情報処理学会論文誌 Vol. 51 No. 12 2149–2168 (Dec. 2010)

創立 50 周年記念論文

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

肥後芳樹[†1] 楠本真二[†1]

これまでにさまざまなコードクローン検出手法が提案されているが，すべての面において他の検出手法よりも優れているものはない．各検出手法は一長一短であり，コードクローン検出を行う状況に応じて適切な検出技術を用いることが重要である．プログラム依存グラフを用いた検出の長所は非連続コードクローンを検出できることである．しかし，その反面，連続コードクローンについては，他の検出技術に比べて検出能力が低い．また，検出に必要な計算コストが高いため，実規模ソフトウェアに対しては適用が難しいという弱点もある．本論文では，これらの弱点を改善するための手法を提案する．提案手法を組み合わせ，1 つのコードクローン検出手法として用いることにより，実規模ソフトウェアから実用的な時間でより適切にコードクローン検出を行うことができる．実際に，提案手法を検出ツールとして実装し，複数のオープンソースソフトウェアに対して評価を行い，その有用性を確認した．

Improvement and Evaluation of Code Clone Detection Using Program Dependency Graph

Yoshiki Higo[†1] and Shinji Kusumoto[†1]

At present, there are various kinds of code clone detection techniques. Each detection technique has merits and demerits, and none of them is superior to any other techniques in every way. PDG-based detection is suitable to detect non-contiguous code clones meanwhile other detection techniques are not suited to detect them. However, PDG-based detection has lower performance for detecting contiguous code clones than line- or token-, or AST-based techniques. Moreover, PDG-based detection is time-consuming, and it is difficult to apply it to actual software systems. This paper proposes a new PDG-based detection method, which can be applied to actual software systems. Also, the proposed method improves a traditional detection algorithm, and it can detect code clones that cannot be detected by existing PDG-based detection methods. We confirmed the usefulness of the proposed method by applying it to multiple real software systems.

1. はじめに

近年，ソフトウェア工学における研究対象の 1 つとしてコードクローンが注目を集めており，これまでにコードクローンに関係するさまざまな研究がなされている[1),22)]．コードクローンとは，ソースコード中に存在する同一または類似したコード片を表す．コードクローンは，コピーアンドペーストや定型処理などのさまざまな理由によりソースコード中に作り込まれる[6)]．

一般的に，コードクローンの存在はソフトウェア保守を困難にするといわれている．たとえば，あるコード片に対して変更を加える場合，もしその部分がコードクローンであれば，対応するすべてのコードクローンに対しても同様の修正の是非を検討しなければならない．また，修正すべきコード片を見落としてしまう危険性も含んでいる．実際にコードクローンが保守に悪影響を与えていたとの報告もされている．Monden らは，COBOL で記述されたソフトウェアに対してコードクローンとそのソースファイルの改版数の関係を調査している[21)]．その調査では，80%以上がコードクローンになっているソースファイルや，200 行以上のコードクローンが存在するソースファイルは，他のソースファイルに比べて改版数が多くなる傾向があることが報告されている．

また，リファクタリングの先駆者 Fowler は，“重複コードは最も優先して取り除くべき不吉な匂いの 1 つ”，と述べている[10)]．文献 10) では，コードクローンの除去方法についても言及しており，たとえば，ある 1 つのクラス内にコードクローンが存在している場合は，重複部分を新たな内部メソッドとして抽出すればよく，同一クラスから派生した複数のクラス間にコードクローンが存在している場合は，重複部分を共通の基底クラスに引き上げることによってコードクローンを除去することが可能である，と述べている．

対象ソフトウェアが大きい場合，チェックすべき個所が膨大な数になること，および人間がすべての重複部分を認識しておくことが現実的ではないため，ツールを用いたコードクローン検出が行われる．コードクローン検出ツールは，対象のソースコードを入力として受

†1 大阪大学大学院情報科学研究科
Graduate School of Information Science and Technology, Osaka University

け取り，その中に含まれるコードクローンの位置情報を出力する．しかし，コードクローンの厳密で普遍的な定義は存在しない．これまでにさまざまなコードクローン検出手法が提案されているが，各手法は独自のコードクローンの定義を持ち，その定義に基づいて検出ツールが作成されている．そのため同じソースコードから検出を行った場合でも検出結果は検出手法ごとに異なる．

既存の検出手法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，メトリクスを用いた検出に大別される[12]．各検出技術は一長一短でありすべての面において他の技術よりも優れているものはない[7),8)]．コードクローン検出を行う状況に応じて，適切な検出技術を選択することが重要である．

プログラム依存グラフ（Program Dependency Graph，以降 PDG）を用いた検出の長所と短所を示す[7),8),17),18)]．

長所　非連続コードクローン（Non-contiguous Code Clone）を検出することができる．非連続コードクローンとは，1 つのコードクローンを構成する要素（プログラムの文や式など）が必ずしもソースコード上で連続していないコードクローンを指す．コピーアンドペーストしたコード片に対して修正漏れが起こるという報告があり[5)]，そのような部分は非連続コードクローンとなるため，非連続コードクローンを検出することはソフトウェア保守の観点から重要である．

短所　行単位，字句単位，および抽象構文木を用いた検出に比べると連続コードクローン（Contiguous Code Clone）の検出能力が劣り，非連続コードクローンについては誤検出が多く含まれるため，検出の精度が低い．また，検出に必要な計算コストが高く，実規模ソフトウェアに対しては適用が難しい．

本論文では，これらの短所を改善した，PDG を用いたコードクローン検出法を提案する．提案手法のキーアイデアは次のとおりである．

PDG 頂点間への実行依存関係の導入と双方向スライスの利用　連続コードクローンの検出能力を高めることを目的とした提案である．従来手法に比べて，プログラムスライスでたどる頂点の範囲を拡大することができる．

スライス基点とする PDG 頂点数の削減　計算コストの削減を目的とした提案である．PDG を用いた手法の計算コストが高いのは 2 つの原因がある．1 つめの原因は，コードクローンを検出するための（同形部分グラフを特定するための）基点となる PDG 頂点の数が非常に多いことである．2 つめの原因は，同形部分グラフを検出すること自体が NP 完全な，難しい問題であるからである．本アイデアは，両面から計算コストを削減するためのものである．

距離の遠い頂点間の依存関係を無視　非連続コードクローンの誤検出を削減するために，ソースコード上で遠く離れた頂点間には依存関係を引かない．本アイデアにより，検出する価値のない重複コードの検出を抑え，検出の精度を上げることができる．また副次的な効果として，プログラムスライスの範囲が狭まるため，計算コストの削減による検出時間の短縮も見込まれる．

以降，2 章では PDG を用いたコードクローン検出とその問題点について述べ，3 章では，それに対する改善手法を提案する．4 章と 5 章では，オープンソースソフトウェアを用いた評価について述べ，6 章では実験結果の妥当性について考察する．また，7 章では関連研究について触れ，最後に 8 章で本論文をまとめる．

2. 準　　備

2.1 プログラム依存グラフ

プログラム依存グラフ（PDG）とは，プログラム内の要素（文）の間に存在する依存関係を表す有効グラフである．PDG の頂点はプログラムの要素であり，辺で結ばれた頂点に依存関係があることを表す．次に，PDG の 2 つの依存関係について説明する．

データ依存　文 s で変数 v を定義し，文 t で変数 v を参照しており，文 s から文 t への経路のうち，変数 v を再定義しないものがある場合，文 s から文 t にデータ依存があるという．

制御依存　文 s が条件文または繰返し文の条件式であり，文 s の条件判定の結果によって文 t を実行するか否かが直接決まる場合，文 s から文 t への制御依存関係があるという．

図 1 は，簡単な PDG の例を表している．if 文の条件式からその内部に存在している文へ制御依存があり，また text などの変数の定義・参照を行っている文の間にはデータ依存があるのが分かる．ラベル<1>は PDG の入り口を表す頂点である．この頂点は便宜上条件式と見なされ，直内にある文や式を表す頂点に対して制御依存辺が引かれる[9)]．以降，本論文では，説明のための PDG としてこの例を用いる．

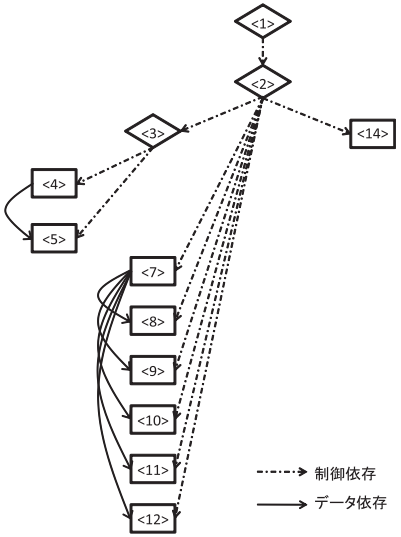
2.2 PDG を用いたコードクローン検出

PDG を用いたコードクローン検出の手順[17)] を示す．

STEP1　PDG のすべての頂点のハッシュ値を求め，ハッシュ値が同じ頂点ごとにグループを作成する．ハッシュ値は頂点が表すプログラム要素の構造に基づいて計算される．

```
 1: String sample1(){
 2:   if(this.trueOrFalse()){
 3:     if(null == this.getPath()){
 4:       Project proj = this.getProject();
 5:       this.setPath(proj.getBaseDir());
 6:     }
 7:     StringBuilder text =
                     new StringBuilder();
 8:     text.append("String A");
 9:     text.append("String B");
10:     text.append("String C");
11:     text.append("String D");
12:     return text.toString();
13:   }else{
14:     return "";
15:   }
16: }
```

(a) ソースコード



(b) PDG

図 1　ソースコードと PDG の例
Fig. 1　A simple example of source code and its PDG.

ハッシュ値を計算する前に，プログラム要素の変換を行うこともある．たとえば，利用している変数やリテラルをその型に変換することが考えられる．この処理を行うことにより，利用している変数やリテラルが異なっていても，それらの型が同じであり，そのプログラム要素の構造が等しければ，同じハッシュ値が生成される．

STEP2　プログラムスライシングを行い，同形部分グラフを検出する．スライス基点は，同じグループに属する頂点のペア (r_1, r_2) であり，2 つのスライシングは同期して行われる．スライシングにより新たにたどった頂点のハッシュ値が等しい場合はそれらを同形部分グラフの頂点として加える．スライシングが下記条件のいずれかを満たすとき，たどった頂点は同形部分グラフに加えられず，スライシングを終了する．

- 新たにたどった頂点のペア (p_1, p_2) が異なるハッシュ値を持つ場合．
- (p_1, p_2) のハッシュ値は等しいが，r_1 のグラフ（または r_2 のグラフ）がすでに p_1（または p_2）を含んでいる場合（無限ループを回避するための処理）．
- (p_1, p_2) のハッシュ値は等しいが，r_1 のグラフ（または r_2 のグラフ）が p_2（または p_1）を含んでいる場合（2 つの同形部分グラフが頂点を共有するの回避するための処理）．

この処理を同じグループに属するすべての頂点のペアに対して行う．スライシング終了後に特定されているグラフのペア（2 つの同形部分グラフ）が本手法において検出されるクローンペアである．

STEP3　あるクローンペア (s_1, s_2) が他のクローンペア (s'_1, s'_2) に含まれている場合 $(s_1 \subseteq s'_1 \cap s_2 \subseteq s'_2)$，そのクローンペアを検出されたクローンペアの集合から削除する．他のクローンペアに包含されたクローンペアをユーザに対して提示する理由はなく，またこれらの存在は検出結果を肥大化させてしまうからである．

