情報検索技術に基づく細粒度ブロッククローン検出

横井 一輝　崔 恩瀞　吉田 則裕　井上 克郎

本研究では情報検索技術を利用したブロッククローン (コードブロック単位のコードクローン) の検出手法を提案する．既存研究において関数単位でコードクローンを検出する手法を提案したが，この手法は検出粒度が大きいため検出漏れが起きるという問題点があった．この問題を解決するために，本手法では，ソースコード中の識別子や予約語に利用される単語に対して重み付けを行い，コードブロックから変換された特徴ベクトル間の類似度を求め，ブロック単位でクローンの検出を行う．この手法は，クラスタリングの手法や特徴ベクトルのデータ構造の工夫を行うことにより．既存手法と比較して，より小さい粒度でコードクローンを検出することができ，かつ検出時間とメモリ使用量を削減できるという利点がある．また評価実験では，既存のコードクローン検出手法と比較を行い，高速に高い精度で検出を行うことができた．

In this paper, we propose an approach for detecting block clones (i.e., block-level code clones) using information retrieval techniques. In the previous study, we have presented an approach for detecting function clones. However, the previous approach misses a number of code clones because it only identifies coarse-grained code clones (i.e., function clones). To mitigate this problem, this study proposes an approach that detects block clones by generating feature vectors for code blocks based on the occurrence of identifiers and reserved keywords and then performing clustering of the generated vectors. Also, we improve a clustering method and a data structure of feature vectors in the proposed approach. It leads not only the detection of fine-grained code clones but also less detection time and low memory consumption, compared to the previous approach. As a case study, we compared the proposed approach with the existing code clone approaches and then confirmed the effectiveness of the proposed approach.

1　まえがき

ソフトウェアの保守における問題のひとつとして，コードクローンが指摘されている [4]．コードクローンとは，ソースコード中に含まれる互いに一致または類似した部分を持つコード片のことであり，一般的に，コードクローンの存在はソフトウェアの保守を困難にすると言われている．コードクローンに対する様々な保守や管理の方法が提案されているが，ソースコードの規模が大きくなるとソースコード中に含まれるコードクローンも膨大な量となり，手作業でそれらを管理することは困難となる．そこで，コードクローンを自動的に検出することを目的とした様々なコードクローン検出手法が提案されている [11] [23]．

山中らは TF-IDF [3] と LSH [14] を利用することによって，意味的に処理が類似した関数単位のコードクローンを検出する手法 (関数クローン検出法) を提案した [32]．コード片単位で検出を行う場合，構文の不完全な部分で終了するコード片など，集約を行うことが困難なコードクローンが多く検出されることがある [33]．一方，関数クローンは処理の内容がまとまっているため，開発者にとって集約の対象になりやすい

Fine-grained Block Clone Detection Based on Information Retrieval Techniques.

Kazuki Yokoi, Katsuro Inoue, 大阪大学大学院情報科学研究科, Graduate School of Information Science and Technology, Osaka University.

Eunjong Choi, 奈良先端科学技術大学院大学先端科学技術研究科, Graduate School of Science and Technology, Nara Institute of Science and Technology.

Norihiro Yoshida, 名古屋大学大学院情報学研究科, Graduate School of Informatics, Nagoya University.

コンピュータソフトウェア, Vol.35, No.4 (2018), pp.16–36.

[研究論文] 2018 年 1 月 17 日受付.

コードクローンを検出できる．山中らの手法では，情報検索技術の一種である TF-IDF [3] を用いて，ソースコード中の識別子や予約語に利用される単語に対して重み付けを行う．そして，重み付けに基づいて各関数を特徴ベクトルに変換し，特徴ベクトル間の類似度を計算することによって，関数クローンの検出を行う．また，近似最近傍探索アルゴリズムの一種である LSH (Locality-Sensitive Hashing) [14] を用いて，特徴ベクトルをクラスタリングすることにより，検出の高速化を行っている．しかし，山中らの手法では検出粒度が関数単位のため，長い関数内の一部にコードクローンが含まれた場合，検出漏れが生じる可能性がある．このような検出漏れを減らすためには，関数単位より小さい粒度で構文上のまとまりがあるコードクローンの検出を行うべきである．

そこで，本研究ではコードブロック単位のコードクローン (ブロッククローン) を検出する手法を提案する．関数単位より小さい粒度であるコードブロック単位で検出を行うことで，関数単位では検出できなかったようなコードクローンが検出でき，検出精度が向上する．本研究ではコードブロックを，関数と，関数内部の if，for，while 文などの中括弧で囲まれた部分と定義する．本手法では，まずソースコードに対して構文解析を行い，コードブロックの抽出を行う．その後，抽出した各コードブロックに対して TF-IDF を用いて特徴ベクトルに変換し，特徴ベクトル間の類似度を計算することでブロッククローンの検出を行う．また本手法でも山中らの手法と同様，LSH を用いた特徴ベクトルのクラスタリングを行い，検出の高速化を行う．

しかし，関数単位より小さい粒度で検出を行うため，検出時間とメモリ使用量が増大する問題が発生する．この問題に対応するため，以下の 2 点の工夫を行った．1 点目は，クラスタリング手法の LSH の変更である．LSH は様々な応用手法が研究されており，本研究では提案手法に適した LSH の検討を行った．その結果，multi-probe LSH [21] と cross-polytope LSH [2] [30] を組み合わせた手法 [2] を用いて特徴ベクトルのクラスタリングを行うことを選択した．multi-probe LSH は，従来の LSH が抱えるメモリ使用量の問題点を改良したアルゴリズムである [18]．また，cross-polytope LSH は TF-IDF を用いた特徴ベクトルのクラスタリングに適したアルゴリズムである [2]．これらを組み合わせた手法を用いることで少ないメモリで高速なクラスタリングが可能となった．2 点目は，特徴ベクトルを表現するデータ構造の変更である．TF-IDF を用いた特徴ベクトルが疎 (ほとんどの要素が 0) である性質に着目し，0 以外の要素のみを保持するデータ構造として実装した．これにより，メモリ使用量や入出力時間の削減を実現した．上記の工夫を行うことで，山中らの関数クローン検出手法より検出粒度が小さいにもかかわらず，より高速な検出を実現し，さらに大規模な検出対象に対しても適用可能となった．

評価実験では，関数クローン検出法 [32] と，字句単位のコードクローン検出ツールである CCFinderX [16] の，2 つの手法を検出精度と検出時間の観点から比較を行った．3 つの C 言語のプロジェクトに対して適用した結果，本手法が総合的に高い精度でより多くのコードクローンを検出することができた．また，本手法は関数クローン検出法や CCFinderX と比較して高速にコードクローンを検出することができた．また，スケーラビリティの評価を行い，1KLOC から 10MLOC までの検出対象において線形的に検出時間が増加することを確認した．

以降，2 章では，コードクローンと関数クローン検出法について述べる．3 章では，本研究で提案するブロッククローン検出法について述べる．4 章では，本手法の評価実験について述べる．5 章では，本研究の考察について述べる．6 章では，関連研究について述べる．最後に，7 章でまとめと今後の課題について述べる．

2 コードクローン

本章では，本研究の背景としてコードクローン，および山中らの関数クローン検出法と，その問題点について述べる．Roy らはコードクローンの定義として，コードクローン間の違いの度合いに基づき以下の 4 つのタイプに分類している [26]．

タイプ 1 空白やタブの有無，コーディングスタイル，コメントの有無などの違いを除き完全に一

致するコードクローン．

タイプ 2　タイプ 1 の違いに加えて，変数名などのユーザー定義名，変数の型などが異なるコードクローン．

タイプ 3　タイプ 2 の違いに加えて，文の挿入や削除，変更などが行われているコードクローン．

タイプ 4　類似した処理を実行するが，構文上の実装が異なるコードクローン．

タイプ 4 のコードクローン間の差異として以下が挙げられる．

- 条件分岐処理や繰り返し処理などの制御構造の実装が異なる．
- 中間媒介変数の利用の有無が存在している．
- 文の並び替えが発生している．

2.1 関数クローン検出法

山中らは TF-IDF と LSH を利用することによって，意味的に処理が類似した関数クローンを検出する手法を提案した [32]．コード片単位でコードクローンの検出を行う場合，構文の不完全な部分で終了するコード片など，集約を行うことが困難であるコードクローンが多く検出される恐れがある [33]．一方，関数クローンは処理の内容がまとまっているため，開発者にとって集約の対象になりやすいコードクローンが検出できる．また関数クローン検出法は，タイプ 1 からタイプ 4 までのコードクローンを検出可能である．この手法は，まず，入力されたソースコード中のワード (例：予約語，識別子名) に基づいて各関数を特徴ベクトルに変換する．そして，特徴ベクトル間の類似度を求めることによってクローンペア (互いに処理が類似したクローンの対) の集合をリストとして出力する．また，類似度の計算の直前に LSH [14] を利用し，特徴ベクトルのクラスタリングを行うことによって，検出の高速化を行っている．

2.2 関数クローン検出法の問題点

関数クローン検出法に対して以下の 2 つの問題点が挙げられる．

1 つ目は，関数単位の検出による問題点である．この手法は，関数全体ではなく，一部のみがコードクローンになっている場合に検出することができない．例えば図 1 のように，長い関数内の一部にコードクローンが含まれる場合，検出漏れが生じる可能性がある．

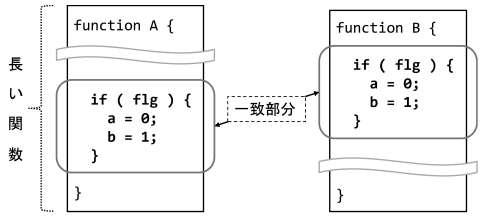


図 1　長い関数内の一部にコードクローンを含む例

2 つ目は，メモリ使用量が多いという問題点である．関数を特徴ベクトルに変換する方法として関数クローン検出法は TF-IDF を用いているが，TF-IDF では全関数で出現したワードの種類数が次元数となるため，特徴ベクトルの次元数が非常に大きくなる傾向にある．特徴ベクトルは各関数に 1 個ずつ与えるため，次元数の大きい特徴ベクトルを多数保持することになり，メモリ使用量が大きくなる．また，高速に検出を行うために LSH を用いてクラスタリングを行っているが，LSH は精度を上げるとメモリ使用量が大きくなる特徴がある．よって，大規模プロジェクト (Linux Kernel など) に対してコードクローンの検出を行う場合，メモリ不足で検出を完了できない恐れがある．

