Vol.2017-SE-196 No.19
2017/7/20

情報検索技術に基づくブロッククローン検出

横井 一輝[1,a)]　崔 恩瀞[2]　吉田 則裕[3]　井上 克郎[1]

概要：ソフトウェア保守における問題の 1 つとしてコードクローン（ソースコード中に存在する同一または類似した部分を持つコード片）が指摘されている．既存の研究において，情報検索技術に基づいて関数単位でコードクローンを検出する手法が提案されたが，検出粒度が大きいため検出漏れを起こすという問題点がある．そこで本研究では，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．これにより，従来の関数単位のコードクローンに加えてより粒度の小さいコードクローンが検出できると考えられる．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，ブロッククローンの検出を行う．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．

キーワード：コードクローン，ソフトウェア保守，情報検索

Block Clone Detection Based on Information Retrieval Techniques

Kazuki Yokoi[1,a)]　Eunjong Choi[2]　Norihiro Yoshida[3]　Katsuro Inoue[1]

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 [3]．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 [5], [11]．

山中らは情報検索技術 [2] 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（関数クローン）を検出する手法を提案した [13]．情報検索技術とは，大量のデータ群から目的に合致したものを取り出すことであり，自然言語で書かれた文書の処理などに利用される [2]．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 [14]．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 法 [2] 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）アルゴリズム [6] 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，

[1] 大阪大学
Osaka University
[2] 奈良先端科学技術大学院大学
Nara Institute of Science and Technology
[3] 名古屋大学
Nagoya University
[a)] k-yokoi@ist.osaka-u.ac.jp

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．ここではコードブロックを，関数と，関数内部の if，for，while 文等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 法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，Multi-Probe LSH [9] を使用して特徴ベクトルのクラスタリングを行う．Multi-Probe LSH とは，従来の LSH のメモリ使用量が多いという問題点を改良したアルゴリズムである．（以降，単に“LSH”と表記した場合は LSH アルゴリズム全般を指し，LSH アルゴリズムを区別する際は“Multi-Probe LSH”，“従来の LSH”という表記を用いる）このことで，より少ないメモリ使用量で高速な検出を実現した．

評価実験では，関数クローン検出法 [13] と CCFinder [7] の 2 つと検出精度と検出時間の観点から比較を行った．CCFinder は神谷らが開発したコードクローン検出ツールであり，字句単位のコードクローン検出が可能である．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，本手法の検出にかかる時間は 3 分以下となり，関数クローン検出法と CCFinder よりも高速にコードクローンを検出することができた．また，検出時間とスケーラビリティの評価も行い，100MLOC の大規模プロジェクトに対して 4 時間程度で検出できることを確認した．

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．最後に，5 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 [12]．

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 のコードクローンとして，以下のものが挙げられる．

- 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

山中らは情報検索技術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．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 [14]．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に基づいて各関数を特徴ベクトルに変換する．ここでワードとは，以下の 2 つを対象とする．

- 変数や関数などにつけられた識別子名を構成する単語
- 条件文や繰り返し文などの構文に利用される予約語

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 [6] 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

2.1 節では，関数クローン検出法の概要について説明した．しかし，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ものを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．

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 法を用いているが，TF-IDF 法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

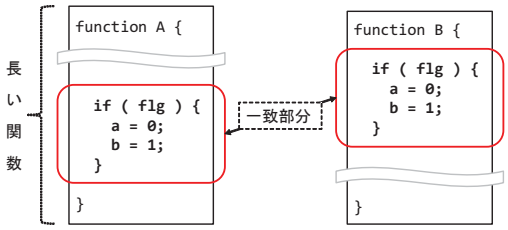

図 1　長い関数内の一部にコードクローンを含む例
Fig. 1　An example of parts of long function containing code clones

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 等）に対してコードクローンの検出を行った場合，メモリ不足で検出を完了できない恐れがある．

