NAIST-IS-MT1051122

修士論文

2段階のクラスタリングを用いた Near-Miss コードクローンの検出

吉岡 俊輔

2012年2月2日

奈良先端科学技術大学院大学
情報科学研究科 情報システム学専攻

本論文は奈良先端科学技術大学院大学情報科学研究科に
修士 (工学) 授与の要件として提出した修士論文である。

吉岡 俊輔

審査委員：

飯田 元 教授	（主指導教員）
関 浩之 教授	（副指導教員）
松本 健一 教授	（副指導教員）
吉田 則裕 助教	（副指導教員）

2段階のクラスタリングを用いた Near-Miss コードクローンの検出*

吉岡 俊輔

内容梗概

ソフトウェアの保守性を低下させる主な要因として，ソースコード中に含まれる重複したコード (コードクローン) の存在が指摘されている．そのためこれまでに多くの研究者によってコードクローンの検出手法が提案されてきた．しかし従来の手法ではステートメントの追加や削除などの変更によって生成された Near-Miss クローンの検出が困難であるという問題があった．また一部のコードクローン検出ツールは Near-Miss クローンの検出は可能ではあるが，検出対象のソースコードの規模が大きくなると計算量が爆発的に増えるという問題があった．本論文では 2 段階に分けたクラスタリングを用いることにより，ソースコードの規模に比例した時間で Near-Miss クローンの検出が可能であることを確認した．

キーワード

コードクローン, リファクタリング支援, LSH アルゴリズム, 頻出データマイニング

* 奈良先端科学技術大学院大学 情報科学研究科 情報システム学専攻 修士論文, NAIST-IS-MT1051122, 2012 年 2 月 2 日.

Near-Miss Clone Detection Technique Using Two Stage Clustering*

Shunsuke Yoshioka

Abstract

Duplicated code is pointed out as one of the main factors that increase the maintainability of software. So far, much research has been done on clone detection. However, previous detection techniques do not detect near-miss clones derived by the addition or the deletion of statements, or take much computational cost for detecting near-miss clones from large scale software. This thesis presents a technique using two-stage clustering for near-miss clone detection, and show the detection time proportional to the size of source code.

Keywords:

code clone, refactoring, LSH algorithm, frequent pattern mining

* Master's Thesis, Department of Information Systems, Graduate School of Information Science, Nara Institute of Science and Technology, NAIST-IS-MT1051122, February 2, 2012.

目 次

図 目 次

表 目 次

1. はじめに

近年，情報技術の急速な進歩によって我々を取り巻く環境はめまぐるしく変化している．情報技術は携帯電話や ATM などありとあらゆるところにソフトウェアとして導入されており，その機能は年々高度化・複雑化している．その結果ソフトウェアのソースコードは大規模化し，同時に品質の悪化が問題となっている．ソフトウェアの品質を悪化させる要因はいくつか存在するが，その中の一つとしてコードクローンの存在が指摘されている [1] [2].

コードクローンとはソースコード中に含まれる重複，もしくは類似したコードの断片（以下「コード片」と呼ぶ）のことを指す．コードクローンがソフトウェアの品質を悪化させる原因のひとつとして，あるコード片に欠陥が含まれていた場合，そのコード片とクローンの関係にある別のコード片にも同様の欠陥が含まれている可能性がある，という点があげられる．例えば図 1 に示す File B と File C の網目部分は File A の網目部分をコピーして作成されたものとする．もし File B の網目部分から欠陥が見つかった場合，同様の欠陥が File A，File C の網目の部分にも含まれている可能性がある．

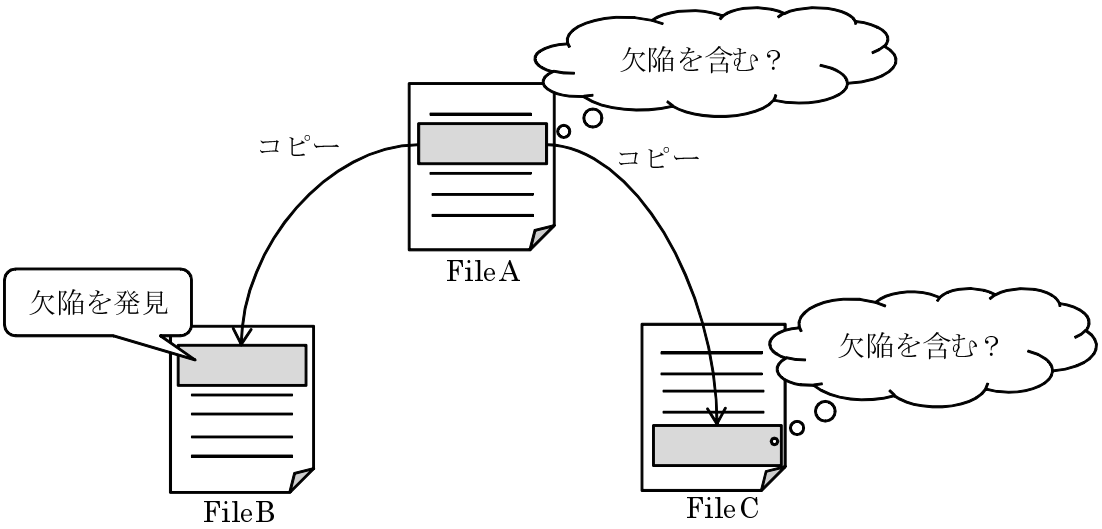


図 1 コードクローンの概要

一般にソフトウェアの開発現場では各開発者の判断でソースコードのコピーアンドペーストが行われているため [3]，どのコード片同士がクローンの関係にあるかといった管理がなされていない．さらに，大規模なソフウェアではソースコードの行数は数百万行から数千万行にもおよび，その中からクローンの関係にあるコード片をすべて見つけ出すことは極めて困難である．これは，開発者がコード片をコピーアンドペーストした後で識別子 (変数名や関数名など) を変更した場合，正規表現をはじめとした従来の検索技術でそのすべてを見つけ出すことが困難なためである．(このように，類似はしているがしかし完全には一致していないコードクローンを「Near-Miss クローン」と呼ぶ)．その結果取り除くべき欠陥が取り除かれないまま製品として出荷され，流通後に問題を引き起こす．

この問題に対し，上記で述べたような変更を認識した上で検索を行う手法が提案されてきた．しかし，これらの手法では，識別子の変更は認識できても他の種類の変更 (式・ステートメントの挿入・削除など) を認識した上で検索を行うことができない，あるいは検索は可能であったとしても計算に膨大な時間がかかるという問題がある．

本論文ではこれらの問題点を踏まえた上で「識別子以外に対して変更を加えたコードクローンを高速に検出するための手法」を提案する．

本論文の章構成は以下の通りである．第 1 章では研究の背景と目的について述べた．次の第 2 章ではコードクローンの定義および，その発生原因について述べる．第 3 章ではこれまでに提案されてきたコードクローンの検出手法について述べる．第 4 章では従来のコードクローンの検出手法の問題点について述べる．第 5 章では新たに提案するコードクローンの検出手法について述べる．第 6 章では提案した検出手法の評価結果について述べる．第 7 章では提案した検出手法に対する考察を述べる．第 8 章では本論文のまとめについて述べる．

2. コードクローン

2.1 コードクローンの定義

コードクローンとはソースコード中の類似したコード片のことを指す．コード片の類似の仕方には様々な形態が考えられる．例えばコード片に含まれる文字列の並びが完全一致している場合，それは類似したコード片の一例としてあげられる．また別の例として，コード片に含まれる識別子 (変数名や関数名など) は異なっていてもそれ以外 (ステートメントの並びや制御の流れ) が同じであるものは類似したコード片と考えられる．このようにコード片の類似の仕方には様々な形態が考えられるが，どのような形態をコードクローンとするか厳密かつ普遍的な定義は存在しない．したがって従来の研究では各研究者が個々にコードクローンを定義していた．しかし 2000 年に入り Bellon らによってコードクローンの類似の形態に基づく分類が行われた [4]．Bellon らはコードクローンを TYPE-1，TYPE-2，TYPE-3 の 3 つの形態に分類している．

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を除いて，完全に一致するコードクローンを指す．図 2 はその具体例である．左右のコード片を比較すると改行の位置や括弧の位置が異なっているが，それ以外は完全に一致している．

TYPE-2

変数名や関数名といった識別子，または変数の型などの一部の予約語やリテラルが異なるコードクローンを指す．図 3 はその具体例である．左右のコード片を比較すると，5 行目の文字列リテラルや 8，9，10 行目の変数名や 11 行目の関数名が異なるが，それ以外は一致している．

TYPE-3

TYPE-2 の変更に加えて文の挿入や削除，入れ替えなどが行われたコードクローンを指す．図 4 はその具体例である．左右のコード片を比較すると

図3で見られた変更に加えて左の8, 9行目のステートメントが削除されている.

```
 1  private void read()
 2  {
 3    if (args.length < 3)
 4    {
 5      print("invalid args");
 6        return ;
 7    }
 8    String in= args[0];
 9    String out = args[1];
10    int max =args.count;
11    create(in, out, max);
12  }
```

```
 1  private void read() {
 2
 3    if (args.length < 3) {
 4
 5      print("invalid args");
 6        return ;
 7    }
 8    String in= args[0];
 9    String out = args[1];
10    int max =args.count;
11    create(in, out, max);
12  }
```

図 2 TYPE-1 クローンの例

```
 1  private void read()
 2  {
 3    if (args.length < 3)
 4    {
 5      print("invalid args");
 6        return ;
 7    }
 8    String in= args[0];
 9    String out = args[1];
10    int max =args.count;
11    create(in, out, max);
12  }
```

```
 1  private void read()
 2  {
 3    if (args.length < 3)
 4    {
 5      print("fatal error");
 6        return ;
 7    }
 8    String In = args[0];
 9    String Out = args[1];
10    int Max = args.count;
11    CreateNew(In, Out, Max);
12  }
```

図 3 TYPE-2 クローンの例

```
 1  private void read()
 2  {
 3    if (args.length < 3)
 4    {
 5      print("invalid args");
 6        return ;
 7    }
 8    String in= args[0];
 9    String out = args[1];
10    int max =args.count;
11    create(in, out, max);
12  }
```

```
 1  private void read()
 2  {
 3    if (args.length < 3)
 4    {
 5      print("fatal error");
 6        return ;
 7    }
 8
 9
10    int Max = args.count;
11    CreateNew(_in,_out,Max);
12  }
```

図 4 TYPE-3 クローンの例

近年のコードクローンの研究では Bellon らの定義に基づいて述べられることが多い．本論文でも最新の研究動向を踏まえ Bellon らの定義を採用する．

2.2 コードクローンの発生原因

コードクローンの発生原因として以下の理由が指摘されている [5] [6].

コピーアンドペースト

ソースコードのコピーアンドペーストによりコードクローンが発生する可能性がある．ソフトウェアの開発現場ではコピーアンドペーストが頻繁に行われている．これはゼロからコードを記述するのではなく，既存のコードを再利用して記述することが多いためである．

定型処理

記述する処理内容が定型的な処理である場合，コードクローンが発生する可能性がある．例えばファイルの入出力やデータベースのクエリーの処理

などは定型的な記述スタイルがあり，誰がコーディングしても類似したコードになる傾向がある．

繰り返し要素

変数の宣言や初期化，文字列の連結処理，switch-case 文などはコードクローンの原因となる可能性がある．これらの処理はソースコード中の一か所にまとめて記述することが多く，また記述の形式が単純構造の繰り返しであることが多い．そのためコードクローンの原因となることがある．図 5 はその具体例である．

```
 1  case 1:
 2       s = "No Errer.";
 3       break;
 4  case 2:
 5       s = "IO Errer.";
 6       break;
 7  case 3:
 8       s = "Invalid IP.";
 9       break;
10  case 4:
11       s = "Invalid Name.";
12       break;
13  case 4:
14       s = "Invalid DNS.";
15       break;
```

```
 1  case 1:
 2       ext = ".jpg";
 3       break;
 4  case 2:
 5       ext= ".png";
 6       break;
 7  case 3:
 8       ext = ".mag";
 9       break;
10  case 4:
11       ext = ".bmp";
12       break;
13  case 4:
14       ext = ".gif";
15       break;
```

図 5 繰り返し要素の例

コードジェネレータによる生成コード

GUI プログラミングにおけるウィンドウの処理コード，あるいは構文解析を行う際の字句解析・構文解析などの処理コードはコードジェネレータにより自動的に生成されることがある．例えば Visual Studio，Eclipse などの RAD は前者の処理コードを自動的に生成する．また yacc/lex，ANTLR な

```
1   this.panel1.BackColor = System.Drawing.Color.Gray;
2   this.panel1.Dock = System.Windows.Forms.DockStyle.Fill;
3   this.panel1.Location = new System.Drawing.Point(0, 0);
4   this.panel1.Name = "tableLayoutPanel1";
5   this.panel1.Size = new System.Drawing.Size(544, 320);
6   this.panel1.TabIndex = 1;
```

```
1   this.panel1.BackColor = System.Drawing.Color.Gray;
2   this.panel1.Dock = System.Windows.Forms.DockStyle.Fill;
3   this.panel1.Location = new System.Drawing.Point(580, 0);
4   this.panel1.Name = "tableLayoutPanel2";
5   this.panel1.Size = new System.Drawing.Size(544, 320);
6   this.panel1.TabIndex = 2;
```

図 6 VisualStudio によって自動生成されたクローンの例

どのコンパイラジェネレータは後者の処理コードを自動的に生成する．これらのツールはある入力値に対して機械的にソースコードを出力するため，コードクローンが生じる原因となる．図 6 は Visual Studio のフォームデザイナによって自動的に生成されたコード片である．

共通機能の抽出失敗

設計段階で共通機能として抽出すべきものを見落とした場合にコーディング時になって類似した処理内容を複数個所で記述しなければならない場合があり，その結果コードクローンが発生することがある．

プログラミング言語の機能不足

プログラミング言語の機能的な制約からコードクローンが発生する可能性がある．例えば C 言語は C++や Java のようにクラスの継承やポリモーフィズムを想定した言語仕様ではない．そのため機能の共通化が困難な場合があり，その結果類似した処理内容を複数箇所に記述しなければならないことがある．

パフォーマンス改善

組み込みシステムにおけるリアルタイム処理など特定の環境下でプログラムを実行する場合，パフォーマンスを考慮したコーディングが必要となる場合がある．パフォーマンスを向上させるための技法として集約可能なコードを意図的に展開して記述するという手法がある．この手法を用いた結果コードクローンが発生する場合がある．

3. 関連研究

これまでに数多くのコードクローンの検出手法が提案されてきた．それらは，トークンの並びに着目した手法，プログラム依存グラフの構造に着目した手法，抽象構文木の構造に着目した手法に大別される．また近年の動向として最近傍探索によるクラスタリングを用いた検出手法がいくつか提案されている．以下，それぞれの手法について簡単に述べる．

3.1 トークンの並びに着目した検出手法

トークンとはソースコード中の終端記号のことを指す．C，C++，Java などでは `int`，`extern`，”`abc` ”，`123` などがトークンに該当する．Kamiya ら，Li らはトークンの並びを着目したコードクローンの検出手法を提案している [6] [7].

