社団法人 電子情報通信学会
THE INSTITUTE OF ELECTRONICS,
INFORMATION AND COMMUNICATION ENGINEERS

信学技報
TECHNICAL REPORT OF IEICE.

# 近傍ハッシュ法を用いた2段階のクラスタリングに基づく Near-miss クローンの検出

吉岡 俊輔† 吉田 則裕† 伏田 享平† 飯田 元†

† 奈良先端科学技術大学院大学情報科学研究科

**あらまし** 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 コード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 コード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 コード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, LSH

# Near-Miss Clone Detection Based on Two-stage Clustering Using Neighborhood Hashing

Shunsuke YOSHIOKA†, Norihiro YOSHIDA†, Kyohei FUSHIDA†, and Hajimu IIDA†

† Graduate School of Information Science, Nara Institute of Science and Technology, Japan

**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 paper presents a technique using two-stage clustering for near-miss clone detection, and show the detection time proportional to the size of source code.

**Key words** Code Clone, LSH

## 1. まえがき

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 [5] [4]．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．

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 [9]，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，大規模なソフト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困難である．その理由は、開発者がコード片をコピーアンドペーストした後で

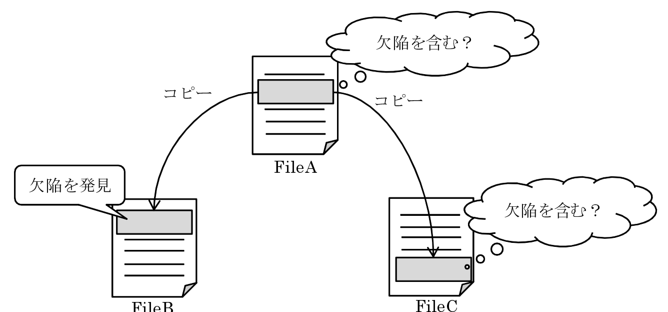


図 1 コードクローンの概要

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難しいためである (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のことを Near-Miss クローンという)．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 Near-Miss コードクローンの検出を行う手法が提案されてきた．しかし従来の手法では上記の変更に対してはうまく検出できても、ステートメントの挿入・削除などの変更に対してはうまく検出を行うことができない，あるいは検出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．

そこで本論文では、ステートメントの挿入・削除などの変更を加えた Near-Miss 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．

## 2. コードクローンとは

コードクローン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 [3]．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．

**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．

**TYPE-3**　TYPE-2 の変更に加えてステートメントの挿入や削除，変更などが行われたコードクローンを指す．

Near-Miss コードクローンは Bellon らの定義では Type-2,3 コードクローンの事を指す．近年のコードクローンの研究では Bellon らの定義の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て Bellon らの定義を用いる．

## 3. 従来の検出手法

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

トークンの並びに着目した検出手法では，ソースコードの先頭から現れるトークンの種類を調べ，同じ種類のトークンの繰り返しを見つけ出すことによりコードクローンの検出を行う [8] [11]．この手法を用いることにより識別子や演算子の違いを吸収してコードクローンの検出を行うことができるため，TYPE-1,2 コードクローンを見つけ出すことができる．一方 TYPE-3 コードクローンを検出することが難しいという問題がある．

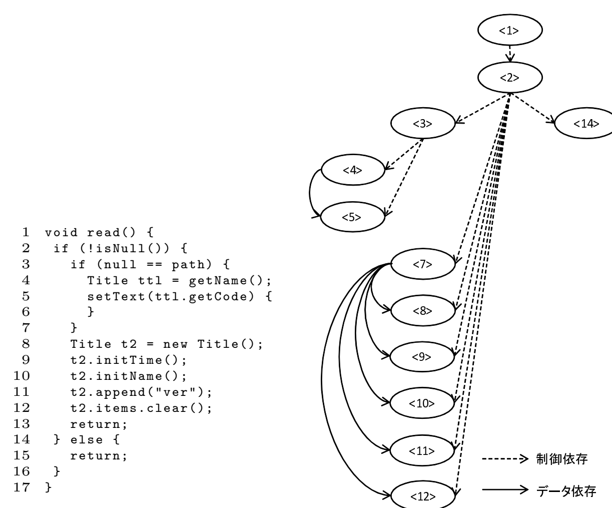


図 2　プログラム依存グラフの例

### 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．ソースコードの静的解析の分野で広く用いられている．図 2 はその具体例である．左側がソースコード，右側は左側のソースコードに対するプログラム依存グラフである．