そこで，上の 2 つの問題点を踏まえた新しいコードクローン検出法の必要性が考えられる．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を提案した．コードブロック単位で検出を行うことで，関数クローン検出法では検出できなかったコードクローンを検出が可能になる．また，空間計算量の少ない特徴ベクトルの実装方法や LSH に変更することで，大規模プロジェクトへの適用が可能となる．

3. 提案する検出手法

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基に，コードブロック単位のクローン検出に対応するよう変更した手法（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　構文解析を行い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 法を利用し，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主な相違点は，コードクローンの検出粒度である．関数クローン検出法では関数の検出のみを行うが，本手法では特徴ベクトルの計算方法も変更し，関数と関数内のコードブロックの両方を検出する．

しかし，本手法は検出粒度を小さくすることで検出対象数が増え，それに伴い検出時間，メモリ使用量が増大する．そのため，特徴ベクトルのクラスタリングでは，LSH [6] の一種であり，空間計算量の改良を行った Multi-Probe LSH [9] を適用した．

3.1 用語の定義

3.1.1 コードブロック

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　関数の“{ }”で囲まれた範囲

条件 2　if，else，for，while，do-while，switch 文の“{ }”で囲まれた範囲

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纏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，入れ子構造の内側のコードブロックも検出可能であり，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 3，4 のいずれかを満たすものをワードとして定義する．

条件 3　予約語

条件 4　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，繰り返し文等でよく利用される 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 等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

条件 5　コードブロック間の類似度が閾値以上

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

条件 6　コードブロック間に共通部分がない

$$CB_1 \cap CB_2 = \phi$$

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

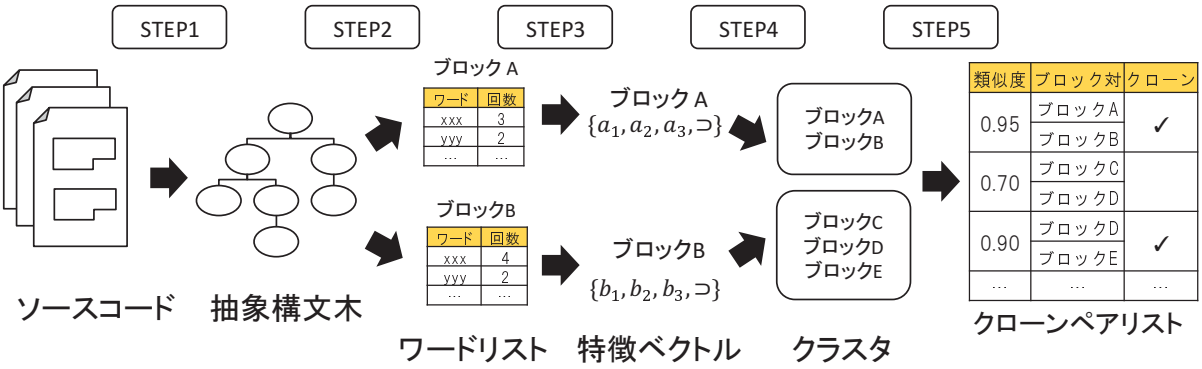

図 2　提案手法の概要
Fig. 2　An overview of our detection approach

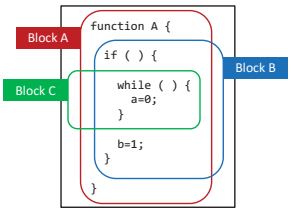

図 3　入れ子構造にあるコードブロック
Fig. 3　Code blocks in nested structure

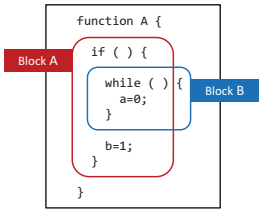

図 4　共通部分があるコードブロックペア
Fig. 4　Code block pair sharing a common part

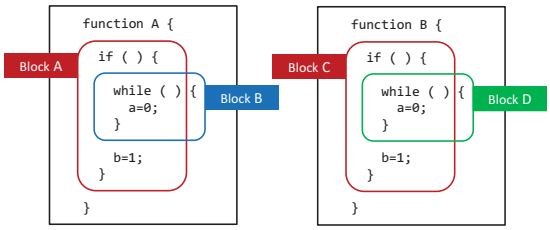

図 5　極大コードブロックペアと重複したコードブロックペア
Fig. 5　Code block pair overlapped with a maximam code block pair

