1 = (v_1, v_2, t_1)$ 、エッジ  $e_2 = (v_3, v_4, t_2)$  とすると、これら2つのエッジの隣接関係は下記の式により定義される。

$$incident(e_1, e_2) := (v_1 = v_3 \wedge v_2 \neq v_4) \vee (v_1 = v_4 \wedge v_2 \neq v_3)$$

$$\vee (v_2 = v_4 \wedge v_1 \neq v_3) \vee (v_2 = v_3 \wedge v_1 \neq v_4)$$

つまり、エッジ  $e_1$  と  $e_2$  が1つのノードを共有していれば、 $incident(e_1, e_2)$  は真となる。

次に、隣接関係を用いてエッジ間の経路を定義する。

**定義4 (エッジ間の経路)** 連結グラフであるため<sup>\*1</sup>、任意のエッジ  $e_i$  とエッジ  $e_j$  の間には経路が存在する。経路が複数存在する場合もあるため、エッジ  $e_i$  と  $e_j$  の間の経路集合を、 $PATHS(e_i, e_j)$  と表す。なお、ここでの経路は閉路を含まないものとする。閉路を含まない経路とは、その経路上に存在する全てのノードを一度しか辿らないものである。

次に、エッジツリーという新しいデータ構造を定義する。

**定義5 (エッジツリー)** エッジツリーのノードは、プログラム依存グラフ  $g = \{e_1, e_2, \dots, e_n\}$  を構成するエッジであり、ツリーのエッジはグラフのエッジ間の隣接関係である。また、エッジツリーのノード数はグラフのエッジ数と等しい (グラフ中のすべてのエッジをノードとし

\*1 厳密には、参照されていない引数があると孤立ノードが存在するが、そのような孤立ノードは無視する。

#### 4 プログラム依存グラフを用いた増分的なコードクローン検出

て持つ)。

図3はエッジツリーの簡単な例を表している。この図に示すように、1つのPDGから複数のエッジツリーが生成される。図3はエッジ $e_1$ を根とする3つのエッジツリーを表している。このように1つのエッジツリーが全ての隣接関係を表現しているわけではない。

次に、エッジツリーの集合を定義する。

**定義6(エッジツリーの集合)**  $TREES(g)$  は、プログラム依存グラフ  $g$  から生成される全てのエッジツリーの集合である。

また、ここで、エッジツリー上の経路を定義する。

**定義7(エッジツリー上の経路)**  $treepath(t, e)$  は、エッジツリー  $t$  において、根からノード  $e$  までの経路である。

定義4と定義7を用いると、定理1は自明である。

**定理1**  $e_r$  をエッジツリー  $t$  の根とすると、下記の式が成り立つ。

$$treepath(t, e) \in PATH(e_r, e)$$

また、連結グラフでは、 $PATH(e_r, e) \neq \emptyset$  は常に真となる。つまり、あるエッジから他の任意のエッジへの経路が少なくとも1つ存在する。つまり、連結グラフからは少なくとも1つのエッジツリーが生成される。

次に、グラフエッジの同値関係を定義する。

**定義8(グラフエッジの同値関係)** 2つのエッジ  $e_1 = (v_1, v_2, t_1)$  と  $e_2 = (v_3, v_4, t_2)$  の同値関係は、下記の式で定義される。

$$e_1 \equiv e_2 := (t_1 \equiv t_2) \wedge (v_1 \equiv v_3) \wedge (v_2 \equiv v_4)$$

$e_1$  と  $e_2$  の依存関係の種類が等しいとき  $t_1 \equiv t_2$  は真となる。 $v_1 \equiv v_2$  は、コードクローン検出の状況に応じた定義をすべきである。たとえば、完全に字面が等しいコード片のみをコードクローンとして検出したい場合は、ノード  $v_1$  と  $v_2$  を表す文字列を直接比較すべきである。また、ある程度の差違も許容して検出したい場合は、何らかの正規化処理を行った後に、比較すべきである。

定義8を用いて、ツリーノード間の経路の同値関係を定義する。

**定義9(ツリーノード間の経路の同値関係)** 2つのツリーノード間の経路  $p_1 = (e_1, e_2, \dots, e_n)$  と  $p_2 = (f_1, f_2, \dots, f_m)$  において、その同値関係は下記の式で定義される。

$$p_1 \equiv p_2 := (|p_1| = |p_2|) \wedge \forall i (e_i \equiv f_i)$$

つまり、ツリーノード間の経路  $p_1$  と  $p_2$  が同数のノードを持ち、各ノードのペアが同値であれば、 $p_1$  と  $p_2$  は同値である。