3 提案する検出手法

2.2 節で述べた 2 つの問題点を踏まえ，新たなコードクローン検出法の必要性が考えられる．そこで，本研究では関数単位より検出粒度を小さくした，コードブロック単位のコードクローン検出法 (ブロッククローン検出法) を提案する．ブロッククローン検出法では，関数クローン単位では検出できなかったコードクローンの検出が可能になる．例えば図 1 のように，長い関数内の一部にコードクローンが含まれる場合も，提案手法では検出が可能である．また，字句単位の検出より処理の内容がまとまっているため，開発者にとって集約などの保守作業の対象となりやすい．以上の観点から本手法には有用性があると考えられる．

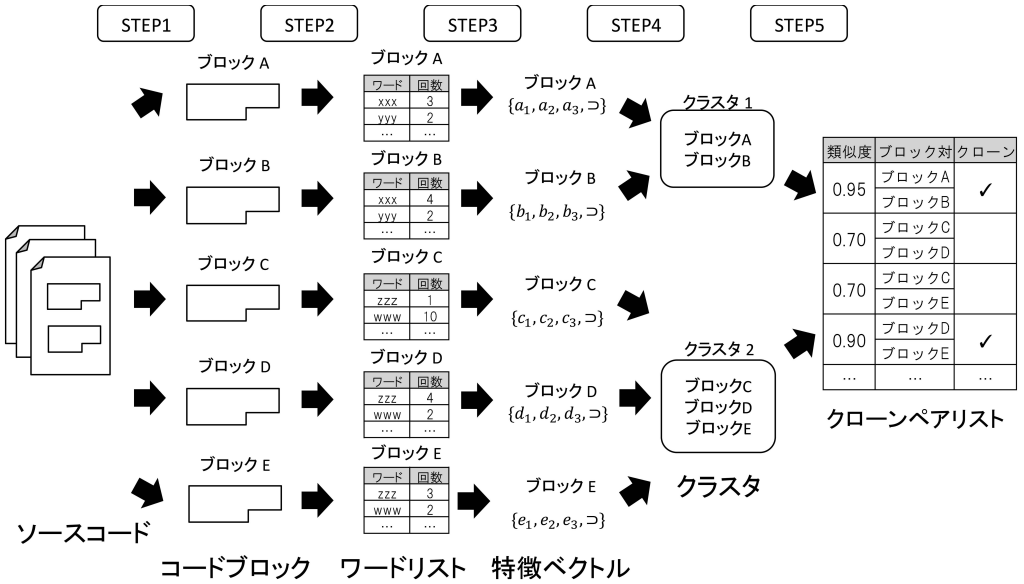


図 2 提案手法の概要

本研究では，2.1 節で説明した山中らの関数クローン検出法をもとに，タイプ 1 からタイプ 4 すべてに対応したブロッククローン検出法を提案する．本手法の概要を図 2 に示す．本手法は主に以下の 5 つのステップで実行される．

STEP 1 ソースコードから抽象構文木を生成し，生成した抽象構文木からコードブロック (3.1.1 節参照) を取り出す．

STEP 2 STEP 1 で抽出した各コードブロックから，ワード (3.1.2 節参照) の抽出を行う．

STEP 3 TF-IDF を利用し，STEP 2 で抽出したワードに重み付けを行い，各コードブロックを特徴ベクトルに変換する．

STEP 4 LSH を利用し，STEP 3 で求めた各コードブロックに対する特徴ベクトルのクラスタリングを行う．

STEP 5 STEP 4 で求めたコードブロックの各クラスタの中で，特徴ベクトル間の類似度の計算を行い，ブロッククローン (3.1.3 節参照) を検出する．

本手法と関数クローン検出法の相違点は，以下の 3 点である．

- コードクローンの検出粒度
- 特徴ベクトルをクラスタリングする手法の選択と実装
- 特徴ベクトルを表現するデータ構造の選択と実装

主な相違点はコードクローンの検出粒度を小さくした点である．本手法では関数単位だけでなく，関数内のコードブロック単位の両方のコードクローンを検出する．検出粒度を関数より小さくすることで，検出精度の向上を実現した (4.1 節参照)．しかし，検出粒度を小さくすると検出対象数が増加し，それに伴い検出時間とメモリ使用量が増大する問題が新たに発生する．この問題に対処するため，さらに 2 つの変更を適用した．

まず，特徴ベクトルのクラスタリングを行う LSH を変更した．関数クローン検出法でも LSH を用いて特徴ベクトルのクラスタリングが行われており，これにより高速な検出を実現している．しかし，近年 LSH は様々な応用手法が研究されており，関数クローン検出法では LSH の応用手法の検討まではされていない．また，LSH によるクラスタリングが占める時間の割合が大きく無視できないため [31]，本手法に適した LSH の検討を行った．その結果，multi-probe LSH [21] と cross-polytope LSH [2] [30] の 2 つの既存手法を組み合わせる手法 [2] に変更した．multi-probe

LSH とは，空間計算量の改良を行った LSH である．また，cross-polytope LSH とは，角度距離に基づいてハッシュ化を行う LSH である．角度距離に基づいた cross-polytope LSH は TF-IDF を用いた特徴ベクトルのクラスタリングが高速に行える．multi-probe LSH を cross-polytope LSH に組み合わせることで，より少ない空間計算量でより高速にクラスタリングが可能となることが示されている [2].

さらに，特徴ベクトルを表現するデータ構造を変更した．特徴ベクトルは，検出対象数の増加に伴いベクトルの次元が高次元になる性質があり，次元が高次元になるほどメモリ使用量とベクトルの計算時間が増大する．関数クローン検出法では，特徴ベクトルのすべての要素の値を保持するデータ構造をしている．関数単位の検出では問題がないが，検出粒度を小さくしコードブロック単位で検出を行うには，上記の高次元の問題が無視できなくなる．そこで，本手法のように TF-IDF を用いた特徴ベクトルでは，ほとんどの要素が 0 である疎なベクトルとなる性質に着目し，非 0 要素のみを保持するデータ構造を選択した．データ構造を疎なベクトルとして実装することで，メモリ使用量や計算時間，さらに入出力時間の削減を行った．

以上の変更により，関数クローン検出法と比較して検出粒度は小さくなったにもかかわらず，より高速な検出を実現し，さらに大規模な検出対象に対しても適用可能となった．

3.1 用語の定義

3.1.1 コードブロック

本研究では，プログラミング言語において，複数の命令文を一括りにまとめたものをコードブロックという．多くのプログラミング言語では，コードブロックを入れ子構造にすることができ，変数のスコープとしての意味を持つことがある．

本手法では，以下の 2 つの条件のいずれかを満たすコードブロックを検出対象とする．対象言語は C 言語と Java 言語とする．

条件 1-1 関数の ‘{ }’ で囲まれた範囲

条件 1-2 if，else，for，while，do-while，switch 文の ‘{ }’ で囲まれた範囲

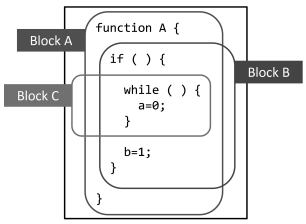


図 3 入れ子構造において親子関係にあるコードブロック

ただし，後に ‘{ }’ が現れない単文の命令文はコードブロックとしてのまとまりがないため検出対象としない．また図 3 の Block A に対する Block B や Block C のように，入れ子構造における親を持つコードブロックを検出可能であり，検出対象を再帰的に探索する．

3.1.2 ワード

本手法では以下の条件 2-1，2-2 のいずれかを満たすものをワードとして定義する．

条件 2-1 予約語

条件 2-2 識別子名を構成する単語

識別子名が複数の単語から構成される場合，以下の方法でワード単位に分割する．

- ハイフンやアンダースコアなどの区切り記号による分割
- 識別子名中の大文字になっているアルファベットによる分割

また，2 文字以下の識別子は，それらをまとめて同一のメタワードとして扱う．例えば，繰り返し文などでよく利用される i や j といった変数は，意味情報が込められていない変数として扱うためである．さらに，条件分岐に用いられる if や while，繰り返しに用いられる for や while などの予約語もワードとして扱う．なお，各ワードの大文字と小文字による区別はつけず，同一のワードとして扱う．

3.1.3 ブロッククローン

本手法におけるブロッククローンについて説明する．最初にブロッククローンペアについて定義する．本手法では，以下の条件 3-1，3-2 の両方を満たすコードブロック対を，ブロッククローンペアと呼ぶ．

条件 3-1 コードブロック間の類似度が閾値以上

$$sim(CB_1, CB_2) \geq p \quad (0 \leq p \leq 1)$$

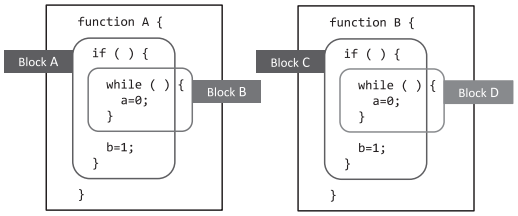


図 4 極大コードブロックペアと重複したコードブロックペア

条件 3-2 入れ子構造において，CB_1 が CB_2 の親でなく，かつ CB_2 が CB_1 の親でない

次に極大ブロッククローンについて定義する．CB_1，CB_2 がブロッククローンペアであり，かつ CB_1，CB_2 それぞれを真に包含する如何なるコードブロックもブロッククローンペアでないとき，CB_1，CB_2 を極大ブロッククローンと呼ぶ．本手法では，極大ブロッククローンをブロッククローンと定義する．条件 3-2 で示したように，ブロッククローンペアはコードブロック間に入れ子構造における親子関係がないことが条件である．コードブロック間に入れ子構造における親子関係が存在する場合，一方のコードブロックが他方を包含していることを示している．例えば，図 3 のコードブロック A と B は親子関係が存在し，包含関係にあるためブロッククローンペアでない．

