Vol.2016-SE-191 No.7
2016/3/14

LSHアルゴリズムを利用した類似ソースコードの検索

川満 直弘[1,a)]　石尾 隆[1,b)]　井上 克郎[1,c)]

概要：本研究では，再利用されたコードの出自を調べるための方法として，与えられたソースファイルに対し，類似するソースファイルを高速に検索する手法を提案する．内容が同一のものだけでなく類似するものを含めて出力することで，再利用の際に変更が加えられているソースファイルにも対応する．また，大量のソースファイルの中から検索を行うため，locality-sensitive hashing を利用することにより，高速な検索を可能にする．この手法によって，再利用したコードの出自を知ることが可能になる．
C 言語で記述された再利用されているライブラリを対象にケーススタディを行った．対象の中にはプロジェクト独自の変更が加えられ，再利用元ファイルと一致しなくなっているものも含まれていた．その結果，92%の精度で 200 プロジェクトの中から再利用元のソースファイルを検出することができた．また，4 ファイルを使用し検索の実行時間を測定したところ，各ファイルについて 1 秒以内で検索を完了した．

1. はじめに

ソフトウェアの開発において，他プロジェクトで開発したソフトウェアの再利用が頻繁に行われている．プロジェクト独自の変更が必要な場合やコンパイル手順を簡単にしたい場合には，再利用元プロジェクトのソースコードをコピーして取り込む，という方法で再利用が行われる．このような再利用には利点がある一方，再利用したコードに含まれる不具合を取り込んでしまう虞がある．再利用したコードを管理するためには再利用したコードがどのプロジェクト，バージョンのものなのかに関する情報が必要である．しかしながら，すべてのプロジェクトにおいてそのような情報が記録されているわけではないことが判明している [20]．

再利用したソースファイルに対し，もしファイルの内容が一致するものが他のプロジェクトに存在するならば，そのプロジェクトに由来する可能性があるといえる．しかし，再利用の際にコードに変更が加えられることがあり，その場合にはソースファイルが単純に一致するかどうかを判定する，という方法では不十分である．出自を求める既存研究として，Inoue ら [8] は Ichi Tracker というシステムを提案した．このシステムは，コード片をクエリとし，コード検索エンジンを利用して検索を行う．また，我々のグループでは，リポジトリ中のソースコードを対象に，バージョンを検出する手法を提案した [12]．この手法では再利用元のリポジトリを用意しなければならない．

本研究では，与えられたソースファイルに対し，類似するソースコードを高速に検索する手法を提案する．検索の高速化のために，locality-sensitive hashing[7] を利用する．提案手法では，まず検索の対象となるソースファイルをデータベースに登録する．この際，LSH アルゴリズムを利用するために，ソースファイルを MinHash[2] の値からなるベクトルに変換する．検索を行う際には，同様に対象のソースファイルをベクトルに変換し，そのベクトルを用いて検索を行う．

手法の評価のために，2 ケースを利用して手法の妥当性を評価した．さらにケーススタディとして実際に再利用されたコードに対し手法を適用し，記録されている再利用元が検出できるか評価を行った．また，手法に要する時間についても評価した．

以降，2 章では関連研究について述べる．3 章では本研究での提案手法について述べ，4 章では本研究においての実装について述べる．5 章では行ったケーススタディについて述べ，6 章で妥当性への脅威について述べる．最後に，7 章を本研究のまとめとする．

2. 関連研究

2.1 ソースコードの再利用

開発者はソフトウェアを開発する際に，他プロジェクトで開発したソフトウェアを再利用しながら開発を行っている．Heinemann ら [6] はオープンソースの Java のプロジェクトを対象に調査を行い，バイナリでの再利用である black-box

[1] 大阪大学大学院情報科学研究科
Osaka University, Suita, Osaka 565-0871, Japan
[a)] n-kawamt@ist.osaka-u.ac.jp
[b)] ishio@ist.osaka-u.ac.jp
[c)] inoue@ist.osaka-u.ac.jp

reuse がソースコードの再利用である whilte-box reuse より優勢であったことを報告した．また，Rubin ら [18] は企業内において，開発したソフトウェアから新しい製品の開発に利用するために，再利用を行っていることを報告している．

