情報処理学会論文誌 Vol.55 No.10 2245–2255 (Oct. 2014)

情報検索技術に基づく高速な関数クローン検出

山中 裕樹[1] 崔 恩瀞[1,a)] 吉田 則裕[2,b)] 井上 克郎[1,c)]

受付日 2014年1月7日, 採録日 2014年6月17日

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．これまでの研究において様々なコードクローン検出手法が提案されてきたが，多くの手法がプログラムの構造的な類似性のみに着目している．また，プログラムの意味的な処理の類似性に着目した手法では，検出時間に膨大な時間がかかる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関数クローン（関数単位のコードクローン）の検出手法を提案する．関数単位のコードクローンは処理の内容がまとまっているため，コード片単位のコードクローンに比べてライブラリ化などの集約の対象になりやすいと考えられる．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うことによって，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関数クローンの検出を行う．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

A High Speed Function Clone Detection Based on Information Retrieval Techniques

Yuki Yamanaka[1] Eunjong Choi[1,a)] Norihiro Yoshida[2,b)] Katsuro Inoue[1,c)]

Received: January 7, 2014, Accepted: June 17, 2014

Abstract: A code clone (i.e., code fragment that has identical or similar fragment to it in the source code) is one of the major problems for software maintenance. So far, a lot of approaches have been developed on the detection of code clones. Several of them focus on semantic similarities based on control and data flow analyses, however they lack the scalability for large-scale source code. In this study, we propose an approach to detect function clones using information retrieval techniques. The proposed approach generates a feature vector for each function based on the occurrence of identifiers and reserved keywords, and then performs clustering of generated vectors. Finally, a set of functions corresponding to vectors in each cluster is detected as a set of semantic clones. As a case study, we applied the proposed approach to open source software systems. The result shows that the proposed approach is able to perform faster and more precise detection of function clones compared to the existing approach based on the similarity between abstract memory states.

Keywords: code clone, software maintenance, information retrieval

1. まえがき

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が指摘されている [1], [2], [3]．コードクローンとは，ソースコード中に存在する同一または類似した部分を持つコード片のことであり，コピーアンドペーストなどの様々な理由により生成される [1], [3]．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いわれている．たとえば，あるコード片を編集する場合，対応するすべてのコードクローンに対しても一貫した編集が必要となる可能性がある．コードクローンを検出することによっ

[1] 大阪大学
Osaka University, Suita, Osaka 565–0871, Japan
[2] 名古屋大学
Nagoya University, Nagoya, Aichi 464–8601, Japan
a) ejchoi@ist.osaka-u.ac.jp
b) yoshida@ertl.jp
c) inoue@ist.osaka-u.ac.jp

て，一貫した編集がなされておらず，不具合を引き起こす危険性があるコード片を検出することが可能である．また，コードクローン中の共通処理に対して，親クラスへの引き上げやライブラリ化といった集約を行うことによって，ソフトウェアの保守性や可読性を向上させることが可能となる [4], [5]．

これまでに様々なコードクローン検出手法が提案されてきたが，その多くの手法がプログラムの構造的な類似性のみに着目している [6]．そのため，同一の処理を実装しているにもかかわらず，for 文と while 文の違いなど構文上の実装が異なる場合はコードクローンとして検出することができる手法は少ない．また，プログラム依存グラフを用いた手法など，構文上の実装に依存せずに，処理内容が類似している部分をコードクローンとして検出する手法もいくつか提案されているが，検出時間に膨大な時間がかかるという問題点がある [7], [8], [9]．

そこで本研究では，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（関数単位のコードクローン）を検出する手法を提案する．情報検索とは，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 [10]．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であるコードクローンが多く検出されることがある [11]．一方，関数単位のコード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と考えられる．

本手法では，情報検索技術を用いて，ソースコード中の識別子や予約語に用いられる単語に対して重み付けを行うことによって，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計算することによって関数クローンの検出を行う．

評価実験では，意味的な類似性に基づく関数クローン検出ツールの 1 つである MeCC [9] と比較実験を行った．MeCC は，大域変数，仮引数，局所変数が関数終了時点で取りうる値を静的解析により推定し，それらの値が類似した関数対を関数クローンとして検出する．比較実験の結果，本手法の方がより高い精度で高速な検出を行うことができた．また，ほぼ同一の処理を実装しているにもかかわらず，条件分岐処理や繰返し処理の実装が異なる関数クローンや，文が並べ替えられている関数クローンを検出することができた．

以降，2 章では，本研究で提案する関数クローン検出手法について述べる．3 章では，本手法の評価実験について述べる．4 章では，評価実験の考察について述べる．5 章では，本研究の関連研究について述べる．最後に，6 章でまとめと今後の課題について述べる．

2. コードクローン検出手法

本章では，本研究で提案する関数クローン検出手法の概要について説明する．コードクローンには，以下の 4 つのタイプが存在する [2], [9]．

タイプ 1：空白の有無，レイアウト，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 の違い加えて，変数名などのユーザ定義名，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，変更などが行われたコードクローン

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，以下のものがあげられる．

- 条件分岐処理や繰返し処理などの制御構造の実装が異なる（図 3，図 5 参照）．
- 中間媒介変数の利用の有無が存在している（図 4 参照）．
- 文の並べ替えが発生している（図 6 参照）．

本研究では，情報検索技術を利用し，上記のすべてのタイプの関数クローンを高速に検出することが目的である．

本手法では，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．

- 変数や関数などに付けられた識別子名を構成する単語
- 条件文や繰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（互いに処理が類似した関数クローンの対）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（Locality-Sensitive Hashing）アルゴリズム [12] を用いて特徴ベクトルのクラスタリングを行うことにより，検出の高速化を図っている．もし，全関数の特徴ベクトル間の類似度を計算する場合，ソースコードの規模が大きくなると検出時間が膨大になってしまうと考えられる．