Kamiya らはトークンの並びに接尾辞木アルゴリズムを適用したコードクローンの検出手法を提案している [6]．接尾辞木アルゴリズムとは文字列中から類似した文字の並びを検出するためのアルゴリズムである．古くは DNA やたんぱく質の解析などに用いられてきた．Kamiya らは本手法をベースに CCFinder というツールを作成し，TYPE-1，TYPE-2 のコードクローンの検出に成功している．

Li らはトークンの並びに頻出系列マイニング [7] を適用したコードクローンの検出手法を提案している [8]．頻出系列マイニングとはあるデータ列中から頻出する連続・非連続のデータ列を検出することを指す．元々はマーケティングなどに用いられていた．Li らは本手法をベースに CP-Miner というツールを作成し，TYPE-1，TYPE-2，TYPE-3 のコードクローンの検出に成功している．

3.2 プログラム依存グラフを用いた検出手法

プログラム依存グラフとはコンパイラの最適化やソースコードの静的解析などに用いられてきた技術で，関数内の制御フローとデータの依存の関係を 2 種類の矢印で表した有効グラフのことを指す．図 7 はその具体例である．左図がソースコード，右図は左図のソースコードに対するプログラム依存グラフである．

```
 1  void read() {
 2   if (!isNull()) {
 3     if (null == path) {
 4       Title ttl = getName();
 5       setText(ttl.getCode) {
 6       }
 7     }
 8     Title t2 = new Title();
 9     t2.initTime();
10     t2.initName();
11     t2.append("ver");
12     t2.items.clear();
13     return;
14   } else {
15     return;
16   }
17  }
```

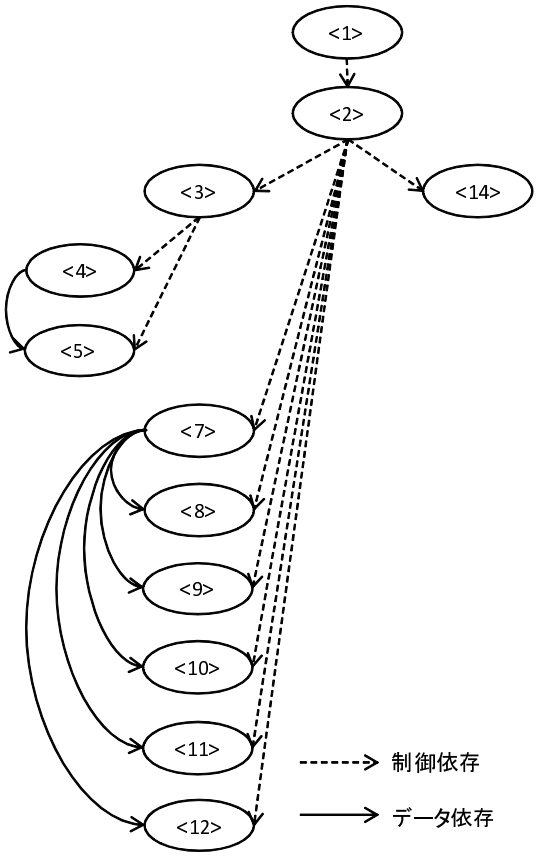


図 7 プログラム依存グラフの例

まずグラフ作成の起点となるステートメントを定める．次にその起点と関わりのあるデータの参照・代入（これらを「データ依存」という），また条件分岐やループなど（これらを「制御依存」という）を抽出し，ステートメントをノード，依存関係（データ依存・制御依存）を矢印とした有向グラフで記述する．図 7 は 1 行目のステートメントを起点とした場合のプログラム依存グラフである．どのステートメントを起点とするかによりグラフの構造が異なる．

プログラム依存グラフを用いたコードクローンの検出では，類似した構造のグラフを探し出すことにより検出を行う．しかし一般的にグラフの構造の比較には時間がかかる．そのため比較方法に工夫を凝らしたコードクローンの検出手法が Komondoor ら，Higo らによって提案されている [9] [10].

Komondoor らはプログラム依存グラフの各ノードをハッシュ値に変換し，同じハッシュ値でノードをグルーピングすることによりグラフの比較にかかる時間の削減に成功している [9]．Komondoor らは本手法により TYPE-1，TYPE-2，TYPE-3 のコードクローンの検出に成功している．

また Higo らは Komondoor らの手法をベースに，プログラム依存グラフにステートメントの実行順序の関係 (実行依存) を追加した，拡張プログラム依存グラフによる検出を行っている [10]．実行依存という新たな概念を追加したことにより Komondoor らの手法よりも検出の精度を高めている．Higo らは本手法を用いて Scorpio というツールを作成し， TYPE-1，TYPE-2，TYPE-3 のコードクローンの検出に成功している．

3.3 抽象構文木を用いた検出手法

抽象構文木とはソースコードの構文構造をツリー状で表したグラフのことを指す．元々はコンパイル時の文法解析やソースコードの静的解析などに用いられてきた．図 8 に抽象構文木の例を示す．抽象構文木を用いたコードクローンの検出では，類似したツリー構造を探し出すことにより検出を行う．しかし一般的にツリーの比較には時間がかかるため，比較方法を工夫したコードクローンの検出手法が Baxter ら，Jiang らによって提案されている [11] [12].

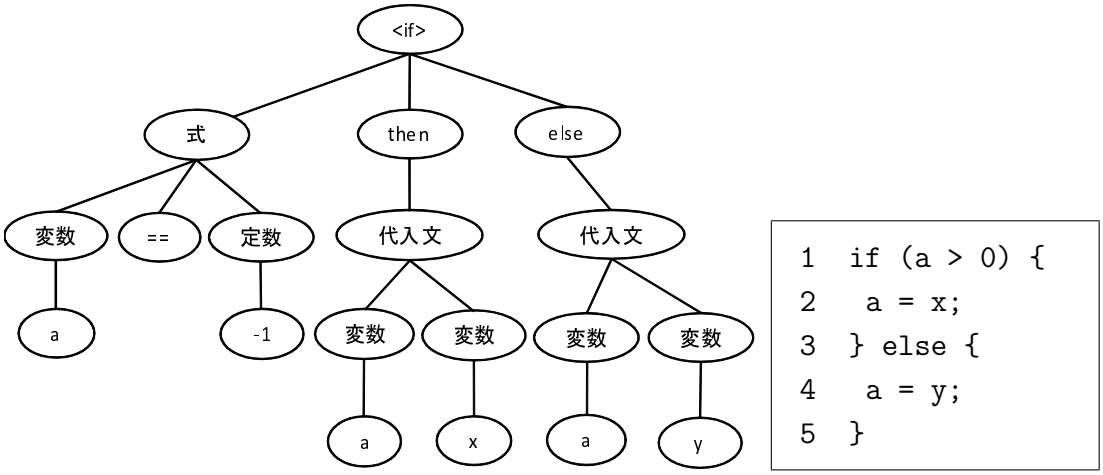


図 8 抽象構文木の例

Baxter らは抽象構文木から各部分木のハッシュ値を計算し，同一のハッシュ値を持つ部分木のみを比較することで計算量の削減を行っている [11]．Baxter らは本手法を用いて CloneDR というツールを作成し，TYPE-1，TYPE-2 のコードクローンの検出に成功している．

Jiang らは抽象構文木の各部分木の特徴量を計算し，後述の最近傍探索によるクラスタリングを用いることでコードクローンの検出を行っている [12]．Jiang は本手法を用いて DECKARD というツールを作成し TYPE-1，TYPE-2，TYPE-3 のコードクローンの検出に成功している．

3.4 クラスタリングを用いた検出手法

近年の動向として，最近傍探索によるクラスタリングを用いた検出手法が多くみられる．最近傍探索とは，あるデータセットを距離空間とみなし，その距離空間内である点と距離の近い別の点を検索するためのアルゴリズムである．元々は類似画像検索や DNA の解析などに用いられていたが，近年になりコードクローンの検出でも用いられるようになった．

前述の DECKARD では，抽象構文木の特徴量をベクトルで表現し，そのベクトルを最近傍探索でクラスタリングすることによりコードクローンの検出を行っ

ている．

Uddin らは関数のメトリクスを計測し，その値を最近傍探索でクラスタリングすることでコードクローンの検出を行っている [13]．最近傍探索には Charikar ’s simhash [14] のアルゴリズムを用いている．Uddin らは本手法を用いて SimCad というツールを作成し TYPE-1，TYPE-2，TYPE-3 のコードクローンの検出に成功している．

4. 従来手法の問題点

従来のコードクローン検出ツールは TYPE-3 のコードクローンを検出することができない，あるいは検出はできても計算に膨大な時間がかかるという問題がある．

トークンの並びに着目した Kamiya ら，Baker ら手法は高速にコードクローンを検出することが可能である．Bellon らの報告によると postgresql のソースコード (約 235,000 行) からコードクローンを検出するのに，Kamiya らの手法では 40 秒，Barker らの手法では 12 秒で処理を終えている [4]．一方欠点としては TYPE-3 のコードクローンを検出できないという点があげられる．1 章で述べた通り，コードクローンの検出結果の実用上 TYPE-3 のコードクローンが検出できないのは望ましくない．

それに対して，抽象構文木を用いた Baxter らの手法やプログラム依存グラフを用いた Krinke らの手法は TYPE-3 のコードクローンを検出することができる．一方欠点として，コードクローンの検出に膨大な時間がかかるという点があげられる．Bellon らの報告によると snns のソースコード (約 115,000 行) からコードクローンを検出するのに Baxter らの手法では 3 時間，Krinke らの手法は 63 時間かかっている．またこれらの検出手法はソースコードの規模が大きくなるにつれて検出に膨大な時間がかかる (つまりスケーラビリティに問題がある) ことが知られている．そのため大規模なソフトウェアになると利用が困難である．

本論文ではこれらの問題を踏まえ,「TYPE-3 のコードクローンを高速に検出する (つまりソースコードの規模に対して一定の時間で検出を終える)」ことを研究の目的とする．

5. 提案手法

5.1 概要

図 9 は提案手法の概要を表したブロック図である．ソースコードを入力データとし，コードクローンの検出結果を出力する．検出の対象となる部分は，代入，演算，関数呼び出し等，具体的な処理が記載されている部分 (多くの言語では関数の定義部分) とし，それ以外の部分 (関数の宣言，構造体の宣言，クラスの構造を記述など) は検出対象としない．検出は 5 つのステップから構成される．

STEP1

ソースコード中から抽出可能なコード片をすべて抽出する．詳細は 5.3.1 節で述べる．

STEP2

検出対象として不適切なコード片を検出対象から除去する．詳細は 5.3.3 節で述べる．

STEP3

コード片に含まれるステートメントの特徴値を計算し，計算結果をベクトルで表現する．詳細は 5.3.2 節で述べる．

STEP4

ステートメントの特徴ベクトルに基づきコード片をクラスタリングする．クラスタリングは 2 段階で構成される．1 段階目は高速に粗くクラスタリングを行う．2 段階目は 1 段階目の結果を利用して高い精度のクラスタリグを行う．この 2 段階に分けたクラスタリングにより検出速度と検出精度を向上させる．詳細は 5.3.4 節で述べる．

STEP5

前のステップで得られたクラスタリングの結果に対して頻出パターンマイニングを用いて関連するクローンセット同士を連結し，最終的なコードク

ローンの検出結果を得る．検出結果はコード片の開始と終了の位置情報で出力される．詳細は 5.3.5 節で述べる．

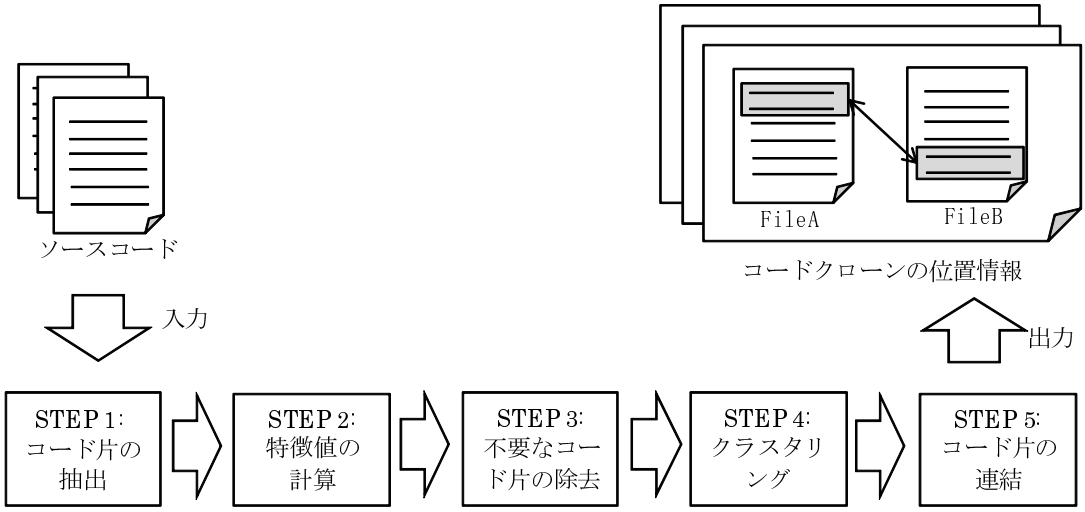


図 9 検出手法の概要

5.2 準備

検出手法の詳細について述べる前の準備として，本提案手法の想定するプログラミング言語と，ステートメントの定義について述べる．

5.2.1 想定するプログラミング言語

本研究で提案する検出手法は，手続き型言語一般 (C，C++，Java，BASIC，FORTRAN，PL/I など) を想定しており，関数型言語や論理プログラミング言語は想定していない．したがって 5 章では手続き型言語一般に対して検出手法の説明を行う．なお説明上ソースコードの具体例を示す場合は Java を用いて説明する．