また，極大ブロッククローンをブロッククローンと定義するとは，言い換えるとそれぞれ入れ子関係にあるコードブロックの類似度が閾値以上の場合，最も外側のコードブロックペアをブロッククローンとするという意味である．例えば，図 4 のコードブロック A と C，B と D それぞれの類似度が閾値以上となる場合，最も外側のコードブロック A と C をブロッククローンとする．

3.2 コードブロックとワードの抽出

本手法でのコードブロックとワードの抽出について説明する．最初にソースコードに対して構文解析を行い，抽象構文木を生成する．本手法では，構文解析には ANTLR v4 †1 を利用する．次に生成した抽象構文木に対して深さ優先探索で検索する．抽象構文木から関数 (3.1.1 節の条件 1-1 を満たすコードブロック) の部分木を取り出し，コードブロックとして抽出する．そして，取り出した部分木から，コードブロック (3.1.1 節の条件 1-2 を満たすコードブロック) の部分木を取り出し，関数の内側に存在するコードブロックを抽出する．コードブロックの抽出後，各コードブロック内に含まれるワードの抽出を行う．

†1 http://www.antlr.org/

3.3 特徴ベクトルの計算

特徴ベクトルの計算では，ワードに対し TF-IDF [3] を利用して重みを計算し，その値を特徴量として各コードブロックを特徴ベクトルに変換する．よって，各コードブロックの特徴ベクトルの次元数はソースコード中に存在する全ワードの種類数となる．本手法では，tf 値はコードブロック中のワードの出現頻度を，idf 値はソースコード中のワードの希少さを表している．tf 値は式 (1)，idf 値は式 (2) で与えられる．

$$tf_{i,j} = \frac{n_{i,j}}{\sum_{k \in b_j} n_{k,j}} \quad (1)$$

$$idf_i = \log \frac{|F|}{|\{f : f \ni w_i\}|} \quad (2)$$

ここでは，$n_{i,j}$ はコードブロック b_j 内におけるワード w_i の出現回数，$\sum_{k \in b_j} n_{k,j}$ はコードブロック b_j における全ワードの出現回数の和，$|F|$ は全関数の数，$|\{f : f \ni w_i\}|$ はワード w_i が出現する関数の数を示している．

関数クローン検出法と比較して，tf 値の求め方をコードブロック単位に変更したが，idf 値の求め方はコードブロック単位に変更せず関数単位のままである．なぜなら，コードブロック単位で idf 値を求めるとワードの重み付けに偏りが生じてしまうからである．あるワードがいくつのコードブロックに含まれるかは出現場所によって異なる．例えば，図 3 の関数内の変数 a はブロック A と B の 2 個のコードブロックに含まれるが，変数 b はブロック A のみにしか含まれない．そのため，ソースコード中の出現回数は同じにもかかわらず，出現するコードブロックの数が異なってしまう．idf 値はワードの希少性に基づいて重み付けを行っているため，偏りを無くすために関数単位で求めた．

また，TF-IDF を用いた特徴ベクトルは，非常に高次元かつ疎となる特性がある．そのため，非 0 要素のみを保持する疎ベクトルとして実装することでメモリ使用量の効率を改善した．さらに，特徴ベクトルの外部記憶装置への入出力処理の時間を高速化した．

3.4 特徴ベクトルのクラスタリング

特徴ベクトルのクラスタリングとして，近似最近傍探索の一種である LSH [14] を用いた．LSH とは，高次元なデータを確率的にハッシュ化し，近似的に近傍点を見つけるアルゴリズムである．クローンペアとなる特徴ベクトルは近傍点で表されるため，LSH を用いてクラスタリングすることで類似した特徴ベクトルを高速に絞り込むことができ，クローン検出の高速化を実現している．

最初に，LSH の一般的な説明を述べる．LSH は，(c, r)-Approximate Nearest Neighbor (ANN) と，(r, cr, p_1, p_2)-sensitive hash を用いて表される．(c, r)-ANN とは，近似最近傍探索の問題定義であり，(r, cr, p_1, p_2)-sensitive hash とは，空間的に距離が近い点が同じハッシュ値を取る確率が高くなる確率的ハッシュ関数である．(c, r)-ANN と，(r, cr, p_1, p_2)-sensitive hash の定義を以下に示す．

R^d 上に距離関数 D を定義する．

(c, r)-ANN　実数 $c > 1$，実数 $r > 0$，任意のクエリ $q \in R^d$ が与えられたとき，$p \in R^d$ を q の正解最近傍点とし，$D(p, q) < r$ ならば，

$$P' = \left\{ p' \in R^d : D(p', q) < cr \right\}$$

で定義される R^d の部分集合 P' を求める問題を (c, r)-ANN と呼ぶ．

(r, cr, p_1, p_2)-sensitive hash　任意の点 $p, q \in R^d$ が与えられたとき，

- if $D(p, q) < r$ then $\Pr[h(p) = h(q)] \geq p_1$
- if $D(p, q) > cr$ then $\Pr[h(p) = h(q)] \leq p_2$

を満たすハッシュ関数 h を (r, cr, p_1, p_2)-sensitive hash と呼ぶ．Pr は条件式が真となる確率を表している．

ここでは一般的な LSH で用いられる定義を示したが，上記のハッシュ関数 h を変更するなど様々な LSH の応用手法が提案されている．関数クローン検出法ではユークリッド空間に対する LSH を実装した E2LSH [1] †2 を用いてクラスタリングを行い検出の高速化を実現しているが，他の LSH の検討までは行われていない [32]．そこで本研究では，本手法により適した LSH の検討を行った．

一般的に LSH を用いて大規模のデータセットを高い精度で求めようとすると，メモリ使用量が非常に大きくなるという問題点がある [18] [21]．そこで，メモリ使用量を改良するため multi-probe LSH [21] という手法が提案されている．LSH のアルゴリズムは，空間的に距離が近い 2 点が同じハッシュ値になる確率が高くなるようなハッシュ関数を用い，同じハッシュ値を取る点を同じバケットに入れることで近傍点の検索を行う．しかし LSH は確率的手法であるため，近い 2 点が偶然に別のバケットに入る可能性が存在する．従来の LSH では複数のハッシュテーブルを用いることで確率的な誤差を少なくし精度を上げている．そのため，大規模なデータセットに対してはより多数のハッシュテーブルが必要となり，LSH のメモリ使用量の増大につながっている．multi-probe LSH は，ある点が入るバケットだけでなく，空間的に距離が近いバケット群も調べるという手法である．これにより，少ないハッシュテーブルでも偶然別のバケットに入った点の見落としを防いでいる．実際に，192 次元のデータセットに対して同じ再現率 (0.90) を得るために，従来の LSH は 49 個のハッシュテーブルが必要だったのに対し，multi-probe LSH は 3 個のハッシュテーブルで検索時間をほぼ落とさずに達成したという結果が示されている [21]．

本手法では，TF-IDF を用いた特徴ベクトルに対し，コサイン類似度によって類似したベクトルの検索を行うが (3.5 節参照)，これらは情報検索の分野でよく用いられる手法であり，上記のようなベクトル空間に対する LSH がいくつかある．その中で，角度距離に基づいてハッシュ化を行う cross-polytope LSH [2] [30] が，理論的に保証されており計算時間の観点から実用的な手法として提案されている．cross-polytope LSH は，実用性の面で優れているとして既に提案されてい

†2 http://www.mit.edu/~andoni/LSH/

た hyperplane LSH [6] と比較し，1.2 倍から 4.0 倍高速に同じ精度で検索できるという結果が示されており，さらに TF-IDF を用いた特徴ベクトルの場合は少なくとも 3 倍高速になるという結果も示されている [2].

Andoni らは multi-probe LSH を cross-polytope LSH に組み合わせた手法 [2] を提案しており，この 2 つの LSH を組み合わせた手法を本研究では選択した．なお，2 つの手法の組み合わせの実装として FALCONN [2] †3 ライブラリを利用した.

3.5 特徴ベクトルの類似度の計算

本手法ではコサイン類似度を用いてクローンペアの判定を行う．コサイン類似度は多次元ベクトルの類似度を表す尺度であり，次元が V である 2 つの特徴ベクトル $\vec{a}, \vec{b}$ 間の類似度は以下の式 (3) で与えられる．

$$sim(\vec{a},\vec{b}) = \cos(\vec{a},\vec{b}) = \frac{\sum_{i=1}^{|V|} a_i b_i}{\sqrt{\sum_{i=1}^{|V|} {a_i}^2}\sqrt{\sum_{i=1}^{|V|} {b_i}^2}} \quad (3)$$

TF-IDF の計算式より，特徴量は常に正の値を取るため，コサイン類似度は 0 から 1 の範囲となる．コサイン類似度が閾値以上であれば，それら 2 つのコードブロックはクローンペアであると判定する．閾値は実行時の引数によって与えられる.

4 評価実験

本章では，本研究で提案したブロッククローン検出法の評価実験について述べる．評価実験では，検出精度，検出時間，スケーラビリティの 3 つの観点から，本手法と既存手法の比較を行い評価した．また，保守対象とならないコードクローンと，コードクローンに対する保守作業の調査も行った．最後にブロッククローンの実例を示し，本手法の特徴について考察する.

4.1 節の本手法と既存手法の検出精度の評価では，本手法の拡張元であり検出粒度が異なる関数クローン検出法と，高速かつスケーラビリティの高い字句単位の検出法の中で代表的である CCFinderX [16] を比較対象として選び評価した．4.4 節の検出時間とスケーラビリティの評価では，本手法の有用性を確認するため，上記の 2 つに加えて，抽象構文木を用いた検出法の Deckard [15]，プログラム依存グラフを用いた検出法の Scorpio [9] [10] の 2 つを比較対象として追加して評価した．さらに 4.7 節では，同じコードブロック単位の検出手法として，粗粒度なコードクローン検出法 [13] を対象とした比較実験も行った †4．なおブロッククローンを検出する閾値は，本実験では関数クローン検出法の論文に基づき，0.9 と設定し検出を行った [32].