再利用をすることによって開発のコストを抑え，ソフトウェアを効率的に開発することが可能になる．再利用されたコンポーネントはそうでないものより不具合が少ないことが Mohagheghi らによって報告されている [15]．

開発者が再利用を行う際にプロジェクト独自の変更が必要な場合，ソースコードを開発中のプロジェクトに取り込む，という方法がとられることがある．例えば，v8monkey[*1] では，APNG(Animated Portable Network Graphics) フォーマットに対応するための修正を libpng[*2] のライブラリに加えている．

ソースコードの再利用は有用である一方，バグやセキュリティに関する脆弱性を抱えているものを使用しないよう，開発者は注意しなければならない．Xia ら [20] はオープンソースのライブラリのコードを再利用しているプロジェクトについて，使用しているライブラリが脆弱性を抱える可能性のあるバージョンであるかどうかを調査した．その結果，調査対象としたプロジェクトのうち，zlib を利用しているプロジェクトでは 31.1%，libcurl を利用しているプロジェクトでは 85.7%，libpng を利用しているプロジェクトでは 92%のプロジェクトが，脆弱性を抱える可能性のあるバージョンを利用していることが判明した．さらに，18.7%のプロジェクトには使用したライブラリについて，どのバージョンを利用したかに関する情報が残っていなかった．加えて，4.9%のプロジェクトではディレクトリ名が変更されたり，再利用したソースコードに他のソースコードが混ぜられ，管理することが難しい状態になっていることが判明した．このような管理のされていないプロジェクトに対して，再利用元を特定することは有用であると考えられる．

ソフトウェアの再利用の検知は，バイナリを対象としたものやソースコードを対象としたものがある．バイナリを対象としたものとして，Davies ら [4], [5] は Java のバイナリから，クラス中のシグネチャを利用する手法を提案した．Sæbjørnsen ら [19] は実行ファイルのバイナリからのコードクローン検出手法を提案した．Qiu ら [17] は，バイナリからライブラリの関数を特定する手法を提案した．

Inoue ら [8] は，Ichi Tracker というシステムを提案した．これは，コードの断片などをクエリとし，そのコード片のクローンを含むソースコードをコード検索エンジンを使用して検索する．システムの出力として見つかったクローンとともに，各クローンについてそのクローンの最終変更日時，どのプロジェクトのコードか，どの程度クエリの内容を含んでいるか，などの情報が得られる．

我々の研究グループ [12] ではリポジトリに含まれているソースコードについて，バージョンを推定する手法を提案した．ソースコードの類似度として最長共通部分列に基づいた類似度 [11] を使用し，もっとも類似度の高いものが再利用元であるという仮定に基づいてバージョンを提示した．この手法には入力として再利用を行ったリポジトリとその再利用元のリポジトリが必要なため，再利用元が不明な場合は利用することができない．

2.2 Locality-Sensitive Hashing による検索

本研究ではソースファイルの再利用を検出するという問題を，クエリとして与えられたファイルに類似したソースファイルを見つける問題の一種ととらえる．大量な検索対象に対する高速化のために locality-sensitive hashing(以下 LSH とも表記) を用いる．LSH は近似最近傍検索や，クラスタリングに用いられるアルゴリズムである [7]．

LSH には複数の適用例がある．Manku ら [14] は，Web でのクローリングにおいて，内容がほぼ重複している Web ページを検出するための手法を提案した．Das ら [3] は，Google News のために協調フィルタリングを使用し，ユーザの行動から記事を推薦する手法を提案した．Brinza ら [1] は，ゲノムワイド関連解析において遺伝子座のクラスタリングのために利用した．Jing ら [10] は，画像検索のために LSH を利用している．

LSH はソースコードにおいても利用されている．Jiang ら [9] らは，類似する部分木を高速に検出する手法をソースコードに用いることでクローンの検出を実現しているが，部分木の特徴ベクトルに対しクラスタリングを行うために LSH を利用した．山中ら [21] は TF-IDF 法を特徴量とし，クラスタリングを行ってタイプ 4 の関数クローン検出の手法を提案したが，高速な検出を行うために特徴ベクトルのクラスタリングに LSH を利用している．