5.2.2 ステートメントの定義

本手法におけるステートメントの定義について述べる．一般に構造化プログラミングをサポートする言語ではステートメント・制御ステートメント・ブロックを用いて処理を記述する [15]．図 10 は Java におけるその具体例である．

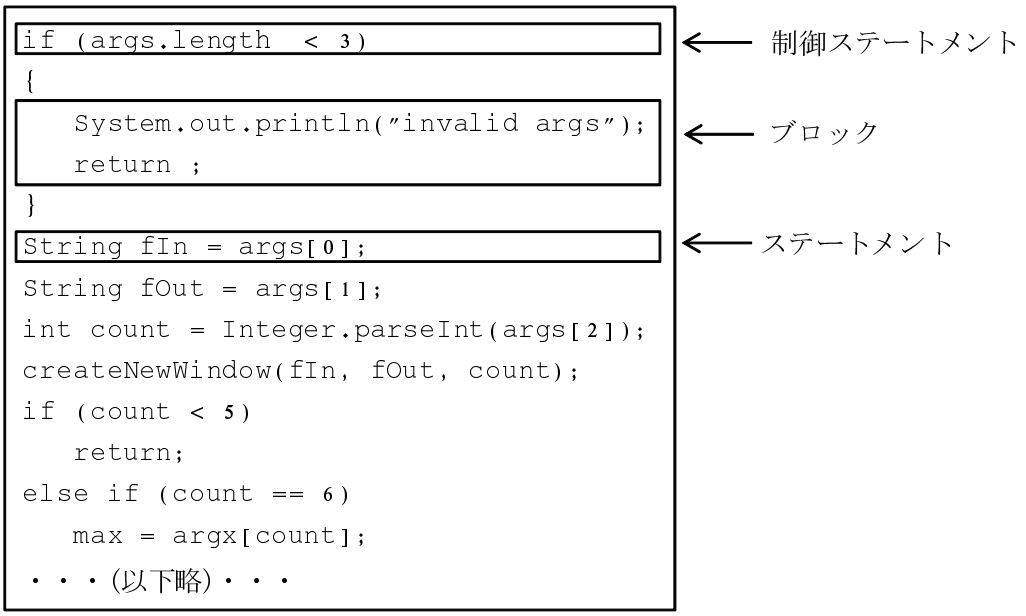


図 10 ステートメント・制御・ブロック

ブロックは複数のステートメントから構成され，ブロック自体を単一のステートメントとして解釈する．また if 文や for 文などの制御ステートメントは処理のフローを制御する特殊なステートメントであると解釈される．したがって以下に示す 3 種類のステートメントが考えられる．

1. ブロックを単一のステートメントと解釈した場合のステートメント
2. 制御ステートメント
3. 上記以外のステートメント

本研究では 2 と 3 の区別はせず，制御文を 3 と同様に一般的なステートメントとして扱う．また 1 のブロック全体に対するステートメントとしての解釈を行わない．したがって図 11 の上部のコード片は下部に示すステートメントのリストとして扱われる．

5.3 検出手順

5.3.1 コード片の抽出

本手法はソースコードから複数のコード片を抽出し，その類似度を評価することでコードクローンを検出する．そのため，まずソースコードからコード片を抽出する．

コード片の抽出はコード片に含まれるステートメントの数に基いて行われる．この値をウィンドウサイズと呼び自然数 w で与える．w はコードクローンの検出時に初期パラメータとして与えられる定数値である．図 12 は w が 5 に固定された場合のコード片の抽出を示している．w の値に基づき，ステートメントリストの先頭から w 個分のステートメント (図中の破線で囲んだ部分) をコード片として抽出する．

次にウィンドウを下方向にスライドすることによりステートメントのリストから複数のコード片を抽出する．図 13 はウィンドウのスライドを示した図である．下図は上図のウィンドウを下方向にスライドした後の状態を表している．スライドの間隔はステートメントの数に基づいて行われ，その間隔をスライドサイズと呼び自然数 s で与える．s は w と同様に初期パラーメータとして与えれる定数値である．図 13 は s を 2 に固定した場合のスライドを表している．

パラメータ w と s によりステートメントのリストから抽出可能なコード片をすべて抽出する (図 14)．

5.3.2 特徴値の計算

次にコード片の類似度を評価するために，ステートメントの特徴値を計算する．ステートメントの特徴として，以下の項目について評価する．

```
if (args.length  < 3) {
   System.out.println("invalid args");
   return ;
}
String fIn = args[0];
String fOut = args[1];
int count = Integer.parseInt(args[2]);
createNewWindow(fIn, fOut, count);
if (count < 5)
   return;
else if (count == 6)
   max = argx[count];
・・・(以下略)・・・
```

```
 1. if (args.length  < 3)
 2. System.out.println("invalid args")
 3. return
 4. String fIn = args[0]
 5. String fOut = args[1]
 6. int count = Integer.parseInt(args[2])
 7. createNewWindow(fIn, fOut, count)
 8. if (count < 5)
 9. return
10. else if (count == 6)
11. max = argx[count]
・・・(以下略)・・・
```

図 11 ソースコードからステートメントのリストへの変換

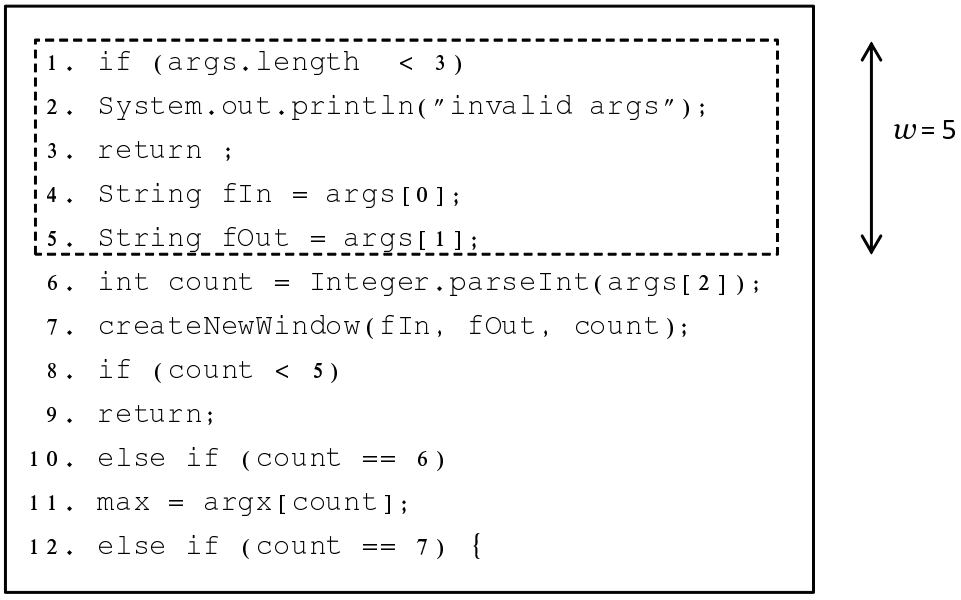


図 12 パラメータ w に基づくコード片の抽出

1. 条件分岐に関係する文か否か
   例 1) `if (p != null)`
   例 2) `else if (p != null)`

2. ループに関係する文か否か
   例.1) `for(int i = 0; i < 16; i++)`
   例.2) `while(true)`

3. ジャンプ命令（ループからの脱出）に関係する文か否か
   例) `break;`

4. ジャンプ命令（ループへの再入）に関係する文か否か
   例) `continue;`

5. 三項演算子を含むか否か
   例) `path = isNull() ? "" : getPath();`

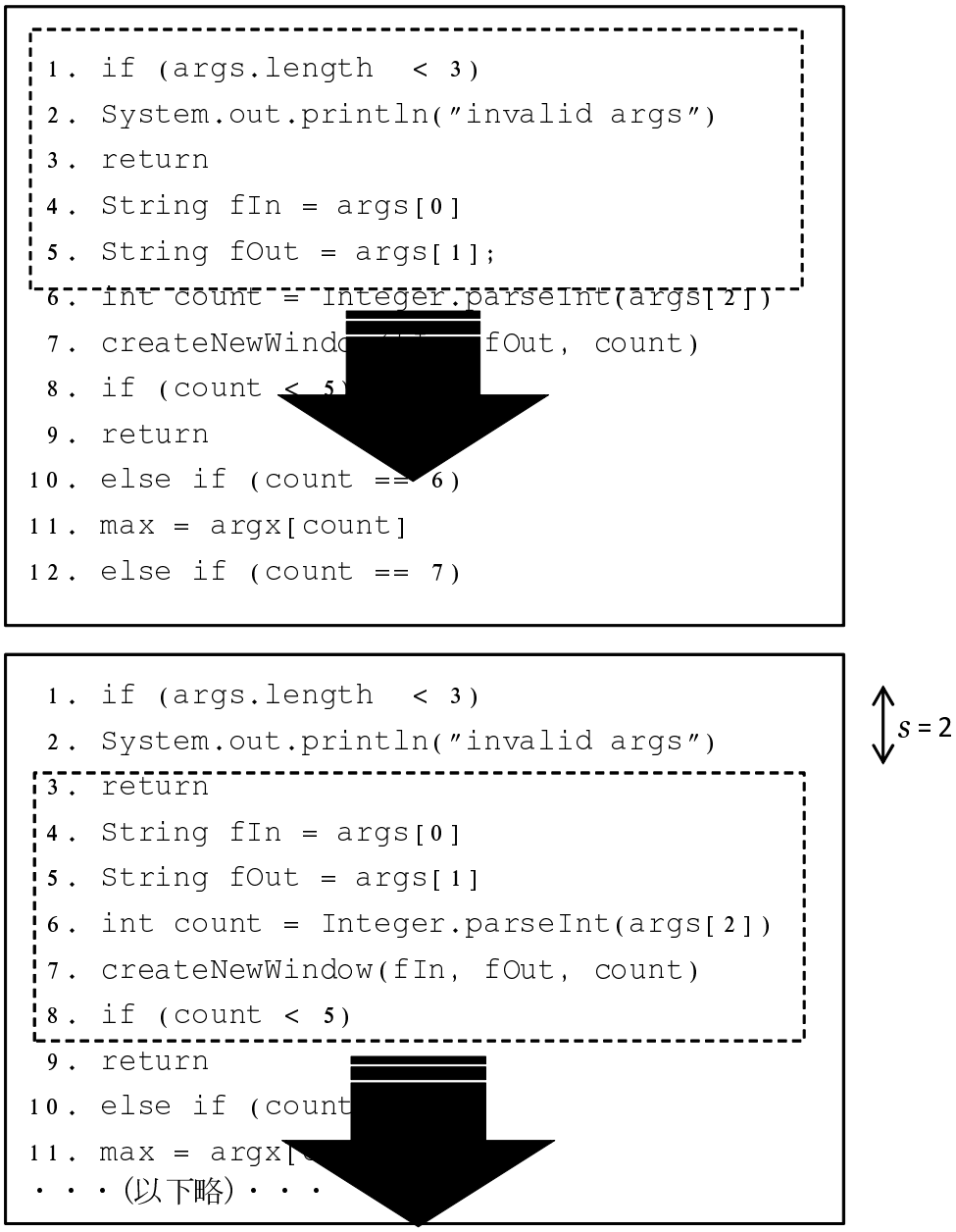


図 13 パラーメータ s に基づくコード片の抽出

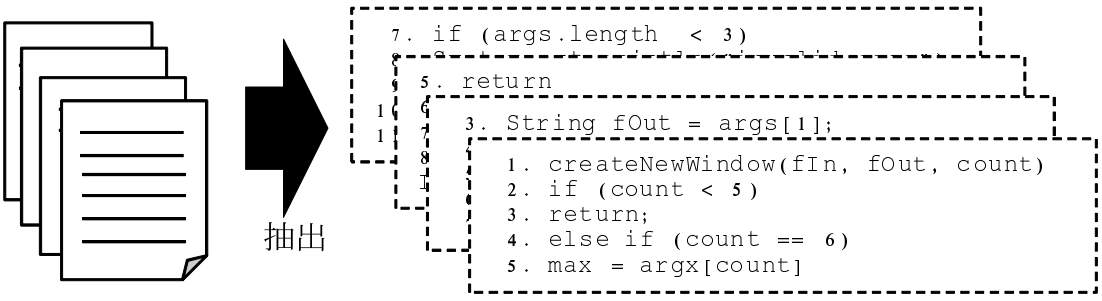


図 14 ソースコードからのコード片の抽出

6. 例外の検出に関係する文か否か
   例) `try`

7. 例外の処理に関係する文か否か
   例) `catch(Exeption e)`

8. 整数値のリテラルを含むか
   例) `i = 1;`

9. 小数値のリテラルを含むか否か
   例) `f = 1.0;`

10. 文字列のリテラルを含むか否か
    例) `str = "text";`

11. 文字のリテラルを含むか否か
    例) `c = 'a';`

12. 変数の代入文か否か
    例) `b = 2;`

これらの項目はソースコードの処理の流れと，局所的特徴を取得するために算出される．例えば条件分岐やループは処理の流れに関係し，三項演算子や各種リ

テラルはソースコード中の一部分でしか用いられないため，上記に示した各項目を評価することによりステートメントの局所的特徴が得られる．

上記の各項目を評価することにより，ステートメントのリストは図 15 に示すように，条件を満たす場合を 1，満たさない場合を 0 とした 0/1 の 2 値ベクトルに変換される．