4.1 節の評価実験においては，検出結果が正しくコードクローンであるか目視で判断するだけでなく，実際に保守対象となり得るかコードクローンの研究者にアンケートを行うことで検出精度を評価した．したがって，単にコード片が外見上類似しているだけでなく，ソフトウェアを保守管理するという観点から実際に有用なコードクローンの評価方法となっている．検出されたコードクローンがどの程度保守の対象になるのかは実際の開発で利用する際に非常に重要となる．既存の研究ではあまり評価されていなかったが，本研究ではこの面についても評価した.

4.1 関数クローン検出法と CCFinderX との比較

本節では関数クローン検出法と CCFinderX の 2 つの既存手法との比較実験について述べる．コードクローンに対する保守作業として，集約や同時修正が挙げられる．そのため，今回は集約または同時修正の保

†3 https://falconn-lib.org/

†4 本実験において，本手法では最小一致トークン数を 30，類似度を 0.9 に，関数クローン検出法では最小一致トークン数を 30，類似度を 0.9 に，CCFinderX では最小一致トークン数を 50，最小トークンタイプ数は 12 に，Deckard では最小一致トークン数を 50，類似度を 0.85，トークンストライドを 2 に，Scorpio では最小一致ノード数を 7 に，粗粒度なコードクローン検出法では最小一致トークン数を 50 と設定した．本手法と関数クローン検出法の各パラメータは既存研究で信頼性が高いと示された値を採用した [32]．CCFinderX，Deckard の各パラメータは既存の実験で実際に用いられた値を採用した [27]．Scorpio のパラメータは既存研究を参考に設定した [10]．粗粒度なコードクローン検出法のパラメータは CCFinderX を参考に設定した.

表 1　検出対象プロジェクト

プロジェクト	バージョン	言語	規模
Apache HTTPD[†5]	2.2.14	C/C++	343 KLOC
PostgreSQL[†6]	8.5.1	C/C++	937 KLOC
Python[†7]	2.5.1	C/C++	435 KLOC

表 2　検出精度の評価

検出手法	検出対象	適合率	再現率	F 値
本手法	Apache HTTPD	**0.90**	**0.74**	**0.81**
	PostgreSQL	0.57	**0.87**	0.69
	Python	**0.90**	0.53	0.67
	合計	**0.68**	**0.70**	**0.69**
関数クローン検出法	Apache HTTPD	0.87	0.53	0.66
	PostgreSQL	**0.83**	0.74	**0.78**
	Python	0.30	0.21	0.25
	合計	0.67	0.47	0.55
CCFinderX	Apache HTTPD	0.70	0.55	0.62
	PostgreSQL	0.13	0.33	0.19
	Python	0.87	**0.63**	**0.73**
	合計	0.57	0.52	0.54

守対象となるコードクローンを検出できるかという観点で検出精度を評価した．本実験では，対象プロジェクトから作成したベンチマークに対する検出精度と検出時間の観点から比較を行う．本実験で検出対象としたプロジェクトの一覧を表 1 に示す．

4.1.1　ベンチマークの作成方法

ベンチマークの作成は以下の 3 ステップで行った．

1. 表 1 のプロジェクトに対し，本手法，関数クローン検出法，CCFinderX の 3 つの手法でコードクローンを検出．
2. 各手法が各プロジェクトから検出したクローンペアから，30 個のクローンペアをランダムサンプリングし，合計 270 個のクローンペア集合を作成．
3. 2 で作成した 270 個のクローンペアに対し，目視で集約または同時修正の保守対象となるコードクローンかの判断を行い，ベンチマークを作成．

なお，ベンチマークに客観性を持たせるため，アンケートにより第三者にコードクローンの判断を依頼した．アンケートの概要を以下に示す．

調査対象　以下の 3 名に依頼

- コードクローンの研究者 1 名
- コードクローンの研究に従事している大学院生 2 名

質問内容　集約または同時修正の保守対象となるか

回答方式　二択 (はい/いいえ)

上記の質問を，検出結果からサンプリングした 270 個のクローンペアに対して行った．そして，過半数である 2 人以上が保守対象と回答したクローンペアを正解とし，本実験で用いる正解クローンペア集合としてベンチマークを作成した．これはアンケートの結果が個人の判断に依存しないようにするためである．本実験では，各プロジェクトの正解クローンペア数は，Apache HTTPD が 74 個，PostgreSQL が 46

†5　http://httpd.apache.org/
†6　http://www.postgresql.org/
†7　http://www.python.org/

個，Python が 62 個となった．

4.1.2 比較結果

本節では，関数クローン検出法と CCFinderX との比較実験の結果について述べる．本実験では，ベンチマークを用いた検出精度と検出時間の観点から比較を行った．検出精度の指標として，適合率，再現率，F 値の 3 つの指標を用いた．本実験における 3 つの指標を表 2 に示す．それぞれの手法と比較して，最も高い値を太字で示している．

適合率

適合率とは，検出結果に対して真に正しかった割合を指し，正確性に関する指標として用いられる．本実験では，各 30 個ずつランダムサンプリングしたクローンペア集合に対して，アンケートにて保守対象となるコードクローンと判断されたクローンペアの割合によって適合率を求めた．今回は過半数である 2 人以上がコードクローンと判断した場合を正解としている．

本手法では，Apache HTTPD と Python において，関数クローン検出法や CCFinderX より高い適合率が得られた．また，PostgreSQL においては，CCFinderX よりは高い適合率が得られたが，関数クローン検出法より低い適合率となった．3 つのすべてのプロジェクトの合計では適合率は 0.68 であり，関数クローン検出法と同程度の適合率，CCFinderX より高い適合率であることが確認できた．

本手法が PostgreSQL において適合率が低くなった原因として，出力処理の連続，同じ識別子の多用，2 文字以下の変数の多用，if 文の連続のクローンが多数あることが確認できた．

再現率

再現率とは，正解集合に対して実際に検出された割合を指し，網羅性に関する指標として用いられる．本実験では，アンケートによって作成したベンチマークの正解集合に対し，各手法が検出したクローンペアの割合によって再現率を求める．

本手法では，Apache HTTPD，PostgreSQL において，関数クローン検出法や CCFinderX より高い再現率が得られた．また，Python においては，関数クローン検出法よりは高い再現率が得られたが，CCFinderX より低い再現率となった．3 つのすべてのプロジェクトの合計では，再現率は 0.70 であり，関数クローン検出法と CCFinderX より再現率が高いことが確認できた．

Python において再現率が低くなった原因として，Python にタイプ 2 のコードクローンが多く含まれることが確認できた (表 5 参照)．表 5 は，検出対象ごとのタイプ別の正解クローン数を示している．本手法は識別子名に基づいて検出を行うため，識別子名の変更を伴うタイプ 2 のコードクローンの検出は不得手である．

F 値

F 値とは，適合率と再現率の総合的な評価として用いられ，適合率と再現率の調和平均によって求められる．

本手法では，Apache HTTPD において F 値が 0.81 であり，関数クローン検出法や CCFinderX より高い F 値が得られた．PostgreSQL においては F 値が 0.69 であり，CCFinderX よりは高い F 値が得られたが，関数クローン検出法より低い F 値となった．また，Python においては F 値が 0.67 であり，関数クローン検出法よりは高い F 値が得られたが，CCFinderX より低い F 値となった．3 つのすべてのプロジェクトの合計の F 値は 0.69 であり，関数クローン検出法と CCFinderX より F 値が高いことが確認できた．

検出時間

検出時間の比較では，表 1 の 3 つのプロジェクトに対する検出時間を測定した．本実験の実行環境は，CPU Intel Xeon 2.80GHz 4core，メモリ 16GB，ハードディスクドライブ，OS Windows 10 64bit である．

検出時間の比較結果を表 3 に示す．本手法では，検出対象すべてのプロジェクトに対して 3 分以下でコードクローンを検出できた．また，関数クローン検出法に対して 3〜4 割程度，CCFinderX に対して 4〜8 割程度と，他の手法よりも短時間で検出することが確認できた．

4.1.3 ベンチマークに含まれたコードクローンのタイプ

本節では，ベンチマークにおいて保守対象と回答されたコードクローンのタイプ別の個数について説明

表 3 検出時間の比較

検出対象	本手法	関数クローン検出法	CCFinderX
Apache HTTPD	1m 39s	4m 7s	2m 1s
PostgreSQL	2m 27s	8m 47s	5m 30s
Python	1m 15s	3m 33s	3m 10s

する．4.1.1 節で実施した，各ツールの検出結果からサンプリングしたコードクローンに対するアンケートにて，保守対象と回答したコードクローンのタイプ (タイプ 1 からタイプ 4) の調査を各回答者に行った．その結果に基づき，各手法が検出した正解クローンのタイプ別の個数と誤検出数を表 4 に示す．今回は，保守対象と回答された検出を正解クローン，保守対象と回答されなかった検出を誤検出とする．なお，回答者によって回答が異なる場合，第 1 著者が最終的な判断を行った．

これより，本手法と関数クローン検出法が実際にタイプ 1 から 4 まで検出可能であることが確認できた．また，CCFinderX はタイプ 1 と 2 が検出可能であり，タイプ 3 と 4 は検出できないことが確認できた．表 4 は各ツールの検出結果から 90 個ずつ抽出した結果の内訳であり，各タイプの検出数は母集団 (各ツールの検出結果) に含まれる絶対数ではないことに注意されたい．例えば，表 4 において，CCFinderX のタイプ 1 より本手法のタイプ 1 の数が多いが，CCFinderX と比べて本手法がより多くのタイプ 1 を検出結果に含んでいることを表していない．

4.2 保守対象と判定されなかったコードクローンの調査

本節では，4.1 節で実施したアンケートにて，ツールによって検出されたが保守対象とならないと回答されたコードクローンについて説明する．今回はアンケートにて，3 人中 1 人以下が保守対象となると回答した検出を誤検出とする．本手法，関数クローン検出法，CCFinderX の 3 つの手法にて誤検出した 88 個のコードクローンを調査し，原因を考察した．その結果を表 6 に示す．今回は複数回出現した原因を手法ごとに掲載している．