プログラム依存グラフを用いた検出では，類似した構造のグラフを探し出すことによりコードクローンの検出を行う [10] [18]．この手法を用いることにより TYPE-1,2,3 コードクローンの検出を行うことができるが，グラフの構造の比較には時間がかかるという欠点がある．

### 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木構造で表したグラフのことを指す．抽象構文木を用いたコードクローンの検出では，類似した木構造を持つ部分木を探し出すことにより検出を行う [2] [7]．この手法を用いることにより TYPE-1,2,3 コードクローンの検出を行うことができるが，しかし木構造の比較には時間がかかるという欠点がある．

### 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．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 [14]．Uddin らはこの手法を用いて TYPE-1,2,3 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

これまで提案されてきたコードクローンの検出手法は TYPE-3 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 [8]，Barker ら [11] の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 [3] 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．一方 TYPE-3 コードクローンを検出することができないという欠点がある．

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 コードクローンを検出することができるが，一方コードクローンの検出に膨大な時間がかかるという欠点がある．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対しては利用が困難である．

本論文ではこれらの問題を踏まえ，TYPE-3 コードクローンを高速に検出する手法を提案する (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

本節では手続き型言語を対象にコードクローンの検出手法を述べる．なお説明上ソースコードを示す必要がある場合は Java を用いる．

図 3 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位置情報として出力する．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** 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．

**STEP3** 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を向上させる．詳細は 5.3.3 節で述べる．

**STEP4** 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コードクローンを連結し，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 5.3.4 節で述べる．

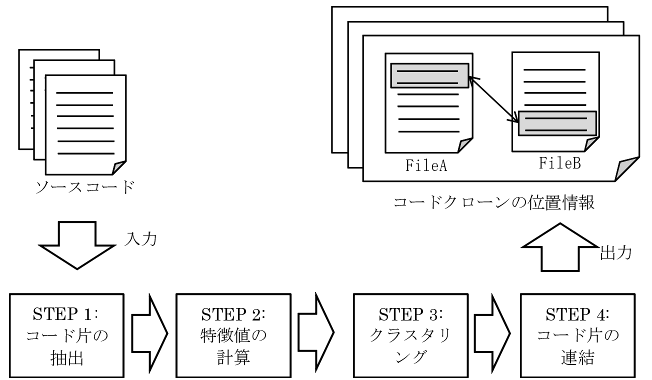


図 3 検出手法の概要

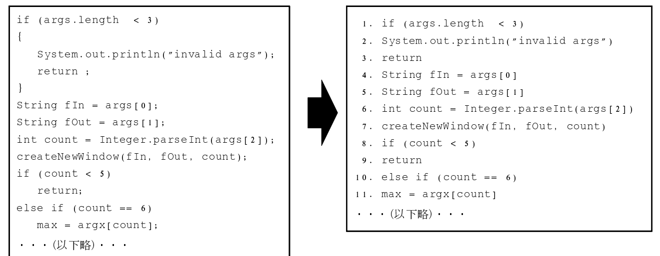


図 4 コード片をステートメントのリストとして解釈

### 5.2 準備

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 [13]．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．

本研究ではステートメントの構造を単純化するために，ブロック自体をステートメントとして解釈しない．また制御ステートメントとそれ以外のステートメントを同種のステートメントとして扱う．したがって図 4 の左側のコード片は右側の様なネスト構造を持たない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 5 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

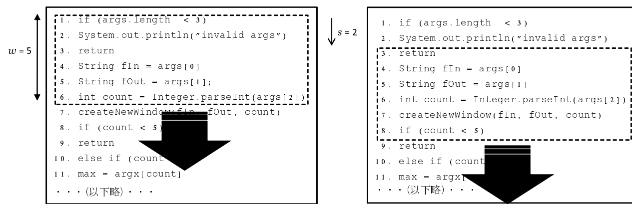

図 5　パラメータ $w$ に基づくコード片の抽出

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．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幅と呼び自然数 $s$ で与える．$s$ は初期パラーメータとして与えれる定数値である．