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 6 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共通部分がないことが条件である．コードブロック間に共通部分が存在する場合，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 4** のコードブロック A と B は共通部分が存在し，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 5** 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

本手法では構文解析を行い，コードブロックの抽出を行う．コードブロック抽出の手法は以下の 6 つのステップに分けられる．本手法では，構文解析に ANTLR v4[*1] を利用した．

STEP I　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．

STEP II　抽象構文木から関数（3.1.1 節の条件 1 を満たすコードブロック）の部分木を取り出す．

STEP III　STEP II で取り出した部分木を最も外側のコードブロックとして抽出する．

STEP IV　STEP II で取り出した部分木から，コードブロック（3.1.1 節の条件 2 を満たすコードブロック）の部分木を取り出す．

STEP V　STEP IV で取り出した部分木を入れ子関係にあるコードブロックとして抽出する．

STEP VI　以降，深さ優先探索で抽象構文木から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．

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 法 [2] 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．

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

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

[*1] http://www.antlr.org/

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 4 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関わらず，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．

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，関数クローン検出法と同様に LSH を用いた．クラスタリングを行うことによって，クローンペアとなりうる候補を絞ることができ，高速なクローンペアの検出が可能となる．しかし，大規模のデータセットを LSH アルゴリズムを用いて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 [8], [9].

LSH のアルゴリズムは，空間的に近接した二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クラスタリングを行う．しかし LSH は確率的手法であるため，近接した二点が偶然に別のバケットに入る可能性がある．従来の LSH では複数のハッシュ関数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関数が必要となり，これにより LSH のメモリ使用量の増大につながっている．

そこで, Lv らはメモリ使用量の改良を行った Multi-Probe LSH[9] を提案した．Multi-Probe LSH は，ある点が入るバケットだけでなく，空間的に近接したバケット群も調べるというものである．これにより，少ないハッシュ関数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 個のハッシュ関数が必要だったのに対し，Multi-Probe LSH は 3 個のハッシュ関数で検索時間をほぼ落とさずに達成したという結果が報告されている [9].

本手法ではメモリ使用量の改良を行うため，Multi-Probe LSH を用いてクラスタリングを行う．なお，Multi-Probe LSH の実装として FALCONN [1][*2] ライブラリを利用した．

表 1　検出対象プロジェクト
Table 1　Target projects

プロジェクト	バージョン	言語	規模
Apache HTTPD[*3]	2.2.14	C/C++	343 KLOC
PostgreSQL[*4]	8.5.1	C/C++	937 KLOC
Python[*5]	2.5.1	C/C++	435 KLOC

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．

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

TF-IDF 法の計算式より，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．

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，本手法と既存手法との比較と，本手法の検出時間とスケーラビリティの評価を行った．

4.1 関数クローン検出法と CCFinder との比較

本節では関数クローン検出法と CCFinder [7] の 2 つの既存手法との比較実験について述べる．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った．CCFinder は国内外の企業・大学で使用されているため，比較対象に追加した．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，CCFinder 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

[*2] https://falconn-lib.org/
[*3] http://httpd.apache.org/
[*4] http://www.postgresql.org/
[*5] http://www.python.org/

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「はい」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．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 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 CCFinder 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．

適合率

本手法では，Apache HTTPD と Python において，関数クローン検出法や CCFinder より高い適合率が得られた．また，PostgreSQL においては，CCFinder 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 つの全てのプロジェクトの合計では適合率 0.68 と，関数クローン検出法と同程度の適合率，CCFinder より高い適合率であることが確認できた．

再現率

本手法では，Apache HTTPD，PostgreSQL において，関数クローン検出法や CCFinder 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，CCFinder より低い再現率となった．3 つの全てのプロジェクトの合計では，再現率 0.70 と，関数クローン検出法と CCFinder より総合的に再現率が高いことが確認できた．