本手法では「2 文字以下の変数の多用」，「if 文の連続」，「同じ識別子の多用」，「出力処理の連続」が原因として確認できた．特に「2 文字以下の変数の多用」と「同じ識別子の多用」は，本手法の識別子名に基づいた検出に起因する．本手法では 2 文字以下の変数を同一のメタワードに置換している．そのため，2 文字以下の変数が多用されている場合，同一のワードが頻出していると判断し，誤検出の原因となる．また，一般的によく使用される識別子名の場合，処理内容が異なっていても同じ識別子名が用いられる傾向がある点も，誤検出の原因になる．

関数クローン検出法では「ツールの不具合」が原因として多く確認できた．これは C 言語のマクロの処理が適切に行われていない点に起因する．本手法ではマクロの処理を適切に行えているため，ツールの不具合を原因とした誤検出は今回出現しなかった．また，処理の内容は異なるが，二重の繰り返し構文の構造を内部に持つ関数も誤検出として見られた．

CCFinderX では case 節やコードの断片，if 文や代入文の連続の原因による誤検出の例が目立ち，これらは Higo らの指摘と同様の結果が得られた [12]．本手法ではコードブロック単位での検出を行うため，case 節やコードの断片といった保守作業の対象となりにくい誤検出は今回出現しなかった．

また，太字で表記した「if 文の連続」は 3 つの手法で共通して確認できた．

4.3 コードクローンに対する保守作業の調査

本節では，コードクローンと実際に適用する保守作業の関連性について，本手法，関数クローン検出法，CCFinderX の 3 つの手法との比較を行う．今回は，4.1 節で実施したアンケートにて 3 人中 2 人以上が保守対象となると判定したコードクローンに対し，どのような保守作業を適用できるかについて以下のアンケートを行った．

表 4 ベンチマークに含まれたコードクローンのタイプ別内訳

検出手法	T1	T2	T3	T4	誤検出	合計
本手法	9	34	27	1	19	90
関数クローン検出法	5	29	25	1	30	90
CCFinderX	4	47	0	0	39	90

表 5 ベンチマークに含まれたコードクローンの内訳 (検出対象ごと)

	Apache HTTPD				PostgreSQL				Python			
検出手法	T1	T2	T3	T4	T1	T2	T3	T4	T1	T2	T3	T4
本手法	4	14	8	1	4	10	3	0	1	10	16	0
関数クローン検出法	1	14	11	0	3	12	10	0	1	3	4	1
CCFinderX	2	19	0	0	2	2	0	0	0	26	0	0

表 6 保守対象と判定されなかったコードクローン

本手法	関数クローン検出法	CCFinderX
2 文字以下の変数の多用	**if 文の連続**	case 節の断片
if 文の連続	同じ識別子の多用	**if 文の連続**
出力処理の連続	処理内容が異なる繰り返し構文の入れ子構造	コードの断片
同じ識別子の多用	ツールの不具合	代入文の連続

調査対象 4.1 節で実施したアンケートと同様

質問内容 どのような保守作業の対象となるか

回答方式 三択 (複数回答可)

- 集約
- 同時修正
- その他 (自由回答形式)

結果を手法ごとに分けて図 5 に示す．なお，回答者によって異なる保守作業が適用された場合，両方の保守作業を適用できることとする．また，その他の保守対象と判断されたクローンペアについては，回答者と相談し同時修正の対象に含めた．

本手法では 90 個中 71 個のクローンペアが保守対象と判断された．その内，36 個が同時修正の対象，35 個が集約と同時修正の対象となった．関数クローン検出法では 90 個中 60 個のクローンペアが保守対象と判断された．その内，29 個が同時修正の対象，31 個が集約と同時修正の対象となった．CCFinderX では 90 個中 51 個のクローンペアが保守対象と判断された．その内，35 個が同時修正の対象，16 個が集約と同時修正の対象となった．また，いずれの手法においても集約のみの対象と判断されたクローンペアは 0 個であった．

関数クローン検出法と比較して，保守作業の割合は同じであるが，保守対象となる検出数が増加している．関数単位の検出法では集約の対象となりやすい点が長所として挙げられている．本手法では検出粒度をコードブロック単位に小さくしたが，関数単位の検出と保守作業の割合は変わらず，さらに保守作業の対象となりやすい傾向にあることが確認できた．また CCFinderX と比較して，同時修正の対象数は同程度であるが，集約と同時修正両方の対象数は増加している．字句単位の検出法では集約を行うことが困難なクローンが多く検出される点が指摘されていた．本手法では処理内容にまとまりを持ったコードブロック単位の検出を行うため，字句単位の検出法より集約の対象となりやすい傾向にあることが確認できた．以上より字句単位の検出法より集約の対象となりやすく，また関数単位の検出法より保守作業の対象となりやすいと言える．

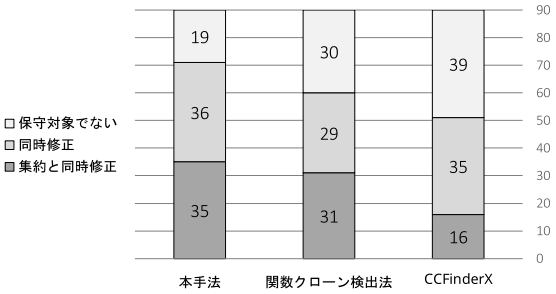


図 5 ベンチマークに含まれたコードクローンに対する保守作業

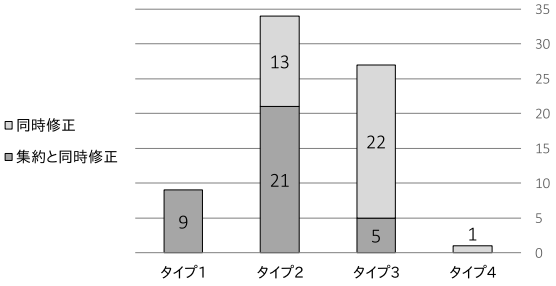


図 6 本手法のタイプ別の保守作業

さらに，本手法においてタイプ別に適用される保守作業の調査を行った．結果を図 6 に示す．これより，タイプ 1 は集約と同時修正のどちらの対象にもなりやすく，タイプが上がるにつれて集約の対象にはなりにくいことが分かった．

4.4 検出時間とスケーラビリティの評価

本節では，提案手法の検出規模ごとの検出時間とスケーラビリティの評価について述べる．検出対象の規模の指標にはコメントや空行を除いた LOC を用いることとした．LOC の計測には cloc †8 を用いた．検出対象は，IJaDataset 2.0 †9 からランダムにファイルを選択し，表 7 に示した LOC ごとのサブセットシステムを作成した．IJaDataset とは，2 万以上の Java プロジェクトから成る，365MLOC のビッグデータセットである．本実験の実行環境は，CPU Intel Xeon 2.80GHz 4core，メモリ 32GB，ソリッドステートドライブ，OS Windows 10 64bit である．また，Java 仮想マシンのスタック領域を 1GB，ヒープ領域を 15GB と設定して実行した．

検出時間の評価結果を表 7 に示す．これにより，10MLOC 規模の検出対象に対して CCFinderX は検出に約 2 時間かかり，関数クローン検出法ではメモリ不足で終了したのに対し，本手法では約 30 分で検出可能であることが確認できた．また，1KLOC から 10MLOC の検出規模に応じて線形的に検出時間が増加することも確認できた．

表 7 より，本手法は関数クローン検出法や CCFinderX より高速に検出できることが確認できた．また，10MLOC 規模のプロジェクトに対して約 30 分で検出できることも確認できた．特徴ベクトルの次元を d，特徴ベクトル集合の大きさを n とした時，クローン検出にかかる時間は $O(dn^2)$ である．しかし，LSH を用いてクラスタリングを行うことで，クローン検出時間は $O(dn^{\rho+1})$ となる．ρ は LSH の鋭敏性を示す値で，LSH に適切なパラメータを与えることで，$\rho \leq 0.2$ に抑えることができる [2]．よって $O(dn^{\rho+1}) < O(dn^2)$ となり，時間計算量は少なくなる．関数クローン検出法と比較して本手法は検出粒度を下げているため n が大きくなるが，LSH を用いることで検出にかかる時間計算量の増加を抑えている．さらに，LSH の実装に用いた FALCONN は高速に LSH を計算できるように実装されたツールであり，クラスタリングを行うツールの変更が検出の高速化の主な要因であると考えられる．また，特徴ベクトルを表現するデータ構造を疎ベクトルとして実装したことにより入出力時間の短縮につながった．入出力時間が検出全体で占める割合も大きいため，特徴ベクトルのデータ構造の変更も検出の高速化に貢献している．

また表 7 より，検出規模に応じて線形的に検出時間が増加することも確認できた．これに関しては，本手法や関数クローン検出法ではコードブロック，関数の抽出，CCFinderX では字句の抽出にかかる時間の割合が全ステップ中で多いため，検出規模に応じて線形的に時間が増加すると考えられる．CCFinderX は字句の抽出にかかる時間が長いため，本手法の方が高速に検出することができた．

さらに，関数クローン検出法では 10MLOC の検出中にヒープ領域不足で検出を終了した．ヒープ領域不

†8 http://cloc.sourceforge.net
†9 http://secold.org/projects/seclone

表 7 検出規模ごとの検出時間

LOC	本手法	CCFinderX	関数クローン検出法
1K	1s	4s	11s
10K	2s	8s	15s
100K	16s	1m 18s	2m 25s
1M	3m 1s	12m 24s	20m 4s
10M	29m 9s	2h 4m 55s	メモリ不足で停止

足は特徴ベクトルの生成中に起こった．これは，特徴ベクトルを疎ベクトルとして実装していないことによる，メモリ使用効率の悪さが原因として考えられる．

さらに，抽象構文木を用いた検出法の Deckard と，我々の知る限り，プログラム依存グラフに基づくクローン検出ツールの中で唯一公開されている Scorpio の 2 つの手法を 10MLOC の検出規模で実験した．その結果，7 日間経過しても検出が完了しなかった．そのため現実的な時間内での検出は完了しないとして実験を終了した．このことより，抽象構文木を用いた検出法やプログラム依存グラフを用いた検出法と比較しても，本手法はスケーラビリティの観点から優位性があると言える．