本手法の概要を**図 1** に示す．本手法は，主に以下の 4 つのステップから構成される．

STEP1：ソースコード中の各関数からワードの抽出を行う．

STEP2：STEP1 で抽出したワードに重み付けを行うことによって，各関数の特徴ベクトルを計算する．

STEP3：LSH アルゴリズム [12] を用いて，STEP2 で求めた特徴ベクトルのクラスタリングを行う．

STEP4：STEP3 で求めたクラスタにおいて，特徴ベクトル間の類似度の計算を行い，関数クローンを検出する．

以降の節で，それぞれのステップの詳細について説明する．

2.1 STEP1：ワードの抽出

まず最初に，ソースコードの各関数に含まれる識別子や

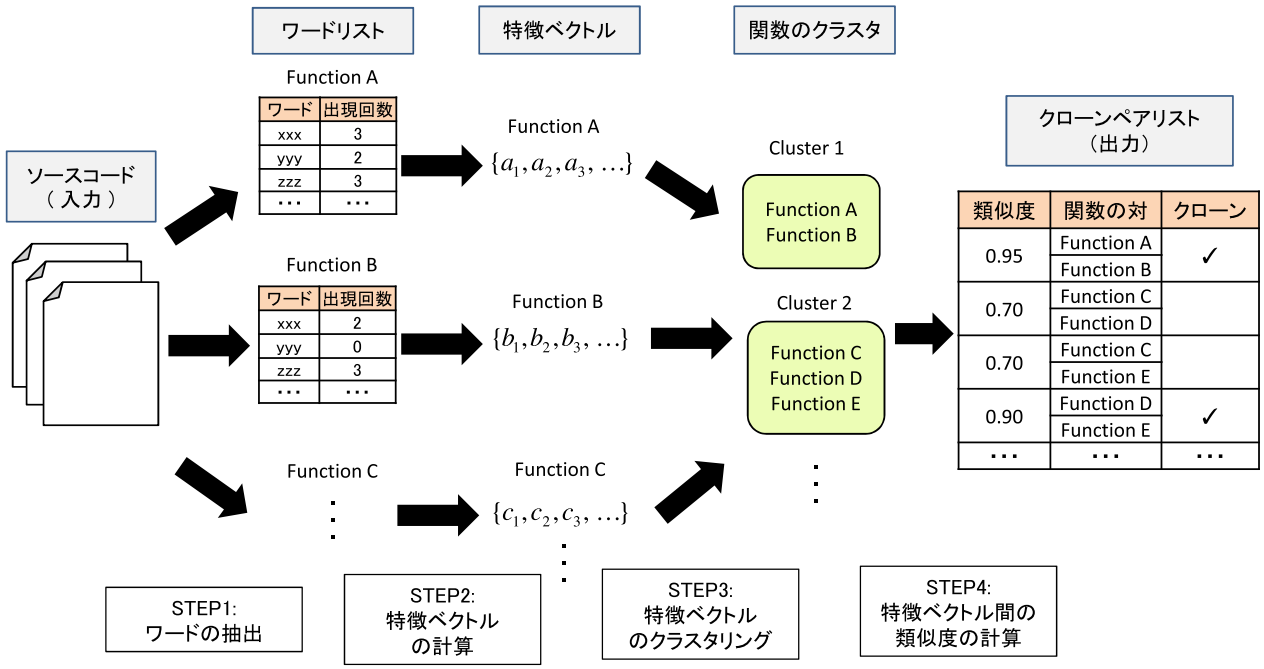


図 1 検出手法の概要
Fig. 1 Overview of our detection approach.

予約語から，ワードの抽出を行う．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（デリミタ）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

たとえば，“value_of_item” という識別子はアンダーバーで分割し，“value”・“of”・“item” という 3 つのワードに分割できる．また，“itemValue” という識別子も大文字で分割し，“item”・“value” という 2 つのワードに分割できる．さらに，2 文字以下の識別子名に対しては，それらをまとめて 1 つのワードとして扱う．この理由は，繰返し文などによく利用される “i” や “j” といった意味情報が込められない変数をすべて同一のものとして扱うためである．このように識別子の情報を利用することによって，各関数が実装する機能を表すことができると考えられる．

また，条件文に用いられる “if” や “switch”，繰返し文に用いられる “for” や “while” といった予約語もワードとして扱う．

2.2 STEP2：特徴ベクトルの計算

次に，STEP1 で抽出したワードに重み付けを行うことによって，各関数を特徴ベクトルに変換する．ここでは，TF-IDF 法 [10] を利用して各ワードの重みを計算し，その値を特徴量として利用する．TF-IDF 法は情報検索において文書の類似性の判定などに利用されており，計算コストが小さいため大規模なソースコードにも適用できると考えられる．TF-IDF 法による値は tf 値（関数中のワードの出現頻度）と idf 値（ソースコード全体のワードの希少さ）の積で与えられる．ワード x の重み w_x の計算式を以下に示す．

$$tf_x = \frac{\text{関数中のワード } x \text{ の出現回数}}{\text{関数中に出現する全ワードの出現回数の合計}}$$

$$idf_x = \log \frac{\text{全関数の数}}{\text{ワード } x \text{ が出現する関数の数}}$$

$$w_x = tf_x idf_x$$

本手法では，全関数中の各ワードに対して重みを計算し，それらを特徴量として用いることによって特徴ベクトルを求める．したがって，各関数の特徴ベクトルの次元はソースコード中に存在する全ワードの数となる．

2.3 STEP3：特徴ベクトルのクラスタリング

このステップでは，STEP2 で計算した各関数の特徴ベクトルに対してクラスタリングを行うことによって，クローンペアと成りうる候補を絞ることを目的とする．

ここでは，近似最近傍探索アルゴリズムの一種である LSH アルゴリズム [12] を用いて特徴ベクトルのクラスタリングを行う．LSH アルゴリズムは，(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である．このアルゴリズムを利用することによって，クエリとして 1 つの特徴ベクトルを与えると，特徴ベクトル集合からそのクエリと近似した特徴ベクトル集合のクラスタを取得することができる．(p_1, p_2, r, c)-Sensitive Hashing と (r, c)-Approximate Neighbor の定義を以下に示す．