STEP4　同じグラフを持つクローンペアからクローンセットを形成する．たとえば，2 つのクローンペア (s_1, s_2)，(s_2, s_3) があった場合，クローンセット $\{s_1, s_2, s_3\}$ が形成される．

2.3　既存手法の問題点

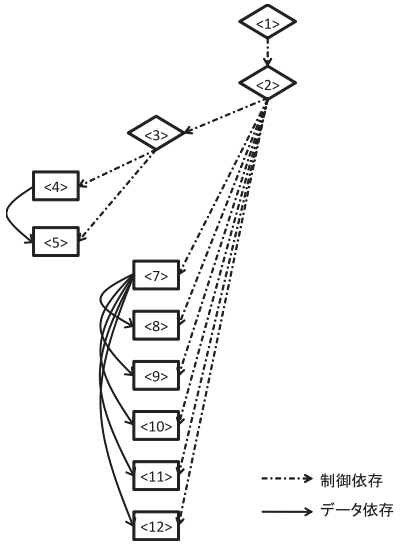
これまでの研究により，PDG を用いたコードクローン検出には検出精度と検出コストの面において問題があることが分かっている[7),17),18)]．

PDG を用いた検出は，他の検出手法に比べて，検出の精度が低い．これには 2 つの理由がある．第 1 の理由は，連続コードクローンの検出能力が低いことである．ソースコード上で隣接して存在しているプログラム要素が必ずしもデータ依存や制御依存を持つわけではないため，PDG 上でそれらをたどることができない．それに対して，行単位や字句単位の検出では，そのような依存関係は考慮せず，プログラムの表面的な字面を比較することによって検出を行っているため，連続コードクローンの検出に長けている．たとえば，図 1 (a) と **図 2** (a) からコードクローンを検出する場合を考える．行単位や字句単位の検出法を用いた場合は，各ソースコードの 3 行目から 11 行目までがコードクローンとして検出される．しかし，PDG を用いた手法では，3 行目から 6 行目までの領域と 7 行目から 12 行目までの領域が異なるコードクローンとして検出される．この理由は，これら 2 つの領域は，2 行目の頂点を介して接続されているが，この頂点は 2 つのソースコードでは異なるハッシュ値を持つためである．また，既存の PDG を用いた検出法では，計算コストの関係により，前向きスライスと後向きスライスのどちらか一方しか用いていない．しかし，前向きスライスでのみ，また後向きスライスでのみでは特定できない同形グラフが PDG には存在するため，十分にコードクローンが検出できているとはいえない．

精度が低い第 2 の理由は，非連続コードクローンの検出において，誤検出が多いことで

```
 1: String sample2(){
 2:   while(this.goOrStop()){
 3:     if(null == this.getPath()){
 4:       Project proj = this.getProject();
 5:       this.setPath(proj.getBaseDir());
 6:     }
 7:     StringBuilder text =
                     new StringBuilder();
 8:     text.append("String A");
 9:     text.append("String B");
10:     text.append("String C");
11:     text.append("String D");
12:     System.out.println(text.toString());
13:   }
14: }
```

(a) ソースコード



(b) PDG

図 2　比較対象ソースコードとその PDG
Fig. 2　Another sample of source code and its PDG.

ある．つまり，PDG を用いて検出された非連続コードクローンの多くは，人間がコードクローンと判断しないような重複コードであることが多い．**図 3** は，あるオープンソースソフトウェアから PDG を用いた検出手法により特定したコードクローンである．++ではじまる行が他のメソッド内のプログラム要素と重複していると特定された部分である．このメソッドでは，読み込んだ文字数と残りの文字数を格納するための変数，totalRead と numToRead が用いられており，文字列を読み込むたびにこれらの変数の値が更新されている．同様の処理が他のメソッドにも存在しており，コードクローンとして検出されていた．たしかに検出されたコードは処理内容としては共通しているが，コードクローンの要素に挟まれた部分（たとえば，283 行から 294 行，299 行から 312 行など）はメソッドごとにまったく処理が異なっていた．このような，人間がコードクローンとは判定しないと思われるコードクローンを検出することにより，検出の精度が低くなってしまう．

また，PDG を用いた検出は，検出に必要な計算コストが非常に高く，実規模ソフトウェアに対して適用が難しい．計算コストが高いのは 2 つの理由からである．第 1 の理由は，ス

```
  278: public int read(byte[] buf, int offset, int numToRead) throws ...
++279:   int totalRead = 0;
  280:   if (this.entryOffset >= this.entrySize) {
  281:     return -1; }
++282:   if ((numToRead + this.entryOffset) > this.entrySize) {
  283:     numToRead = (this.entrySize - this.entryOffset); }
  284:   if (this.readBuf != null) {
  285:     int sz = (numToRead > this.readBuf.length) ?
  286:              this.readBuf.length : numToRead;
  287:     System.arraycopy(this.readBuf, 0, buf, offset, sz);
  288:     if (sz >= this.readBuf.length) {
  289:       this.readBuf = null;
  290:     } else {
  291:       int newLen = this.readBuf.length - sz;
  292:       byte[] newBuf = new byte[newLen];
  293:       System.arraycopy(this.readBuf, sz, newBuf, 0, newLen);
  294:       this.readBuf = newBuf; }
++295:     totalRead += sz;
++296:     numToRead -= sz;
  297:     offset += sz; }
++298:   while (numToRead > 0) {
  299:     byte[] rec = this.buffer.readRecord();
  300:     if (rec == null) {
  301:       // Unexpected EOF!
  302:       throw new IOException("unexpected EOF with " + numToRead
  303:                             + " bytes unread"); }
  304:     int sz = numToRead;
  305:     int recLen = rec.length;
  306:     if (recLen > sz) {
  307:       System.arraycopy(rec, 0, buf, offset, sz);
  308:       this.readBuf = new byte[recLen - sz];
  309:       System.arraycopy(rec, sz, this.readBuf, 0, recLen - sz);
  310:     } else {
  311:       sz = recLen;
  312:       System.arraycopy(rec, 0, buf, offset, recLen); }
++313:     totalRead += sz;
++314:     numToRead -= sz;
  315:     offset += sz; }
++316:   this.entryOffset += totalRead;
  317:   return totalRead; }
```

図 3　頂点が離れたデータ依存を持つコードクローン
Fig. 3　Code clone having nodes that are separately located in the source code.

ライス基点として使用される頂点のペア数が非常に多いことである．そのため，プログラムスライシングの回数も非常に多くなり計算コストが高い原因となっている．第 2 の理由は，同形部分グラフを特定することそのものが，NP 完全な，難しい問題であるからである．

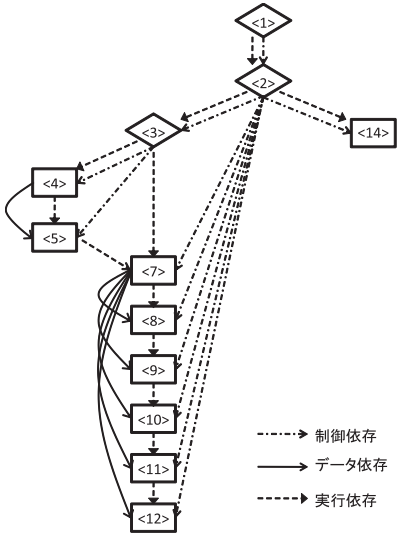


図 4　図 1 (b) の PDG に実行依存を導入した結果
Fig. 4　Result that execution dependency was introduced into Fig.1 (b).

3. 提 案 手 法

筆者らは文献 16) で提案されている検出手法を実装し，同形部分グラフが特定される過程および検出されたコードクローンを分析した．その結果，2.3 節で述べた問題点を改善する手法を考案することができた．本章では，それらの改善手法について述べる．

3.1 連続コードクローン検出能力の改善

3.1.1 実行依存の導入

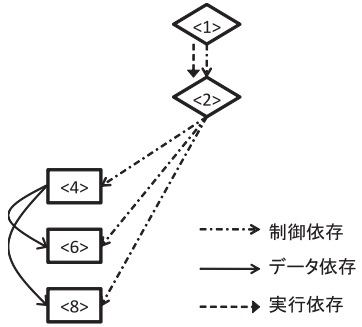
2.1 節で説明した従来の PDG に，新たにもう 1 種類の依存，**実行依存**を加える．実行依存はプログラム要素間の実行の順序を表す依存関係である．PDG 上の 2 つの頂点において，そのうちの 1 つの頂点（頂点 A とする）のプログラム要素が，他方の頂点（頂点 B とする）のプログラム要素が実行された直後に実行される可能性がある場合は，頂点 B から頂点 A に向かって実行依存辺が引かれる．実行依存を導入することにより，データ依存や制御依存がない隣接プログラム要素をたどることが可能となり，連続コードクローンの検出能力向上が期待できる．

```
 1: void sample3(){
 2:   if(this.trueOrFalse()){
 3:     ...
 4:     float rate = 0.05;
 5:     ...
 6:     float taxA = priceA * rate;
 7:     ...
 8:     float taxB = priceB * rate;
 9:     ...
10:   }
11: }
```

(a) ソースコード A

```
 1: void sample4(){
 2:   while(this.goOrStop()){
 3:     ...
 4:     float rate = 0.05;
 5:     ...
 6:     float taxA = priceA * rate;
 7:     ...
 8:     float taxB = priceB * rate;
 9:     ...
10:   }
11: }
```

(b) ソースコード B



(c) 生成される PDG

図 5　後向きスライスでは検出できない同形部分グラフの例
Fig. 5　Isomorphic subgraph that backward slice cannot detect.

図 4 は図 1 (b) の PDG に実行依存辺を加えたものである．図 4 では，実行依存辺により 3 行目から 6 行目までの領域と 7 行目から 12 行目までの領域が直接接続されているため，行単位や字句単位の検出を行った場合と同様に，これらの領域をまとめて 1 つのコードクローンとして検出できる．

3.1.2 双方向スライスの利用

提案手法では，前向きスライスと後向きスライスの 2 つを用いて同形部分グラフの特定を行う．その理由は，後向きスライスを用いて検出される一部の同形部分グラフは前向きスライスでは検出されず，またその逆もあるからである．

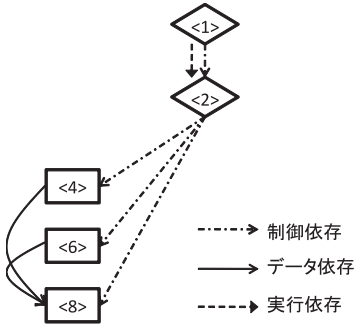
図 5 と**図 6** はそのような同形部分グラフ（4，6，8 行目の頂点からなる）の例を表して

```
 1: void sample5(){
 2:   if(this.trueOrFalse()){
 3:     ...
 4:     int price = this.getPrice();
 5:     ...
 6:     int tax = this.getTax();
 7:     ...
 8:     int amount = price + tax;
 9:     ...
10:   }
11: }
```

(a) ソースコード A

```
 1: void sample6(){
 2:   while(this.goOrStop()){
 3:     ...
 4:     int price = this.getPrice();
 5:     ...
 6:     int tax = this.getTax();
 7:     ...
 8:     int amount = price + tax;
 9:     ...
10:   }
11: }
```

(b) ソースコード B



(c) 生成される PDG

図 6　前向きスライスでは検出できない同形部分グラフの例
Fig. 6　Isomorphic subgraph that forward slice cannot detect.