4.5 ブロッククローンの実例

本節では，4.1 節で実施したアンケートにて保守対象のコードクローンと回答されたクローンペアの中から，本手法が検出したブロッククローンの実例を示す．破線で囲まれたコード片がブロッククローンを表している．

図 7 は同じ関数内に存在するタイプ 1 のブロッククローンである．関数クローン検出法では関数単位の検出のため，同じ関数内でコピーアンドペーストを行うことで発生したコードクローンを検出できなかった．

図 8 は，図 8 (a) の 708 行目と 712 行目に文の挿入が行われたタイプ 3 のブロッククローンである．関数単位では，図 8 (a) の 697 から 699 行目，719 から 723 行目のコード片に対応する部分が図 8 (b) にない．よって，関数単位では類似していないため関数クローン検出法では検出できなかった．

図 9 は，ファイルの出力処理を行うタイプ 4 のブロッククローンである．どちらもファイルの入出力関連の処理を行う関数だが，図 9 (b) の 364 から 366 行目にて個別の処理を行うため，関数クローン検出法では検出できなかった．

また，字句単位の検出である CCFinderX では，図 7 のコードクローンを検出することができる．図 8 は文の挿入が行われていない部分の検出はできるが，それではコード断片のクローンとなってしまう．よって集約などの保守対象となりづらい．図 9 はタイプ 4 のクローンであり，CCFinderX はタイプ 2 のクローンまでしか対応していないため検出することができない．さらに，Deckard では，図 7 のコードクローンを検出することができるが，図 8，図 9 は検出することができなかった．なお，唯一公開されているプログラム依存グラフに基づく検出ツールである Scorpio は，C 言語に対応していないため適用できなかった．

ブロッククローンの実例より，関数単位より検出粒度を小さくすることで，関数単位では検出できないが本手法で検出できるクローンを示すことができた．また，コードブロック単位でまとまっているため，集約などの保守対象となりやすいことも実例から確認できた．また，実際に本手法はタイプ 1 から 4 の検出に対応しており，さらに他の手法では検出できなかったコードクローンの検出も可能であった．以上の観点から本手法の有用性が確認できた．

4.6 関数クローン検出法と CCFinderX と比較したときの本手法の特徴

評価実験の結果から，本手法が関数クローン検出法や CCFinderX より優れた点は以下の点であると考え

```
248: APU_DECLARE(apr_status_t) apr_rmm_destroy(apr_rmm_t *rmm)
249: {
250:     apr_status_t rv;
251:     rmm_block_t *blk;
252:
253:     if ((rv = APR_ANYLOCK_LOCK(&rmm->lock)) != APR_SUCCESS) {
254:         return rv;
255:     }
256:     /* Blast it all --- no going back :) */
257:     if (rmm->base->firstused) {
258:         apr_rmm_off_t this = rmm->base->firstused;
259:         do {
260:             blk = (rmm_block_t *)((char*)rmm->base + this);
261:             this = blk->next;
262:             blk->next = blk->prev = 0;
263:         } while (this);
264:         rmm->base->firstused = 0;
265:     }
266:     if (rmm->base->firstfree) {
267:         apr_rmm_off_t this = rmm->base->firstfree;
268:         do {
269:             blk = (rmm_block_t *)((char*)rmm->base + this);
270:             this = blk->next;
271:             blk->next = blk->prev = 0;
272:         } while (this);
273:         rmm->base->firstfree = 0;
274:     }
275:     rmm->base->abssize = 0;
276:     rmm->size = 0;
277:
278:     return APR_ANYLOCK_UNLOCK(&rmm->lock);
279: }
```

Apache HTTPD: /srclib/apr-util/misc/apr_rmm.c

図 7　同じ関数内に存在するブロッククローン (タイプ 1)

られる．

- 保守対象となりにくいブロックの一部をコードクローンとして検出することがなく，また関数よりも小さい単位の検出が可能であるため，字句単位の手法や関数クローン検出手法と比べて，保守 (集約や同時修正) 対象のコードクローンを多く検出可能．
- 字句単位の手法や関数クローン検出手法だけでなく，抽象構文木やプログラム依存グラフを用いた手法と比べてもスケーラビリティが高いことから，大規模なソースコード集合に対して適用可能．

これらの点から，大規模なソースコード集合に対して，保守対象のコードクローンを検出する場合に最も適していると考えられる．

一方で評価実験の結果から，短い変数名が多く出現するなど，変数名に処理内容が反映されてないソースコードに対して本手法は不向きであり，そのようなソースコードに対しては他の手法の利用を検討すべきである．また，時間が十分にあり，かつ 100 万行未満のソースコード集合が対象であれば，本手法に加えて抽象構文木やプログラム依存グラフを用いた手法の利用も合わせて検討すべきである．

```
690: strop_swapcase(PyObject *self, PyObject *args)
691: {
692:         char *s, *s_new;
693:         Py_ssize_t i, n;
694:         PyObject *newstr;
695:         int changed;
696:
697:         WARN;
698:         if (PyString_AsStringAndSize(args, &s, &n))
699:                         return NULL;
700:         newstr = PyString_FromStringAndSize(NULL, n);
701:         if (newstr == NULL)
702:                         return NULL;
703:         s_new = PyString_AsString(newstr);
704:         changed = 0;
705:         for (i = 0; i < n; i++) {
706:                         int c = Py_CHARMASK(*s++);
707:                         if (islower(c)) {
708:                                         changed = 1;
709:                                         *s_new = toupper(c);
710:                         }
711:                         else if (isupper(c)) {
712:                                         changed = 1;
713:                                         *s_new = tolower(c);
714:                         }
715:                         else
716:                                         *s_new = c;
717:                         s_new++;
718:         }
719:         if (!changed) {
720:                         Py_DECREF(newstr);
721:                         Py_INCREF(args);
722:                         return args;
723:         }
724:         return newstr;
725: }
```

(a) Python: /Modules/stropmodule.c

```
2308: string_swapcase(PyStringObject *self)
2309: {
2310:         char *s = PyString_AS_STRING(self), *s_new;
2311:         Py_ssize_t i, n = PyString_GET_SIZE(self);
2312:         PyObject *newobj;
2313:
2314:         newobj = PyString_FromStringAndSize(NULL, n);
2315:         if (newobj == NULL)
2316:                         return NULL;
2317:         s_new = PyString_AsString(newobj);
2318:         for (i = 0; i < n; i++) {
2319:                         int c = Py_CHARMASK(*s++);
2320:                         if (islower(c)) {
2321:                                         *s_new = toupper(c);
2322:                         }
2323:                         else if (isupper(c)) {
2324:                                         *s_new = tolower(c);
2325:                         }
2326:                         else
2327:                                         *s_new = c;
2328:                         s_new++;
2329:         }
2330:         return newobj;
2331: }
```

(b) Python: /Objects/stringobject.c

図 8　文の挿入が行われたブロッククローン (タイプ 3)

4.7　粗粒度なコードクローン検出法と比較したときの本手法の特徴

同じコードブロック単位の検出手法として，粗粒度なコードクローン検出法 [13] を対象として比較実験を行った．粗粒度なコードクローン検出法とは，タイプ 1 と 2 のブロック単位のコードクローンを検出する手法である．本節ではその実験結果について述べる．

最初に，粗粒度なコードクローン検出法の検出精度を比較した．本実験では，4.1.1 節で作成したベンチマークを用いて適合率，再現率，F 値を求めた．ここ

```
334: APR_DECLARE(apr_status_t) apr_file_flush(apr_file_t *thefile)
335: {
336:     apr_status_t rv = APR_SUCCESS;
337:
338:     if (thefile->buffered) {
339:         file_lock(thefile);
340:         rv = apr_file_flush_locked(thefile);
341:         file_unlock(thefile);
342:     }
343:     /*
344:      * (コメント省略)
345:      */
346:     return rv;
347: }
```

(a) Apache HTTPD:
/srclib/apr/file_io/unix/readwrite.c

```
349: APR_DECLARE(apr_status_t) apr_file_sync(apr_file_t *thefile)
350: {
351:     apr_status_t rv = APR_SUCCESS;
352:
353:     file_lock(thefile);
354:
355:     if (thefile->buffered) {
356:         rv = apr_file_flush_locked(thefile);
357:
358:         if (rv != APR_SUCCESS) {
359:             file_unlock(thefile);
360:             return rv;
361:         }
362:     }
363:
364:     if (fsync(thefile->filedes)) {
365:         rv = apr_get_os_error();
366:     }
367:
368:     file_unlock(thefile);
369:
370:     return rv;
371: }
```

(b) Apache HTTPD:
/srclib/apr/file_io/unix/readwrite.c

図 9 ファイルの出力処理を行うブロッククローン (タイプ 4)

では，ベンチマークにて保守対象と判定されたクローンペアを正解クローン，判定されなかったクローンペアを誤検出と定義する．また，粗粒度なコードクローン検出法が検出したクローンペアを検出クローンと定義する．適合率は，検出クローンの中から，正解クローンと一致した集合と誤検出と一致した集合の和集合の内，正解クローンと一致した集合の割合によって求める．再現率は，正解クローン集合の内，検出クローンの中で正解クローンと一致した集合の割合によって求める．F 値は適合率と再現率の調和平均によって求める．実験の結果，粗粒度なコードクローン検出法の検出精度は表 8 のとおりとなった．粗粒度なクローン検出法はタイプ 1 と 2 のコードクローンしか検出しないため，保守対象となりやすく適合率は高い値となった．しかし，タイプ 3 や 4 を検出することができず検出数が少ないため再現率は低い値となり，総合的な評価の F 値は低い値となることが確認できた．

また，粗粒度なコードクローン検出法の検出時間も求めた．実験の結果を表 9 に示す．これにより粗粒度なコードクローン検出法は高速に検出ができることが確認できた．粗粒度なコードクローン検出法は，各コードブロックに対して，正規化後 (変数名の置換など) の文字列からハッシュ値を求め，同じハッシュ値を持つコードブロックをコードクローンとして検出する．ハッシュ値を用いた比較を行うことで，文字列を用いた比較を行う場合と比べて，小さいコストで 2 つのブロックの比較を行うことができる [13]．