次に、エッジツリーの同値関係を定義する。

**定義10(エッジツリーの同値関係)** 2つのエッジツリー  $t_1 \in TREES(g_1)$  と  $t_2 \in TREES(g_2)$  において、その同値関係は下記の式で定義される。なお、グラフ  $g$  のエッジ数を  $|g|$  と表す。

$$t_1 \equiv t_2 := (|g_1| = |g_2|) \wedge \exists (e_1, e_2, \dots, e_{|g_1|}) \exists (f_1, f_2, \dots, f_{|g_2|}) \forall k \left( \bigcup_{1 \leq i \leq |g_1|} \{e_i\} = g_1 \right. \\ \left. \wedge \bigcup_{1 \leq j \leq |g_2|} \{w_j\} = g_2 \wedge treepath(t_1, e_k) \equiv treepath(t_2, f_k) \right)$$

つまり、エッジツリー  $t_1$  と  $t_2$  はエッジの数が等しく、 $t_1$  の任意のノードから根への経路と同値関係にある経路が  $t_2$  に存在しており、かつ  $t_2$  の任意のノードから根への経路と同時関係にある経路が  $t_1$  に存在している場合に、同値となる。

上記の定義を用いることによって、クローンペアは次のように定義される。

**定義11(クローンペア)** プログラム依存グラフ  $g_1$  と  $g_2$  上のサブグラフ  $s_1$  と  $s_2$  において、下記の式が真になる場合に、 $s_1$  と  $s_2$  はクローンペアとなる。

$$clonepair(s_1, s_2) := (s_1 \cap s_2 = \emptyset) \wedge \exists t_1 \exists t_2 (t_1 \in TREES(s_1) \wedge t_2 \in TREES(s_2) \wedge t_1 \equiv t_2)$$

定義11のクローンペアの中には出力する必要のないものが含まれている。そのようなコードクローンを取り除いた定義を以下に示す。

**定義12(出力されるクローンペア)**

$outputclonepair(s_1, s_2) := clonepair(s_1, s_2) \wedge \neg \exists (s'_1, s'_2) (clonepair(s'_1, s'_2) \wedge s_1 \subset s'_1 \wedge s_2 \subset s'_2)$   
提案手法では、 $outputclonepair$  を満たすクローンペアが出力される。

#### 4.2 解析処理

解析処理の入力と出力を以下に示す。

入力 更新されたソースファイル

出力 入力ファイル内のグラフエッジ群

図2の上部は解析処理の概要を表している。解析処理では、以下の手順で行われる。

1. 更新されたソースファイルが解析モジュールへと渡される
2. 解析モジュールはソースファイルからPDGを構築し、それに含まれるエッジの情報をエッジデータベースに保存する

手順2.では、既にそのファイルの古いバージョンの情報が保存されている場合は、その情報を削除したのち、新しいバージョンの情報が追加される。

#### 4.3 検出処理

検出処理の入力と出力を以下に示す。

## 5 プログラム依存グラフを用いた増分的なコードクローン検出

入力 コードクローン検出対象ファイル名

出力 入力として与えられたファイルに関連するクローンペアの集合

図2の下部は検出処理の概要を表している。検出処理は下記の手順で行なわれる。

1. ユーザはコードクローンを検出したいファイルを指定する
2. 検出モジュールはそのファイル名をクエリとしてエッジデータベースに問い合わせる
3. エッジデータベースは、クエリのファイル内に存在するエッジと等価なエッジ群を返す
4. 検出モジュールは、エッジデータベースより返されたエッジ群から、クローンペアを構築しユーザに提示する。

検出処理を実現するためには、エッジ群からクローンペアを生成するアルゴリズムが必要である。まず、メソッドを入力として、そのメソッドと関連のあるクローンペアの集合を出力するアルゴリズムを Algorithm 1 に示す。なお、あるメソッドと関連のあるクローンペアとは、そのクローンペアのコードクローンのうち少なくとも一方がそのメソッド内に存在しているクローンペアである。