(p_1, p_2, r, c)-Sensitive Hashing

実数 $p_1, p_2 (0 \leq p_1, p_2 \leq 1)$，実数 $r (0 < r)$，実数 $c (1 < c)$，$\mathbf{R}^d$ 空間に対するベクトル集合 V 上の任意の点 v_i，v_j が与えられたとき，

$$\begin{cases} \text{if } D(v_i, v_j) < r \text{ then } Prob[h(v_i) = h(v_j)] > p_1 \\ \text{if } D(v_i, v_j) > cr \text{ then } Prob[h(v_i) = h(v_j)] < p_2 \end{cases}$$

を満たすハッシュ関数 h を (p_1, p_2, r, c)-Sensitive Hashing と呼ぶ．ここで，D は $\mathbf{R}^d$ 上の距離関数，$Prob$ は条件式が真となる確率を表している．

(r, c)-Approximate Neighbor

$\mathbf{R}^d$ 空間に対するベクトル集合 V 上の任意の点（クエリ）v が与えられたとき，

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

で定義される V の部分集合 U をクエリ v に対する (r, c)-Approximate Neighbor と呼ぶ．ここで，c と r の定義は (p_1, p_2, r, c)-Sensitive Hashing で利用した定義と共通である．

$\mathbf{R}^d$ 空間に対する特徴ベクトル集合 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 H_l が以下の式で与えられたとき，

$$H_l = (h_{l,1}(v), h_{l,2}(v), \ldots, h_{l,K}(v)) \in \mathbf{R}^K$$

V 上の任意の点 v に対して L 個の K 次元のベクトルが得られる．LSH アルゴリズムでは，これらの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ことができる．なお，本手法では LSH アルゴリズムの実装である E2LSH [13][*1] を利用している．

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，確率に関するパラメータを $\rho = \log_{p_2} p_1$ としたとき，LSH のクラスタリングの時間計算量は $O(dn^\rho \log n)$ と表される．一方，全関数に対して特徴ベクトル間の類似度を計算する場合の時間計算量は $O(dn^2)$ となる．したがって，本ステップであらかじめクラスタリングを行い，クローンペアと成り得る候補を絞ることによって，検出時間にかかる計算コストを削減できると考えられる．

2.4 STEP4：特徴ベクトルの類似度の計算

最後に，STEP3 で求めた各クラスタ中の関数の対に対して，コサイン類似度を用いてクローンペアであるか否かの判定を行う．コサイン類似度は多次元ベクトルの類似度を測定するものであり，次元が d 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で表すことができる．

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

TF-IDF 法の計算式から分かるように，特徴量はつねに正の値となるため，コサイン類似度は 0 から 1 の範囲となる．もし，コサイン類似度が閾値以上であれば，その関数の対はクローンペアであると判定する．

3. 評価実験

本章では，2 章で述べた関数クローン検出手法の評価実験について述べる．本実験で適用の対象としたオープンソースソフトウェアプロジェクトを**表 1** に示す．

なお，本実験では結果の信頼性を重視するため，関数クローンとして検出する類似度の閾値を 0.9 とし，E2LSH が自動決定したパラメータ（**表 A·1**）のみを用いて関数クローンの検出を行った．

また，Java 言語における setter や getter などの比較的小さい関数を検出の対象から除外するため，他の研究 [14], [15] と同様に 30 トークン以上の関数を対象に実験を行った．以降，3.1 節では，コーパスを用いた本手法の検出精度の評価について述べる．3.2 節では，既存手法との比較実験について述べる．最後に 3.3 節で，本手法で検出できた関数クローンの実例を示す．

3.1 コーパスを用いた検出精度の評価

最初に，Tempero らのコーパス [16][*7] を用いた評価実験について述べる．このコーパスは，コードクローン検出ツール CMCD [17] の出力結果と著者らの目視による判断に基づいて作成されている．Bellon らのベンチマーク[*8] ではタイプ 3 までのコード片単位のコードクローンを対象としているが，Tempero らのコーパスではタイプ 1 からタイプ 4 のメソッド単位のコードクローンを対象としている．そのため，本研究では Temepro らのコーパスを用いて評価実験を行った．

本実験では 2 つの Java プロジェクト（Apache Ant，ArgoUML）に対して検出精度の評価を行った．これらの Java プロジェクトは，他のコードクローン検出手法の評価実験

表 1 適用プロジェクト
Table 1 Target projects.

プロジェクト名	バージョン	言語	規模
Apache Ant[*2]	1.8.4	Java	109 KLOC
ArgoUML[*3]	0.34	Java	192 KLOC
Python[*4]	2.5.1	C/C++	435 KLOC
Apache HTTPD[*5]	2.2.14	C/C++	343 KLOC
PostgreSQL[*6]	8.5.1	C/C++	937 KLOC

[*1] http://www.mit.edu/~andoni/LSH/
[*2] http://ant.apache.org/
[*3] http://argouml.tigris.org/
[*4] http://www.python.org/
[*5] http://httpd.apache.org/
[*6] http://www.postgresql.org/
[*7] http://www.qualitascorpus.com/clones/
[*8] http://softwareclones.org/

表 2 コーパスを用いた検出精度
Table 2 Evaluation of detection accuracy based on the corpus.