次に，粗粒度なコードクローン検出法が検出したコードクローンのタイプ別の個数について調査した．本実験では，CCFinderX が検出したクローンペア集合に対して，4.1.3 節のアンケートにより得られたコードクローンのタイプ情報を基に，タイプ別の個数を検出対象ごとに算出した．結果を表 10 に示す．これにより，粗粒度なコードクローン検出法はタイプ 2 のクローンのみ検出することが分かった．検出法の理論上ではタイプ 1 のコードクローンも検出可能であるが，今回はタイプ 1 のコードクローンは確認できなかった．これは，元々検出対象となっている CCFinderX が検出したコードクローンにタイプ 1 のコードクローンが少なかったことが原因として挙げられる．

さらに，粗粒度なコードクローン検出法の検出規模ごとの検出時間とスケーラビリティについても述べる．本実験では 4.4 節と同一のデータセットを用いて実験を行った．実験の結果を表 11 に示す．これにより，検出規模に比例して線形的に検出時間が増加することも確認できた．これは，検出時間においてファイル読み込みとコードブロックの抽出に最も時間を占めていることが原因として挙げられる．

最後に，粗粒度なコードクローン検出法の検出したコードクローンに対する保守作業の調査を行った．本実験では，CCFinderX が検出したクローンペア集合に対して，4.3 節のアンケートにより得られた保守作業の調査結果をもとに，実際に適用される保守作業の割合と，保守対象と判定されなかった原因を求めた．実際に適用される保守作業は，同時修正の対象と

表 8　粗粒度なコードクローン検出法の検出精度の評価

検出手法	検出対象	適合率	再現率	F 値
粗粒度クローン検出法	Apache HTTPD	**0.91**	0.14	0.24
	PostgreSQL	**0.87**	0.28	0.43
	Python	**0.95**	0.32	0.48
	合計	**0.91**	0.24	0.38
本手法	Apache HTTPD	0.90	**0.74**	**0.81**
	PostgreSQL	0.57	**0.87**	**0.69**
	Python	0.90	**0.53**	**0.67**
	合計	0.68	**0.70**	**0.69**

表 9　粗粒度なコードクローン検出法の検出時間

検出対象	粗粒度クローン検出法	本手法
Apache HTTPD	8s	1m 39s
PostgreSQL	16s	2m 27s
Python	8s	1m 15s

表 10　粗粒度なコードクローン検出法の検出したコードクローンのタイプ別内訳

検出対象	T1	T2	T1, T2 合計
Apache HTTPD	0	4	4
PostgreSQL	0	1	1
Python	0	13	13
合計	0	18	18

表 11　粗粒度なコードクローン検出法の検出規模ごとの検出時間

LOC	粗粒度クローン検出法	本手法
1K	0s	1s
10K	1s	2s
100K	8s	16s
1M	1m 15s	3m 1s
10M	12m 38s	29m 9s

なるクローンペアが 15，集約と同時修正の対象となるクローンペアが 3 となった．また，保守対象と判定されなかった原因として，出力処理の連続，代入文の連続，単調な関数呼び出しの 3 例が挙げられた．

以上の結果から，タイプ 1 や 2 だけでなく，3 と 4 のコードクローンも検出可能で網羅性が高く，検出の正確性と網羅性の総合評価でも高い点において本手法の方が優れている．ただし，粗粒度なコードクローン検出法はコードブロック単位の検出を行うため，トークン単位の検出法である CCFinderX で確認されたようなコード片の断片を検出することはなく，検出の正確性が高くなっている．また，タイプ 1 と 2

のコードクローンのみを検出するため，高速な検出が可能となっている．今回の評価実験から，本手法の補完的な検出方法として粗粒度なコードクローン検出法の利用も検討できる．

5 考察

5.1 本手法の拡張性

本手法の実装は，現在 C 言語と Java 言語にのみ対応している．しかし，本手法では ANTLR を用いて構文解析を行っており，ANTLR にて構文解析を行うための文法ファイルが 100 種類以上用意されていることから，他の言語への拡張が容易に可能である．

5.2 評価実験の妥当性

適合率，再現率，F 値で示される検出精度に関して，本実験では 3 つの C 言語のプロジェクトに対して関数クローン検出法と CCFinderX との比較を行うことによって，本手法の有用性を示した．しかし，今後は他の言語で実装されたプロジェクトに対して適用したり，他のツールとの比較を行ったりするなど，一般性を示す必要性がある．

また，本実験で用いたパラメータの値によって実験結果は変わってくる．特に CCFinderX の実験結果は最小一致トークン数の値が影響している可能性があるが，最小一致トークン数を下げると再現率は上昇するが適合率が低下する．

特徴ベクトルの計算方法として本手法では TF-IDF を用いているが，他にも LSI (Latent Semantic Indexing) [3] や，LDA (Latent Dirichlet Allocation) [5] など次元圧縮を行い計算時間の短縮を行った手法がある．またワードの共起頻度に基づいて次元圧縮を行うことで，ワードの類義性も計算可能となる．これらと TF-IDF を比較することで，検出速度と検出精度の観点から比較を行う必要がある．

そして，LSI や LDA を用いて特徴ベクトルを計算する場合，特徴ベクトルの生成や類似度の計算にかかる時間が変化する．たとえば，次元圧縮を行うため特徴ベクトル生成に時間が増加するが，次元削減により類似度計算の時間が減少するなどが考えられる．したがって，LSI や LDA などを用いた場合に各ステップにかかる時間を詳細に調査する必要がある．

また本研究では我々が作成したコードクローンベンチマークを対象に検出精度の評価実験を行ったが，他にも大規模なベンチマーク [29] も提案されている．このような他のベンチマークに対しても本手法の有用性が確認できるのか評価する必要がある．さらに今回比較を行っていない手法との比較も行い，本手法の検出結果の特徴についてより詳細に分析する必要もある．

6 関連研究

構文の類似性に着目した手法として，字句単位の検出手法や，抽象構文木 (ソースコードの構文構造を木構造で表したグラフ) を用いた検出手法が存在する．字句単位の検出手法では，ソースコードをトークン列に変換し，共通トークン列をコードクローンとして検出する [16] [20]．また，抽象構文木を用いた検出手法では，ソースコードを抽象構文木に変換し，類似した部分木をコードクローンとして検出する [15] [4]．本研究では，提案手法と字句単位の検出ツールである CCFinderX と比較し，提案手法が適合率や再現率，スケーラビリティにおいて優れていることを確認した．また，スケーラビリティの実験において，抽象構文木を用いた検出ツールである Deckard が 10MLOC のソースコード集合を対象とした実験において現実的な時間で検出処理を終えることができなかったこと，および Deckard が検出できないが提案手法が検出できるコードクローンの実例を確認した．

また，プログラムの処理の類似性に着目した手法として，プログラム依存グラフ (プログラム内の要素間に存在する依存関係を表した有効グラフ) を用いた検出手法が存在する．この手法では，ソースコードからプログラム依存グラフを構築し，類似した部分グラフを探索することによって，タイプ 3 や 4 のコードクローンを検出することが可能である [9] [10] [19]．しかし，プログラム依存グラフの比較の計算コストが高く，検出に時間がかかってしまうという問題点がある．また，プログラム言語ごとにプログラム依存グラフを構築する機構を用意する必要性がある．本研究では，プログラム依存グラフを用いた検出ツールである Scorpio [9] [10] が 10MLOC のソースコード集合を

対象とした実験において現実的な時間で検出処理を終えることができなかったことを確認した．

LCS (Longest Common Subsequence) アルゴリズムを利用した検出手法として NiCad が存在する [24] [25]．NiCad は，ソースコードの各行を要素とする列に対して LCS アルゴリズムを適用して，固有の行の割合が閾値以下であればコードクローンとして検出する．閾値未満であれば，固有の行を許容するため，タイプ 3 のコードクローンを検出することができる．NiCad は文献により実装方法に差異があり，またチューニングにより検出速度が大きく異なる [7] [24] [25]．今後，本研究が提案する手法と比較を行う必要があるが，各実装に応じたチューニングを適切に行いながら，比較実験を行う必要があると考えられる．

山中らのツール [32] 以外の関数クローン検出ツールとして MeCC [17] がある．MeCC は，記号実行を行うことによって，ソースコード中の各関数が終了した時点における抽象的なメモリの状態の予測を行う．そして，メモリの状態が類似した関数をコードクローンとして検出する．山中らの論文 [32] において，山中らの関数クローン検出ツールの方が MeCC より正確に関数クローンを検出できることが確認されている．本論文の実験では，山中らの関数クローン検出ツールと比較して，提案手法の方が適合率や再現率，スケーラビリティにおいて優れている点が多いことを示した．

Marcus らは，クエリとして与えられたソースファイルと類似した部分を，LSI (Latent Semantic Indexing) [3] を用いてソースコード全体から検索する手法を提案している [22]．彼らが提案する手法はクエリを与える必要があるため，本研究の提案手法とは目的が異なる．しかし，提案手法においても LSI を利用し識別子間の潜在的意味を解析することで，再現性を向上させることができる可能性がある．

我々の研究グループでは，トークン列が等価なファイルを高速に検出するツール FCFinder の開発を行った [28]．本研究では，識別子や予約語の類似性に着目し，ブロック単位のコードクローンを検出する手法を提案した．本研究が提案する手法は，2 つのブロックに含まれるトークン列が異なっていても，識別子や予約語の集合が類似していれば，それらブロックはブロッククローンとして検出される．

Duala-Ekoko らは，クローン領域を抽象的に記述し，バージョン間でのクローンの移動を追跡し管理する手法を提案している [8]．ソフトウェアの開発過程においてコードクローンの変更管理を行う場合，ソースコードの編集により行情報は変更されるため，従来の行情報を用いたクローン領域の記述方法ではコードクローンを追跡できない．そこで彼らは，クローン領域を抽象的に記述する手法を提案した．彼らの提案手法はコードクローン追跡のための手法であり，コードクローン検出を目的とする本研究とは目的が異なる．彼らの手法では，構文，構造，語彙の情報を組み合わせてクローン領域を抽象的に記述することでクローン追跡を可能にしている．しかし，本手法はクローン追跡の必要性がないためファイルと行の情報で記述されたクローン領域をコードブロックとして扱っている．