3. 提案手法

本研究では，検索のクエリとして与えられたソースファイルに対し，大量のソースファイルの中からソースコードの内容がクエリに類似するものを検索する手法を提案する．LSH を用いることにより，高速な検索を可能にする．

本手法ではさらに，検索された結果に対して類似度の推定値を提示する．この推定値は効率的に求めることができ，検索結果に含まれる各ソースファイルについて，推定値ではあるがクエリとの類似度を個別に知ることができる．

まず本手法で用いるソースファイル間の類似度を定義し，LSH について説明したのち，提案手法と詳細について説明する．

[*1] https://github.com/zpao/v8monkey
[*2] http://www.libpng.org/pub/png/libpng.html

3.1 ソースファイル間の類似度の定義

本研究で用いるソースファイル間の類似度は，n-gram と Jaccard 係数を利用した類似度である．この類似度を用いる利点として，関数単位の並べ替えなどの影響を過度に受けないという利点がある．また，TF-IDF 法のように，類似度を求める対象の 2 ファイル以外は類似度に影響することはない．具体的な計算方法は，以下のステップからなる．

(1) ソースファイルの字句分割

ソースファイル s からコメントを取り除き，字句の列に分割し，T_s を得る．コメントを取り除くことで，コメント量の大小や，コメントの違いが類似度に影響を与えることを防ぐ．ソースファイル t にも同様の操作を行い，字句列 T_t を得る．

(2) 字句列からの n-gram の抽出

字句列 T_s から長さ n の部分文字列を取り出すことにより，n-gram の多重集合 M_s へ変換する．

たとえば，$n = 3$ のとき，要素列 $e_1, e_2, e_3, ..., e_m$ に対して得られる n-gram は，$\{\{(\$, \$, e_1), (\$, e_1, e_2), (e_1, e_2, e_3), (e_2, e_3, e_4), ..., (e_{m-1}, e_m, \$), (e_m, \$, \$)\}\}$ となる．ただし，要素$は要素列の先頭および末尾の要素に付加するダミーの要素である．先頭及び末尾にダミーの要素を付加することによって，列のすべての要素が n 個の n-gram に含まれる．同様にして，字句列 T_t から多重集合 M_t を得る．

(3) n-gram の多重集合からの集合への変換

多重集合 M_s の各要素に番号を付加することによって集合に S_s に変換する．各要素に付加する番号は，ある要素 e について，変換前の多重集合に要素 e が k 個含まれている場合，各要素に 1 から k の値が付加される．

たとえば，要素 a, b, c からなる多重集合 $\{\{a, a, b, b, b, c\}\}$ をこの手法により集合に変換した場合，得られる集合は $\{(1, a), (2, a), (1, b), (2, b), (3, b), (1, c)\}$ となる．

同様にして，多重集合 M_t から集合 S_t を得る．ただし，付加する番号は M_s に依存せず，M_s と M_t に同じ要素が含まれていたとしても，M_t から S_t に変換する過程で付加する番号は 1 から順に与える．

(4) 集合に対する Jaccard 係数の算出

ソースファイル s, t の類似度を，集合 S_s と S_t を用いて以下の式で算出する．

$$\mathrm{Jaccard}(S_s, S_t) = \frac{|S_s \cap S_t|}{|S_s \cup S_t|}$$

3.2 Locality-Sensitive Hashing

locality-sensitive hashing は類似検索に利用される手法の一つである [7]．本研究では，以下の式で表される関係が成り立つハッシュ関数の族 F を利用して LSH を構成する [16]．

$$\Pr_{h \in F}[h(x) = h(y)] = sim(x, y)$$

ただし，$sim(x, y)$ は x, y 間の類似度であり，$0 \leq sim(x, y) \leq 1$ である．類似度として Jaccard 係数を用いる場合，F として MinHash が利用可能である．