	クローンペア数	適合率	再現率	タイプ 1	タイプ 2	タイプ 3	タイプ 4
Apache Ant	474	92.2%	62.3%	56	139	220	22
ArgoUML	880	96.4%	52.7%	222	219	371	33

でも一般的に利用されており，自動生成コードが存在していないことが知られている．また，本実験では手作業でクローンペアの各タイプの分類を行っており，複数のプロジェクトに対して適用するコストが大きいと考えたため，この 2 つのプロジェクトに対してのみ適用を行った．

結果を表 2 の適合率と再現率に示す．適合率は，全検出結果に対してコーパスで正解集合と判定されているクローンペアの割合を表している．再現率は，コーパス中のクローンペアに対して本手法によって検出されたクローンペアの割合を表している．実験の対象とした 2 つのプロジェクトに対して再現率は 60%前後であるが，適合率は 90%を超えており，誤検出をほとんど含まず関数クローンを検出することができた．Bellon らが行ったタイプ 1〜3 を対象としたコードクローン検出ツールの比較評価 [18] によると，各ツールの再現率は 8%〜46%である．比較評価の方法が異なるため，単純に再現率を比較することはできないが，本手法の 60%前後の再現率から，実用的な検出結果が得られることが期待できる．

さらに，表 2 には正解集合と判定したクローンペアを手作業で各タイプに分類した結果を示している．この表から分かるように，本手法を用いることによって，すべてのタイプの関数クローンを検出することができた．

3.2 既存手法との比較

次に，既存手法との比較実験について述べる．本実験では，関数クローン検出ツールの 1 つである MeCC [9] との比較を行った．MeCC では，静的解析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．ここでメモリとは，ADDR（大域変数，仮引数，局所変数などの関数で利用されている変数）と GV（ADDR が取りうる値の集合）の組の集合で表される．メモリの状態の類似度の計算例を図 2 に示す．この例では，2 つの関数 Function A と Function B のメモリ MEM_A と MEM_B の類似度を求めている．

まず最初に，以下の式を用いて 2 つの関数中の ADDR の全組合せに対して類似度を計算する．

$$sim_{addr}(ADDR_X, ADDR_Y) = \frac{2 \times |GV_X \cap GV_Y|}{|GV_X| + |GV_Y|}$$

ここで GV_X と GV_Y は，それぞれ，$ADDR_X$ と $ADDR_Y$ が取り得る値の集合を意味する．次に，貪欲法を用いて，

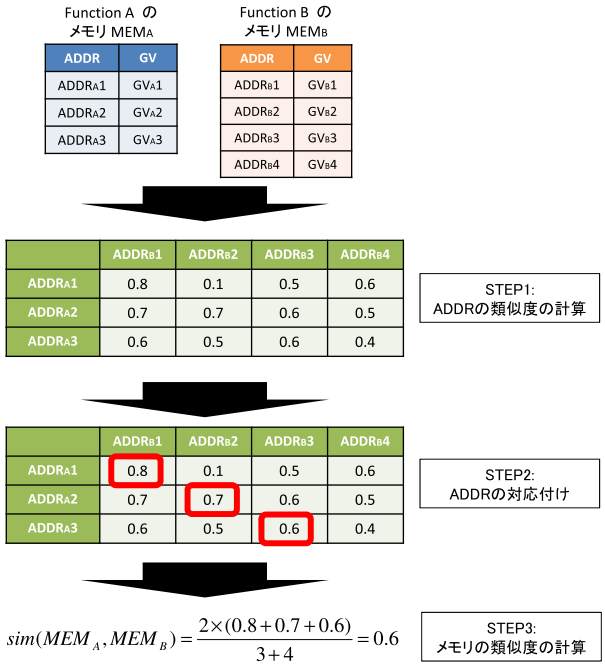


図 2 MeCC の概要
Fig. 2 Overview of MeCC.

高い類似度から順に ADDR の対応付けを行う．

最後に，以下の式を用いて，メモリ MEM_A と MEM_B の類似度の計算を行う．

$$sim(MEM_A, MEM_B) = \frac{2 \times SUM(sim_{addr})}{|MEM_A| + |MEM_B|}$$

ここで，$SUM(sim_{addr})$ は，2 つの関数間で対応する ADDR の類似度の合計値を意味する．そして，これらの類似度が閾値以上であれば，コードクローンとして検出を行う．

本手法で MeCC を比較の対象とした主な理由は，他の検出手法に比べて，高い精度でタイプ 3 とタイプ 4 の関数クローンを数多く検出できることを評価実験で示しているためである．また，大規模なプロジェクトに対しても，実時間で検出することが可能である．さらに，検出結果をウェブサイト上[*9]で公開しているため，比較対象として選択した．

MeCC は C 言語が対象であるため，その評価実験で利用している 3 つの C プロジェクト（Python, Apach HTTPD, PostgreSQL）に対して比較を行った．以降，検出精度と検出時間の比較について述べる．

3.2.1 検出精度の比較

検出精度の比較では，3 つのプロジェクトに対する適用

[*9] http://ropas.snu.ac.kr/mecc/

表 3 MeCC との検出精度と検出クローンセット数の比較
Table 3 Comparison of accuracy and the number of detected clone sets with MeCC.

		誤検出の割合	正解集合			
			タイプ 1	タイプ 2	タイプ 3	タイプ 4
本手法	Python	6.5%	19	103	159	21
	Apache HTTPD	4.6%	71	100	190	11
	PostgreSQL	5.3%	57	230	341	17
	合計	5.4%	147	433	690	59
MeCC	Python	14.7%	3	127	82	13
	Apache HTTPD	12.5%	2	84	71	10
	PostgreSQL	16.9%	9	120	88	14
	合計	15.0%	14	331	241	37

表 4 再現性の評価
Table 4 Evaluation of recall.

	30 トークン以上	全関数
Python	97.5%	61.2%
Apache HTTPD	62.3%	40.0%
PostgreSQL	88.1%	72.7%
合計	86.5%	66.2%

表 5 Roy らのベンチマークを用いた比較
Table 5 Comparison of accuracy using benchmark created by Roy et al.