#### 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，コード片に含まれる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値として以下の項目について評価する．

- 条件分岐に関係する文か否か
- ループに関係する文か否か
- ジャンプ命令（ループからの脱出）に関係する文か否か
- ジャンプ命令（ループへの再入）に関係する文か否か
- 三項演算子を含むか否か
- 例外の検出に関係する文か否か
- 例外の処理に関係する文か否か
- 整数値のリテラルを含むか
- 小数値のリテラルを含むか否か
- 文字列のリテラルを含むか否か
- 文字のリテラルを含むか否か
- 変数の代入文か否か

これらの項目はソースコードの処理の流れと，局所的情報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，ステートメントの局所的情報が得られる．

上記の各項目を評価することにより，各ステートメントは，条件を満たす場合を 1 満たさない場合を 0 とした 2 値ベクトルで表現される．

#### 5.3.3　クラスタリング

コード片に含まれるステートメントの特徴ベクトルの値に基づき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) [6] を用いる．LSH はあるクエリーのベクトルに対して値が近ければ近いほどクエリーのベクトルと同じクラスターに分類される確率が高いという特徴を持つ．LSH は $d$ をデータセットの次元，$n$ をデータセットの数，$\rho$ を LSH の確率に関するパラメータとすると，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．こ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ことによりデータセットの数を削減する．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**　データセットの数を削減するために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 $d$，ウィンドウサイズを $w$ 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．

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

上記の式に基づき得られた各コード片の特徴ベクトルを LSH を用いて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内の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，各ステートメントが個別に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ステートメント単位の情報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．具体的には 1 段階目の結果得られた各クラスタのコード片をステートメント単位で比較する．例えば一段階目のクラスタリングの結果得られたあるコード片のクラスタを $A = \{c_1, c_2, \cdots, c_n\}$ とすると，$c_1$ の各ステートメントと $A \setminus \{c_1\}$ の各コード片のステートメントを比較し，$c_1$ のステートメントと類似したステートメントを一定以上含むものを抽出して $c_1$ と共に新たなクラスターを作成する．

#### 5.3.4　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ウィンドウサイズ $w$ 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．本手法ではこの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．具体的にはメソッド内の存在するコードクローンの集合をトランザクションとして頻出パターンマイニングを行い，関連するコードクローンのパターンを抽出して連結する．

## 6. 実　　験

### 6.1　実 験 概 要

5. 節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．実行環境は，OS は Windows7 Professional 64bit，CPU は Intel Core i7-2600K(1 コアのみ使用)，メモリは 16GB である．解析対象のプロジェクトは netbeans-javadoc，eclipse-ant，jdk1.5.0，j2sdk1.4.0-javax-swing，eclipse-jdtcore，lucene，soot，vuze で全て Java の OSS である．解析対象のプロジェクトは，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．ただし，LOC は 5.2 節の定義に基づく．

なおクラスタリングには LSH の実装の E2LSH を用いた [1]. E2LSH は LSH のパラメータを自動的に決定する機能を持つ．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．本実験では E2LSH が自動的に算出した値を用いた．

またコード片の連結には頻出パターンマイニングの実装の LCM [15] を用いた．

### 6.2 結果と考察

#### 6.2.1 検出対象の規模と検出時間の関係

図 6 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線形の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 5.3.3 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線形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線形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を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個々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．

#### 6.2.2 検出されたコードクローンの具体例

図 7 は提案手法で検出されたコードクローンの具体例である．+記号の部分がコードクローンとして検出された部分である．内容は X.509 の認証に関する処理である．左側の 4 行目では配列変数に添字を用いてメソッドを呼び出しているのに対して，右側の 5,6 行目では配列の要素を一度変数に代入してからメソッドを呼び出している．左右のコード片は構文木やプログラム依存グラフの形状が異なるため既存の手法では検出が困難である．