いる．図 5 では，変数 rate が同形部分グラフにおいて 2 度参照されている．よって，“<4> → <6>” と “<4> → <8>” の 2 つのデータ依存辺が存在している．このような場合だと，同形部分グラフのどの頂点から後向きスライスを行っても，この同形部分グラフを特定することはできない．

図 6 では，2 つの変数（price と tax）が 8 行目の頂点で参照されている．よって，“<4> → <8>” と “<6> → <8>” の 2 つのデータ依存辺が存在している．このような形状の同形部分グラフの場合，どの頂点から前向きスライスを行っても，この同形部分グラフを特定することはできない．

つまり，PDG 上に存在する同形部分グラフを最大限に特定するためには，双方向のスライスを用いる必要がある．

3.2　計算コストの削減

本節では，コードクローン検出に必要な計算コストの削減を行う手法を提案する．3.2.1 項では，コスト削減のメインとなる手法を提案し，3.2.2 項では，2 つの簡易的なコスト削減手法を提案する．本節で提案する 3 つの手法は独立で用いることも同時に用いることも可能である．

3.2.1　(1) 実行依存辺に着目した頂点の集約

1 つ目のコスト削減手法は，PDG の頂点数を減らすことにより，コードクローン検出に必要な計算量を削減する．また，検出能力の向上も考慮に入れ，これまでの研究によりコードクローン検出に悪影響を与えるとされているソースコード上の繰返し処理部分[13] からのコードクローン検出を抑える．ソースコード上の繰返し処理とは，図 1 (a) の 8 行目から 11 行目のような，文単位で同様の処理を繰り返し行っている部分を指す．このような処理部分から検出されるコードクローンは，人間が必要とするコードクローン情報となることはほとんどないことがこれまでに示されている[13]．たとえば，8 行目の頂点と 10 行目の頂点から，フォワードスライスを用いて実行依存をたどることにより，8，9 行目と 10，11 行目が同一処理として検出されるが，そのようなコードクローンを人間が必要とすることはないだろう．

提案手法では，8 行目から 11 行目までの処理をそれぞれ異なる頂点とするのではなく，まとめて 1 つの頂点として PDG を構築する．このように頂点を集約することにより，上記のような必要のないコードクローンの検出処理を抑えつつ，計算コストの削減も実現することができる．具体的には，この手法では，下記のすべての条件を満たす頂点群が 1 つの頂点として集約される．

条件 1　文 s から文 t へ，実行依存辺のみをたどることにより到達可能な経路 $s, u_1, u_2, \cdots, u_k, t$（以降，経路 R）が存在する．

条件 2　経路 R に存在するすべての頂点（頂点 s と t を含む）は，実行依存の入力辺と出力辺を 1 つずつ持つ（if 文や while 文などの条件式，およびそれらが終わった後の合流点を含まない）．

条件 3　経路 R 上に存在するすべての頂点は同一ハッシュ値を持つ．

条件 4　経路 R を包含するいかなる経路も，条件 2 と 3 を同時に満たさない．

経路 R に含まれる頂点群を 1 つに集約した頂点を m とする．以降，頂点 m を始点・終点とする，データ依存辺，制御依存辺，実行依存辺について説明する．

データ依存辺　頂点 p を終点とするデータ依存辺の始点となっている頂点の集合を $data_{to}(p)$，経路 R に含まれる頂点の集合を P とすると，集約された頂点 m へのデータ依存辺を持つ頂点の集合は以下の式により定義される．

$$data_{to}(m) = \bigcup_{p \in P} data_{to}(p) \cap \overline{P} \quad (1)$$

同様に，頂点 p を始点とするデータ依存辺の終点となっている頂点の集合を $data_{from}(p)$ とすると，頂点 m からのデータ依存辺を持つ頂点の集合は以下の式により定義される．

$$data_{from}(m) = \bigcup_{p \in P} data_{from}(p) \cap \overline{P} \quad (2)$$

制御依存辺　条件 2 より，P に含まれるすべての頂点は，同一の頂点からの制御依存辺を持つ．提案手法では，頂点 m も同一の頂点からの制御依存辺を持つとする．頂点 p を終点とする制御依存辺の始点となっている頂点の集合を $control_{to}(p)$ とすると，下記の式により定義される．

$$control_{to}(m) = control_{to}(s) \quad (3)$$

また，条件 2 より，P に含まれる頂点を始点とする制御依存辺は存在しない．提案手法では，頂点 m を始点とする制御依存辺も存在しないとする．よって，頂点 m を始点とする制御依存辺の終点となっている頂点の集合 $control_{from}(m)$ は，下記の式により定義される．

$$control_{from}(m) = \emptyset \quad (4)$$

実行依存辺　頂点 m を終点・始点とする実行依存辺を持つ頂点の集合 $execute_{to}(m)$，$execute_{from}(m)$ は，それぞれ以下の式で定義される．

$$execute_{to}(m) = execute_{to}(s) \quad (5)$$

$$execute_{from}(m) = execute_{from}(t) \quad (6)$$

図 4 の PDG に対して，提案手法の頂点集約を行った PDG を**図 7** に示す．この例では，図 1 (a) に表すソースコードの 8 行目から 11 行目までの 4 つ頂点が，集約に必要な 4 つの条件を満たしているため，1 つの頂点として集約が行われている．

提案手法の頂点集約を行うことにより，下記の 2 つの計算コストを削減することができる．

- 集約した頂点がプログラムスライシング上に現れる場合，頂点をたどる計算コストを抑えることができる．たとえば，図 1 (a) と図 2 (a) において，各ソースコードの 7 行目の文をスライス基点としてスライシングを行う場合，頂点を集約しない場合は，7 行目から 12 行目までの頂点をたどるのに，PDG 上で 5 ホップを必要とするが，頂点集約を行うことにより，2 ホップでたどることができる．
- 集約した頂点がスライス基点となる場合，集約しない場合に比べて頂点の総ペア数が減るため，行うプログラムスライシングの数を削減することができる．たとえば，頂点集約を行わない場合は，図 1 (a) と図 2 (a) では，各ソースコードの 8 行目から 11 行目に存在する計 8 個の文はすべて同一ハッシュ値を持つため，${}_8C_2 = 28$ 個のペアについてスライシングが行われる．しかし，それらのペアのうちの多くは，ソースコード上で隣接した文であり，そのような頂点のペアを基点としてコードクローンを検出する必要はないと筆者らは考える．一方頂点集約を行った場合は，各ソースコードの 8 行目から 11 行目の 4 つの頂点が 1 つに集約されるため，同一ハッシュ値を持つ頂点数は 2 つとなり，そのペアに対してのみプログラムスライシングが行われることになる．

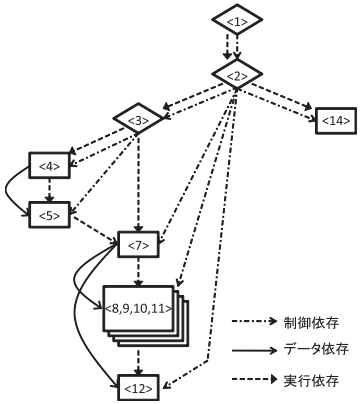


図 7　図 4 の PDG に対して頂点集約を行った結果
Fig. 7　Result that the merging method was applied to Fig.4.

3.2.2　その他の簡易的な改善手法

本項では，その他の軽微な計算コスト削減手法について述べる．

(2) 大きすぎるグループはスライス基点として用いない　検出処理の STEP2 において，n 個の頂点が同じハッシュ値を持つ場合，それらのペアをスライス基点とする組合せの数は $n(n-1)/2$ となる．実規模ソフトウェアの場合，同じハッシュ値を持つ頂点の数は，数千あるいはそれ以上になる場合があり，すべての組合せに対して同形部分グラフを特定する処理は非常に高い計算コストを必要とする．しかし，数千もの頂点が同じハッシュ値を持つ理由は，それらがプログラムで頻繁に必要とされる処理であるからであり

（たとえば，変数宣言 `int i;` と `int j;` や，リターン文 `return a;` と `return b;`），それらがコピーアンドペーストにより生成されたわけではない．そこで，本論文では，同じハッシュ値を持つ頂点数の閾値を設け，その閾値よりも多くの数の頂点数が同じハッシュ値を持つ場合は，それらはスライス基点としては用いない方法を提案する．

(3) 小さいメソッドの要素はハッシュ化しない　一般的にコードクローン検出では，検出するコードクローンの最小の大きさを指定する．提案手法はメソッド単位で PDG を構築しているため，検出する最小の大きさよりも小さい PDG からは決してコードクローンが検出されることはない．そのため，検出処理の STEP1 において，このようなメソッドの PDG に含まれる頂点をハッシュ化しないことにより，STEP2 において無駄な頂点のペアをスライス基点とすることがなくなり，検出処理の向上が見込まれる．

3.3　誤検出の削減

ツールを用いたコードクローン検出では，ある程度誤って検出されたコードクローンが検出結果に含まれる[7),8)]．ここで「誤って検出されたコードクローン」とは検出ツールが定めたコードクローンの定義には従っているが，人間がコードクローンとは判断しないコード片を指す．筆者らは，PDG を用いた検出における誤検出は，依存関係にある頂点間のソースコード上での位置関係を調べることにより，ある程度削減できると考えている．

たとえば，図 3 のような，大きなメソッド内において離れて存在しているプログラム要素のみが重複した処理になっている場合は，コードクローンとして検出するメリットは少ないと筆者らは考えた．そこで，このような重複処理をコードクローンとして検出することを防ぐために，本論文では，頂点間のソースコード上での距離に閾値を設ける．そして，2 つの頂点間に依存関係がある場合でも，その距離が閾値以上である場合は，依存辺を引かない．このようにすることで，図 3 で紹介したようなコードクローンの検出を抑えることが可能になる．なお，この処理は，データ依存と制御依存についてのみ行う．実行依存は，たとえソースコード上での位置が離れていても，それらの頂点の要素は連続して実行されるという関係があり，筆者らはこの関係は重要と考えたため，閾値は設けない．

3.4　実　　装

提案手法をツールとして実装した[4)]．提案手法において，最も計算コストが高いのは検出処理の STEP2（同形部分グラフの特定）である．この処理では，莫大な数（ハッシュ値が等しい頂点のペア数）の同形部分グラフ特定の計算を行う．しかし，各ペアの検出結果は互いに影響を及ぼさないため，これらの計算は並列に行うことが可能である．実装したツールでは，同形部分グラフ特定の処理を行うスレッドの数をツール実行の際にコマンドライン引数として与えることができる．このように実装を行うことによって，近年普及してきた複数の物理 CPU やマルチコア CPU を搭載したコンピュータの資源を有効に利用し，高速に検出処理を行うことができる．

4.　各提案手法の評価

実装したツールを用いて，オープンソースソフトウェアに対して実験を行った．この実験の目的は，3 章で述べた各提案手法の効果を評価することである．具体的な評価項目は下記の 4 つとなる．

評価項目 a1　双方向スライスを用いた効果の評価.
評価項目 a2　実行依存の導入した効果の評価.
評価項目 a3　コスト削減手法を用いた効果の評価.
評価項目 a4　誤検出削減手法を用いた効果の評価.

この実験では，**表 1** に示すソフトウェアを対象とした．この表の「行数」はソースコードの総行数を表す．本実験では，6 つ以上のプログラム要素からなるコードクローンを検出した[⋆1]．以降，対象ソフトウェアは表 1 に記載している省略名を用いて呼ぶ．

4.1　準　　備

4.1.1　正解クローン

この実験では，文献 3) で公開されているデータを検出すべきコードクローン群とした．このコードクローン群の情報は下記の手順で作成された．

(1)　文献 7) の筆者（Stefan Bellon）が検出対象ソフトウェアを決め，6 人のコードクロー

表 1　対象ソフトウェア
Table 1　Target software.