	タイプ 1	タイプ 2	タイプ 3	タイプ 4
ベンチマーク	3	4	5	4
本手法	3	2	5	4
MeCC	3	4	4	4

結果を手作業で正解集合と誤検出に分類し，その割合を比較した．さらに，正解と判断した関数クローンをタイプ 1 からタイプ 4 に分類し，タイプごとの検出数を比較した．なお，MeCC ではクローンセット（互いに類似した関数クローンの集合）単位でタイプの分類を行っている．そのため，本実験においても，クローンペアをクローンセット単位に変換し分類を行っている．

本手法と MeCC における，検出精度とタイプごとの検出クローンセット数の比較を**表 3** に示す．MeCC の検出結果は，文献 [9] の表 2 および表 3 より抜粋した[*10]．MeCC では全検出結果に対して，10%以上の誤検出を含んでいるが，本手法では誤検出の割合が 10%未満であった．また，クローンセットの検出数においても，MeCC よりも多くの関数クローンを検出することができた．特に，タイプ 1，タイプ 3，タイプ 4 の検出はすべてのプロジェクトにおいて MeCC を上回っていることを確認することができた．

また，本実験では MeCC に対する本手法の再現性について評価を行った．再現性とは，MeCC と本手法で検出できた関数クローンの和集合に対する，本手法の検出数の割合を示している．この指標は文献 [18] でも用いられている．結果を**表 4** に示す．なお，本手法では 30 トークン以上の関数のみを対象としている．そのため，MeCC の全検出結果に対する再現性と，30 トークンでフィルタリングした場合について評価を行った．その結果，全検出結果に対して再現性は約 50%前後であるが，30 トークン以上のより大きな関数では 62%～98%と高い再現性を示すことを確認することができた．

さらに，Roy らのベンチマーク [2] を用いて，タイプごとの検出精度の評価を行った．このベンチマークでは，タイプ 1 からタイプ 4 までの合計 16 個の関数単位のクローンペアが用意されており，文献 [9] の評価実験でも利用されている．

表 5 は，ベンチマークで用意されているクローンペアのタイプごとの個数と，本手法で検出することができたクローンペアのタイプごとの個数を示している．結果として，全体で 16 個中 14 個のクローンペアを検出することができた．また，タイプ 1，タイプ 3，タイプ 4 において，本手法を用いてすべてのクローンペアを検出することができた．MeCC は，if 文が挿入されたタイプ 3 のクローンペアを検出することができなかった．これは，if 文が挿入されたことにより，その直後に存在するメソッド呼出し文が実行される条件が変化したためであると考えられる．本手法は，if 文が追加されても，出現する変数名が変化しなかったため，検出することができたと考えられる．

一方で，タイプ 2 では，2 つのクローンペアの検出を行うことができなかった．これらのクローンペアは，元の変数名が 1 文字のアルファベットに省略されており，意味を持たない変数名に変換されていたため，本手法ではコードクローンとして検出することができなかったと考えられる．

3.2.2 検出時間の比較

検出時間の比較では，MeCC の評価実験で利用されたワークステーションと同様の環境（OS: Ubuntu 64-bit, CPU: Intel Xeon 2.40 GHz，RAM: 8.0 GB）で本手法を実行し，3 つのプロジェクトに対する検出時間を測定した．

[*10] 各表には，MeCC のオプションとして Similarity = 80%および MinEntry = 50 を指定したときの検出結果が掲載されている．各オプションについては文献 [9] を参照されたい．

表 6 検出時間の比較
Table 6 Comparison of detection time.

	本手法		MeCC
	パラメータ計算あり	パラメータ計算なし	
Python	4 m 39 s	2 m 13 s	65 m 26 s
Apache HTTPD	4 m 30 s	1 m 43 s	310 m 34 s
PostgreSQL	8 m 51 s	4 m 39 s	428 m 32 s

```
1:  public void log(String message,int loglevel){
2:      if (managingPc != null) {
3:          managingPc.log(message,loglevel);
4:      }else {
5:          if (loglevel > Project.MSG_WARN) {
6:              System.out.println(message);
7:          }else {
8:              System.err.println(message);
9:          }
10:     }
11: }
```

(a) ant/util/ConcatFileInputStream.java
（if 文を用いた条件分岐処理）

```
1:  public void log(String message,int loglevel){
2:      if (managingPc != null) {
3:          managingPc.log(message,loglevel);
4:      }else {
5:          (loglevel > Project.MSG_WARN ? System.out :
                              System.err).println(message);
6:      }
7:  }
```

(b) ant/util/ConcatResourceInputStream.java
（三項演算子を用いた条件分岐処理）

図 3 条件分岐処理が異なるタイプ 4 の関数クローン（Apache Ant）
Fig. 3 Type-4 function clones with different conditional statements.

結果を**表 6** に示す．表中のパラメータ計算とは，特徴ベクトルのクラスタリングで利用している E^2LSH に対するものである．E^2LSH 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．したがって，1 つの適用対象に対してパラメータの値を一度計算すれば，その後の検出においてもその値を再利用することが可能であると考えられる．

結果として，パラメータ計算を行う場合において，9 分以下で関数クローンを検出できることを確認できた．また，パラメータ計算を行わない場合においてはすべてのプロジェクトに対して 5 分以下で関数クローンを検出することができた．一方，MeCC では静的解析に膨大な時間がかかるため，検出時間が長くなる傾向がある．本手法では，関数を特徴ベクトルに変換し，あらかじめ LSH アルゴリズムを用いたクラスタリングを行っている．そのため，MeCC よりも本手法の方が高速に検出を行うことができたと考えられる．

```
1:  static ap_conf_vector_t *
2:  create_default_per_dir_config(apr_pool_t *p)
3:  {
4:      void **conf_vector = apr_pcalloc(p, sizeof(void *) *
5:                           (total_modules +DYNAMIC_MODULE_LIMIT));
6:      module *modp;
7:
8:      for (modp = ap_top_module; modp; modp = modp->next) {
9:          dir_maker_func df = modp->create_dir_config;
10:         if (df)
11:             conf_vector[modp->module_index] = (*df)(p, NULL);
12:     }
13:     return (ap_conf_vector_t *)conf_vector;
14: }
```