---

### Algorithm 1 detect ( $m$ )

---

**Input:**  $m$ : メソッド

**Output:**  $C$ : メソッド  $m$  に関連するクローンペアの集合

- 1:  $C \leftarrow \emptyset$
  - 2: **for all**  $e_1$  such that  $e_1 \in m$  **do**
  - 3:   **for all**  $e_2$  such that  $e_2 \equiv e_1 \wedge e_2 \neq e_1$  **do**
  - 4:      $C \leftarrow C \cup \text{create}(e_1, e_2, \emptyset)$
  - 5:   **end for**
  - 6: **end for**
  - 7: **return**  $C$
- 

アルゴリズムの主要な部分を説明する。

- 1行目 出力するクローンペアの集合を保存する変数  $C$  を空集合で初期化する。
- 2行目 入力として与えられたメソッド  $m$  に含まれるエッジを1つずつ取ってくる（このエッジを  $e_1$  とする）。なお、この処理は、メソッド ID を元にしたデータベースへの問い合わせで実現している。入力として与えられたメソッドのソースコードが解析されるわけではない。

- 3行目 データベース検索を用いて、エッジ  $e_1$  と等価なエッジ（このエッジを  $e_2$  とする）を1つずつ取ってくる。

- 4行目 エッジ  $e_1$  と  $e_2$  を引数として、アルゴリズム *create* を用いてクローンペア情報を構築し、 $C$  に加える。

---

### Algorithm 2 create ( $e_1, e_2, U$ )

---

**Input:**  $e_1, e_2$ : エッジのペア,  $U$ : すでにチェックしたエッジの集合

**Output:**  $(S_1, S_2)$ :  $u_1 \in S_1$  かつ  $u_2 \in S_2$  である *outputclonepair*

- 1:  $(S_1, S_2) \leftarrow (\{e_1\}, \{e_2\})$
  - 2:  $U \leftarrow U \cup \{e_1, e_2\}$
  - 3: **for all**  $e_x$  such that  $\text{incident}(e_1, e_x) \wedge e_x \notin U$  **do**
  - 4:   **for all**  $e_y$  such that  $e_x \equiv e_y \wedge \text{incident}(e_2, e_y) \wedge e_y \notin U$  **do**
  - 5:      $(S_x, S_y) \leftarrow \text{create}(e_x, e_y, U)$
  - 6:      $(S_1, S_2) \leftarrow (S_1 \cap S_x, S_2 \cap S_y)$
  - 7:   **end for**
  - 8: **end for**
  - 9: **return**  $(S_1, S_2)$
- 

次にアルゴリズム *create* (Algorithm 2) を説明する。このアルゴリズムは、エッジのペアを入力として受け取り、そのエッジを含むクローンペアを出力する。以下に *create* アルゴリズムの主要な部分を説明する。

- 1行目 引数で与えられた与えられたエッジのペア ( $e_1$  と  $e_2$ ) を出力するクローンペアの要素 ( $S_1$  と  $S_2$ ) とする。
- 2行目 引数で与えられた2つのエッジをチェック済みエッジ集合に加える。
- 3行目  $e_1$  と隣接関係にあり、またチェックされていないエッジを1つずつ取ってくる（このエッジを  $e_x$  とする）。
- 4行目  $e_2$  と隣接関係にあり、まだチェックされておらず、 $e_x$  と等価なエッジを1つずつ取ってくる（このエッジを  $e_y$  とする）。
- 5行目  $e_x, e_y$ , およびこれまでにチェックしたエッジの集合を引数として、アルゴリズム *create* を再起呼び出しする。

## 6 プログラム依存グラフを用いた増分的なコードクローン検出

6行目 5行目の結果得られた、クローンペアを成すエッジの集合を  $S_1$  と  $S_2$  に加える。

### 5. 実 装

提案手法をツールとして実装した。規模は約 10,000 行であり、PDG の構築に必要なソースコード解析には、著者らの研究グループで開発している MASU<sup>13)</sup> を利用した。現在のところ、対象言語は Java のみであるが、ソースコードから PDG を構築するモジュールを実装すれば、他のプログラミング言語に対しても適用することができる。

提案手法では、PDG ノードの同値関係については定義していないため、実装したツールでは、以下の定義を用いている。

定義 13 (PDG ノードの同値関係)  $s_1$  と  $s_2$  をプログラムの要素とし、 $v_1$  と  $v_2$  はそれらを表す PDG のノードとすると、同値関係は以下の式で定義される。

$$v_1 \equiv v_2 := \text{normalize}(s_1) = \text{normalize}(s_2)$$

*normalize* は正規化処理を行う関数であり、変数とリテラルを下記のルールで正規化を行う。  
変数 変数毎に特殊な字句が用意される。型が同じであっても違う変数は違う字句に置換される。