ソフトウェア	省略名	行数
netbeans-javadoc	netbeans	14,360
eclipse-ant	ant	34,744
eclipse-jdtcore	jdtcore	147,634
j2sdk1.4.0-javax-swing	swing	204,037

⋆1 文献 7) では，6 行以上からなる重複コードをコードクローンとして検出，比較している．本論文で提案する手法では，PDG の頂点は文や式であるため，1 つの頂点が 1 行のソースコードと対応することが多い．本論文では，6 つ以上のプログラム要素からなるコードクローンの検出を行い，文献 7) の検出ツールとの比較を行った．この比較の詳細については，5 章を参照されたい．

ン検出ツールの開発者に検出を依頼．Bellon はコードクローン検出ツールの開発者ではないため，中立的な立場で検出対象ソフトウェアを選んだといえる．

(2) 各開発者は，自身が開発した検出ツールを用いて，対象ソフトウェアから重複コードを検出．所定のフォーマットを用いて，検出した重複コードの位置情報を Bellon に送付．

(3) Bellon は各開発者から送られた重複コードの全ペア（325,935 個）のうちの 2%を乱数を用いて選択し，それらが本当にコードクローンであるかを手作業により判定．ツールが検出した重複コードがそのままコードクローンと判断される場合もあるが，Bellon が必要に応じて重複コードを加工して[⋆1]コードクローンと判断した場合もある．また，コードクローンとは判断されない重複コードもあった．この結果 4 つの対象ソフトウェアから 4,789 個のコードクローンのペアが抽出された．

以降，この実験では，ツールが検出した重複コードを**クローン候補**（clone candidates），Bellon が抽出したコードクローンを**正解クローン**（clone references）と呼ぶ．なお，正解クローンは，Bellon により以下の 3 種類に分類されている．

Type1　空白やタブを除いて表面上まったく同一なコードクローンのペア，

Type2　変数名やメソッド名など，ユーザ定義の識別子が異なるコードクローンのペア，

Type3　Type1 や Type2 が許容する変更に加え，文単位や式単位で異なる部分が存在するコードクローンのペア．

Type1 と Type2 は連続コードクローンであり，Type3 は非連続コードクローンである．

4.1.2　評 価 基 準

この実験では，クローン候補が正解クローンかどうかを **good 値**と **ok 値**を用いて判断し，**再現率**を算出する．文献 7) と同様，good 値と ok 値の閾値として 0.7 を用いた．また，2 つの検出結果がどの程度似ているのかを調査するために**被覆率**を用いた．good 値と ok 値，再現率および被覆率の定義は付録に示す．なお，この実験で用いた正解クローンは，コードクローンの母集団（対象ソフトウェアに含まれるすべてのコードクローン）ではない．そのため，下記の点に留意されたい．

- 再現率の絶対値そのものは意味のある値ではない．正解クローンの母集合を用いていないため，再現率の絶対値が検出能力を正しく表しているとは限らないためである．この値はあくまでも複数の検出結果を相対的に比較するためのものである．
- 正解クローンの母集団が不明なため，適合率は算出できていない．

また，コードクローン検出に必要な計算コストを定量的に評価するため，この実験では**頂点比較回数**を調査した．頂点比較回数とは，検出処理（2.2 節を参照）の STEP2 で同形部分グラフを検出するために，2 つのプログラムスライスを同時に実行し，たどった頂点のハッシュ値を比較した回数である．頂点比較回数が多いほど，同形部分グラフの検出に必要なハッシュ値の比較回数が多いことになり，より高い計算コストが必要であったことが分かる．

⋆1 ここでの加工とは，ツールが検出した重複コードから一部分を取り除くことや，重複コードの周辺コードもコードクローンに含めることを指す．

表 2　検出の設定
Table 2　Detection configuration.

検出 ID	検出の設定		
	後向きスライス	前向きスライス	実行依存
A1	○	×	×
A2	×	○	×
A3	○	○	×
B1	○	×	○
B2	×	○	○
B3	○	○	○

表 3　単方向スライスを用いた検出結果の，双方向スライスを用いた検出結果に対する被覆率
Table 3　Content rates between detections using one-way slicing and two-way slicing.

ソフトウェア	被覆率	ソフトウェア	被覆率
netbeans	$coverage$(A3,A1) = 21.57% $coverage$(A3,A2) = 50.65% $coverage$(B3,B1) = 75.79% $coverage$(B3,B2) = 82.27%	ant	$coverage$(A3,A1) = 44.61% $coverage$(A3,A2) = 55.45% $coverage$(B3,B1) = 64.10% $coverage$(B3,B2) = 70.30%
jdtcore	$coverage$(A3,A1) = 54.97% $coverage$(A3,A2) = 65.74% $coverage$(B3,B1) = 77.65% $coverage$(B3,B2) = 80.31%	swing	$coverage$(A3,A1) = 45.63% $coverage$(A3,A2) = 56.77% $coverage$(B3,B1) = 73.60% $coverage$(B3,B2) = 75.84%

4.2　評価項目 a1：双方向スライスを用いた効果

双方向スライスを用いて検出されるコードクローンが，どちらか一方のスライスを用いて検出されるコードクローンに対してどの程度拡大しているのかを式 (13) を用いて調査した．なお，本節（4.2 節）と次節（4.3 節）では，**表 2** に示す設定を用いてコードクローンを検出した．調査結果を**表 3** に示す．ソフトウェアによってばらつきはあるものの被覆率は約

表 4　双方向スライスと実行依存の利用による正解クローン検出数の変化
Table 4　Number of detected clone references with and without two-way slicing and execution dependency.

(a) good 値を用いた場合

検出 ID	netbeans		ant		jdtcore		swing	
	連続	非連続	連続	非連続	連続	非連続	連続	非連続
正解クローン	39	16	28	2	986	359	740	37
A1	4	2	3	0	48	8	56	6
A2	7	1	4	1	219	62	76	11
A3	8	3	3	1	177	60	100	12
B1	11	4	6	0	288	61	90	12
B2	11	5	7	1	247	67	107	13
B3	12	4	5	1	204	60	105	12

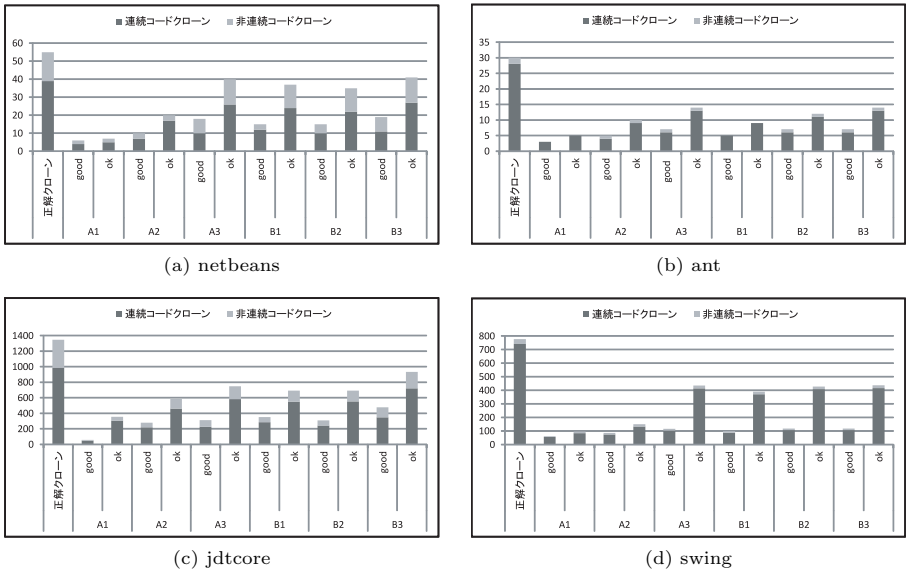
(b) ok 値を用いた場合

検出 ID	netbeans		ant		jdtcore		swing	
	連続	非連続	連続	非連続	連続	非連続	連続	非連続
正解クローン	39	16	28	2	986	359	740	37
A1	7	2	4	0	297	50	93	12
A2	17	3	9	1	462	126	135	16
A3	20	3	9	1	484	135	413	21
B1	23	13	9	0	558	143	380	19
B2	22	13	11	1	554	138	412	20
B3	24	13	11	1	572	153	418	23

22%〜約 82%であり，双方向スライスを用いることによって検出される同形部分グラフが拡大していることが分かる．これは，3.1.2 項で紹介したような，1 つのスライスのみでは完全に特定できない同形部分グラフが存在したためである．

次に双方向スライスを用いることによって正解クローンの検出数がどのように変化したのかを調査した．調査結果を**表 4** に示す．またこの表を可視化したグラフを**図 8** に示す．これらより，双方向スライスを用いた検出では，どちらか一方のスライスを用いた検出に比べ，多くの場合においてより多くの正解クローンを検出できていることが分かる．一部の場合では，双方向スライスを用いた検出の場合に，正解クローンの検出数が少なくなっていた（たとえば，ant における連続コードクローンの検出など）．この原因は，双方向スライスを用いることにより，コードクローンに含まれるプログラム要素が必要以上に増えてしまい，正解クローンとの good 値が閾値を下回ってしまったクローン候補があったためである．

最後に，双方向スライスを用いることによってどの程度計算コストが増加したのかを頂点



(a) netbeans　(b) ant　(c) jdtcore　(d) swing

図 8　表 4 の棒グラフを用いた可視化
Fig. 8　A bar chart representation of Table 4.

比較回数を用いて調査した．調査の結果を**表 5** に示す．この表より計算コストは，後向きスライスのみを用いた場合に最も低く，双方向スライスを用いた場合に最も高くなっていることが分かる．双方向スライスを用いることによる計算コストの増加の程度はソフトウェアによって異なるが，最も増加率が大きかったのは，netbeans の A1 と A3 の間であり，頂点比較回数が約 800 倍になっていた．以上の調査により，双方向スライスを用いることにより検出できた正解クローンの数は増えたが，検出に必要な計算コストも増加したことが分かった．

4.3　評価項目 a2：実行依存を用いた効果

実行依存を用いた場合に検出されるコードクローンが，用いなかった場合に検出されるコードクローンに対してどの程度拡大しているのかを調査した．この調査では，後者の前者に対する被覆率を式 (13) を用いて算出した．調査結果を**表 6** に示す．この表から分かるように，被覆率は後向きスライスの場合が最も低く，双方向スライスの場合が最も高い．この原因は，後向きスライスを用いてデータ依存辺と制御依存辺をたどるのみでは，到達可能な

表 5　双方向スライスと実行依存の利用による頂点比較回数の変化
Table 5　Number of node comparisons with and without two-way slicing and execution dependency.

ソフトウェア	検出 ID	頂点比較回数	ソフトウェア	検出 ID	頂点比較回数
netbeans	A1	135,385	ant	A1	567,891
	A2	551,331		A2	4,942,013
	A3	110,356,247		A3	10,388,934
	B1	327,889		B1	951,183
	B2	663,348		B2	5,289,877
	B3	101,104,540		B3	11,295,018
jdtcore	A1	58,155,404	swing	A1	16,123,841
	A2	346,223,464		A2	331,698,602
	A3	1,382,195,451		A3	525,716,585
	B1	118,144,589		B1	28,427,935
	B2	408,615,802		B2	342,132,984
	B3	1,431,922,235		B3	528,963,280

表 6　実行依存を用いなかった検出結果の，実行依存を用いた検出結果に対する被覆率
Table 6　Content rates between detection with and without execution dependency.