F 値

本手法では，Apache HTTPD において F 値 0.81 と，関数クローン検出法や CCFinder より高い F 値が得られた．PostgreSQL においては F 値 0.69 と，CCFinder 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 0.67 と，関数クローン検出法よりは高い F 値が得られたが，CCFinder より低い F 値となった．3 つの全てのプロジェクトの合計の F 値は 0.69 となり，関数クローン検出法と CCFinder より総合的に再現率が高いことが確認できた．

表 2　検出精度の評価
Table 2　Evaluation of detection accuracy

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
CCFinder	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

表 3　検出時間の比較
Table 3　Comparison of detection time

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

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，検出対象全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ができた．また，関数クローン検出法に対して 3〜4 割程度，CCFinder に対して 4〜8 割程度と，比較手法よりも短時間で検出することが確認できた．

4.2　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．計測には cloc[*6] を用いた．検出対象は，IJaDataset 2.0[*7] からランダムにファイルを選択し，**表 4** に示した LOC ごとのサブセットシステムを作成した．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 4 に示す．これにより，本手法では 100MLOC の検出対象に対して 4 時間程度で検出可能

*6　http://cloc.sourceforge.net
*7　http://secold.org/projects/seclone

表 4　検出規模ごとの検出時間
Table 4　Detection time for varying input size

LOC	1K	10K	100K	1M	10M	100M
時間	1s	2s	15s	2m 53s	32m 59s	4h 5m 17s

であることが確認できた．

4.3　ブロッククローンの実例

本節では，アンケートにて保守対象のコードクローンと判断されたクローンペアの中から，本手法によって検出したブロッククローンの実例を示す．

同じ関数内に存在するブロッククローン

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などして関数内で発生したコードクローンを検出できなかったが，本手法では検出可能である例を確認できた．

長い関数内の一部が一致するブロッククローン

90 行以上の長い関数内の一部が一致するブロッククローンである．関数単位では異なる処理を行っているため，関数クローン検出法ではこのようなコードクローンを検出できなかった．しかし，本手法では検出可能である例を確認できた．

文の挿入が行われたブロッククローン

関数単位では，類似した処理を行うタイプ 4 のコードクローンであるが，関数クローン検出法では実際に検出できなかった．しかし検出粒度を下げることで，本手法では検出可能である例を確認できた．

類似した処理を行うブロッククローン

図 6 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．赤色のコード片と緑色のコード片が一致箇所を表している．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 6(b) の apr_file_sync 関数が，図 6(a) の apr_file_flush 関数に加えて独自の処理を行うため，関数クローン検出法では検出できなかった．しかし検出粒度を下げることで各関数の共通部分を見つけ出し，図 6 のように検出可能である例を確認できた．