リテラル リテラルの型毎に特殊な字句が用意される。型が同じであれば、値が違っていても同じ字句に置換される。

提案手法では、PDG に含まれるエッジはデータベースに保存される。エッジはデータベースに登録される際に、そのハッシュ値も登録される。そして、そのハッシュ値を用いてデータベースからの等価なエッジの取得処理が行われる。

実装では、ハッシュ値を求めるアルゴリズムとして MD5 を用いている。定義 2 に示すように、本論文では、エッジ  $e$  は  $(v_1, v_2, t)$  で表現している。よって、 $e$  のハッシュ値は  $v_1, v_2$ , および  $t$  を結合したバイト列の MD5 値として求める。

### 6. 評 価

本章では、提案手法を評価するために行った実験について述べる。この実験は以下の 2 つの実験からなる。

実験 A 提案手法の検出速度を評価するための実験。効率的にコードクローンが検出できているかどうかを確認する。

実験 B 提案手法の検出精度を評価するための実験。PDG を用いた検出法として、検出結果が適切であるかどうかを確認する。

なお、この実験は、下記に示す仕様のワークステーションを用いて行った：

CPU Intel Xeon E5405 (quad-core, 2.0GHz)

Memory 8GB

OS Windows 7 Enterprise (64bit)

実験対象は、Ant である。Ant を実験対象とした理由を下記に示す。

- Ant は Java 言語を用いて開発されている。現在のところ、提案手法を実装したツールが対応しているのは Java 言語のみである。
- Ant の最新リビジョンの規模が約 20 万行あり、また開発期間も数年と実験対象規模として十分なため。

#### 6.1 実験 A : 検出時間の評価

この実験では、下記の 2 つの項目について評価を行った。

項目 1 Ant の全てのリビジョンからコードクローン検出を行い、その時間を測定した。このような検出処理は、コードクローンに関係する研究で頻繁に行われている。近年、コードクローンの存在がどの程度ソフトウェア保守作業を阻害しているのかを調査する試みが行われている<sup>8),11),12)</sup>。また、コードクローンの履歴情報を可視化する手法も提案されている<sup>2)</sup>。このような研究では、増分的な検出法を用いることによって、効率的に検出処理を行える。

項目 2 あるコード片からバグを検出した場合は、そのコード片とコードクローンになっている部分についても同様のバグがないかを調査することが望ましい。本項目ではそのような状況を想定して、ある 1 つのファイルが指定されたときに、それとコードクローンになっている部分を検出するのに必要な時間を測定する。

##### 6.1.1 項目 1 の評価

この項目の評価は下記の手順で行われた。

STEP1 最初のリビジョンをチェックアウトする。

STEP2 最初のリビジョンからコードクローンを検出する。この STEP では、全てのソースファイルに対して解析処理と検出処理が実行される。

STEP3 次のリビジョンをチェックアウトする。

STEP4 STEP3 でチェックアウトしたリビジョンからコードクローン検出を行う。この STEP では、アップデートされたファイルについてのみ解析処理と検出処理が実行される。

STEP5 もし次のリビジョンが存在すれば、STEP3 に戻る。

この評価では、STEP2 と STEP4 の合計時間を計測した。その結果、5,903 リビジョンか

## 7 プログラム依存グラフを用いた増分的なコードクローン検出

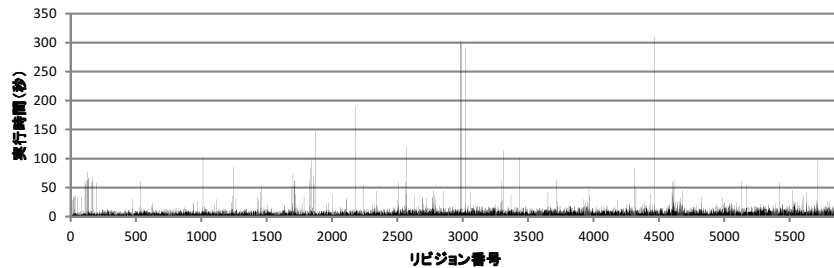


図4 Antの各リビジョンからコードクローンを検出するのに必要な時間  
Fig. 4 Elapsed time to detection for every revision