ソフトウェア	被覆率	ソフトウェア	被覆率
netbeans	$coverage$(B1,A1) = 25.76%	ant	$coverage$(B1,A1) = 57.43%
	$coverage$(B2,A2) = 59.15%		$coverage$(B2,A2) = 74.10%
	$coverage$(B3,A3) = 96.08%		$coverage$(B3,A3) = 93.93%
jdtcore	$coverage$(B1,A1) = 65.01%	swing	$coverage$(B1,A1) = 60.86%
	$coverage$(B2,A2) = 76.91%		$coverage$(B2,A2) = 71.53%
	$coverage$(B3,A3) = 94.03%		$coverage$(B3,A3) = 95.67%

頂点が少ないからである（表 5 では頂点比較回数が少なく，表 4 では正解クローン検出数が少ない）．そして，データ依存と制御依存のみで到達可能な頂点が少ない方が，実行依存を用いることにより到達可能な頂点の増加の程度が大きくなっていることが分かる．後向きスライスを用いた場合では，実行依存辺の有無の被覆率が 26%〜65%なのに対して，双方向スライスを用いた場合の被覆率は 94%〜96% と高い．いずれのスライスを用いた場合においても，被覆率は 100%ではなく，実行依存辺を用いることによって検出されるクローンが大きくなったという結果であった．

次に実行依存を用いることによって正解クローンの検出数がどのように変化したのかを調査した．調査結果を表 4 に示す．この表より，すべての場合において，実行依存を用いることにより，正解クローンの検出数は増加もしくは変化なしであった．3.1.1 項で述べたように，実行依存の導入は連続コードクローンの検出能力の向上を目的としていたが，非連続コードクローンについても正解クローンの検出数が増えていた．

表 7　計算コスト削減手法による頂点比較回数の変化（括弧内の数値は削減手法未適用の場合に対する割合）
Table 7　Number of node comparisons with computational cost reduction methods (numbers in parentheses indicate the rate of the detection without applying the methods.)

ソフトウェア	頂点比較回数（$\times 10^3$）						
	(1) 頂点集約	(2) 大きいグループ					(2) 小さいメソッド
		閾値 100	閾値 200	閾値 300	閾値 500	閾値 1,000	
netbeans	39,507 (39.1%)	46,276 (45.8%)	100,713 (99.6%)	100,713 (99.6%)	100,713 (99.6%)	101,104 (100.0%)	100,755 (99.7%)
ant	9,818 (86.9%)	2,056 (18.2%)	4,436 (39.3%)	7,388 (65.4%)	7,388 (65.4%)	7,388 (65.4%)	8,944 (78.3%)
jdtcore	1,031,635 (72.0%)	42,941 (3.0%)	344,209 (24.0%)	393,344 (27.0%)	615,456 (43.0%)	699,470 (48.8%)	1,380,464 (96.4%)
swing	408,196 (77.2%)	16,584 (3.1%)	35,860 (6.6%)	41,774 (7.9%)	100,153 (18.9%)	137,676 (26.0%)	438,090 (82.8%)

また，表 5 は実行依存を用いた場合と用いなかった場合の PDG 頂点の比較回数を表している．変化の様子はすべてのソフトウェアで共通しており，後向きスライスを用いた場合は約 2 倍から 3 倍，前向きスライスを用いた場合は 2 倍未満，双方向スライスを用いた場合にはほとんど変わっていない．これは，後向きスライスを用いた場合では，実行依存辺を用いることにより，新たにたどる頂点が大きく増えたのに対して，前向きスライスや双方向スライスを用いた場合では，実行依存辺を用いてもそれほど増えていないことを表している．以上の調査により，実行依存を用いることにより，正解クローンの検出数は増えたが，検出に必要なコストも大きくなったことが分かった．

4.4　評価項目 a3：計算コスト削減手法の効果

この評価では，すべての対象ソフトウェアにおいて最も計算コストが高かった B3 の設定を用いた．

表 7 は，各計算コスト削減手法を用いた場合の頂点比較回数と計算コスト削減手法未適用の場合に対する割合を表している．頂点集約を行った場合，手法未適用の場合に比べて計算コストは 39%〜87%であり，すべてのソフトウェアで計算コストが削減できていた．特に ant 以外の 3 つのソフトウェアについて削減率が高い．頂点数や辺数は数%しか減少していないことを考慮すると（**表 8**），提案手法を使うことにより，検出のボトルネックとなっている部分を集約しているといえるだろう．大きいグループを用いない場合は，閾値が小さ

表 8　PDG の規模（従来 PDG は制御依存とデータ依存を持つ PDG，実行依存付き PDG は従来 PDG に実行依存を加えた PDG，頂点集約 PDG は，実行依存付き PDG に対して頂点集約を行った PDG を表す）
Table 8　Size of PDGs.

ソフトウェア	PDG	頂点数	辺数		
			データ依存	制御依存	実行依存
netbeans	従来 PDG	6,557	4,700	5,626	0
	実行依存付き PDG	6,557	4,700	5,626	6,144
	頂点集約 PDG	6,060	4,362	5,131	5,647
ant	従来 PDG	12,505	11,269	10,423	0
	実行依存付き PDG	12,505	11,269	10,423	12,379
	頂点集約 PDG	12,126	10,998	10,073	12,002
jdtcore	従来 PDG	77,493	91,617	64,701	0
	実行依存付き PDG	77,493	91,617	64,701	77,980
	頂点集約 PDG	73,885	88,443	61,263	74,595
swing	従来 PDG	82,824	75,560	68,310	0
	実行依存付き PDG	82,824	75,560	68,310	78,110
	頂点集約 PDG	78,783	73,026	64,370	74,050

表 9　計算コスト削減手法を用いた検出結果の，用いなかった検出結果に対する被覆率
Table 9　Content rates between detections with and without the computational cost reduction methods.

ソフトウェア	計算コスト削減手法未適用の検出結果に対する被覆率						
	(1) 頂点集約	(2) 大きいグループ					(3) 小さいメソッド
		閾値 100	閾値 200	閾値 300	閾値 500	閾値 1,000	
netbeans	98.45	100%	100%	100%	100%	100%	100%
ant	99.15	96.58%	99.48%	99.82%	99.82%	99.82%	100%
jdtcore	99.40	91.25%	94.62%	96.49%	97.35%	99.00%	100%
swing	98.45	93.94%	96.43%	97.34%	98.52%	99.70%	100%

いほど，また，対象ソフトウェアが大きいほど，計算コスト削減手法を用いない場合に対する削減率が高くなっている．たとえば，jdtcore の閾値 100 の場合は，削減手法を用いない場合に比べて頂点比較回数が 3.0%にまで削減できている．小さいメソッドを用いない場合は，削減手法を用いない場合に比べて最大約 78%に削減できている．

次に，計算コスト削減手法を用いた場合に検出されるクローン候補は，用いない場合に検出されるクローン候補に対してどの程度の被覆率があるのかを，式 (13) を用いて調査した．**表 9** は，各ソフトウェアにおける被覆率をまとめたものである．頂点集約を行った場合は，どのソフトウェアにおいても 98%以上の被覆率があり，検出結果がほどんど変化していない．大きいグループを用いない場合は，どのソフトウェアにおいても 90%以上の被覆率があり，閾値が大きいほど被覆率も高いことが分かる．小さいメソッドを用いない手法では，そもそも小さいメソッドからはコードクローンが検出されることはないため，検出結果にまったく影響を与えず，すべてのソフトウェアにおいて被覆率 100%となっている．

表 10　計算コスト削減手法を用いた場合の正解クローン検出数の変化
Table 10　Number of detected clone references with the cost reduction method.

(a) good 値を用いた場合

		netbeans		ant		jdtcore		swing	
		連続	非連続	連続	非連続	連続	非連続	連続	非連続
手法未適用		11	8	6	1	347	130	105	12
(1) 頂点集約		12	9	6	1	358	130	103	13
(2) 大きいグループ	閾値 100	11	8	6	1	153	56	101	12
	閾値 200	11	8	6	1	210	73	103	12
	閾値 300	11	8	6	1	325	128	103	12
	閾値 500	11	8	6	1	326	128	104	12
	閾値 1,000	11	8	6	1	345	129	104	12
(3) 小さいメソッド		11	8	6	1	347	130	105	12

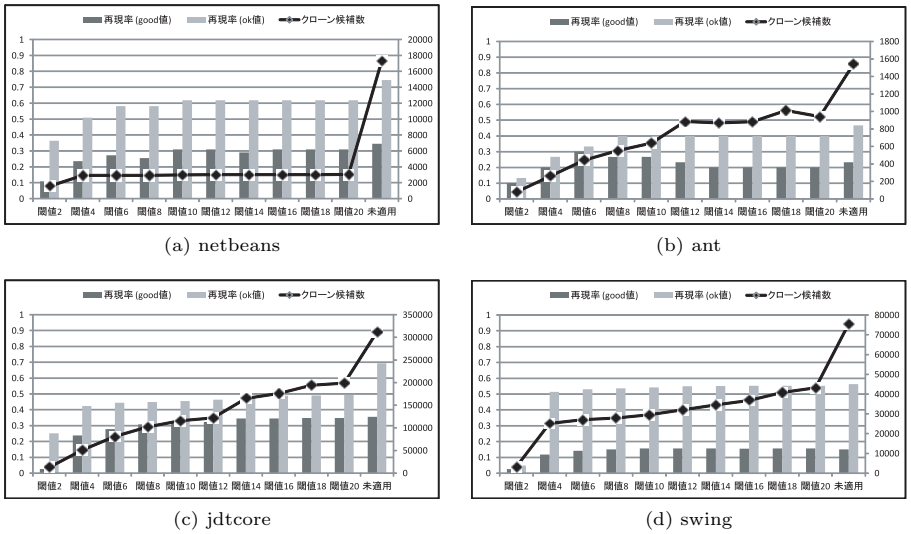
(b) ok 値を用いた場合

		netbeans		ant		jdtcore		swing	
		連続	非連続	連続	非連続	連続	非連続	連続	非連続
手法未適用		27	14	13	1	720	214	415	22
(1) 頂点集約		27	14	13	1	726	214	416	23
(2) 大きいグループ	閾値 100	27	14	13	1	495	128	413	21
	閾値 200	27	14	13	1	548	149	414	22
	閾値 300	27	14	13	1	698	205	414	22
	閾値 500	27	14	13	1	699	205	415	22
	閾値 1,000	27	14	13	1	720	210	415	22
(3) 小さいメソッド		27	14	13	1	720	214	415	22

次に，計算コスト削減手法を用いることによる正解クローン検出数への影響を調査した．調査結果を**表 10** に示す．頂点集約を行った場合は，正解クローンの検出数が少なくなることはほとんどなく，swing の good 値を用いた場合のみ検出結果が悪くなってしまっていた．また，頂点集約により新しく正解クローンを検出できる場合もあった．これは，連続した同一処理部分全体をまとめて 1 つの頂点とすることにより，正解クローンとの good 値，ok 値が閾値を上回った（人間がコードクローンと判断する部分に近づいた）場合があったことを示している．大きいグループをスライス基点として用いない場合は，netbeans と ant においては正解クローンの検出数は変化がなかったが，jdtcore と swing については，正解

表 11　誤検出削減手法を用いた検出結果の，用いなかった検出結果に対する被覆率
Table 11　Content rates between detections with and without the false positive reduction method.