次に，図 8 は肥後らによって 2010 年に発表されたプログラム依存グラフによる最新の検出手法 [18] で検出されたコードクローンである．+記号の部分がコードクローンとして検出された部分である．内容は Java のソースコードの解析に関する処理である．提案手法は図 8 のコードクローンをうまく検出することができなかった．理由としては，コード片の抽出時におけるウィンドウサイズ $w$ の値に対して図 8 のコード片のサイズが小さかったためと考えられる．このことから，本手法だけでは図 8 の様な細切れの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検出することができないと思われる．

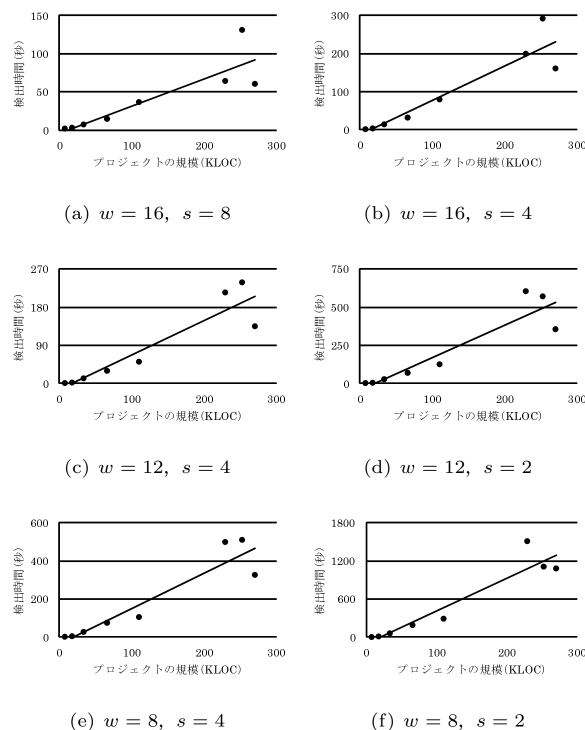


図 6 解析対象のサイズと検出時間の関係 (図中のプロットは左端から順に javadoc，ant，jdk1.5.0，swing，jdtcore，lucene，soot，vuze)

## 7. 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が分かった．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，コードクローンの検出用途に応じて既存の手法と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 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 [16] [17] [12] 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** 本研究は，日本学術振興会 科学研究費補助金 基盤研究（C）（課題番号:22500027），および研究活動スタート支援

```
 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 7　検出されたコードクローンの例

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

```
 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 ) {
```

図 8　肥後等の手法 [18] では検出されたが，提案手法では検出されなかったコードクローンの例

（課題番号:22800040,22800043）の助成を得た．

## 文　　献


[1] Alexandr Andoni and Piotr Indyk. E2LSH 0.1 User Manual, 2005. http://www.mit.edu/ andoni/LSH/.

[2] I.D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier. Clone detection using abstract syntax trees. In *Software Maintenance, 1998. Proceedings. International Conference on*, pages 368 –377, nov 1998.

[3] Stefan Bellon, Rainer Koschke, Giuliano Antoniol, Jens Krinke, and Ettore Merlo. Comparison and Evaluation of Clone Detection Tools. *IEEE Transactions on Software Engineering*, 33(9):577–591, 2007.

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

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

[6] 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, pages 604–613, New York, NY, USA, 1998. ACM.

[7] 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, pages 96–105, Washington, DC, USA, 2007. IEEE Computer Society.

[8] 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*, 28:654–670, 2002.

[9] 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*, pages 83–92, Washington, DC, USA, 2004. IEEE Computer Society.

[10] 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, pages 40–56, London, UK, 2001. Springer-Verlag.

[11] 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*, 32:176–192, 2006.

[12] 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, pages 106–115, Washington, DC, USA, 2007. IEEE Computer Society.

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

[14] 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*, pages 13–22. IEEE Computer Society, 2011.

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

[16] 岡原 聖, 真鍋 雄貴, 山内 寛己, 門田 暁人, 松本 健一, and 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. In **電子情報通信学会技術研究報告 ，ソフトウェアサイエンス**, pages 7–11. 社団法人電子情報通信学会, 2008-12-11.

[17] 肥後 芳樹, リビエリ シモネ, 松下 誠, and 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**, 48(11):3510–3519, 2007-11-15.

[18] 肥後 芳樹 and 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**, 51(12):2149–2168, 12 2010. 情報処理学会創立 50 周年記念論文.