4.4　考察

本節では，提案手法，および評価実験の結果についての議論を行い，本手法の有用性，拡張性，および評価実験の妥当性についての考察を行う．

検出精度

山中らが行った Tempero らのコーパスを用いた評価実験では，関数クローン検出法は適合率が 90%を超えるという報告がされている [13]．また，沼田らが行ったバグを含むコード片に対する関数クローン検出手法と CCFinder の比較実験により，関数クローン検出法が CCFinder より高い適合率を得られることも報告されており，関数クローン検出法の有用性が確認できる [10]．しかし，関数クローン検出法では関数単位の検出のために検出漏れが生じている可能性を考え，検出粒度をコードブロック単位に小さくした本手法を提案した．本実験の結果，3 つのプロジェクトの合計に対して適合率，再現率，F 値の 3 つの指標で関数クローン検出法や CCFinder より高い値が得られ，検出精度の観点で本手法の有用性を確認できた．

```
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) 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) httpd/srclib/apr/file_io/unix/readwrite.c

図 6　ファイルの出力処理を行うブロッククローン
Fig. 6　Block clones writing data to a file

検出時間とスケーラビリティ

本手法は関数クローン検出法や CCFinder より高速に検出できることが確認できた．また，100MLOC という大規模プロジェクトに対して 4 時間程度で検出できることも確認できた．検出の高速化の理由として，Multi-Probe LSH を用いてクラスタリングを行うことで，メモリ使用量が減り，より効率的な計算が行えることにより高速化に繋がったと考えられる．

ブロッククローンの実例

ブロッククローンの実例によって，長い関数内の一部が一致するコードクローンや，同じ関数内に存在するコードクローンなど，関数クローン検出法では検出できなかったコードクローンを確認できた．よって，関数クローン検出法による検出漏れの削減を示せた．

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可能である．

また，本手法ではコードブロックを“{ }”に囲まれた範囲によって抽出している．しかし，字下げによるコードブロックの抽出や，ある一定行数のまとまりをコードブロックとして抽出するなど，抽出方法に他の手法を適用することも可能である．

評価実験の妥当性

本実験では，3 つの C 言語のプロジェクトのみに対して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，他の言語で実装された多くのプロジェクトに対して適用し，一般性を示す必要性がある．また，今回はベンチマークの作成において，アンケートにて過半数である 2 人以上が保守対象となりうるコードクローンと判定した場合を正解として扱った．しかし，より正確性を上げるためには，意見が割れたクローンペアについて議論を行う必要性も考えられる．

5. まとめと今後の課題

本研究では，情報検索技術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，意味的に処理内容が類似したブロッククローンの検出を行う．また，LSH アルゴリズム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 CCFinder の 2 つの手法と比較を行った．その結果，本手法が総合的に高い精度で多くのコードクローンを検出できることが確認できた．また，検出時間は 3 分以下となり，関数クローン検出法と CCFinder より高速にコードクローンを検出できた．さらに，関数クローン検出法では検出できなかったコードブロック単位のコードクローンを検出できた．

今後の課題として，以下が挙げられる．

- ワードの重みの計算に，LSI（Latent Semantic Indexing）[2] や，LDA（Latent Dirichlet Allocation）[4] を用いた手法と比較を行う必要がある．
- 本手法では“{ }”によってコードブロックの定義を行っているが，コードブロックの定義や抽出方法を再考する必要がある．
- 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，MeCC などの他のツールとの比較を行う必要がある．

謝辞 本研究におい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，渋谷健介氏に深く感謝する．本研究は JSPS 科研費 25220003，16K16034，15H06344 の助成を受けた．

参考文献

[1] Andoni, A., Indyk, P., Laarhoven, T., Razenshteyn, I. and Schmidt, L.: Practical and Optimal LSH for Angular Distance, *Proceedings of the 28th International Conference on Neural Information Processing Systems*, pp. 1225–1233 (2015).

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

[3] Baxter, I. D., Yahin, A., Moura, L., Anna, M. S. and Bier, L.: Clone detection using abstract syntax trees, *Proceedings of International Conference on Software Maintenance*, pp. 368–377 (1998).

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

[5] 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, pp. 1465–1481 (2008).

[6] Indyk, P. and Motwani, R.: Approximate nearest neighbors: towards removing the curse of dimensionality, *Proceedings of the 30th annual ACM symposium on Theory of computing*, pp. 604–613 (1998).

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

[8] 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, pp. 256–268 (2014).

[9] Lv, Q., Josephson, W., Wang, Z., Charikar, M. and Li, K.: Multi-probe LSH: efficient indexing for high-dimensional similarity search, *Proceedings of the 33rd international conference on Very large data bases*, pp. 950–961 (2007).

[10] Numata, S., Yoshida, N., Choi, E. and Inoue, K.: *On the Effectiveness of Vector-Based Approach for Supporting Simultaneous Editing of Software Clones*, pp. 560–567, Springer International Publishing (2016).

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

[12] 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, pp. 470–495 (2009).

[13] 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, pp. 2245–2255 (2014).

[14] Yamanaka, Y., Choi, E., Yoshida, N., Inoue, K. and Sano, T.: Applying clone change notification system into an industrial development process, *Proceedings of the 21st International Conference on Program Comprehension*, pp. 199–206 (2013).