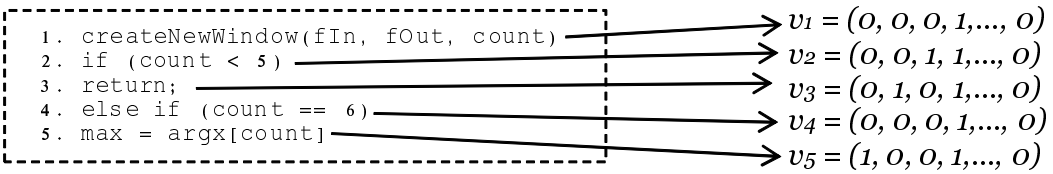


図 15 ステートメントの特徴ベクトルへの変換

5.3.3 繰り返し要素の除去

ソースコードの記述内容によっては，短いコード片が繰り返し出現する場合がある．例えば変数の宣言や図 5 の switch-case 文などが挙げられる．コードクローンの検出の目的にもよるが，一つのコード片の内部に類似したコード片が繰り返し出現するコードクローンは，検出結果の実用上一般的には有益でないといわれている．この問題に対し Higo らは繰り返し要素を多く含むコード片を検出結果から取り除くために RNR(Ratio of Non-Repeated token) というメトリクスを提唱している [16] [17]．RNR は 0 から 1 の値をとり，0 に近いほどコード片内に繰り返し要素を多く含むという特徴を持つ．本手法では Higo らが提唱した RNR をベースとした RNR-S(Ratio of Non-Repeated Statements) というメトリクスを提案する．図 16 に RNR-S の算出アルゴリズムを示す．

図 16 のアルゴリズムはコード片に含まれるステートメントの特徴ベクトルのリストを入力データとし，コード片の繰り返しの度合いを数値化して返す．例えばあるコード片の特徴ベクトルの並びが

$$\boldsymbol{abcabcabcabcabc}(\text{ベクトル}\ \boldsymbol{abc}\ \text{の 5 回繰り返し})$$

```
GetRNRS(v)
 1  length ← v の要素数
 2  index ← 2
 3  count ← 0
 4  while index ≤ length
 5  do
 6      i ← 0
 7      while i < index/2
 8      do
 9          if v_index から v_index−i のベクトルの並びが
10             v_index−i−1 から v_index−2i−1 のベクトルの並びと一致している
11             then count ← count + 1
12          i ← i + 1
13      index ← index + 1
14  return 1 − count/length
```

図 16 RNR-S の算出アルゴリズム

とすると，図 16 の 6 行目の変数 i に対し，9 行目の条件式は
i=1 のとき ‘***a***’ は左端から見て連続した繰り返しを持たないため false.
i=2 のとき ‘***ab***’ は左端から見て連続した繰り返しを持たないため false.
i=3 のとき ‘***abc***’ は左端から見て連続した繰り返しを持たないため false.
i=4 のとき ‘***abca***’ は左端から見て連続した繰り返しを持たないため false.
i=5 のとき ‘***abcab***’ は左端から見て連続した繰り返しを持たないため false.
i=6 のとき ‘***abcabc***’ は左端から見て連続した繰り返しを持つため true(‘***abc***’).
i=7 のとき ‘***abcabca***’ は左端から見て連続した繰り返しを持つため true(‘***bca***’).
(以下略)
となるため RNR-S の値は．5/15(　0.33) となる．

RNR-S を用いて 5.3.1 節で抽出したコード片からコードクローンの検出対象として不適切なものを除去する．

5.3.4 クラスタリング

類似したコード片を検出するために，5.3.1 節，5.3.3 節で得られたコード片のクラスタリングを行う．クラスタリングには最近傍探査アルゴリズムの一種である LSH(Locality Sensitive Hashing) [18] を用いる．LSH および LSH のベースとなる (p_1, p_2, r, c)-Sensitive Hashing と (r, c)-Approximate Neighbor の定義を定義 1，定義 2，定義 3 に示す．

定義 1 ($(\boldsymbol{p_1}, \boldsymbol{p_2}, \boldsymbol{r}, \boldsymbol{c})$-Sensitive Hashing)
p_1，p_2 を 0 以上 1 以下の実数，r を 0 以上の実数，c を 1 以上の実数とし，$\mathfrak{R}^d$ 空間に対してベクトル点集合 $\boldsymbol{V}$ が与えられたとき，$\boldsymbol{V}$ 上の任意の点を $\boldsymbol{v_i}$，$\boldsymbol{v_j}$ とし，$\mathcal{D}$ を $\mathfrak{R}^d$ 上の距離関数とするとき

$$\begin{cases} \mathcal{D}(\boldsymbol{v_i}, \boldsymbol{v_j}) < r & \text{のとき } h(\boldsymbol{v_i}) = h(\boldsymbol{v_j}) \text{ となる確率が } p_1 \text{以上} \\ \mathcal{D}(\boldsymbol{v_i}, \boldsymbol{v_j}) > cr & \text{のとき } h(\boldsymbol{v_i}) = h(\boldsymbol{v_j}) \text{ となる確率が } p_2 \text{以下} \end{cases}$$

を満たすハッシュ関数 h を (p_1, p_2, r, c)-*Sensitive Hashing* という．

定義 1 は距離が近いベクトル同士は同じハッシュ値を取る確率が高く，距離の遠いベクトル同士は異なるハッシュ値をとる確率が高いということを示している．図 17 は上記の定義の概略図である．ベクトル $\boldsymbol{v_1}$，$\boldsymbol{v_2}$，$\boldsymbol{v_3}$ に対し $h(\boldsymbol{v_1})$ と $h(\boldsymbol{v_2})$ は p_1 以上の確率で同じハッシュ値を持つが，$h(\boldsymbol{v_1})$ と $h(\boldsymbol{v_3})$ は p_2 以下の確率で同じハッシュ値を持つ．

なお LSH はハッシュ関数の具体的な実装を定めていないが，一般的には Datar の関数 [19] が用いられる．Datar の関数はベクトル $\boldsymbol{p}$ を入力データ，$\boldsymbol{a}$ を $||\boldsymbol{a}|| = 1$ かつ $\dim(\boldsymbol{p}) = \dim(\boldsymbol{a})$ を満たすベクトルの乱数，b を $0 \leq b \leq w$ を満たす乱数とした場合，下記の関数で与えられる．

$$h(\boldsymbol{p}) = \left\lfloor \frac{\boldsymbol{a} \cdot \boldsymbol{p} + b}{w} \right\rfloor$$

定義 2 ((r, c)-Approximate Neighbor)
$\mathfrak{R}^d$ 空間に対してベクトル点集合 $\boldsymbol{V}$ が与えられたとき，$\mathcal{D}$ を $\mathfrak{R}^d$ 上の距離関数とし，$\boldsymbol{V}$ 上の任意の点（以下「クエリ」と呼ぶ）を $\boldsymbol{v}$ としたとき

$$\boldsymbol{U} = \{\boldsymbol{u} \in \boldsymbol{V} | \mathcal{D}(\boldsymbol{v}, \boldsymbol{u}) \leq cr\}$$

で定義される $\boldsymbol{V}$ の部分集合 $\boldsymbol{U}$ をクエリ $\boldsymbol{v}$ に対する (r, c)-*Approximate Neighbor* という．ただし c と r の定義は定義 1 と共通である．

LSH とは上記の (p_1, p_2, r, c)-Sensitive Hashing と (r, c)-Approximate Neighbor を用いたクラスタリング手法の枠組みを定義したものである．

定義 3 (LSH)
$\mathfrak{R}^d$ 空間に対してベクトル点集合 $\boldsymbol{V}$ が与えられ，(p_1, p_2, r, c)-*Sensitive Hashing* を用いた関数の族 $h_{l,k}(1 \leq l \leq L, 1 \leq k \leq K)$ からなる関数 $\mathcal{H}_l$ を以下の式で与えた時,

$$\mathcal{H}_l = (h_{l,1}(\boldsymbol{v}), h_{l,2}(\boldsymbol{v}), ...h_{l,K}(\boldsymbol{v})) \in \mathfrak{R}^K$$

$\boldsymbol{V}$ 上の任意の点 $\boldsymbol{v}$ に対して $\mathcal{H}_l$ から L 個の K 次元のベクトルが得られる．*LSH* は L 個の K 次元ベクトルをそれぞれハッシュテーブルの鍵とすることで高速に近傍点を求める枠組みを定義したものである．

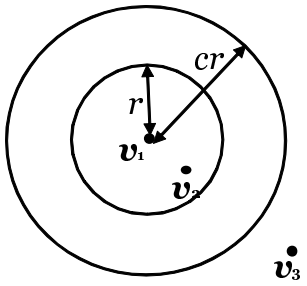


図 17 LSH の概略図

LSH は $d = \dim(\boldsymbol{V})$，$n = \mathrm{card}(\boldsymbol{V})$，$\rho = \log_{p_2} p_1$ とすると，時間計算量は $O(dn^{\rho} \log n)$ となる．これは d と時間計算量が高々線形の関係でしかないことを示している．一方 n に対しては，非線形を含む多項式時間であることを示している．つまり LSH の計算量は，ベクトル空間の次元に対しては頑健であるが，データセットの個数に対しては脆弱であるという特徴を持つ．

本手法では高次元ベクトルのクラスタリングを行うため LSH を用いるが，上記の脆弱性を補うためにクラスタリング 2 段階に分ける．1 段階目のクラスタリングでは複数のステートメントをマージしてコード片単位でクラスタリングを行う．2 段階目のクラスタリングでは 1 段階目の結果に基づきステートメント単位でクラスタリングを行う．

1 段階目のクラスタリング

本手法では前述の LSH を用いて 5.3.1 節，5.3.3 節で得られたコード片のクラスタリングを行う．まずコード片に含まれるの全ステートメントの特徴ベクトルの論理和をとる．コード片に n 個のステートメントが含まれているとし，特徴ベクトルの次元を m とするとコード片の特徴ベクトル c は下記の式で与えられる．

$$\boldsymbol{c} = (\sum_{k=1}^{w} v_{1,k}, \sum_{k=1}^{w} v_{2,k}, \cdots \sum_{k=1}^{w} v_{m,k}) \in \{0,1\}^m$$

上記の式に基づき各コード片を特徴ベクトルに変換し LSH でクラスタリングを行うことによりコードクローンの検出を行う．

2 段階目のクラスタリング

1 段階目のクラスタリングはコード片に含まれるステートメントの特徴ベクトルの論理和をとっているため，各ステートメントの持つ情報が欠損している．2 段階目のクラスタリングでは，1 段階目のクラスタリングで失われた情報欠損を補うためにステートメント単位でクラスタリングを行う．図 18 は 2 段階目のクラスタリングのアルゴリズムである．図 18 の引数 $\boldsymbol{R}$ は 1 段階目のクラスタリン

```
SecondStageClustering(R, w)
 1  H ← ∅
 2  while 1 ≤ card(R)
 3  do
 4      q ← R の任意の要素
 5      R ← R\{q}
 6      i ← 0
 7      while i < card(R)
 8      do
 9          G ← ∅
10          j ← 0
11          V = R の i 番目のコード片に含まれるステートメントの特徴ベクトル集合
12          while j < w
13          do
14              u ← q に含まれる j 番目のステートメントの特徴ベクトル
15              if V が u の近傍ベクトルを含む
16                 then V ← V から u の近傍の任意の特徴ベクトルを一つ取り除く
17          if card(V) ≤ g
18             then
19                  G ← G ∪ {R の i 番目のコード片 }
20      if 1 ≤ card(G)
21         then
22              G ← G ∪ {q}
23              H ← H ∪ {G}
24  return H
```

図 18 2 段階目のクラスタリング

グの結果得られたクラスタ，w はウィンドウサイズで，返り値は $\boldsymbol{R}$ を細分化したクラスタのリストである．$\boldsymbol{R}$ に含まれるコード片の類似度をステートメント単位で比較し，$\boldsymbol{R}$ をさらに細かなクラスタに分割する必要があるかどうかを判別している．

5.3.5 コード片の連結

クラスタリングの結果得られたコードクローンのサイズはステートメント数 w で固定されている．しかしソースコードに存在するコードクローンのサイズは固定長ではない．本手法ではこれらを問題を解決するために頻出パターンマイニングを用いて関係するコードクローンを連結する．

頻出パターンマイニングとは，ある集合族の各集合に頻出する要素の集合を抽出することを指す．形式的には集合 $\boldsymbol{I} = \{i_1, i_2, \cdots i_n\}$ を与えた時，$\boldsymbol{T}_1, \boldsymbol{T}_2, \cdots \boldsymbol{T}_m$ を $\boldsymbol{I}$ の部分集合とすると，$\boldsymbol{T}_1, \boldsymbol{T}_2, \cdots \boldsymbol{T}_m$ 間で頻出する要素の集合を求めることを指す．たとえば $\boldsymbol{I} = \{a, b, c, d, e, f\}$，$\boldsymbol{T}_1 = \{a, b, c\}$，$\boldsymbol{T}_2 = \{a.b.d, e\}$，$\boldsymbol{T}_3 = \{a, b, c, d, e\}$，$\boldsymbol{T}_4 = \{a, d, e, f\}$ という集合があったとする．$\boldsymbol{T}_1, \boldsymbol{T}_2, \boldsymbol{T}_3, \boldsymbol{T}_4$ 間の頻出パターンは {a} は 4 回，{b} は 3 回，{c} は 2 回，{d} は 3 回，{f} は 1 回，{a,b} は 3 回，{a,c} は 2 回，{a,d} は 3 回，{a,e} は 3 回，{a,f} は 1 回，{b,c} は 2 回，$\cdots$ {a,b,c,d,e,f} は 0 回となる