ら15時間2分でコードクローン検出を完了できた。なお、PDGベースのコードクローン検出ツールScorpio<sup>6)</sup>を利用して、増分的でない検出を各リビジョンから行ったところ、約3,300リビジョンを1日で解析できたが、システムが大きくなるに従って検出時間も長くなり、全てのリビジョンからの検出を終えるのに約4日半を要した。

また、リビジョン毎の検出時間を図4に示す。この図より、ほとんどのリビジョンにおいて、コードクローン検出が10~20秒程度で完了していることがわかる。その一方で、いくつかのリビジョンでは、1分以上の長い時間を必要としていた。これらのリビジョンでは多くのファイルが更新されており、最も数が多いもので733のファイルが更新されていた。

### 6.1.2 項目2の評価

項目2の評価は、Antの最新リビジョン(リビジョン番号:1,075,177, ファイル数:804, 行数:203,580)に対して、下記の手順で行った。

**STEP1** 全てのファイルに対して解析処理を実行

**STEP2** 各ファイルに対して検出処理を実行

表1は、STEP1とSTEP2に要した時間を表している。STEP1ではデータベースを作成するために、3分18秒が必要であった。STEP2では、単一ファイルに対する検出処理は最大

表1 項目2の実行時間  
Table 1 Detection time in Context 2

処理	STEP1	STEP3		
		平均値	中央値	最大値
実行時間	3分18秒	2.9秒	2.2秒	12.8秒

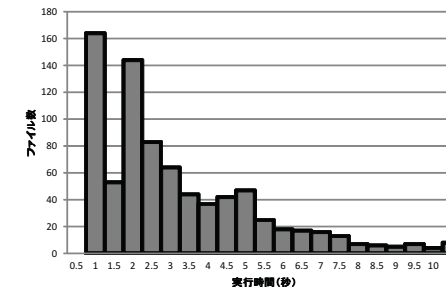


図5 最新リビジョンの各ファイルに対する検出時間  
Fig. 5 Detection time for every source files of the latest revision

でも12.8秒であり、項目2の状況で実用的に利用可能な検出速度といえる。

図5は、各ファイルに対する検出時間のヒストグラムを表している。X軸は、500ミリ秒毎に区切られており、検出時間が500n以上500(n+1)未満であったファイルが、左からn番目の区域に入る。例えば、ラベルが1の区域は、検出時間が500ミリ秒以上1秒未満であったファイルの数を表している。この図より、多くのファイルに対する検出処理が数秒以内で終わっていることがわかる。

### 6.2 実験B: 検出精度の評価

検出精度の評価では、既存のPDGを用いた検出ツールであるScorpio<sup>6)</sup>と検出結果の比較を行った。この評価でも、実験Aの項目1と同様に、Antの最新リビジョンを検出対象とした。比較対象としてScorpioを用いたのは以下の2つの理由のためである。

- Scorpioは提案手法と同じく実行依存付きのPDGを用いて検出処理を行うため、同型部分グラフをコードクローンとして検出する方法と、提案手法の検出法の差違を適切に比較することができる。
- Scorpioは著者らの研究グループで開発されたツールであり、検出の設定方法を熟知しているため、提案手法となるべく同じ設定にして検出処理を行うことができる。

また、提案手法とScorpioが検出したコードクローンが一致するかどうかの判定には、Bellonらが提案したgood値とok値<sup>4)</sup>を用いた。good値とok値の詳細な定義については紙面の都合上割愛するが、いずれも、2つのクローンペアのオーバーラップの程度を0%~100%の割合で表す指標である。この実験では、これらの値の閾値として70%を用いた。

804個のソースファイルから、提案手法は831個、Scorpioは724個のクローンペアを検

## 8 プログラム依存グラフを用いた増分的なコードクローン検出

出した。表2は、Scorpioが検出したクローンペアの集合を正解集合とした場合の、提案手法の適合率と再現率を表している。この表より、適合率と再現率は共に1に近い値であり、提案手法の検出結果はScorpioの検出結果と近いことがわかる。つまり、提案手法は従来のPDGを用いた検出法と検出結果が極めて似ている。