ソフトウェア	誤検出削減手法未適用の検出結果に対する被覆率									
	閾値 2	閾値 4	閾値 6	閾値 8	閾値 10	閾値 12	閾値 14	閾値 16	閾値 18	閾値 20
netbeans	48.62%	64.92%	72.22%	74.70%	76.72%	78.08%	79.55%	80.29%	80.64%	81.41%
ant	21.29%	45.11%	52.94%	57.85%	62.64%	65.60%	68.25%	69.08%	72.00%	72.52%
jdtcore	26.31%	55.32%	66.15%	72.06%	75.78%	78.66%	80.59%	81.99%	83.02%	84.11%
swing	21.25%	51.95%	63.43%	69.98%	74.25%	77.58%	79.74%	81.39%	82.86%	84.07%



(a) netbeans　(b) ant

(c) jdtcore　(d) swing

図 9　誤検出削減手法を用いた場合の再現率の変化
Fig. 9　Recall with the false positive reduction method.

クローンの検出数が少なくなっており，閾値が小さいほど正解クローンの検出数は少なかった．小さいメソッドをハッシュ化しない手法は，検出結果に影響を与えないため，正解クローンの検出数が減ることはなかった．

以上の調査結果より，次のようにまとめることができる．頂点集約手法は，検出に悪影響を与えてしまうことはほとんどなく，計算コストを削減することが可能である．また，検出結果の再現率向上に役立つ場合もある．大きいグループを用いない手法は，適切に閾値を設定すれば，正解クローンの検出数をほとんど減らすことなく，計算コストを削減すること

表 12　誤検出削減手法を用いた場合の正解クローン検出数の変化
Table 12　Number of detected clone references with the false positive reduction method.

(a) good 値を用いた場合

		netbeans		ant		jdtcore		swing	
		連続	非連続	連続	非連続	連続	非連続	連続	非連続
手法適用	閾値 2	5	1	3	0	34	0	20	0
	閾値 4	10	3	6	0	306	15	92	0
	閾値 6	12	3	8	1	355	20	107	3
	閾値 8	11	3	7	1	385	30	108	9
	閾値 10	13	4	7	1	383	39	112	9
	閾値 12	13	4	6	1	369	66	111	10
	閾値 14	12	4	5	1	362	102	111	10
	閾値 16	13	4	5	1	361	103	110	10
	閾値 18	13	4	5	1	363	105	111	10
	閾値 20	13	4	5	1	362	106	111	11
手法未適用（閾値なし）		11	8	6	1	347	130	105	12

(b) ok 値を用いた場合

		netbeans		ant		jdtcore		swing	
		連続	非連続	連続	非連続	連続	非連続	連続	非連続
手法適用	閾値 2	12	8	4	0	238	100	36	1
	閾値 4	17	11	8	0	442	128	386	13
	閾値 6	19	13	9	1	478	119	397	15
	閾値 8	19	13	11	1	493	110	400	17
	閾値 10	21	13	11	1	499	113	404	17
	閾値 12	21	13	11	1	509	114	408	19
	閾値 14	21	13	11	1	516	136	409	19
	閾値 16	21	13	11	1	522	136	410	19
	閾値 18	21	13	11	1	524	136	410	19
	閾値 20	21	13	11	1	527	136	410	19
手法未適用（閾値なし）		27	14	13	1	720	214	415	22

表 13　誤検出削減手法による頂点比較回数の変化（括弧内の数値は削減手法未適用の場合に対する割合）
Table 13　Number of node comparisons with the false positive reduction method (numbers in parentheses are the rates for the detections without the method).

ソフトウェア	頂点比較回数 ($\times 10^3$)									
	閾値 2	閾値 4	閾値 6	閾値 8	閾値 10	閾値 12	閾値 14	閾値 16	閾値 18	閾値 20
netbeans	292 (0.29%)	516 (0.51%)	682 (0.67%)	766 (0.76%)	819 (0.81%)	854 (0.84%)	884 (0.87%)	917 (0.91%)	937 (0.93%)	951 (0.94%)
ant	1,201 (10.63%)	2,520 (22.31%)	3,253 (28.80%)	3,781 (33.47%)	4,118 (36.65%)	4,445 (39.35%)	4,724 (41.82%)	5,167 (45.75%)	5,336 (47.24%)	6,207 (54.95%)
jdtcore	62,075 (4.34%)	127,972 (8.94%)	172,778 (12.07%)	203,759 (14.23%)	227,327 (15.86%)	249,763 (17.44%)	267,660 (18.69%)	284,758 (19.89%)	303,180 (21.17%)	316,941 (22.13%)
swing	39,724 (7.51%)	94,256 (17.82%)	139,401 (26.35%)	169,343 (32.01%)	194,798 (36.83%)	216,956 (41.02%)	235,201 (44.46%)	250,743 (47.40%)	264,984 (50.09%)	277,357 (52.43%)

が可能である．小さいメソッドを用いない手法は，計算コストの削減率は 3 つの手法の中で最も低いが，検出結果に影響を与えないため，正解クローンの検出数を減らすことなく，計算コストを削減可能である．

4.5　評価項目 a4：誤検出削減手法の効果

誤検出削減手法を用いた効果を検出結果の再現率とクローン候補数を用いて評価した．この調査でも，B3 の設定を用いた．なお，3.2 節で述べた計算コスト削減手法は用いていない．

まず，誤検出削減手法を用いた検出の，用いなかった検出に対する被覆率を**表 11** に示す．netbeans の場合は，誤検出削減手法を用いることによってクローン候補に含まれるプログラム要素の数が 49%〜81%になっており，他の 3 つのソフトウェアでは，21%〜84%になっている．このことより，誤検出削減手法を用いることにより，クローン候補に含まれるプログラム要素がすべてのソフトウェアにおいて減少していることが分かる．

再現率とクローン候補数の変化の様子を**図 9** に示す．誤検出削減手法を用いることによってクローン候補の数が減るために再現率は低くなる，と筆者らは予測していた．しかし，ant と jdtcore の good 値を用いた場合は，手法を用いない場合に比べて再現率が高くなっていた．この原因は，誤検出削減手法を用いることによって遠く離れた頂点をたどらなくなった結果，正解クローンとの間の good 値が閾値を超えたコードクローンが存在したからである．誤検出削減手法を用いた場合の正解クローン検出数の変化を**表 12** に示す．この表より，good 値を用いた場合にはすべてのソフトウェアにおいて，適切な閾値を設けることにより連続コードクローンについては正解クローンの検出数が増加していた．一方 ok 値を用いた場合には，誤検出削減手法を用いることにより，正解クローンの検出数が減ってしまっていた．

また，誤検出削減手法を用いることにより，PDG においてたどる頂点の数は減るため，副次的な効果として検出速度の向上が見込まれる．この点については，検出に必要であった頂点比較回数を調査することにより評価した．**表 13** に調査結果を示す．計算コスト削減の程度はソフトウェアによってかなり差があり，netbeans の場合は，すべての閾値において，計算コストは 1%未満にまで削減できている．一方，ant や swing の場合には，閾値が大きい場合でも約 50%削減できていた．

5. 提案手法を用いたコードクローン検出法の評価

本論文で提案した手法を組み合わせて用いた場合に，1 つのコードクローン検出法としてどの程度の性能があるのか下記の項目で評価した．

評価項目 b1　既存の PDG を用いた検出手法との検出能力および計算コストの比較．

評価項目 b2　他の検出手法との検出能力の比較．

この評価でも，対象ソフトウェアは，表 1 に示すものを用いた．なお，この評価では，できるだけ多くの正解クローンを検出することを第 1 に考え，下記の設定を用いてコードクローン検出を行った．

用いたスライス：双方向スライス
実行依存：用いる
頂点集約：用いる
大きなグループはスライス基点にしない：用いる（閾値 1,000）
小さいメソッドはハッシュ化しない：用いる
遠く離れた頂点間には辺を引かない：用いない

表 14　既存手法と提案手法の正解クローン検出数
Table 14　Number of clone references detected by the proposed method and the exsting methods.

(a) good 値を用いた場合

検出 ID	netbeans		ant		jdtcore		swing	
	連続	非連続	連続	非連続	連続	非連続	連続	非連続
正解クローン	39	16	28	2	986	359	740	37
既存手法 A1	4	2	3	0	48	8	56	6
既存手法 A2	7	1	4	1	219	62	76	11
提案手法	12	9	6	1	354	130	109	11

(b) ok 値を用いた場合

検出 ID	netbeans		ant		jdtcore		swing	
	連続	非連続	連続	非連続	連続	非連続	連続	非連続
正解クローン	39	16	28	2	986	359	740	37
既存手法 A1	7	2	4	0	297	50	93	12
既存手法 A2	17	3	9	1	462	126	135	16
提案手法	27	14	13	1	725	210	410	20

5.1　評価項目 b1：PDG を用いた既存手法との比較

この評価では，前向きスライスを使った検出と後向きスライスを用いた検出（表 2 の A1 と A2）と，提案手法を組み合わせて用いた検出の差異を比較した．**表 14** は，検出された正解クローンの数を表している．この表より，提案手法は，すべての場合において既存手法と同数またはより多くの正解クローンを検出できていることが分かる．よって，提案手法を組み合わせて用いたコードクローン検出手法は，従来の手法に比べ検出能力が高いといえる．

表 15 は各検出における頂点比較回数と，検出に要した時間を表している．なお，検出時間は，8 個の論理 CPU を搭載したワークステーションにおいて，同形部分グラフを特定する処理に 6 つのスレッドを割り当てて検出した場合の値である．対象ソースコードの読み込み開始からコードクローン検出結果の出力終了までの時間を表している．この表より，swing を除く 3 つのソフトウェアにおいては，既存の手法に比べて検出に必要であった時間が増加していることが分かる．しかし，netbeans と ant については，検出時間は 40 秒足らずであり，検出時間の増加は特に問題にはならない．jdtcore の場合は検出時間は約 2,200 秒であり，既存手法と比較してかなり実行時間が長くなっている[⋆1]．この結果よりさらなる計算コスト削減手法が必要なことが分かる．

⋆1 2,200 秒というのは，検出時間としてはかなり長いが，計算コスト削減手法によってかなり短縮されている．表 2 の B3 の設定で，計算コスト削減手法をまったく適用せずに検出処理を行うと約 8,200 秒を要した．

表 15　既存手法と提案手法の計算コスト
Table 15　Number of node comparisons of the proposed method and the existing methods.

ソフトウェア	検出 ID	頂点比較回数	検出時間
netbeans	既存手法 A1	135,385	19 秒
	既存手法 A2	551,331	19 秒
	提案手法	38,916,578	39 秒
ant	既存手法 A1	567,891	31 秒
	既存手法 A2	4,942,013	33 秒
	提案手法	7,471,474	34 秒
jdtcore	既存手法 A1	58,155,404	416 秒
	既存手法 A2	346,223,464	671 秒
	提案手法	535,181,098	2,283 秒
swing	既存手法 A1	15,158,651	212 秒
	既存手法 A2	331,698,602	375 秒
	提案手法	60,209,476	142 秒

5.2　評価項目 b2：他の手法との比較

他の手法と比較して，提案手法がどの程度の精度でコードクローンを検出できているのかを評価した．この実験では，文献 7) で用いられているツールを比較対象として用いた．なお，検出ツールの空白行や括弧のみの行の扱いの差による影響を抑えるため，対象ソフトウェアのソースコードは，空白行のみの行を削除し，また括弧のみの行は削除しその括弧は直前の行に付け足す正規化が行われている．各ツールの特徴を**表 16** に示す[⋆2]．

表 17 は各ツールによって検出されたクローン候補の数を表している．Scorpio（提案手法）と Dup，そして CCFinder は非常に多くのクローン候補を検出しているのに対して，CloneDR や Duploc は検出数が少ない．なお，スケーラビリティの問題により，Duploc は swing からはコードクローン検出ができなかった．

図 10 は，各ツールが検出した正解クローンの数を表している．この図より，各対象ソフトウェアについては下記のようにまとめることができる．

- netbeans の場合は，CCFinder が最も多くの正解クローンを検出できていた．しかし，非連続コードクローンに限定した場合は，good 値と ok 値の両者において Scorpio が最も多くの正解クローンを検出できていた．
- ant の場合は，CCFinder が ok 値を用いた評価においてすべての正解クローンを検出

⋆2 文献 7) では表 16 に示すツールに加えて Krinke が開発したツールも比較されているが，このツールは Java 言語には対応していないため，本実験では用いていない．

表 18　既存の検出ツールでは検出できなかったが，Scorpio では検出できた正解クローンの数
Table 18　Number of clone references that Scorpio detected but the other tools did not detect.