本手法では 2 段階目のクラスタリングの結果得られた要素に属するコード片からなる集合を $\boldsymbol{I}$ とし，メソッドに含まれるクローンの関係にあるコード片の集合を $\boldsymbol{T}$ とした頻出データ間イニングを行い，出現回数が 2 以上の集合を関連するクローンとして連結し，最終的なコードクローンの検出結果として出力する．

6. 実験

6.1 実験概要

5 章で述べたコードクローンの検出手法を実装し実験を行った．以下に実装および実験の詳細を示す．

実装言語

C#，C

使用したライブラリ

E2LSH [20]

5.3.4 節で述べた LSH アルゴリズムの実装に使用した．

LCM [21]

5.3.5 節で述べた頻出データマイニングの実装に使用した．

実行環境

OS

Windows7 Professional 64bit

CPU

Intel Core i7-2600K(1 コアのみ使用)

メモリ

16GB

解析対象言語

Java1.4

解析対象プロジェクト

netbeans-javadoc

eclipse-ant

jdk1.5.0

j2sdk1.4.0-javax-swing

eclipse-jdtcore

lucene

soot

vuze

なお以降は1ステートメント (ステートメントの定義は5.2.2節を参照) を1(LOC) と記述する．

解析対象のプロジェクトは規模の小さいものから順にnetbeans-javadocは7,686 (LOC)，eclipse-antは17,734 (LOC)，jdk1.5.0は33,647 (LOC)，j2sdk1.4.1-javax-swingは66,099 (LOC)，eclipse-jdtcoreは110,274 (LOC)，luceneは229,499 (LOC)，sootは252,857 (LOC)，vuzeは270,638 (LOC) である．

本研究で用いたE2LSHはLSHの各種パラメータを自動的に決定する機能を持つ [20]．この機能はデータセットの中からいくつかのデータをランダムに選択し，その値の傾向を見て適当なパラメータを自動的に設定する．本手法ではE2LSHが自動的に算出した値を用いている．

6.2 実験結果

6.2.1 コードクローンの検出時間と検出量

表1にコードクローンの検出結果を示す．検出にあたり，ウィンドウサイズ w を16，12，8の3種類，スライド幅 s を $w/2$，$w/4$ の2種類，ギャップサイズ g を $w-2$ の値で設定し，RNR-Sメトリクスを0.5でフィルタリング設定を行い検出した (つまり各プロジェクトに対して計6通りのパラメータ設定を行った)．表の「検出時間」は2段階目のクラスタリングの結果を得るまでにかかった時間,「検出量」は2段階目のクラスタリングで得られたコードクローンのステートメントの総数を表している．

図19の各グラフは表1の w と検出量の関係を表したグラフである．プロジェクトの種類に基づき8つのグラフに分類した．横軸は w，縦軸は検出量である．プロットは s の値 ($s = w/2$，$s = w/4$) に基づき2種類に分類した．プロットに対する近似直線もあわせて示す．このグラフから w と検出量は比例の関係にあることが分かる．

図 20 の各グラフは表 1 の w と検出時間の関係を表したグラフである．プロジェクトの種類に基づき 8 つのグラフに分類した．横軸は w，縦軸は検出量である．プロットは s の値 ($s = w/2$，$s = w/4$) に基づき 2 種類に分類した．プロットに対する近似直線もあわせて示す．このグラフから w と検出時間は比例の関係にあることが分かる．

図 21 の各グラフは表 1 の検出時間と検出量の関係を表したグラフである．プロジェクトの種類に基づき 8 つのグラフに分類した．横軸は検出時間，縦軸は検出量である．プロットは s の値 ($s = w/2$，$s = w/4$) に基づき 2 種類に分類した．プロットに対する近似直線もあわせて示す．このグラフから検出時間と検出量は比例の関係にあることが分かる．

6.2.2 プロジェクトの規模とコードクローンの検出時間

図 22 はプロジェクトの規模と検出時間の関係を表したグラフである．プロットは左から順に netbeans-javadoc，eclipse-ant，jdk1.5.0，j2sdk1.4.1-javax-swing，eclipse-jdtcore，lucene，soot，vuze である．またプロットに対する近似直線もあわせて示す．このグラフからプロジェクトの規模と検出時間は比例の関係にあることが分かる．

また図 23 は各プロジェクトの検出時間の内訳 (比率) である．棒グラフは左端から順に，構文解析および RNR-S のフィルタリングにかかった時間，1 段階目のクラスタリングにかかった時間，2 段階目のクラスタリングにかかった時間，コード片の連結にかかった時間を表している．1 段階目のクラスタリングおよび 2 段階目のクラスタリングが処理時間の大半を占めていることが分かる．

6.2.3 コードクローンの品質

コードクローンの研究分野では，検出結果の品質を計るための共通の物差しとなる評価手法が確立されていない．その理由として，品質は検出時間や検出量と異なり定量的な評価が難しく，またどれをクローンとみなすか (クローンか，偶然の一致か) 人の主観に依存するところもあり，万人の合意を得るに至っていな

いという点があげられる．そのため現状では各研究者によって独自の評価手法が用いられている．具体例をあげると，検出の結果得られたコード片を例示してそれでもって品質について述べたとする [12]，あるいは研究者自身が人為的にコードクローンを作成しそのコードクローンを検出できたかどうかで評価を行う (つまり研究者自身が自分でコードクローンの正解集合を作成する) [13] 等があげられる．

一方 Bellon らは文献 [4] で，複数のコードクローン検出ツールを用いて性能の比較実験を行っている．この実験では計 6 種類のコードクローン検出ツールを用いて，それぞれのツールが正確にコードクローンを検出しているかどうかを評価実験している．具体的にはコードクローン検出ツールが出力した全コード片の 2%を Bellon らがランダムに選択し，それがコードクローンかどうかを Bellon らの主観で評価した．なお文献 [4] は Bellon らはコードクローン検出ツールの開発者ではないため評価結果は客観的であると主張している．Bellon らはこの研究で用いたデータを公開しており，本論文ではそのデータをコードクローンの正解集合と見なし評価する．

また Bellon らは文献 [4] でコードクローン検出ツールの性能を評価するための指標として good 値と ok 値を提唱している．以下にその定義を示す．

定義 4 (good 値)
2 つのコード片 (f_1, f_2) の重なりの程度をしめす overlap 値を以下の式で定義する．なお，$lines(f)$ はコード片 f に含まれるステートメントの集合を表す．

$$overlap(f_1, f_2) = \frac{lines(f_1) \cap lines(f_2)}{lines(f_1) \cup lines(f_2)} \tag{1}$$

クローンの関係にあるコード片集合 (本手法の場合はクラスタリングで得られたコード片の集合) の中から任意に選出した 2 つの要素のペアをクローンペアという．good 値はある 2 つのクローンペア p_1，p_2 に対し，上記の overlap 値

を用いて以下の式で定義される

$$good(p_1, p_2) = min(overlap(p_1.f_1, p_2.f_1), \\ overlap(p_1.f_1, p_2.f_1)) \quad (2)$$

定義 5 (ok 値)
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示す contain 値を以下の式で定義する．

$$contain(f_1, f_2) = \frac{lines(f_1) \cap lines(f_2)}{lines(f_1)} \quad (3)$$

good 値はある 2 つのクローンペア p_1，p_2 に対し，上記の overlap 値を用いて以下の式で定義される

$$ok(p_1, p_2) = min(max(contain(p_1.f_1, p_2.f_1), \\ contain(p_2.f_1, p_1.f_1), \\ max(contain(p_1.f_2, p_2.f_2), \\ contain(p_2.f_2, p_1.f_2)) \quad (4)$$

good 値はクローンペア (p_1, p_2) の全領域から見てどれだけオーバーラップしているかを表した指標である．一方 ok 値はクローンペア (p_1, p_2) の p_1 の領域あるいは p_2 の領域から見てどれだけオーバーラップしているかを表す指標である．

Bellon は netbeans-javadoc の正解集合を公開している．表 3 と図 24 は netbeans-javadoc の 2 段階目のクラスタリングの結果に対する ok 値と good 値の数値と箱ひげ図である．また表 4 と図 25 は 2 段階目のクラスタリングの結果を連結したものに対する ok 値と good 値の数値と箱ひげ図である．ok 値または good 値が 0 以上のもののみ掲示した．

6.2.4 他の検出ツールとの比較

本提案手法の検出時間を他のコードクローン検出ツール (Scorpio，DECKARD，CloneDR) と比較した．結果を表 5 に示す．DECKARD と CloneDR の検出時間は文献 [12] から引用している．文献 [12] では CPU が Xeon 2GHz，メモリーが 1GB で実験を行っている．また Scorpio の検出時間は文献 [10] から引用している．ただし，文献 [10] ではどのような構成の計算機で実験を行ったか明記されていない．提案手法と実験を行った計算機の構成が異なるため単純な値の比較はできないが参考として提示した．

6.2.5 メトリクス RNR-S の効果

表 6 にメトリクス RNR-S によってフィルタリングされたコード片の量を示す．「RNR-S のフィルタリング対象のコード片」の列に示されている値は下記の式より算出した値である．

$$\frac{RNR-S \text{によって削減されたコード片の総数}}{\text{抽出されたコード片の総数}}$$

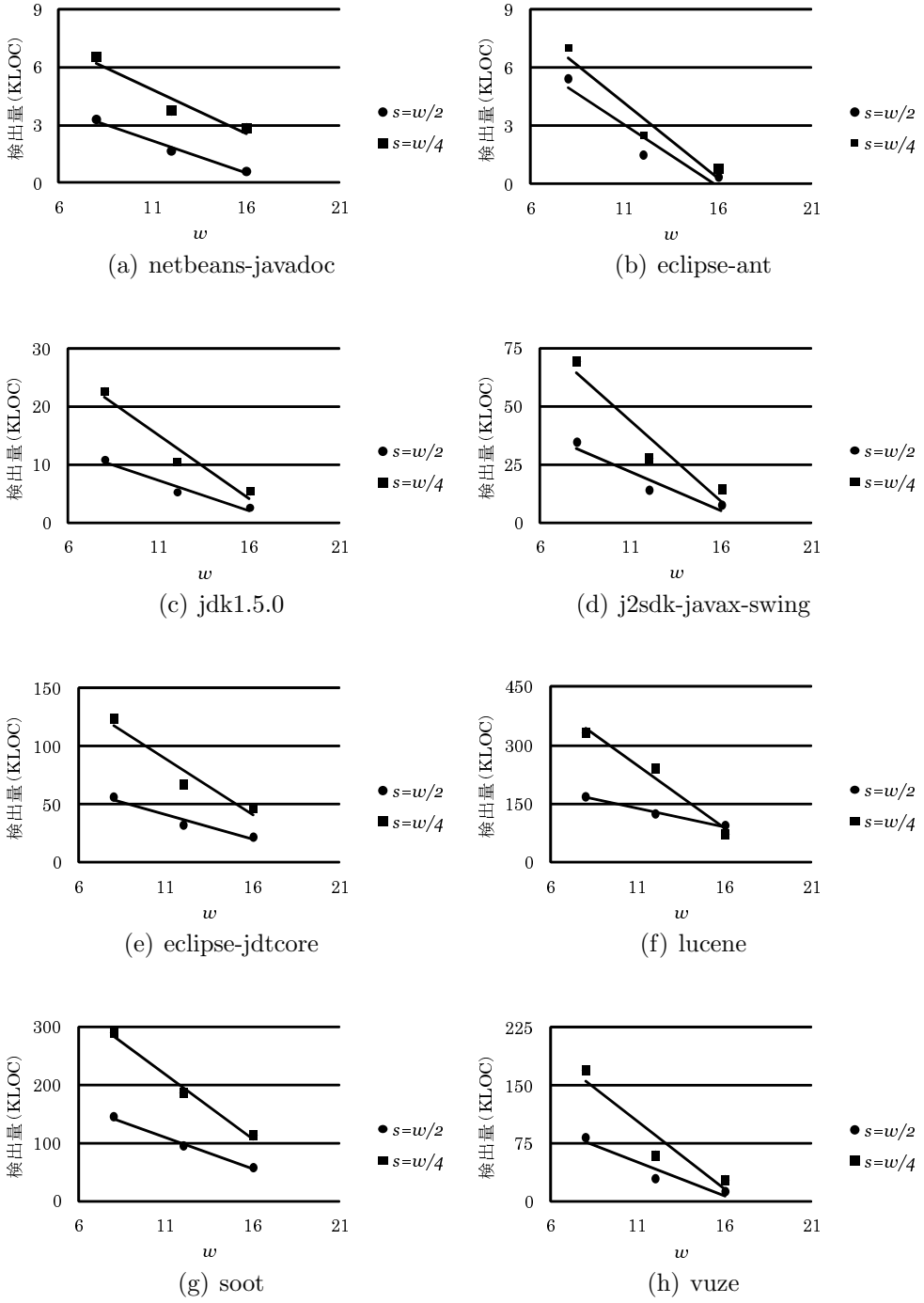
6.2.6 本手法で得られた TYPE-3 コードクローンの例

本検出手法によって得られた TYPE-3 のコードクローンの例を図 26 に示す．

表 1: 検出時間と検出量の関係

プロジェクト	w	s	g	検出時間 (秒)	検出量 (LOC)
	16	8	14	1.83	608
	16	4	14	1.90	2,864
netbeans-javadoc	12	6	10	1.30	1,668
	12	3	10	3.43	3,744
	8	4	6	2.83	3,304
	8	2	6	4.40	6,528
	16	8	14	2.82	352
	16	4	14	3.88	800
eclipse-ant	12	6	10	2.55	1,488
	12	3	10	5.70	2,496
	8	4	6	5.54	5,424
	8	2	6	13.74	7,016
	16	8	14	7.16	2,656
	16	4	14	14.64	5,440
jdk1.50	12	6	10	12.50	5,316
	12	3	10	28.72	10,500
	8	4	6	28.12	10,896
	8	2	6	62.13	22,712
	16	8	14	14.56	7,168
	16	4	14	32.32	13,808
j2sdk-javax-swing	12	6	10	30.29	13,560
	12	3	10	70.55	27,300
	8	4	6	75.84	34,200
	8	2	6	191.40	68,768
	16	8	14	36.52	21,728
	16	4	14	80.06	46,752

eclipse-jdtcore	12	6	10	51.49	33,136
	12	3	10	126.56	66,648
	8	4	6	105.99	56,208
	8	2	6	291.88	123,864
vuze	16	8	14	60.42	13,200
	16	4	14	160.94	28,272
	12	6	10	135.16	29,784
	12	3	10	356.29	59,784
	8	4	6	326.89	82,640
	8	2	6	1081.97	169,208
soot	16	8	14	131.03	56,080
	16	4	14	291.82	111,568
	12	6	10	238.62	93,672
	12	3	10	571.42	184,176
	8	4	6	510.22	143,736
	8	2	6	1110.08	288,800
lucene	16	8	14	64.15	94,672
	16	4	14	199.45	73,040
	12	6	10	214.90	123,972
	12	3	10	605.05	240,120
	8	4	6	499.09	168,016
	8	2	6	1509.64	331,608