(a) server/config.c（中間媒介変数を利用した実装）

```
1:  static ap_conf_vector_t *
2:  create_server_config(apr_pool_t *p, server_rec *s)
3:  {
4:      void **conf_vector = apr_pcalloc(p, sizeof(void *) *
5:                           (total_modules + DYNAMIC_MODULE_LIMIT));
6:      module *modp;
7:
8:      for (modp = ap_top_module; modp; modp = modp->next) {
9:          if (modp->create_server_config)
10:             conf_vector[modp->module_index]
11:                                 = (*modp>create_server_config)(p, s);
12:     }
13:     return (ap_conf_vector_t *)conf_vector;
14: }
```

(b) server/config.c（中間媒介変数を利用しない実装）

図 4 中間媒介変数の利用が異なるタイプ 4 の関数クローン（Apache HTTPD）
Fig. 4 Type-4 function clones with and without a temporal variable.

```
1:  static const XML_Char *
2:  poolCopyStringN(STRING_POOL *pool, const XML_Char *s, int n)
3:  {
4:      if (!pool->ptr && !poolGrow(pool))
5:          return NULL;
6:      for (; n > 0; --n, s++) {
7:          if (!poolAppendChar(pool, *s))
8:              return NULL;
9:      }
10:     s = pool->start;
11:     poolFinish(pool);
12:     return s;
13: }
```

(a) Modules/expat/xmlparse.c（for 文を用いた繰返し処理）

```
1:  static const XML_Char * FASTCALL
2:  poolCopyString(STRING_POOL *pool, const XML_Char *s)
3:  {
4:      // ※例外処理漏れの可能性がある
5:      do {
6:          if (!poolAppendChar(pool, *s))
7:              return NULL;
8:      }while (*s++);
9:      s = pool->start;
10:     poolFinish(pool);
11:     return s;
12: }
```

(b) Modules/expat/xmlparse.c（do-while 文を用いた繰返し処理）

図 5 繰返し処理が異なるタイプ 4 の関数クローン（Python）
Fig. 5 Type-4 function clones with different implementations of iterative processing.

3.3 関数クローンの実例

本手法では文の並べ替え，中間変数の利用の有無，if 文と三項演算子を用いた条件分岐処理の置き換え，for 文と while 文を用いた繰返し処理の置き換えなどの違いが存在する関数クローンを検出することができた．**図 3**，**図 4**，**図 5**，**図 6** に本手法で検出した関数クローンの例を示す．これらは，MeCC では検出することができなかった関数クローンである．なお，図中では 2 つの関数の差異を色付けして表現している．

```
1:  static PyObject *
2:  dequereviter_next(dequeiterobject *it)
3:  {
4:      PyObject *item;
5:      if (it->counter == 0)
6:          return NULL;
7:      if (it->deque->state != it->state) {
8:          it->counter = 0;
9:          PyErr_SetString(PyExc_RuntimeError,
10:                                  "deque mutated during iteration");
11:         return NULL;
12:     }
13:     assert (!(it->b == it->deque->leftblock &&
14:                                  it->index < it->deque->leftindex));
15:
16:     item = it->b->data[it->index];
17:     it->index--;
18:     it->counter--;
19:     if (it->index == -1 && it->counter > 0) {
20:         assert (it->b->leftlink != NULL);
21:         it->b = it->b->leftlink;
22          it->index = BLOCKLEN - 1;
23:     }
24:     Py_INCREF(item);
25:     return item;
26: }
```

(a) Modules/collectionsmodule.c

```
1:  static PyObject *
2:  dequeiter_next(dequeiterobject *it)
3:  {
4:      PyObject *item;
5:      if (it->deque->state != it->state) {
6:          it->counter = 0;
7:          PyErr_SetString(PyExc_RuntimeError,
8:                                  "deque mutated during iteration");
9:          return NULL;
10:     }
11:     if (it->counter == 0)
12:         return NULL;
13:     assert (!(it->b == it->deque->rightblock &&
14:                              it->index > it->deque->rightindex));
15:
16:     item = it->b->data[it->index];
17:     it->index++;
18:     it->counter--;
19:     if (it->index == BLOCKLEN && it->counter > 0) {
20:         assert (it->b->rightlink != NULL);
21:         it->b = it->b->rightlink;
22          it->index = 0;
23:     }
24:     Py_INCREF(item);
25:     return item;
26: }
```

(b) Modules/collectionsmodule.c

図 6 文の並べ替えが存在するタイプ 4 の関数クローン（Python）
Fig. 6 Type-4 function clones with reordered statements.

図 3 は，条件分岐処理が置き換わっている例である．図 3 (a) では if-else 文を用いて標準出力と標準エラー出力の条件分岐処理を実装しているが，図 3 (b) では三項演算子を用いて条件分岐処理を実装している．

図 4 は，中間媒介変数の利用の有無が存在する関数クローンである．図 4 (a) では 9 行目で変数 df を新たに定義し，それを 10〜12 行目で中間媒介変数として利用している．一方，図 4 (b) では中間媒介変数を利用せずに実装を行っている．

図 5 は，繰返し処理が置き換わっている例である．図 5 (a) では for 文を用いて繰返し処理を実装しているが，図 5 (b) では while 文を用いてほぼ同一の繰返し処理を実装している．さらに，図 5 (a) では 4〜5 行目で例外の判定が行われているのに対して，図 5 (b) で行われていない．このような文の挿入の有無はバグを含んでいる可能性があると考えられる．

図 6 は，文の並べ替えが発生している例である．図 6 (a) の最初の if 文（赤色のコード片）と 2 番目の if 文（青色のコード片）が図 6 (b) では入れ替わっている．