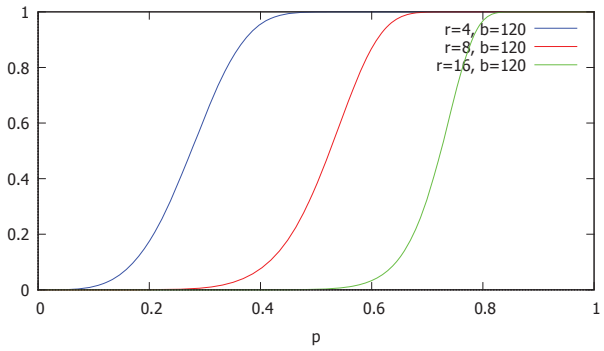

図 1 同じバケットに入る確率
Fig. 1 Probabilities of mapping to the same bucket

MinHash は，集合間の類似度を高速に推定する手法である．集合中の各要素についてハッシュ値を求め，その中の最小値を求める．同様の操作を別の集合に対しても行った時，それら 2 つの最小値が一致する確率は，2 つの集合に対する Jaccard 係数に等しい．

文献 [13] の方法を用いて LSH を構成する．2 集合について考える．ハッシュ関数を $r \times b$ 個用意し，各集合について r 次元の MinHash の値からなるベクトルを b 個求める．2 集合間において，各ハッシュ関数に対応するベクトル同士を比較する．この時，2 集合間での MinHash の値が一致する確率，すなわち Jaccard 係数を p とすると，b 個のベクトルのうち 1 つ以上のベクトルが一致する確率は，$1 - (1 - p^r)^b$ で表される．これらのベクトルをハッシュテーブルのキーとし，ベクトルが一致した場合に同じバケットに入っていると扱うことにより，類似度 p の要素同士が確率 $1 - (1 - p^r)^b$ で同じバケットに入る，ということを実現する．

図 1 に，実際にパラメータの値を与えた際の $1 - (1 - p^r)^b$ のグラフを示す．パラメータは $b = 120$ であり，r については $r = 4, r = 8, r = 16$ の 3 通りである．図に示すとおり，パラメータを調整することで，同じバケットに入る確率を調整することができる．

3.3 LSH を利用したソースファイルの検索

提案手法は，データベース上で LSH を利用した検索を行う．データベースを用いることにより，検索対象のファイル数が大きく，必要なデータすべてを主記憶に収めることができない場合にも対応する．

手法は 2 つのステップからなる．登録ステップでは複数のソースファイルを入力し，データベースに登録する．検索ステップでは，クエリとなるソースファイル q を与え，検索結果である類似ファイルの集合と，それぞれの結果のファイルとソースファイル q との類似度の推定値を出力

する．

3.3.1 検索対象のデータベースへの登録

このステップでは，検索の対象となるソースファイルを入力とし，データベースに登録する．

まず LSH を構成する MinHash において使用するハッシュ関数を $R \times B$ 個用意する．R, B は LSH の検索パラメータ r, b の上限となる．用意するハッシュ関数は番号の付加された n-gram からハッシュ値への関数であり，それらを h_i とする．ただし，$1 \leq i \leq R \times B$ である．次に検索の対象となるソースファイルをデータベースに登録するが，登録する情報はソースファイル 1 件に対し，各ハッシュ関数から求められる MinHash の値もデータベースに登録する．h_i を用いてソースファイルから MinHash の値を求める関数を m_i と表す．ソースファイル f に対し，番号の付加された n-gram を g，ソースファイルから番号の付加された n-gram の集合への関数を G とすると，$m_i(f) = \min_{g \in G(f)} h_i(g)$ である．m_i を利用し，ソースファイル f に対し，$m_1(f), m_2(f), \ldots, m_{R \times B}(f)$ を同時に登録する．

この登録した MinHash の値を利用して LSH を構成し検索を行うため，検索の高速化のためにインデックスを作成する．R 個の MinHash の値を一組とし，B 組それぞれに対してインデックスを作成する．すなわち，$(m_1(f), m_2(f), \ldots, m_R(f))$ を 1 組としてインデックス I_1，$(m_{R+1}(f), m_{R+2}(f), \ldots, m_{R \times 2}(f))$ を 1 組としてインデックス I_2，というようにインデックスを構成し，合計 B 組のインデックスを構成する．