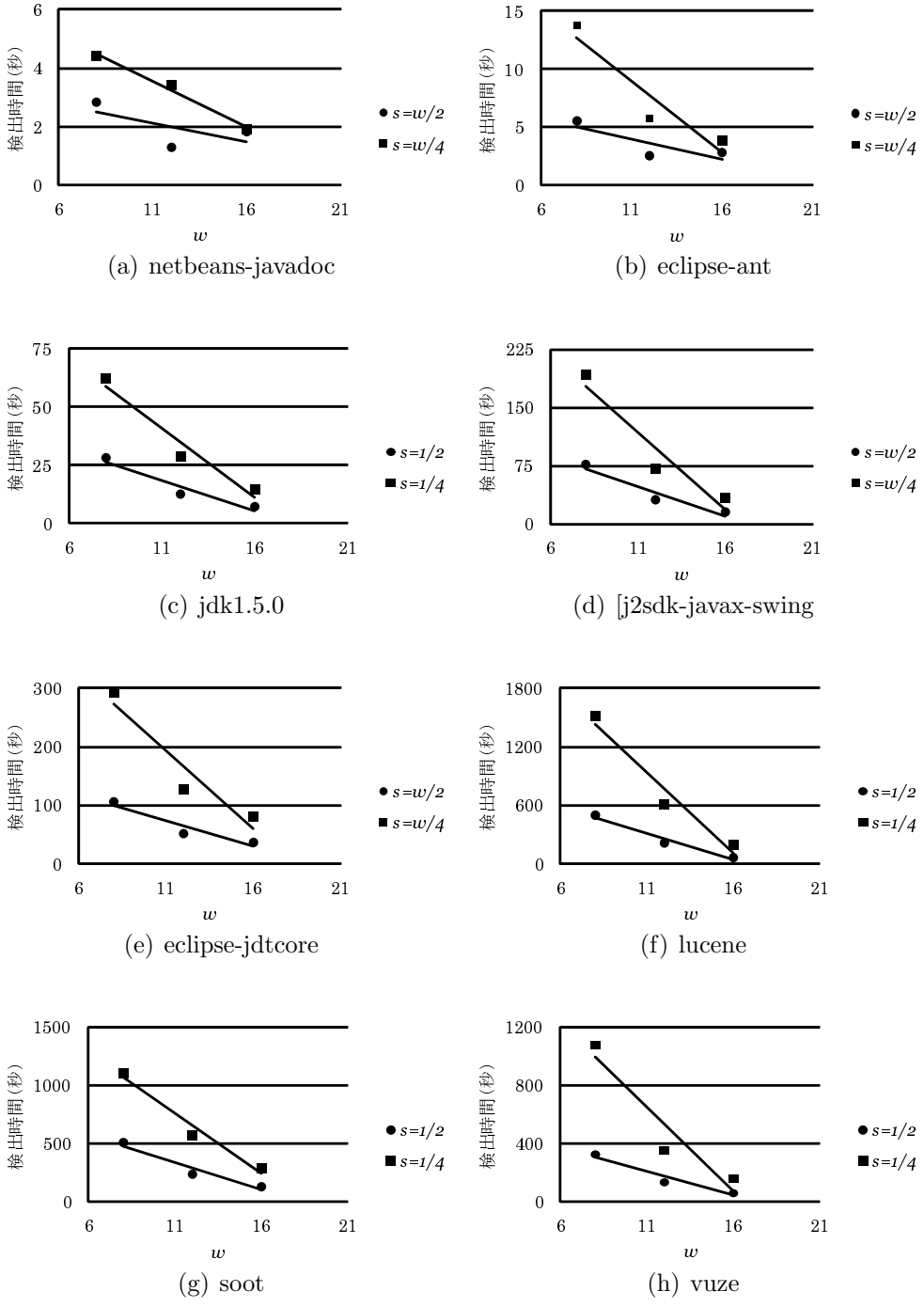
(a) netbeans-javadoc (b) eclipse-ant

(c) jdk1.5.0 (d) j2sdk-javax-swing

(e) eclipse-jdtcore (f) lucene

(g) soot (h) vuze

図 19 w と検出量の関係



(a) netbeans-javadoc (b) eclipse-ant

(c) jdk1.5.0 (d) [j2sdk-javax-swing

(e) eclipse-jdtcore (f) lucene

(g) soot (h) vuze

図 20 w と検出時間の関係

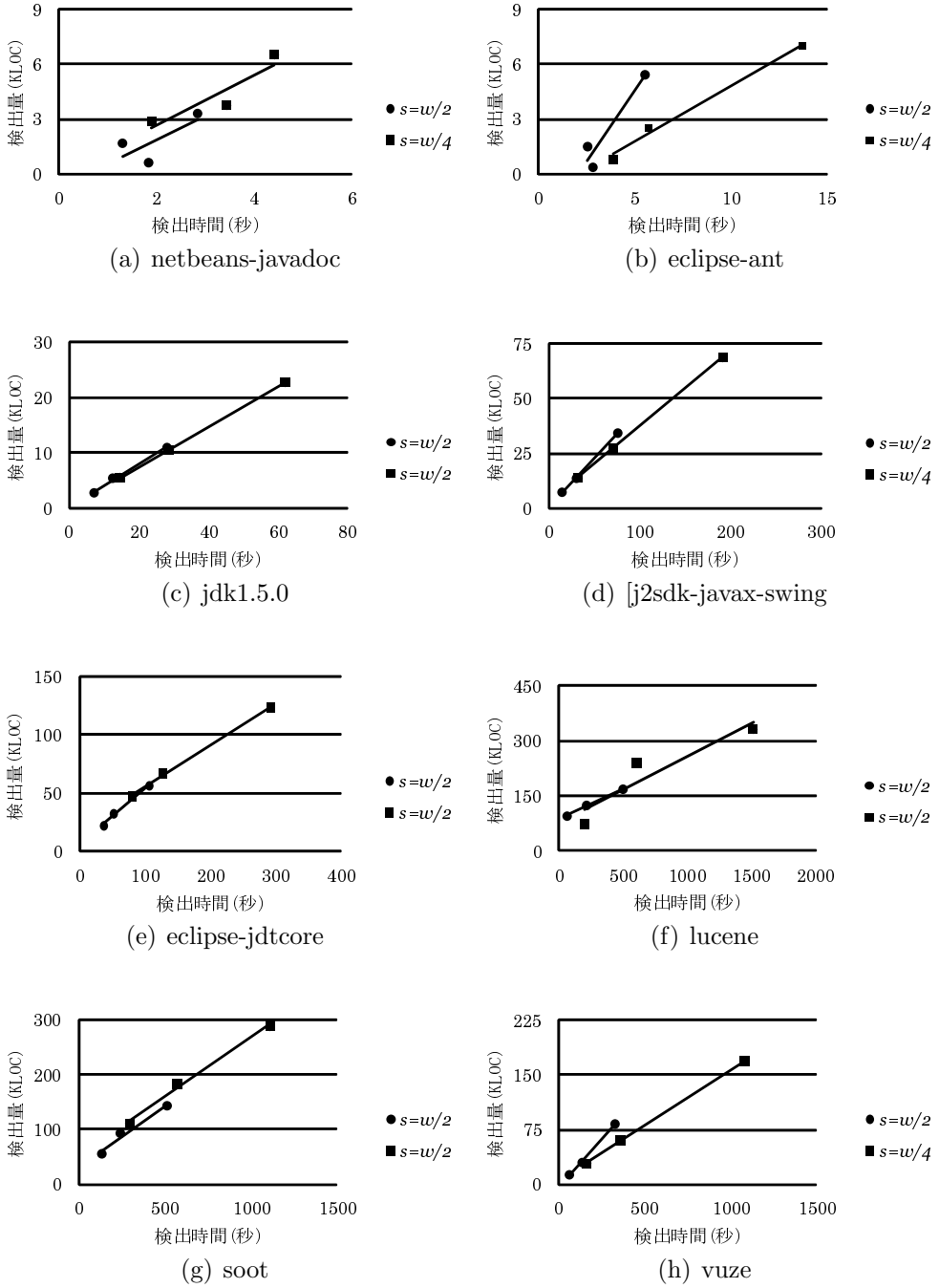


図 21 検出時間と検出量の関係

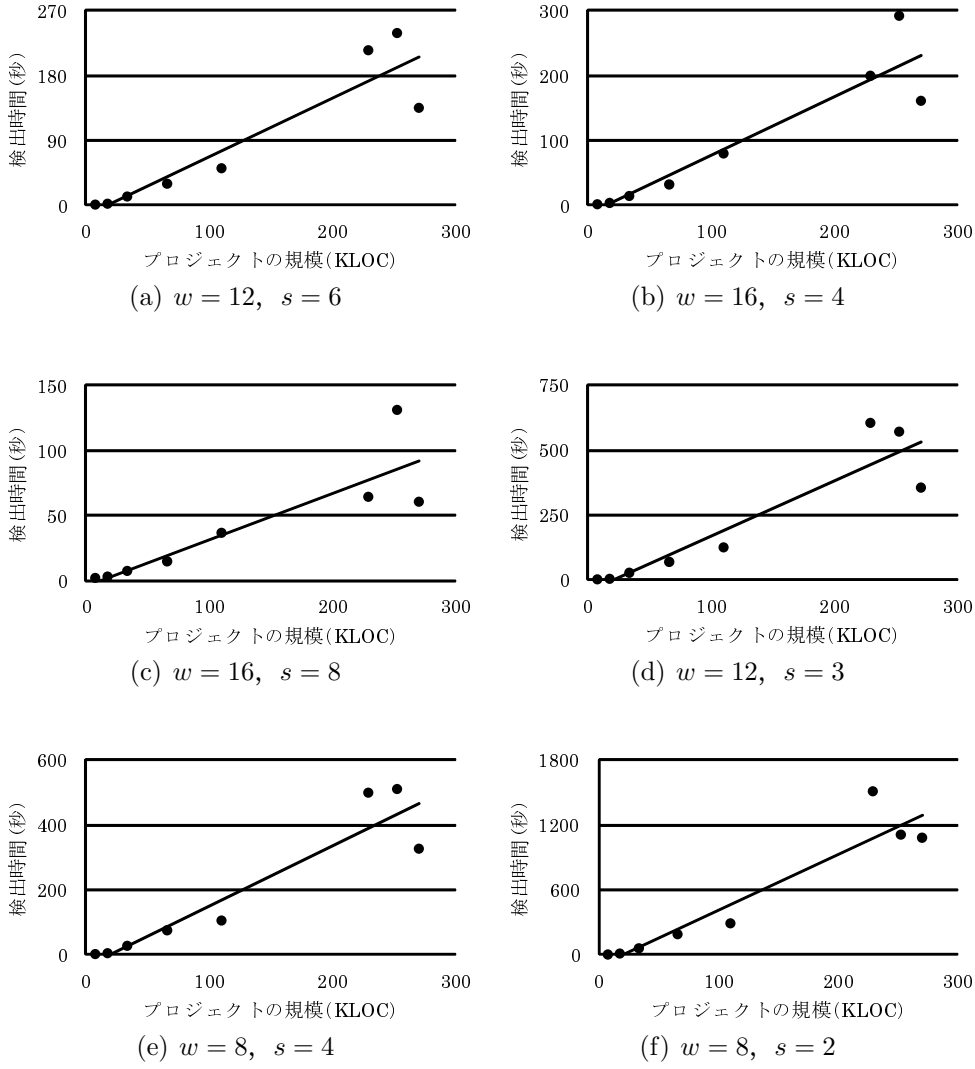
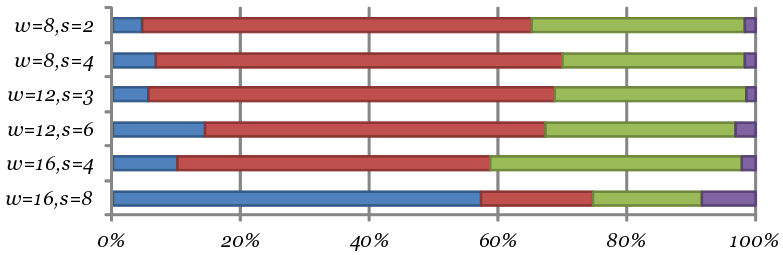
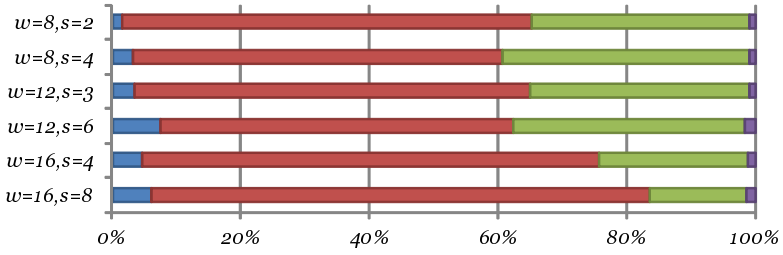