4. 考察

4.1 本手法の有用性

3 章では，コーパスを用いた評価実験と MeCC との比較実験を行い，高い適合率で検出を行うことができた．3.3 節では，本手法で検出したタイプ 4 の関数クローンの実例を示した．プログラム依存グラフを用いた検出手法では，図 6 のような関数は依存グラフそのものが異なってしまうため，検出することができない可能性が高い．一方，本手法では関数を特徴ベクトルに変換するため，文が並べ替えられている関数クローンの検出は容易であるといえる．

また，検出時間の評価では，3 つの C プロジェクトに対して適用し，MeCC より高速に検出を行うことができた．本手法では特徴ベクトルに対して LSH アルゴリズムを用いたクラスタリングをあらかじめ行っている．そのため，アルゴリズムの時間計算量から，ソースコードの規模に対してほぼ比例に近い時間で検出時間が増えていくと考えられる．評価実験からもソースコードのサイズと検出時間に線形的な関係がある結果が得られた．したがって，100 万行を超える他の大規模ソースコードに対しても十分適用することが可能であると考えられる．

表 7 は，3 つの C プロジェクトに対して不具合を引き起

表 7 不具合を引き起こす可能性があるクローンセットの検出数
Table 7 Number of clone sets including exploitable bug.

	本手法	MeCC
Python	25	23
Apache HTTPD	35	27
PostgreSQL	53	20

```
1: void sumProd(int n) {
2:     float sum=0.0; //C1
3:     float prod =1.0;
4:     for (int i=1; i<=n; i++){
5:         sum=sum + i;
6:         prod = prod * i;
7:         foo(sum, prod);
8:     }
9: }
```

```
1: void sumProd(int n){
2:     float s=0.0; //C1
3:     float p =1.0;
4:     for (int j=1; j<=n; j++){
5:         s=s + j;
6:         p = p * j;
7:         foo(s, p);
8:     }
9: }
```

図 7 本手法で検出できなかった関数クローン

Fig. 7 Function clones that our approach is not capable of detecting.

こす危険性があるクローンセットの数を手作業で調査した結果を示している．ここで不具合とは，図 5 (b) のように，特定の値の場合における条件分岐が欠如していることが原因で例外が起きる可能性がある関数クローンを意味している．MeCC と比較すると，本手法の方がより多くの不具合の候補を検出することができている．これらは，リファクタリングによって 1 つに集約することによって，その後の一貫した修正漏れや不具合の発生を防ぐことができることができると考えられる．

また，本手法では字句解析を行い，識別子と予約語の判定を行うだけで，関数を特徴ベクトルに変換することができる．したがって，C 言語や Java 言語だけでなく，他のプログラミング言語についても容易に本手法を適用することができると考えられる．さらに，コンパイルができないソースコードに対しても関数クローンの検出を行うことが可能である．

一方で，本手法ではタイプ 2 のコードクローンを検出できない場合がある．**図 7** は，MeCC などの他の手法で検出することができたが，本手法で検出できなかった関数クローンの実例である．この例では，変数名の識別子名が 1 文字のアルファベット（変数名の頭文字）に置換されている．このように，意味を持たない変数名に修正されたタイプ 2 の関数クローンは，本手法では検出することができない．トークンベースなどの既存検出ツールとの併用や，略語を考慮したワード抽出部分の改善など，さらに手法の改良を行っていく必要がある．

4.2 評価実験の妥当性

本手法の実装は，現在 Java 言語と C 言語にのみ対応している．そのため，2 つの Java プロジェクトと 3 つの C プロジェクトに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ている．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，一般性を示す必要がある．

また，評価実験では手作業でコードクローンの判定やタイプの分類を行っている．しかし，コーパスを用いた評価実験も行っているため一部の検出精度においては信頼できると考えられる．

本手法と MeCC はそれぞれ検出オプションを設定することができるため，それら設定が実験結果に影響している可能性がある．同一の検出オプションを設定できることが望ましいが，本手法と MeCC は異なる検出オプションを持つため困難である．たとえば，本手法では小規模の関数を検出しないためにトークン数の指定し，MeCC ではメモリ（3.2 節参照）の最小サイズ（MinEntry）を指定するが，トークン数とメモリのサイズを換算することは難しい．

5. 関連研究

構文の類似性に着目した手法として，トークンベースの検出手法や，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．トークンベースの検出手法では，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 [14], [19]．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 [3], [15]．これらの手法では，高速にコードクローンの検出を行うことができるが，3.3 節で示したタイプ 4 のコードクローンの検出を行うことが困難である．

Yuan らは，類似したトークンを持つメソッドやクラスをコードクローンとして検出するツール CMCD [17] を開発している．このツールは，タイプ 4 のコードクローンを検出することが可能であり，本研究の評価実験で利用した Tempero らのコーパス [16] を作成する際に利用されている．しかし，CMCD は定量的な評価が行われておらず，ツールも公開されていないため，本研究で提案した手法と比較することが困難である．そのため，本研究では，検出精度と検出時間について定量的に評価が行われている MeCC [9] と比較実験を行った．

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 4 のコードクローンを検出することが可能である [7], [8]．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．

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）[10] 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 [20]．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ツール FCFinder を開発を行った [21]．本研究では，識別子の類似性に着目し，関数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 つの関数に含まれるトークン列が異なっていても，識別子の集合が類似していれば，それら関数は関数クローンとして検出される．

6. 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関数クローン検出手法の提案を行った．本手法では，ソースコード中の各ワードに対して重み付けを行い，それらを特徴量として各関数の特徴ベクトルを計算する．そして，特徴ベクトル間の類似度を計算することによって，タイプ 4 の関数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，高速な関数クローンの検出を実現した．