7 まとめと今後の課題

本研究では，TF-IDF と LSH を利用したブロッククローン検出手法の提案を行った．本手法では，構文解析を行いコードブロックの抽出を行い，コードブロック中の識別子や予約語に利用されている単語からワードを抽出する．そして，TF-IDF を利用して各ワードに対する重みを計算し，その重みを特徴量として各コードブロックを特徴ベクトルに変換する．その後，特徴ベクトル間の類似度を計算することによって，意味的に処理が類似したブロッククローンの検出を行う．また，multi-probe LSH を cross-polytope LSH に組み合わせた手法を用いてあらかじめ特徴ベクトルのクラスタリングを行うことによって，高速なブロッククローンの検出を実現した．

評価実験では，3 つの C プロジェクトに対し，検出精度と検出時間の観点から，関数クローン検出法と CCFinderX の 2 つの手法と比較を行った．比較した結果，本手法が高い精度かつ高速にコードクローンを検出することが確認できた．また，スケーラビリティの評価では，1KLOC から 10MLOC の検出規模に応じて，線形的に検出時間が増加することを確認できた．

今後の課題として，5.2 節にて述べたように以下が

挙げられる．

- 今回は特徴ベクトルの計算に TF-IDF を用いたが，LSI (Latent Semantic Indexing) [3] や，LDA (Latent Dirichlet Allocation) [5] といった次元圧縮を行う手法がある．これら次元圧縮を用いた手法との比較を行う必要がある．
- LSI や LDA などを用いた場合，検出の各ステップにかかる時間を詳細に調査する必要がある．
- 他の大規模プロジェクトに対して適用し，本手法の有用性を評価する必要がある．さらに，他の検出手法との比較を行う必要がある．

謝辞 本研究に対して，様々なご協力をいただいた日本電気株式会社前田直人氏，渋谷健介氏，大阪大学石津卓也氏，沼田聖也氏 (現新日鉄住金ソリューションズ株式会社)，徳井翔梧氏に深く感謝する．本研究は JSPS 科研費 JP25220003，JP18H04094，JP16K16034 の助成を受けた．

参 考 文 献

[1] Andoni, A. and Indyk, P.: Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions, *Proc. of the FOCS*, 2006, pp. 459–468.

[2] Andoni, A., Indyk, P., Laarhoven, T., Razenshteyn, I., and Schmidt, L.: Practical and Optimal LSH for Angular Distance, *Proc. of NIPS*, 2015, pp. 1225–1233.

[3] Baeza-Yates, R. and Ribeiro-Neto, B.: *Modern information retrieval: The concepts and technology behind search*, Addison-Wesley, 2011.

[4] Baxter, I. D., Yahin, A., Moura, L., Anna, M. S., and Bier, L.: Clone detection using abstract syntax trees, *Proc. of ICSM*, 1998, pp. 368–377.

[5] Blei, D. M., Ng, A. Y., and Jordan, M. I.: Latent dirichlet allocation, *Journal of machine Learning research*, Vol. 3, No. Jan (2003), pp. 993–1022.

[6] Charikar, M. S.: Similarity estimation techniques from rounding algorithms, *Proc. of STOC*, 2002, pp. 380–388.

[7] Cordy, J. R. and Roy, C. K.: Tuning Research Tools for Scalability and Performance: The NiCad Experience, *Science of Computer Programming*, Vol. 79 (2014), pp. 158–171.

[8] Duala-Ekoko, E. and Robillard, M. P.: Clone Region Descriptors: Representing and Tracking Duplication in Source Code, *ACM Transactions on Software Engineering and Methodology*, Vol. 20, No. 1 (2010), pp. 3:1–3:31.

[9] 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 (2010), pp. 2149–2168.

[10] Higo, Y. and Kusumoto, S.: Enhancing Quality of Code Clone Detection with Program Dependency Graph, *Proc. of WCRE*, 2009, pp. 315–316.

[11] 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, Vol. J91-D, No. 6 (2008), pp. 1465–1481.

[12] 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 (2007), pp. 985–998.

[13] 堀田圭佑, 楊嘉晨, 肥後芳樹, 楠本真二: 粗粒度なコードクローン検出手法の精度に関する調査, 情報処理学会論文誌, Vol. 56, No. 2 (2015), pp. 580–592.

[14] Indyk, P. and Motwani, R.: Approximate nearest neighbors: towards removing the curse of dimensionality, *Proc. of STOC*, 1998, pp. 604–613.

[15] Jiang, L., Misherghi, G., Su, Z., and Glondu, S.: DECKARD: Scalable and Accurate Tree-Based Detection of Code Clones, *Proc. of ICSE*, 2007, pp. 96–105.

[16] Kamiya, T., Kusumoto, S., and Inoue, K.: CCFinder: a multilinguistic token-based code clone detection system for large scale source code, *IEEE Transactions on Software Engineering*, Vol. 28, No. 7 (2002), pp. 654–670.

[17] Kim, H., Jung, Y., Kim, S., and Yi, K.: MeCC: memory comparison-based clone detector, *Proc. of ICSE*, 2011, pp. 301–310.

[18] 古賀久志: ハッシュを用いた類似検索技術とその応用, 電子情報通信学会基礎・境界ソサイエティ *Fundamentals Review*, Vol. 7, No. 3 (2014), pp. 256–268.

[19] Komondoor, R. and Horwitz, S.: Using slicing to identify duplication in source code, *Proc. of SAS*, 2001, pp. 40–56.

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

[21] Lv, Q., Josephson, W., Wang, Z., Charikar, M., and Li, K.: Multi-probe LSH: efficient indexing for high-dimensional similarity search, *Proc. of VLDB*, 2007, pp. 950–961.

[22] Marcus, A. and Maletic, J. I.: Identification of high-level concept clones in source code, *Proc. of ASE*, 2001, pp. 107–114.

[23] Rattan, D., Bhatia, R., and Singh, M.: Software clone detection: A systematic review, *Information and Software Technology*, Vol. 55, No. 7 (2013), pp. 1165–1199.

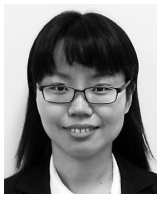
[24] Roy, C. K. and Cordy, J. R.: An Empirical Study of Function Clones in Open Source Software, *Proc. of WCRE*, 2008, pp. 81–90.

[25] Roy, C. K. and Cordy, J. R.: NICAD: Accurate Detection of Near-Miss Intentional Clones Us-

ing Flexible Pretty-Printing and Code Normalization, *Proc. of ICPC*, 2008, pp. 172–181.
[26] Roy, C. K., Cordy, J. R., and Koschke, R.: Comparison and evaluation of code clone detection techniques and tools: A qualitative approach, *Science of Computer Programming*, Vol. 74, No. 7 (2009), pp. 470–495.
[27] Sajnani, H., Saini, V., Svajlenko, J., Roy, C. K., and Lopes, C. V.: SourcererCC: Scaling code clone detection to big-code, *Proc. of ICSE*, 2016, pp. 1157–1168.
[28] 佐々木裕介, 山本哲男, 早瀬康裕, 井上克郎: 大規模ソフトウェアシステムを対象としたファイルクローンの検出, 電子情報通信学会論文誌, Vol. J94-D, No. 8 (2011), pp. 1423–1433.
[29] Svajlenko, J., Islam, J. F., Keivanloo, I., Roy, C. K., and Mia, M. M.: Towards a Big Data Curated Benchmark of Inter-project Code Clones, *Proc. of ICSME*, 2014, pp. 476–480.
[30] Terasawa, K. and Tanaka, Y.: Spherical LSH for Approximate Nearest Neighbor Search on Unit Hypersphere, *Proc. of WADS*, 2007, pp. 27–38.
[31] 徳井翔梧, 吉田則裕, 崔恩瀞, 井上克郎: 局所性鋭敏型ハッシュを用いたコードクローン検出のためのパラメータ決定手法, 電子情報通信学会技術研究報告, Vol. 117, No. 477, 2018, pp. 57–62.
[32] 山中裕樹, 崔恩瀞, 吉田則裕, 井上克郎: 情報検索技術に基づく高速な関数クローン検出, 情報処理学会論文誌, Vol. 55, No. 10 (2014), pp. 2245–2255.
[33] Yamanaka, Y., Choi, E., Yoshida, N., Inoue, K., and Sano, T.: Applying clone change notification system into an industrial development process, *Proc. of ICPC*, 2013, pp. 199–206.

横 井 一 輝

2017 年大阪大学基礎工学部情報科学科卒業．現在，大阪大学大学院情報科学研究科博士前期課程 2 年．コードクローン検出手法の研究に従事．



崔　恩瀞

2015 年大阪大学大学院情報科学研究科博士後期課程修了．同年同大学大学院国際公共政策研究科助教．2016 年奈良先端科学技術大学院大学情報科学研究科助教．2018 年より同大学先端科学技術研究科助教 (改組による)．博士 (情報科学)．コードクローン管理やリファクタリング支援手法に関する研究に従事．

吉 田 則 裕

2004 年九州工業大学情報工学部知能情報工学科卒業．2009 年大阪大学大学院情報科学研究科博士後期課程修了．同年日本学術振興会特別研究員 (PD)．2010 年奈良先端科学技術大学院大学情報科学研究科助教．2014 年名古屋大学大学院情報科学研究科附属組込みシステム研究センター准教授．2017 年より同大学大学院情報学研究科附属組込みシステム研究センター准教授 (改組による)．博士 (情報科学)．コードクローン分析手法やリファクタリング支援手法に関する研究に従事．

井 上 克 郎

1984 年大阪大学大学院基礎工学研究科博士後期課程修了 (工学博士)．同年大阪大学基礎工学部情報工学科助手．1984 年〜1986 年，ハワイ大学マノア校コンピュータサイエンス学科助教授．1991 年大阪大学基礎工学部助教授．1995 年同学部教授．2002 年より大阪大学大学院情報科学研究科教授．ソフトウェア工学，特にコードクローンやコード検索などのプログラム分析や再利用技術の研究に従事．