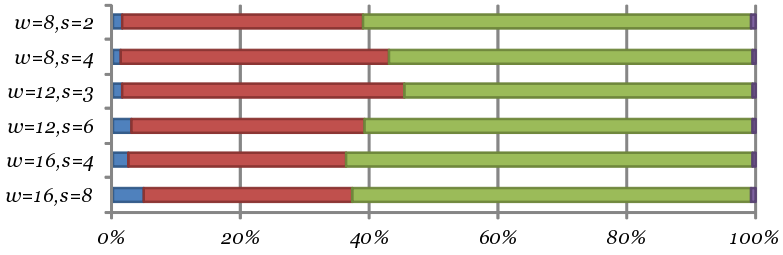
図 22　解析対象のサイズと検出時間の関係 (図中のプロットは左端から順に javadoc，ant，jdk1.5.0，swing，jdtcore，lucene，soot，vuze)



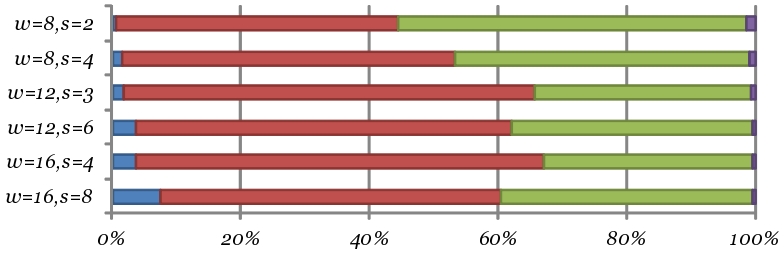
(a) netbeans-javadoc



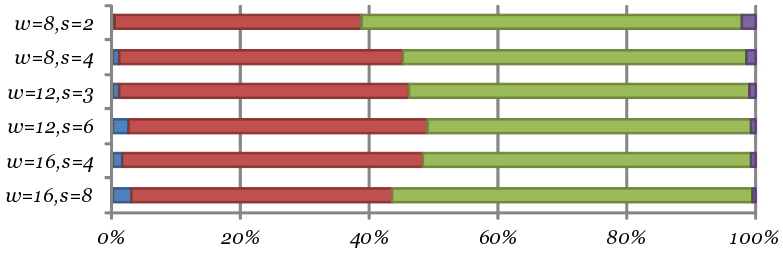
(b) eclipse-ant



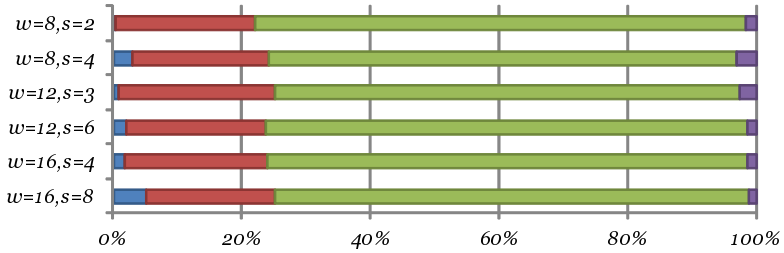
(c) jdk1.5.0



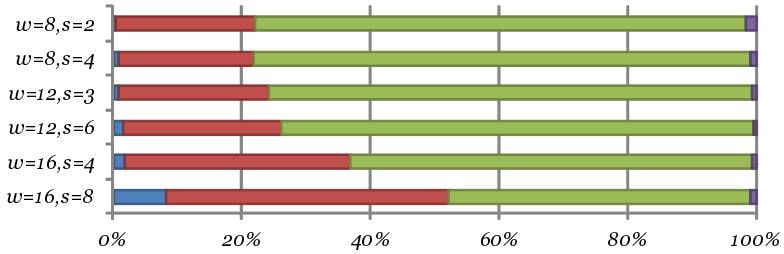
(e) j2sdk-javax-swing



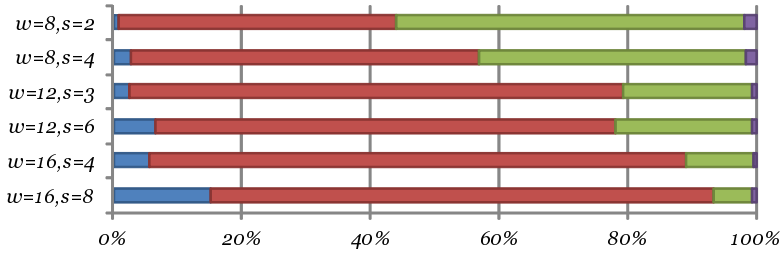
(e) eclipse-jdtcore



(f) lucene



(g) soot



(g) vuze

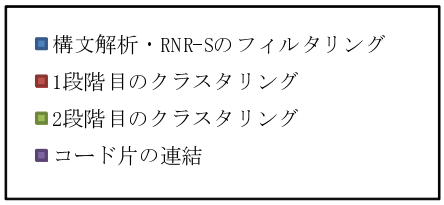


図 23 検出時間の内訳

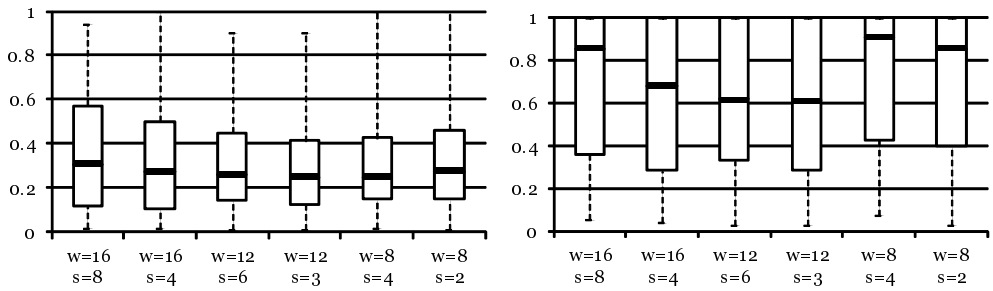
表 3 コード片の連結前の good 値と ok 値

(a) good 値

w	16	16	12	12	8	8
s	8	4	6	3	4	2
最大値	0.94	0.94	1.00	0.90	1.00	1.00
75%	0.57	0.50	0.45	0.41	0.43	0.46
中央値	0.31	0.27	0.26	0.25	0.25	0.28
25%	0.12	0.11	0.14	0.12	0.15	0.15
最小値	0.02	0.02	0.02	0.02	0.02	0.02

(b) ok 値

w	16	16	12	12	8	8
s	8	4	6	3	4	2
最大値	0.94	0.94	1.00	0.90	1.00	1.00
75 %	0.57	0.50	0.45	0.41	0.43	0.46
中央値	0.31	0.27	0.26	0.25	0.25	0.28
25%	0.12	0.11	0.14	0.12	0.15	0.15
最小値	0.02	0.02	0.02	0.02	0.02	0.02



(a) good 値　　(b) ok 値

図 24 コード片の連結前の good 値と ok 値

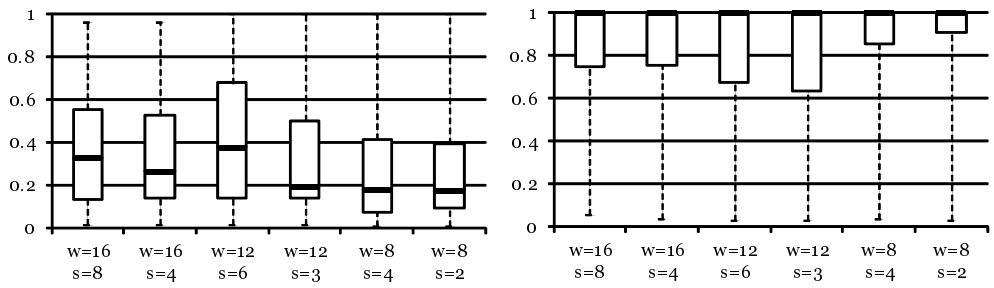
表 4　コード片の連結後の good 値と ok 値

(a) good 値

w	16	16	12	12	8	8
s	8	4	6	3	4	2
最大値	0.96	0.96	1.00	1.00	1.00	1.00
75%	0.55	0.53	0.68	0.50	0.41	0.39
中央値	0.33	0.26	0.38	0.19	0.17	0.18
25%	0.13	0.14	0.14	0.14	0.07	0.09
最小値	0.01	0.02	0.01	0.01	0.01	0.01

(b) ok 値

w	16	16	12	12	8	8
s	8	4	6	3	4	2
最大値	1.00	1.00	1.00	1.00	1.00	1.00
75%	1.00	1.00	1.00	1.00	1.00	1.00
中央値	1.00	1.00	1.00	1.00	1.00	1.00
25%	0.75	0.76	0.68	0.64	0.86	0.90
最小値	0.06	0.04	0.03	0.03	0.04	0.03



(a) good 値　　(b) ok 値

図 25　コード片の連結後の good 値と ok 値

表 5 検出にかかった時間

	javadoc	ant	jdk1.5.0	swing	jdtcore	lucene	soot	vuze
提案手法 ($w = 16, s = 8$)	2	3	7	15	37	64	131	60
提案手法 ($w = 16, s = 4$)	2	4	15	32	80	199	292	161
提案手法 ($w = 12, s = 6$)	1	3	12	30	51	215	239	135
提案手法 ($w = 12, s = 3$)	3	6	29	71	127	605	571	356
提案手法 ($w = 8, s = 4$)	3	6	28	77	106	499	510	327
提案手法 ($w = 8, s = 2$)	4	14	62	194	292	1510	1110	1082
他の手法 (DECKARD)	-	-	9000	-	-		-	-
他の手法 (CloneDR)			7200					
他の手法 (SCORPIO)	34	2283	39	142	-		-	-

表 6: RNR-S によってフィルタリングされたコード片の
比率

	w	s	RNR-S のフィルタリング対象のコード片
eclipse-jdtcore	16	8	7%
	16	4	7%
	12	6	6%
	12	3	7%
	8	4	4%
	8	2	5%
eclipse-ant	16	8	3%
	16	4	2%
	12	6	1%
	12	3	1%
	8	4	2%
	8	2	2%
netbeans-javadoc	16	8	6%
	16	4	15%
	12	6	12%
	12	3	13%
	8	4	12%
	8	2	12%
j2sdk1.4.0-javax-swing	16	8	5%
	16	4	5%
	12	6	4%
	12	3	4%
	8	4	4%
	8	2	4%
	16	8	5%

vuze	16	4	5%
	12	6	4%
	12	3	4%
	8	4	4%
	8	2	4%
jdk1.5.0	16	8	12%
	16	4	12%
	12	6	9%
	12	3	10%
	8	4	8%
	8	2	8%
soot	16	8	18%
	16	4	19%
	12	6	12%
	12	3	12%
	8	4	11%
	8	2	11%
lucene	16	8	11%
	16	4	11%
	12	6	9%
	12	3	9%
	8	4	8%
	8	2	8%

```
 1  List signers =
JarVerifier.getSignersOfJarEntry(jceCipherURL);
 2  for (Iterator t = signers.iterator(); t.hasNext();) {
 3    X509Certificate[] chain = (X509Certificate[])t.next();
 4    if (chain[0].equals(jceCertificate)) {
 5      signers = null ;
 6      break ;
 7  }
 8  if (signers != null) {
 9    throw new SecurityException("Jurisdiction policy files are
" + "not signed by trusted signers!");
10  }
```

```
 1  List paths = JarVerifier.convertCertsToChains(certChains);
 2  boolean found = false;
 3  for (Iterator t = paths.iterator(); t.hasNext();) {
 4    X509Certificate [] path = (X509Certificate[])t.next();
 5    X509Certificate cert = path[0];
 6    if (cert.equals(signer)) {
 7      found = true;
 8      break;
 9    }
10  }
11  if (found == false) {
12    throw new SecurityException("Jurisdiction policy files are
" + "not signed by trusted signers!");
13  }
```

図 26 提案手法によって得られた TYPE-3 のコードクローンの例

7. 考察

7.1 検出手法のスケーラビリティ

図 22 からプロジェクトの規模と検出時間は比例に近い関係にあることが分かる．本来ならば LSH アルゴリズムの時間計算量は 5.3.4 節で述べた通り $O(dn^{\rho}\log n)$ であるためプロジェクトの規模と計算時間は比例の関係にはならない．しかしながら今回は 2 段階のクラスタリングを用いたことにより比例に近い関係を示したと考えられる．

なお今回の 2 段階のクラスタリングは分割統治法の一種と考えられる．そのためウィンドウサイズの複数回変更させることで 2 段階のみならず多段階クラスタリングを行うことができる．また 2 段階目のクラスタリングは 1 段階目のクラスタリングで形成されたクラスタを母集合とする．そのため 2 段階目のクラスタリングでは各データセットがそれぞれ独立しており，分散処理技術の適用が可能である．以上のことからさらに処理速度を改善することができる可能性がある．