3.3.2 類似ソースファイルの検索

このステップでは，与えられたソースファイルに類似するソースファイルを，データベースから検索する．検索には，クエリとなるソースファイルに加え，LSH のパラメータ $1 \leq r \leq R, 1 \leq b \leq B$ を与える．与えられたソースファイルに対して MinHash の値を求め，その値から検索を行う．$1 \leq i \leq b$ について，インデックス I_i を用いて，

$$S_i(q) = \{f | \forall j \in [1, r]. m_{j+R \times (i-1)}(q) = m_{j+R \times (i-1)}(f)\}$$

なる S_i を求める．この集合 S_i の和集合，すなわち $\bigcup_{1 \leq i \leq b} S_i(q)$ が検索の結果，つまりクエリと類似するソースファイルの集合である．

また，検索結果に含まれるそれぞれのソースファイルに対して，クエリとの類似度の推定値を併せて出力する．類似度は最尤推定を用いて推定する．

検索結果中のある類似ソースファイルについて，集合 $S_1, S_2, ..., S_b$ 中の x 個の集合に含まれているとする．類似度が p で表される時，r 次元のベクトルについて x 個のベクトルが一致し，$b - x$ 個不一致となる確率，p に対する尤度関数 $L(p)$ は

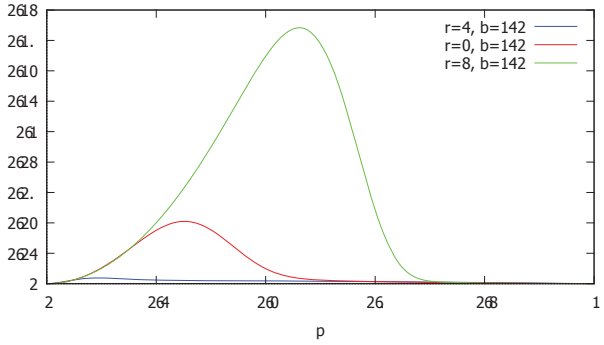

図 2 $b = 120$ の場合の推定量の平均二乗誤差
Fig. 2 MSEs of the estimator when $b = 120$

$$L(p) = (p^r)^x (1 - p^r)^{b-x} \binom{b}{x}$$

で表される．したがって，類似度 p に対する最尤推定量 $\hat{p}$ は，$0 \leq p \leq 1$ であることを踏まえると，

$$\hat{p} = \sqrt[r]{\frac{x}{b}}$$

となる．ここに示す通り，類似度の推定値がとりうる値は，$x \geq 1$ の場合，b 通りに限定される．$r = 1$ の場合は推定値としてとりうる値の間隔は等しく，MinHash を用いて Jaccard 係数を推定する方法そのものである．一方，$r > 1$ の場合，推定値のとりうる値の間隔は 1 に近づくにつれ狭くなり，1 に近いほど細かく，0 に近いほど粗くなる．

推定量の偏り $B(\hat{p})$，分散 $Var(\hat{p})$，平均二乗誤差 $MSE(\hat{p})$ は

$$B(\hat{p}) = \sum_{x=0}^{b} \sqrt[r]{\frac{x}{b}} (p^r)^x (1 - p^r)^{b-x} \binom{b}{x} - p$$

$$Var(\hat{p}) = \sum_{x=0}^{b} \left(\sqrt[r]{\frac{x}{b}} \right)^2 (p^r)^x (1 - p^r)^{b-x} \binom{b}{x} - \left(\sum_{x=0}^{b} \sqrt[r]{\frac{x}{b}} (p^r)^x (1 - p^r)^{b-x} \binom{b}{x} \right)^2$$

$$MSE(\hat{p}) = Var(\hat{p}) + (B(\hat{p}))^2$$

となる．

例として図 **2** にパラメータが $b = 120$ であり，$r = 2, r = 4, r = 8$ の場合の推定量の平均二乗誤差について p の値を変えながら図示した．横軸は p の値であり，縦軸は平均二乗誤差である．p の値が 1 付近に近づくと平均二乗誤差が 0 に近づくことを示している．