(a) good 値を用いた場合

ソフトウェア	Dup		CloneDR		CCFinder		CLAN		Duploc		ALL	
	連続	非連続	連続	非連続	連続	非連続	連続	非連続	連続	非連続	連続	非連続
netbeans	7	9	10	9	6	7	6	8	8	9	1	6
ant	2	0	2	0	3	0	0	0	0	0	2	0
jdtcore	224	128	245	129	250	125	83	115	355	132	21	108
swing	46	13	62	12	86	13	38	12	-	-	4	11

(b) ok 値を用いた場合

ソフトウェア	Dup		CloneDR		CCFinder		CLAN		Duploc		ALL	
	連続	非連続	連続	非連続	連続	非連続	連続	非連続	連続	非連続	連続	非連続
netbeans	7	6	22	13	7	3	18	12	15	5	1	3
ant	0	0	8	0	0	0	3	0	2	0	0	0
jdtcore	73	105	518	204	75	152	196	96	721	211	1	15
swing	15	7	206	20	2	4	305	20	-	-	0	0

表 16　比較対象ツール
Table 16　Detection tools compared in this experiment.

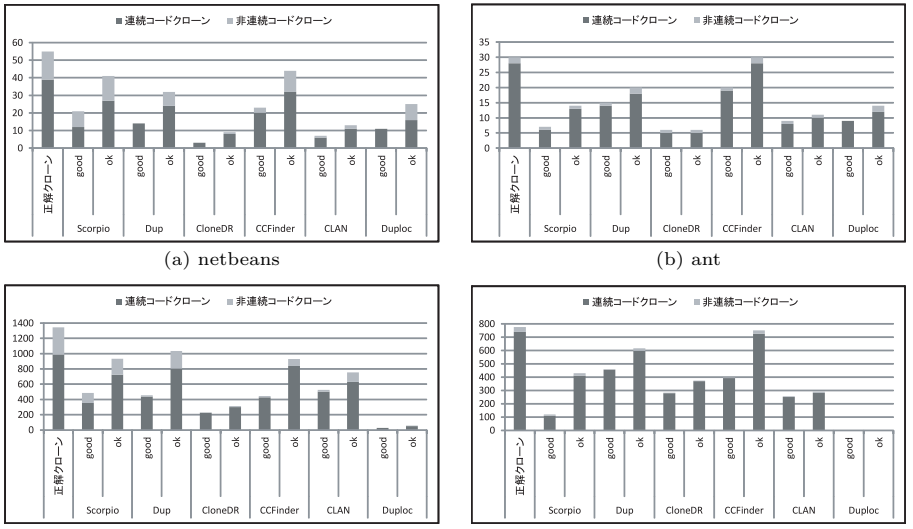
開発者	ツール名	検出方法
Baker	Dup	字句
Baxter	CloneDR	抽象構文木
Kamiya	CCFinder	字句
Merlo	CLAN	メトリクス
Rieger	Duploc	行

表 17　各ツールによって検出されたクローン候補の数
Table 17　Number of clone candidates detected by each of the tools.

ソフトウェア	検出ツール					
	Scorpio	Dup	CloneDR	CCFinder	CLAN	Duploc
netbeans	10,201	344	33	5,552	80	223
ant	1,339	245	42	865	88	162
jdtcore	182,464	22,589	3,593	26,049	10,111	710
swing	31,518	7,220	3,766	21,421	2,809	-

できていた．Scorpio は，good 値を用いた評価では 5 番目，ok 値を用いた評価では，3 番目の検出数であった．

- jdtcore の場合は，good 値と ok 値の評価において，Scorpio は 2 番目の検出数であっ



(a) netbeans　(b) ant

(c) jdtcore　(d) swing

図 10　各ツールによって検出された正解クローンの数
Fig. 10　Number of clone references detected by each of the tools.

た．非連続コードクローンに限定した場合は，good 値を用いた評価において Scorpio が最も多くの正解クローンを検出できていた．この場合は，Scorpio は 130 個の非連続な正解クローンを検出できていたが，2 番目に検出数が多い CLAN では，24 個の正解クローンしか検出できていなかった．

- swing の場合は，good 値を用いた評価において Scorpio の検出結果は良くなく，再現率は約 15%であった．一方 ok 値を用いた評価はあまり悪くはなく，ほぼ半数の正解クローンを検出できていた．

最後に，他のツールが検出できなかった正解クローンを，Scorpio がどの程度検出できていたのかを調査した．調査結果を**表 18** に示す．たとえば，good 値を用いた評価では，netbeans において，Dup は検出できなかったが Scopio が検出できた正解クローンが 16 個あったことが分かる．また，最右列の ALL は，他のどのツールも検出できなかったが Scorpio が検出できた正解クローンの数を表している．この表より，Scorpio は他のどのツールに対しても新たに正解クローンを検出できており，新しい検出法として有益であることが分かる．特に，多くの場合において，Scorpio は既存のツールでは検出できない連続コードクローンを検出できており，PDG を用いた検出手法の弱点であった，連続コードクローンの検出能力が他の手法に比べて劣る点を改善できていることが分かる．

6. 実験結果の妥当性について留意すべき点

本論文の実験では，Bellon らが作成したコードクローンの正解集合を用いて提案手法の有効性について調査を行った．しかし，この正解集合は対象ソフトウェアに含まれるすべてのコードクローンではない．そのため，すべての正解クローンを対象にして同様の実験を行った場合は，実験結果が異なる可能性がある．

また，すべての正解クローンが得られていないため，検出結果の適合率については調査ができていない．コードクローン検出手法の適合率とは，検出結果のどの程度が正解クローンなのかを表す指標であり，検出手法の能力を評価するうえで非常に重要な尺度である．再現率と適合率の両方の変化を見ることにより初めて公正な評価となる．

図 10 で示すように，ほぼすべての場合において，1 つのツールですべての正解クローンを検出できてはいない．これは，現状では十分にクローン検出ができるツールはないことを示している．本論文では，PDG を用いたコードクローン検出の改善手法を提案し，実験により検出結果が従来手法に比べて改善されていることを示した．しかし，すべての正解クローンが検出できているわけではないため，さらなる改善手法および新しい検出手法が必要である．

7. 関 連 研 究

Komondoor らは初めて PDG を用いたコードクローン検出を行った[17]．彼らの手法では，ソースコード中の文と条件式が PDG の頂点である．すべての頂点からハッシュ値が算出され，ハッシュ値が等しい頂点が同じグループに分類される．グループ内のすべての頂点のペアから，それらを含む同形部分グラフが求められ，それらがコードクローンとされる．同形部分グラフの特定には主に後向きスライスが用いられ，前向きスライスは，条件式の頂点が同形部分グラフに追加されたときのみ用いられる．文献 17) の実験では，PDG を用いた手法の高い検出能力が示されたが，約 11,000 行のソフトウェアからの検出に約 90 分を要する[⋆1]など，その実用性の低さも顕著であった．

Krinke は細粒度 PDG を用いたコードクローン検出手法を提案している[18]．細粒度 PDG では，頂点は抽象構文木の頂点とほぼ 1 対 1 で対応しており，通常の PDG に比べて頂点数がはるかに多くなる．そのため，単純に検出処理を行った場合は，Komondoor らの手法[17] に比べてさらに大きな計算コストを必要とする．この問題に対処するため，Krinke は k-limited search という手法を提案している．k-limited search では，同形部分グラフはスライス基点の k ホップ内でのみ探索されるため計算コストを抑えることができる．しかし，最も大きな同形部分グラフが特定されない可能性がある．Krinke の手法では，前向きスライスが用いられており，計算コストを抑えるために，スライス基点となる頂点は条件式の頂点に限定されている．文献 18) の実験では，k-limited search により計算コストを大幅に抑えれることが示された．k が 20 以下の場合は，検出処理は数分以内で完了したが，k が 25 以上の場合は，約 25,000 行のソフトウェアからの検出に約 46 時間を要する[⋆2]など，大きなコードクローンを検出するには，高い計算コストが必要であった．

Jia らは，非連続コードクローンを検出するための複合型検出手法を提案している[14]．まず，字句単位での検出手法を用いて連続コードクローンを検出する．その後，その周辺にあるコードで，検出された連続コードクローンと制御依存やデータ依存があり，対応するコードの周辺にも同様の周辺コードがある場合は，その周辺コードがコードクローンに併合され

⋆1 この計算時間は 2000 年に公開された文献 17) に掲載されていた結果である．現在のコンピュータを用いれば，この検出時間に比べてかなり短い時間で計算を終えることが可能であろう．
⋆2 この計算時間は 2001 年に公開された文献 18) に掲載されていた結果である．現在のコンピュータを用いれば，この検出時間に比べてかなり短い時間で計算を終えることが可能であろう．

る．実験では，既存の PDG を用いた検出ツールである Duplix[18] に比べて高速にコードクローンを検出することに成功している．彼らの手法は，連続コードクローンと非連続コードクローンの双方を検出するのに長けている．本論文で提案する手法との違いは，Jia らの手法はコードクローンの核となる一定以上の長さの連続した重複コードが必要であるが，提案手法ではそのような核となる部分は必要ない．しかしながら，Jia らの手法は提案手法に比べて高速にコードクローンを検出することが可能である．

Liu らは PDG を用いたプログラム剽窃検出法を提案している[19]．彼らはプログラム剽窃で起こるであろう修正を想定し，そのような修正はソースコード上の表面的な偽装（変数名を変える，依存関係のない文の順序を変えるなど）であるため PDG は変化しないことを表し，PDG を用いた検出が有効であると主張している．彼らはメソッド単位の剽窃を目的としているが，すべてのメソッドの PDG を詳細に比較するのは高い計算コストを必要とするため，計算コストを抑える 2 つのフィルタリング手法も提案している．1 つ目のフィルタリングは，もし比較する 2 つのメソッドのサイズが大きく異なる場合は，それらの PDG は比較しないというものである．2 つ目のフィルタリングは，もし比較する 2 つのメソッドの PDG 内に存在している頂点の種類とその数を表すヒストグラムの形が大きく異なる場合は，それらの PDG は比較しないというものである．これら 2 つのフィルタリング手法を用いることによって，PDG の比較に要する計算コストの約 90%が削減されたことが実験により示された．Liu らの手法はメソッド単位での重複を見つけることを前提にしており，メソッドの一部分が類似している場合は検出することができない．