7.2 パラメータと検出結果の関係

図 19，図 20，図 21 からパラメータ w，検出時間，検出量はそれぞれ比例に近い関係にあることが分かる．またパラーメータ s は比例関係の係数 (図中の近似直線の傾き) に影響を及ぼすことが分かる．この事実から検出対象のプロジェクトの規模と各種パラメータより，コードクローンの検出を完了するまでの時間を予想することができる．

7.3 コード片の連結がコードクローンの品質に及ぼす影響

図 24 と図 25 を比較すると，コード片を連結したことにより ok 値の値が向上したことが確認できる．向上した理由として式 (3) の分母が不変のまま分子の値が大きくなったため ok 値が上昇したと思われる．また good 値は式 (1) の分母と分子がともに大きくなったためほとんど変化が見られなかったと思われる．

ok 値が改善され，good 値はほとんど悪化が見られないことから頻出データマイニングによるコード片の連結は 2 段階目のクラスタリングによって得られたコードクローンの品質を高めるうえで有効な手段であるといえる．

7.4 メトリクス RNR-S の効果

メトリクス RNR-S の算出および RNR-S に基づくコード片の除去に必要とされる処理時間は，表 6 から他の処理フェーズと比較して短時間であることが分かる．

また RNR-S によって除去されたコード片として図 27 に示すものがあった．図 27 から RNR-S のアルゴリズムは文献 [16] 文献 [17] の RNR と類似した性質を持つメトリクスであることが確認された．

```
    argument = arguments [ 0 ] ;
    break ;
case ExceptionTypeProblemBase + NotFound :
    filter = CLASSES ;
    argument = arguments [ 1 ] ;
    break ;
case ReturnTypeProblemBase + NotFound :
    filter = CLASSES — INTERFACES ;
    argument = arguments [ 1 ] ;
    break ;
case ImportProblemBase + NotFound :
    filter = IMPORT ;
    argument = arguments [ 0 ] ;
    break ;
case UndefinedType :
    filter = CLASSES — INTERFACES ;
```

図 27 取り除かれたコード片の例

7.5 既存の手法との比較

7.5.1 検出速度

表 6 から Scorpio は ant の処理に極端に時間がかかっている．一方本手法はプロジェクトの種類によらず，プロジェクトの規模に比例して検出に時間がかかっていることが分かる．したがって本手法は検出時間の安定性において Scorpio よりも優位性があると言える．

7.5.2 検出されたコード片

図 26 は本手法によって得られた TYPE-3 のコードクローンである．図 26 のコードクローンはトークンの並びが完全に異なり，プログラム依存グラフの構造や抽象構文木の構造も異なる．そのため 3 章で述べた既存の手法では図 26 に示したコードクローンを検出することは困難である．一方，図 28 は Scorpio では検出できたが，提案手法では検出することができなかったコードクローンの例である．本手法は一定サイズのコード片単位でクラスタリングを行うため，図 28 で示したような，クローンの関係にあるステートメントが飛び飛びに出現するコードクローンをうまく抽出できなかったのではないかと思われる．

```
1  DataFolder df;
2  JavaDataObject jdo;
3  TopManager.getDefault().setStatusText( ResourceUtils.getBundledString( "MSG_GeneratingList" ) );
4  for( int i = 0; i < activatedNodes.length; ++i )
5    if ((df = (DataFolder)activatedNodes[i].getCookie( DataFolder.class )) != null ) {
6      addPackage( df );
7    } else if ((jdo = (JavaDataObject)activatedNodes[i].getCookie( JavaDataObject.class )) != null ) {
```

```
1  DataFolder df;
2  JavaDataObject jdo;
3  //load javadoc setting (need for recursive parameter)
4  ExternalJavadocSettingsService javadocS =
(ExternalJavadocSettingsService)TopManager.getDefault().getServices().find(ExternalJavadocSettingsService.class);
5  for( int i = 0; i < activatedNodes.length; ++i ) {
6    if ((df = (DataFolder)activatedNodes[i].getCookie( DataFolder.class )) != null ) {
7      if( !isAvailableFile(df.getPrimaryFile()) )
8        continue;
9        String pck = df.getPrimaryFile().getPackageName('.');
10     if( existsJdoFilesInFolder(df) && !pckList.contains(pck)){
11       pckList.add(pck);
12       list.add(pck);
13     }
14     if( javadocS == null || javadocS.getRecursive () )
15       list.addAll(parseFolders(df));
16     } else if ((jdo = (JavaDataObject)activatedNodes[i].getCookie(JavaDataObject.class)) != null ) {
```

図 28 Scorpio によって検出された TYPE-3 クローンの例

8. おわりに

本論文では最近傍探査アルゴリズムによるクラスタリングを用いたコードクローンの検出手法を提案した．この手法ではクラスタリングを 2 段階に分けることにより高速に (検出対象のソースコードの規模に対して比例した時間で) 処理を行うことが可能であることを示した．また従来のトークンの並びやプログラム依存グラフ，抽象構文木などを用いた検出手法とは異なる新たな検出手法を用いたことにより従来手法では検出が困難なタイプのコードクローンの検出に成功した．一方従来手法では検出できたが提案手法では検出できないタイプのコードクローンも存在するため，提案手法と既存の手法の組み合わせによる検出が望ましい．

本研究の今後の展望としては検出対象の特徴 (ソースコードの規模，コーディング規約，検出対象の言語) に応じたパラメータの自動選出やクラスタリングを 2 段階だけではなく多段階で行うことによる高速化などがあげられる．また本手法の 2 段階目のクラスタリングは並列処理が可能であるため分散処理による高速化が考えられる．並列分散処理は CPU のマルチコア化やグリッドコンピューティングの普及により，近年情報技術の分野で最も注目を浴びているキーワードの一つである．コードクローンの検出技術の分野でも並列分散処理を利用した手法がいくつか提案されており [22] [23] [24] 本論文で示した検出手法においても並列分散処理による高速化が期待できる．

謝辞

本研究を遂行するにあたり多くの方々にご指導，ご助言，ご協力を賜りました．この場を借りてこれまでお世話になった方々に感謝の意を表したいと思います．

主指導教員であり本論文の審査委員を務めていただいた飯田元教授に対し，厚く御礼を申し上げたいと思います．修士二年になってから研究テーマを変更することになりましたが，その際に飯田先生の研究室に移籍することを快く承諾して頂きました．また研究を行う上で必要となる PC やネットワーク等の環境を貸していただきました．研究内容に関してもいくつかのご助言を賜りました．心より厚く御礼申し上げます．

副指導教員であり本論文の審査委員を務めていただいた関浩之教授からは，研究の中間報告等を通じて貴重なご意見を賜りました．心より厚く御礼申し上げます．

副指導教員であり本論文の審査委員を務めていただいた松本健一教授に対し，厚く御礼を申し上げたいと思います．大学院に入学してからの最初の一年間は松本先生の研究室に所属し，その間に熱意のあるご指導を賜りました．特に研究に対すると姿勢と研究発表の仕方に関してご指導を賜りました．修士二年になり研究室は変わりましたが，この間松本先生より賜ったご指導がなければ本研究を成し遂げることは不可能でした．心より厚く御礼申し上げます．

副指導教員であり本論文の審査委員を務めていただいた吉田則裕助教からは，修士二年の一年間通じて多くのご助言を賜りました．コードクローンの専門家である吉田先生のご助言があったからこそ，本研究を進めていくことができました．また論文執筆においても多くのご助力を賜りました．心より厚く御礼申し上げます．

静岡大学の森崎修司助教からは大学院に入学してからの最初の一年間，資料作成，プレゼンテーション技術，論文執筆など研究の基礎ともいえることすべてに渡りご指導いただきました．研究テーマこそ異なるものの，修士一年のときに森崎先生の元でご指導を賜ったからこそ本研究を成し遂げることができました．心より厚く御礼申し上げます．

奈良先端科学技術大学院大学の伏田享平特任助教からは研究内容について多く

のご意見を賜りました．また論文の執筆においても多くのご助力を賜りました．心より厚く御礼申し上げます．

奈良先端科学技術大学院大学の Raula Kula 氏からは英語論文の執筆に関してご協力を賜りました．心より厚く御礼申し上げます．

奈良先端科学技術大学院大学の大平雅雄助教からは本論文の構成に関していくつかのご助言を賜りました．心より厚く御礼申し上げます．

大阪大学の肥後芳樹助教からは本研究の評価手法に関してご助言を賜りました．心より厚く御礼申し上げます．

参考文献

[1] Andrew Hunt and David Thomas. *The pragmatic programmer: from journeyman to master*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[2] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, Boston, MA, USA, 1999.

[3] Miryung Kim, Lawrence Bergman, Tessa Lau, and David Notkin. An ethnographic study of copy and paste programming practices in oopl. In *Proceedings of the 2004 International Symposium on Empirical Software Engineering*, pp. 83–92, Washington, DC, USA, 2004. IEEE Computer Society.

[4] Stefan Bellon, Rainer Koschke, Giuliano Antoniol, Jens Krinke, and Ettore Merlo. Comparison and evaluation of clone detection tools. *IEEE TSE*, Vol. 33, No. 9, pp. 577–591, 2007.

[5] Shinji Uchida, Akito Monden, Naoki Ohsugi, Toshihiro Kamiya, Ken ichi Matsumoto, and Hideo Kudo. Software analysis by code clones in open source software. *Journal of Computer Information Systems*, Vol. XLV, No. 3, pp. 1–11, April 2005.

[6] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. Ccfinder: A multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering*, Vol. 28, pp. 654–670, 2002.

[7] Xifeng Yan, Jiawei Han, and Ramin Afshar. Clospan: Mining closed sequential patterns in large datasets. In *In SDM*, pp. 166–177, 2003.

[8] Zhenmin Li, Shan Lu, Suvda Myagmar, and Yuanyuan Zhou. Cp-miner: Finding copy-paste and related bugs in large-scale software code. *IEEE Transactions on Software Engineering*, Vol. 32, pp. 176–192, 2006.

[9] Raghavan Komondoor and Susan Horwitz. Using slicing to identify duplication in source code. In *Proceedings of the 8th International Symposium on Static Analysis*, SAS '01, pp. 40–56, London, UK, 2001. Springer-Verlag.

[10] 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, 12 2010. 情報処理学会創立 50 周年記念論文.

[11] Ira D. Baxter, Andrew Yahin, Leonardo Moura, Marcelo Sant' Anna, and Lorraine Bier. Clone detection using abstract syntax trees, 1998.

[12] Lingxiao Jiang, Ghassan Misherghi, Zhendong Su, and Stephane Glondu. Deckard: Scalable and accurate tree-based detection of code clones. In *Proceedings of the 29th international conference on Software Engineering*, ICSE '07, pp. 96–105, Washington, DC, USA, 2007. IEEE Computer Society.

[13] Md. Sharif Uddin, Chanchal K. Roy, Kevin A. Schneider, and Abram Hindle. On the effectiveness of simhash for detecting near-miss clones in large scale software systems. In Martin Pinzger, Denys Poshyvanyk, and Jim Buckley, editors, *WCRE*, pp. 13–22. IEEE Computer Society, 2011.

[14] Moses S. Charikar. Similarity estimation techniques from rounding algorithms. In *Proceedings of the thiry-fourth annual ACM symposium on Theory of computing*, STOC '02, pp. 380–388, New York, NY, USA, 2002. ACM.

[15] A. J. Perlis and K. Samelson. Preliminary report: international algebraic language. *Commun. ACM*, Vol. 1, pp. 8–22, December 1958.

[16] 肥後芳樹, 吉田則裕, 楠本真二, 井上克郎. 産学連携に基づいたコードクローン可視化手法の改良と実装 (情報システム開発, 特集産学連携論文). 情報処理学会論文誌, Vol. 48, No. 2, pp. 811–822, 2007-02-15.

[17] Yoshiki Higo, Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue.

Method and implementation for investigating code clones in a software system. *Inf. Softw. Technol.*, Vol. 49, pp. 985–998, September 2007.

[18] Piotr Indyk and Rajeev Motwani. Approximate nearest neighbors: towards removing the curse of dimensionality. In *Proceedings of the thirtieth annual ACM symposium on Theory of computing*, STOC '98, pp. 604–613, New York, NY, USA, 1998. ACM.

[19] Mayur Datar, Nicole Immorlica, Piotr Indyk, and Vahab S. Mirrokni. Locality-sensitive hashing scheme based on p-stable distributions. In *Proceedings of the twentieth annual symposium on Computational geometry*, SCG '04, pp. 253–262, New York, NY, USA, 2004. ACM.

[20] E Lsh and Alexandr Andoni. E 2 lsh 0.1 user manual. *Interface*, pp. 0–22, 2005.

[21] Takeaki Uno, Masashi Kiyomi, and Hiroki Arimura. *LCM ver . 2 : Efficient Mining Algorithms for Frequent / Closed / Maximal Itemsets Algorithms for Efficient Enu- meration*, Vol. 04. Citeseer, 2004.

[22] 岡原聖, 真鍋雄貴, 山内寛己, 門田暁人, 松本健一, 井上克郎. コードクローンの長さに基づくプログラム盗用確率の実験的算出. 電子情報通信学会技術研究報告. SS, ソフトウェアサイエンス, Vol. 108, No. 362, pp. 7–11, 2008-12-11.

[23] 肥後芳樹, リビエリシモネ, 松下誠, 井上克郎. 大規模ソースコードを対象としたコードクローンの検出と可視化 (ソフトウェアテスト技法・保守技術). 情報処理学会論文誌, Vol. 48, No. 11, pp. 3510–3519, 2007-11-15.

[24] Simone Livieri, Yoshiki Higo, Makoto Matushita, and Katsuro Inoue. Very-large scale code clone analysis and visualization of open source programs using distributed ccfinder: D-ccfinder. In *Proceedings of the 29th international conference on Software Engineering*, ICSE '07, pp. 106–115, Washington, DC, USA, 2007. IEEE Computer Society.