4. 実装

4.1 類似度について

手法の対象を C 言語のソースファイルとし，類似度で用いる n-gram の n の値として，$n = 3$ とした．

異なるファイル間での類似度と，同じファイルの別バー

表 1　データセット
Table 1　Dataset

パッケージ	バージョン#1	バージョン#2
original-awk	2012-12-20	2011-08-10
mgcv	1.8-4	1.7-12
seaview	4.3.1	4.5.3.1
fglrx-installer	8.960	15.200
fglrx-installer-updates	15.200	8.960
foreign	0.8.62	0.8.48
libhdhomerun	20140604	20120128
r-cran-eco	3.1-4	3.1-6
r-cran-xml	3.6-2	3.98-1.1
rt-tests	0.89	0.83

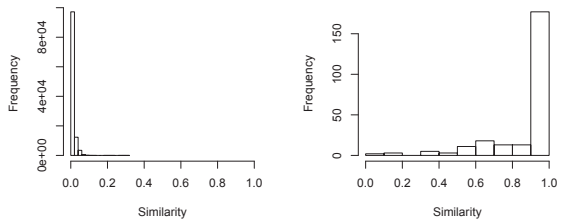

図 3　類似度の分布
Fig. 3　Distributions of similarity

ジョン間での類似度の違いについて調査した．調査には Ubuntu の apt で手に入るソースコード，10 パッケージ，505 ファイルを対象とした．**表 1** に示す各パッケージの 2 バージョンの orig.tar.gz から得たソースコードを利用した．

対象となったファイルすべての組み合わせの間で類似度を求めた結果を**図 3** に類似度に対する頻度として示す．左図はパッケージが異なり，かつファイル名も異なるファイル間での類似度を表しており，右図は同一パッケージ，同一パス，同一ファイル名であり別の tar ファイル，つまりバージョンが異なるファイル間の類似度を表している．左図には 113735 件の類似度が含まれており，右図には 245 件の類似度が含まれている．ただし，右図中で 101 件について，類似度が 1.0 であった．この結果より，3-gram を使用した類似度は二つのファイルが同一ファイルの別バージョンか否かの判別に有効と考えられる．

4.2　ハッシュ関数について

提案手法では番号の付加された n-gram に対するハッシュ関数を複数用意する必要がある．今回は，簡便な方法として以下のような実装を行った．

番号の付加された 3-gram(num, t_1, t_2, t_3) に対するハッシュ関数 h_i によるハッシュ値は，

$$(((num \times 65537) + t_1) \times 65537 + t_2) \times 65537 + t_3) \times a_i + b_i$$

によって求める．ただし，num は 3-gram に付加された番号，t_1, t_2, t_3 は n-gram の要素，つまり字句に対する Java の hashCode メソッドの値，a_i, b_i はそれぞれハッシュ関数ごとに用意する定数である．演算は 64bit で行われ，演算中のオーバーフローは下位 64bit が残される．a_i は 64bit 乱数と 1 とのビット論理和をとることによって，b_i は 64bit 乱数によって用意した．

5.　ケーススタディ

5.1　検索結果および類似度の推定値の評価

実際の検索において，クエリに対する類似度の高いものが検索の結果に表れ，類似度の低いものが結果に表れないということを確認するために，libpng および libcurl に対して実験を行った．

5.1.1　libpng の png.c

libpng のリポジトリ [*3] 中に含まれる，すべてのリビジョンの c ファイル，h ファイルをデータセットとし，実験を行った．具体的には，`git rev-list --all --objects` によって得られたリストの中から，.c ファイルおよび.h ファイルを対象とした．対象のファイル数は 20977 である．libpng に含まれ，`v1.6.0` のタグがついているコミットのファイル png.c をクエリとして検索を行った．検索のパラメータは，$r = 8, b = 120$ である．

検索の結果に表れたファイル数は 716 であり，これはデータセット全体の 3.4%である．検索の結果に表れなかったものの中での類似度の最高値は 0.482 である．また，検索に表れたものの中で類似度の最低値は 0.598 である．

図 4 に，検索結果中のファイルとクエリ間の類似度について，推定値と実際の値との関連を示す．x 軸は類似度の推定値，y 軸は実際の類似度を表す．

5.1.2　libcurl の url.c

libcurl のリポジトリ [*4] に含まれる，すべてのリビジョンの c ファイル，h ファイルをデータセットとし，実験を行った．対象のファイル数は 23273 である．libcurl に含まれ，`curl-7_47_0` のタグがついているコミットの url.c をクエリとして検索を行った．