評価実験では，2 つの Java プロジェクトと 3 つの C プロジェクトに対して適用を行い，高い精度で関数クローンを検出することができた．さらに，既存手法である MeCC との比較を行い，検出数，検出精度，検出時間の観点からも本手法が有益であることを確認することができた．

今後の課題として，以下があげられる．

- LSI や LDA（Latent Dirichlet Allocation）[22] を用いた手法と検出精度と検出時間の観点から比較を行う必要がある．
- 類義語や同位語の関係を用いたワードのクラスタリングなどを行うことによって，リネームが行われた場合のコードクローンの検出精度を向上させる．
- 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 以外の他のツールとの比較を行う必要がある．

謝辞 本研究は JSPS 科研費 26730036，25220003 の助成を得たものである．

参考文献

[1] 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).
[2] 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).
[3] Baxter, I.D., Yahin, A., Moura, L., Sant'Anna, M. and Bier, L.: Clone detection using abstract syntax trees, *Proc. ICSM '98*, pp.368–377 (1998).
[4] Fowler, M.: *Refactoring: Improving the design of existing code*, Addison-Wesley Longman Publishing Co., Inc. (1999).
[5] 肥後芳樹，神谷年洋，楠本真二，井上克郎：コードクローンを対象としたリファクタリング支援環境，電子情報通信学会論文誌，Vol.88, No.2, pp.186–195 (2005).
[6] Rattan, D., Bhatia, R. and Singh, M.: Software clone detection: A systematic review, *Information and Software Technology*, Vol.55, pp.1165–1199 (2013).
[7] 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 (2010).
[8] Komondoor, R. and Horwitz, S.: Using slicing to identify duplication in source code, *Proc. SAS '01*, pp.40–56 (2001).
[9] Kim, H., Jung, Y., Kim, S. and Yi, K.: MeCC: Memory comparison-based clone detector, *Proc. ICSE '11*, pp.301–310 (2011).
[10] Baeza-Yates, R. and Ribeiro-Neto, B.: *Modern information retrieval: The concepts and technology behind Search* (*2nd Edition*), Addison-Wesley Professional (2011).
[11] Yamanaka, Y., Choi, E., Yoshida, N., Inoue, K. and Sano, T.: Applying clone change notification system into an industrial development process, *Proc. ICPC '13*, pp.199–206 (2013).
[12] Indyk, P. and Motwani, R.: Approximate nearest neighbors: Towards removing the curse of dimensionality, *Proc. STOC '98*, pp.604–613 (1998).
[13] Andoni, A. and Indyk, P.: Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions, *Comm. ACM*, Vol.51, No.1, pp.117–122 (2008).
[14] Kamiya, T., Kusumoto, S. and Inoue, K.: CCFinder: A multilinguistic token-based code clone detection system for large scale source code, *IEEE Trans. Software Engineering*, Vol.28, No.7, pp.654–670 (2002).
[15] Jiang, L., Misherghi, G., Su, Z. and Glondu, S.: DECKARD: Scalable and accurate tree-based detection of code clones, *Proc. ICSE '07*, pp.96–105 (2007).
[16] Tempero, E.D.: Towards a curated collection of code clones, *Proc. IWSC '13*, pp.53–59 (2013).
[17] Yuan, Y. and Guo, Y.: CMCD: Count matrix based code clone detection, *Proc. APSEC '11*, pp.250–257 (2011).
[18] Bellon, S., Koschke, R., Antoniol, G., Krinke, J. and Merlo, E.: Comparison and evaluation of clone detection tools, *IEEE Trans. Software Engineering*, Vol.33, No.9, pp.577–591 (2007).
[19] Li, Z., Lu, S., Myagmar, S. and Zhou, Y.: CP-Miner: Finding copy-paste and related bugs in large-scale software code, *IEEE Trans. Software Engineering*, Vol.32, No.3, pp.176–192 (2006).
[20] Marcus, A. and Maletic, J.I.: Identification of high-level concept clones in source code, *Proc. ASE '01*, pp.107–114 (2001).
[21] 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, pp.1423–1433 (2011).
[22] Blei, D.M., Ng, A.Y. and Jordan, M.I.: Latent dirichlet allocation, *The Journal of Machine Learning Research*, Vol.3, pp.993–1022 (2003).

付　　録

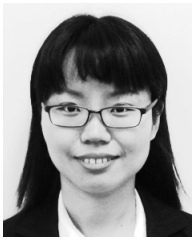
3 章において E2LSH が自動決定したパラメータを表 A·1 に示す．各パラメータの詳細は，E2LSH のマニュアル（http://www.mit.edu/~andoni/LSH/manual.pdf）を参照されたい．

表 A·1 E2LSH のパラメータ
Table A·1 E2LSH parameters.

プロジェクト名	k	m	L
Apache Ant	12	14	91
ArgoUML	12	14	91
Python	16	22	231
Apache HTTPD	16	22	231
PostgreSQL	16	22	231

山中 裕樹

平成 24 年大阪大学基礎工学部情報科学科卒業．平成 26 年大阪大学院情報科学研究科博士前期課程修了．現在，株式会社日立製作所に勤務．在学中，コードクローン管理に関する研究に従事．



崔 恩瀞 （学生会員）

平成 24 年大阪大学大学院情報科学研究科博士前期課程修了．現在，大阪大学大学院情報科学研究科博士後期課程 3 年．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 （正会員）

平成 16 年九州工業大学情報工学部知能情報工学科卒業．平成 21 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）．平成 22 年奈良先端科学技術大学院大学情報科学研究科助教．平成 26 年より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．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 （フェロー）

昭和 59 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．昭和 59～61 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．平成 3 年大阪大学基礎工学部助教授．平成 7 年同学部教授．平成 14 年大阪大学大学院情報科学研究科教授．平成 23 年 8 月より大阪大学大学院情報科学研究科研究科長．ソフトウェア工学，特にコードクローンやコード検索等のプログラム分析や再利用技術の研究に従事．