### 7. おわりに

本論文では、PDGを用いた増分的なコードクローン検出手法を提案した。また、提案手法を実装したプロトタイプを紹介し、それを用いて行った実験についても述べた。この実験により、提案手法は、開発履歴の全リビジョンからのコードクローン検出を想定した実験では、従来の約14%の時間で検出を完了することができ、バグを発見したファイルとのコードクローン検出を想定した実験では、平均2.9秒で検出処理を終えることができた。また、提案手法と増分的な検出法ではない従来手法との検出結果の比較では、従来手法の検出結果を正解集合とした場合の、提案手法の検出結果は再現率が99%以上、適合率が92%以上となり、提案手法は従来手法とほとんど結果が変わらないことを示した。

今後は、さらに検出時間を短縮するために手法を改良する予定である。現在は、コードクローンの検出は、対象ファイルを与えることで行っているが、メソッドを与えることで検出ができるように改良を行う。バグ修正などのソースコード変更では、1つのソースファイル全体に渡って変更が加えられる場合よりも、その中の一部のメソッドのみが変更される場合が多いと著者らは考えている。メソッド単位での検出を実現することによって、前回までの検出結果を再利用できる割合が増えるため、より高速に検出処理を行うことができる。

### 謝 辞

本研究は一部、文部科学省「次世代IT基盤構築のための研究開発」(研究開発領域名:ソフトウェア構築状況の可視化技術の開発普及)の委託に基づいて行われた。また、日本学術振興会科学研究費補助金基盤研究(A)(課題番号:21240002)および萌芽研究(課題番号:23650014)、文部科学省科学研究費補助金若手研究(B)(課題番号:22700031)の助成を得た。

表2 提案手法の再現率と適合率

Table 2 Precision and recall of the proposed method

評価項目	good	ok
適合率	0.922	0.970
再現率	0.990	1.000

### 参 考 文 献

- 1) : Clone Detection Literature, <http://www.cis.uab.edu/tairasr/clones/literature/>.
- 2) Adar, E. and Kim, M.: Visualization and Exploration of Code Clones in Context, *Proc. of the 29th International Conference on Software Engineering*, pp.762–766 (2007).
- 3) Baker, B.S.: Parameterized Duplication in Strings: Algorithms and an Application to Software Maintenance, *SIAM Journal on Computing*, Vol.26, No.5, pp.1343–1362 (1997).
- 4) Bellon, S., Koschke, R., Antoniol, G., Krinke, J. and Merlo, E.: Comparison and Evaluation of Clone Detection Tools, *IEEE Transactions on Software Engineering*, Vol.31, No.10, pp.804–818 (2007).
- 5) Göde, N. and Koschke, R.: Incremental Clone Detection, *Proc. of the 13th European Conference on Software Maintenance and Reengineering* (2009).
- 6) 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).
- 7) 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.J91-D, No.6, pp.1465–1481 (2008).
- 8) Hotta, K., Sano, Y., Higo, Y. and Kusumoto, S.: Is Duplicate Code More Frequently Modified than Non-duplicate Code in Software Evolution?: An Empirical Study on Open Source Software, *Proc. of the 4th International Joint ERCIM/IWPSE Symposium on Software Evolution*, pp.73–82 (2010).
- 9) Hummel, B., Juergens, E., Heinemann, L. and Conradt, M.: Index-Based Code Clone Detection: Incremental, Distributed, Scalable, *Proc. of the 26th IEEE International Conference on Software Maintenance*, pp.1–9 (2010).
- 10) Kamiya, T., Kusumoto, S. and Inoue, K.: 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).
- 11) Krinke, J.: Is Cloned Code more stable than Non-Cloned Code?, *Proc. of the 8th IEEE International Working Conference on Source Code Analysis and Manipulation*, pp.57–66 (2008).
- 12) Lozano, A. and Wermelinger, M.: Assessing the effect of clones on changeability, *Proc. of the 24th International Conference on Software Maintenance*, pp.227–236 (2008).
- 13) 三宅達也, 肥後芳樹, 楠本真二, 井上克郎: 多言語対応メトリクス計測プラグイン開発基盤MASUの開発, *電子情報通信学会論文誌 D*, Vol.J92-D, No.9, pp.1518–1531 (2009).
- 14) Rysselberghe, F. and Demeyer, S.: Evaluating Clone Detection Techniques from a Refactoring Perspective, *Proc. of the 19th IEEE International Conference on Automated Software Engineering*, pp.336–339 (2004).