Gabel らは意味的に等しい処理部分をコードクローンとして検出するスケーラビリティの高い手法を提案している[11]．この手法は彼らの研究グループにおいてかつて提案された抽象構文木（AST）を用いた検出法[15] を発展させたものである．彼らは semantic thread というものを提案し，PDG での同形グラフ検出を AST における同形部分木検出に対応させている．彼らは商用のソースコード解析ツール CodeSurfer[2] を利用して，実用的なツールを開発している．開発したツールは複数のオープンソースソフトウェアに対して適用されている．実験によりそのツールのコードクローン検出は非常に高速であり，たとえば，PDG が与えられた後の処理時間は，Linux カーネル全体からのコードクローン検出に要するのに数分である．また実験では，提案手法と以前提案した AST を用いた手法の検出結果を比較し，前者は非連続コードクローンを検出できるため，後者に比べて大きなコードクローンを検出できていることを述べた．

筆者らは，提案手法と，Jia らの手法[14]，Gabel らの手法[11] において検出されるコードクローンがどのように違うのかに興味を持っている．これら 3 つの手法は，連続コードクローンと非連続コードクローンを検出する高い能力を持っているが，用いている検出技術は互いに異なる．本論文で提案した手法は，PDG のみを用いた手法であり，Jia らは字句単位の検出とデータフロー・制御フロー解析を組み合わせたもの，Gabel らは PDG と AST を組み合わせた手法である．用いている検出技術の違いが，検出されるコードクローンの差にどのように効いているのかを調査することは興味深い研究である．

8. おわりに

本論文では，PDG を用いたコードクローン検出法の性能を改善する手法を提案した．提案内容は，(1) 双方向スライスを用いる，(2) 実行依存の導入，(3) 計算コスト削減手法，(4) 誤検出削減手法の 4 つであり，各提案内容が，コードクローン検出能力の向上，および計算コストの削減に対して有効であることを実験により確かめた．また，本論文の実験では，提案内容を他の検出ツールと比較した．その結果，提案手法は，必ずしも他の手法に比べて多くのコードクローンを検出できるとは限らないが，他の手法では検出できていないコードクローンを検出できてきていることが分かった．

また，現在は，メソッド単位の PDG を用いてコードクローンを検出しているが，今後は対象システム全体の PDG からの検出を行う予定である．これにより，複数のメソッドにまたがる処理を 1 つのコードクローンとして検出することが可能となり，より有益で興味深いコードクローンの検出が見込まれる．また，複数のソフトウェア間でのコードクローン検出も予定している．現在のところ，PDG を用いた検出手法でソフトウェア間のコードクローンを検出している研究はない．Livieri らは 7,000 以上のシステムからコードクローンを検出し，検出されたコードクローンの分析を行っている[20] が，彼らは字句単位でのコードクローンを検出しているため，非連続コードクローンは分析の対象外であった．システム間の非連続コードクローンを検出し，分析することで新たな発見があると筆者らは期待している．

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（C）（課題番号：20500033），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．

参 考 文 献

1) Clone Detection Literature. http://www.cis.uab.edu/tairasr/clones/literature/
2) CodeSurfer. http://www.grammatech.com/
3) Detection of Software Clones. http://bauhaus-stuttgart.de/clones/
4) Scorpio. http://www-sdl.ist.osaka-u.ac.jp/higo/cgi-bin/moin.cgi/scorpio-e/
5) Balint, M., Girba, T. and Marinescu, R.: How Developers Copy, *Proc. 14th IEEE International Conference on Program Conprehension*, pp.56–68 (2006).
6) Baxter, I., Yahin, A., Moura, L., Anna, M. and Bier, L.: Clone Detection Using Abstract Syntax Trees, *Proc. International Conference on Software Maintenance 98*, pp.368–377 (1998).
7) Bellon, S., Koschke, R., Antoniol, G., Krinke, J. and Merlo, E.: Comparison and Evaluation of Clone Detection Tools, *IEEE Trans. Softw. Eng.*, Vol.31, No.10, pp.804–818 (2007).
8) Burd, E. and Bailey, J.: Evaluating Clone Detection Tools for Use during Preventative Maintenance, *Proc. 2nd IEEE International Workshop on Source Code Analysis and Manipulation*, pp.36–43 (2002).
9) Ferrante, J., Ottenstein, K.J. and Warren, J.D.: The Program Dependence Graph and Its Use in Optimization, *ACM Trans. Prog. Lang. and Syst.*, Vol.9, No.3, pp.319–349 (1987).
10) Fowler, M.: *Refactoring: Improving the design of existing code*, Addison Wesley (1999).
11) Gabel, M., Jiang, L. and Su, Z.: Scalable Detection of Semantic Clones, *Proc. 30th International Conference on Software Engineering*, pp.321–330 (2008).
12) 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).
13) Higo, Y., Kamiya, T., Kusumoto, S. and Inoue, K.: Method and Implementation for Investigating Code Clones in a Software System, *Information and Software Technology*, Vol.49, No.9-10, pp.985–998 (2007).
14) Jia, Y., Binkley, D., Harman, M., Krinke, J. and Matsusita, M.: KClone: A Proposed Approach to Fast Precise Code Clone Detection, *Proc. 3rd International Conference on Software Clones* (2009).
15) Jiang, L., Misherghi, G., Su, Z. and Glondu, S.: DECKARD: Scalable and Accurate Tree-based Detection of Code Clones, *Proc. 29th Intenational Conference on Software Engineering*, pp.96–105 (2007).
16) Komondoor, R. and Horwitz, S.: Using Slicing to Identify Duplication in Source Code, *Proc. 8th International Symposium on Static Analysis*, pp.40–56 (2001).
17) Komondoor, R. and Horwitz, S.: Semantics-Preserving Procedure Extraction, *Proc. 27th ACM SIGPLAN-SIGACT on Principles of Programming Languages*, pp.155–169 (2000).
18) Krinke, J.: Identifying Similar Code with Program Dependence Graphs, *Proc. 8th Working Conference on Reverse Engineering*, pp.301–309 (2001).
19) Liu, C., Chen, C., Han, J. and Yu, P.S.: GPLAG: Detection of Software Plagiarism by Program Dependence Graph Analysis, *Proc. 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pp.872–881 (2006).
20) Livieri, S., Higo, Y., Matsushita, M. and Inoue, K.: Very-Large Scale Code Clone Analysis and Visualization of Open Source Program Using Distributed CCFinder: D-CCFinder, *Proc. 29th International Conference on Software Engineering*, pp.106–115 (2007).
21) Monden, A., Nakae, D., Kamiya, T., Sato, S. and Matsumoto, K.: Software Quality Analysis by Code Clones in Industrial Legacy Software, *Proc. 8th IEEE International Software Metrics Symposium*, pp.87–94 (2002).
22) Roy, C.K. and Cordy, J.R.: A Survey on Software Clone Detection Research, Technical report, School of Computing, Queen's University (2007).

付　　録

定義 1　2 つのコード片（f_1 と f_2）の重なりの程度を以下の式で定義する．なお，$lines(f)$ はコード片 f に含まれる行の集合を表す．

$$overlap(f_1, f_2) = \frac{|lines(f_1) \cap lines(f_2)|}{|lines(f_1) \cup lines(f_2)|} \tag{7}$$

定義 2　あるコード片（f_1）が他のコード片（f_2）に含まれている程度を以下の式で定義する．

$$contain(f_1, f_2) = \frac{|lines(f_1) \cap lines(f_2)|}{|lines(f_1)|} \tag{8}$$

定義 3　2 つのクローンペア（p_1 と p_2）の good 値は以下の式で定義される．

$$good(p_1, p_2) = min(overlap(p_1.f_1, p_2.f_1), \\ overlap(p_1.f_2, p_2.f_2)) \tag{9}$$

図 11 には 2 つのクローンペア（p_1 と p_2）が存在している．この場合，good 値は，

$$good(p_1, p_2) = min\left(\frac{5}{8}, \frac{6}{8}\right) = \frac{5}{8}$$

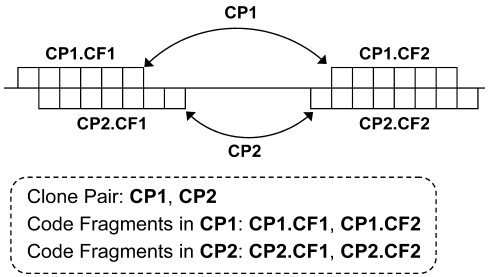


図 11　good 値と ok 値の算出例
Fig. 11　Sample calculation of good and ok values.

となる．閾値が 0.7 の場合，$\frac{5}{8} \leq 0.7$ であるため，2 つのクローンペアは一致しない．

定義 4　2 つのクローンペア（p_1 と p_2）の ok 値は以下の式で定義される．

$$\begin{aligned} ok(CP_1, CP_2) = min(&max(contained(CP_1.CF_1, CP_2.CF_1),\\ &\quad contained(CP_2.CF_1, CP_1.CF_1)),\\ &max(contained(CP_1.CF_2, CP_2.CF_2),\\ &\quad contained(CP_2.CF_2, CP_1.CF_2))) \end{aligned} \tag{10}$$

図 11 の例を用いた場合，ok 値は，

$$ok(p_1, p_2) = min\left(max\left(\frac{5}{6}, \frac{5}{7}\right), max\left(\frac{6}{6}, \frac{6}{8}\right)\right) = \frac{5}{6}$$

となる．閾値が 0.7 の場合，$0.7 \leq \frac{5}{6}$ であるため，2 つのクローンペアは一致する．

定義 5　コードクローンの正解集合を S_{refs}，ok 値を用いてコードクローン検出結果 R から抽出した正解クローンの集合を $S_{ok}(R)$，good 値を用いた正解集合を $S_{good}(R)$ としたとき，R の再現率はそれぞれ以下の式で定義される．

$$Recall_{ok}(R) = \frac{|S_{ok}(R)|}{|S_{refs}|} \tag{11}$$

$$Recall_{good}(R) = \frac{|S_{good}(R)|}{|S_{refs}|} \tag{12}$$

定義 6　コードクローンの検出結果を r_1，r_2，各検出結果に含まれるプログラムの要素[⋆1]の集合を $T(r_1)$，$T(r_2)$ としたとき，r_1 の r_2 に対する被覆率は以下の式で定義される．

$$coverage(r_1, r_2) = 100 \times \frac{|\, T(r_1) \cap T(r_2) \,|}{|\, T(r_1) \,|} \tag{13}$$

被覆率は，r_1 が r_2 の要素をすべて含んでいるときに 100（最大値），まったく含んでいないときに 0（最小値）となる．

(平成 22 年 1 月 25 日受付)
(平成 22 年 7 月 26 日採録)

肥後　芳樹（正会員）
平成 14 年大阪大学基礎工学部情報科学科中退．平成 18 年同大学大学院博士後期課程修了．平成 19 年同大学院情報科学研究科コンピュータサイエンス専攻助教．博士（情報科学）．ソースコード分析，特にコードクローン分析やリファクタリング支援に関する研究に従事．電子情報通信学会，IEEE 各会員．

楠本　真二（正会員）
昭 63 年大阪大学基礎工学部卒業．平成 3 年同大学大学院博士課程中退．同年同大学基礎工学部助手．平成 8 年同講師．平成 11 年同助教授．平成 14 年同大学大学院情報科学研究科助教授．平成 17 年同教授．博士（工学）．ソフトウェアの生産性や品質の定量的評価に関する研究に従事．電子情報通信学会，IEEE，IFPUG 各会員．

⋆1 プログラムの要素は，用いる検出手法によって異なる．たとえば，行単位での検出手法であればプログラム要素は行であり，字句単位での検出手法であれば字句となる．提案手法は PDG を用いた手法であるのでプログラムの要素は，PDG の頂点，つまりプログラムの文となる．