検索の結果に表れたファイル数は 374 ファイルであり，これはデータセット全体の 1.6%である．検索の結果に表れなかったものの中での類似度の最高値は 0.578 である．また，検索に表れたものの中での類似度の最低値は 0.572 である．

5.2　ファイルの出自検索能力の評価

類似度の推定値を用いて再利用元のプロジェクト，ファイル，バージョンを特定することができるかケーススタディを行った．ケーススタディの対象は，v8monkey[*5] 中

[*3] `git://git.code.sf.net/p/libpng/code`
[*4] `https://github.com/bagder/curl.git`
[*5] `https://github.com/zpao/v8monkey.git`

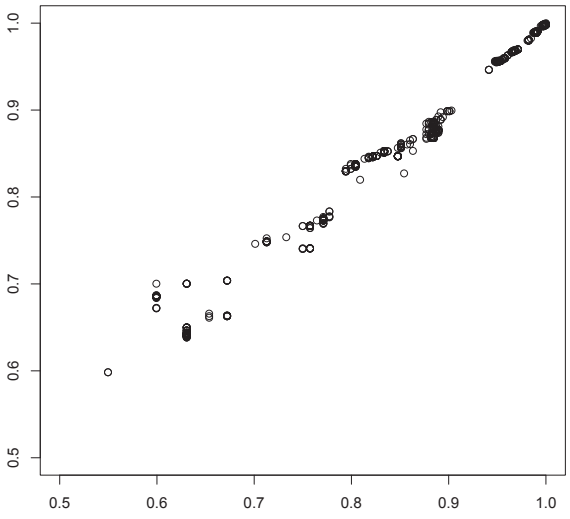

図 4　png.c をクエリとした際の類似度の推定値と実際の類似度
Fig. 4　Estimated values of similarity and actual value of similarity when the query is png.c

に含まれる libpng のソースコードである．v8monkey には複数のコミットメッセージに，libpng のどのバージョンに更新したかが記録されており，複数のソースコードに Animated Portable Network Graphics フォーマットに対応するためと思われる変更が行われている．

検索の対象として，libpng プロジェクトを含む 200 プロジェクトのリポジトリを用意した．これらのプロジェクトは，GitHub において 2016 年 1 月 31 日に”lib language:c”というクエリを用いて検索を行い，その検索結果の上位 200 件のリポジトリを同年 2 月 1 日に得たものである．

実験の手順として，まず検索対象のプロジェクト中の，履歴中に含まれるすべての.c ファイルおよび.h ファイルをデータベースに登録した．登録された件数は 567113 件である．次に v8monkey のコミットのうち libpng のどのバージョンにアップデートしたかが明記されているコミットを列挙した．この条件を満たすコミットは計 14 件見つかった．それらのコミットで新たに加えられた，もしくは変更されたファイルのうち，modules/libimg/png 以下に含まれるファイルをクエリとし，検索を行った．ただし，以下のファイルについては評価対象から除外した．

- コメントのみからなるソースファイル．
- mozpngconf.h．このファイルは libpng に存在しないファイルであり，v8monkey の開発者が追加したものと思われる．

対象となったファイルは，18 ファイル，延べ 197 件である．このうちの 57 件については，再利用元のファイルと内容が一致し，それ以外の 140 件は再利用元とファイルの内容が一致しなかった．検索のパラメータは，$r = 8, b = 120$ とした．

表 2　分類の結果
Table 2　Categorized results

	基準 1		
	満たす	満たさない	計
基準 2 を満たす	177	10	187
基準 2 を満たさない	5	5	10
計	182	15	197

検索の結果，すべての結果について再利用元の該当するバージョンのファイルが検索結果に表れた．クエリ 197 件に対し，検索結果に表れたファイルの数の合計は，120001 件であった．各クエリで検索結果に表れるファイル数は，最低 167 件，最高 1031 件，平均 609 件であった．検索結果には libpng のファイル以外に，libpng を利用している 3 プロジェクト計 97 件含まれていた．

さらに，各クエリについて，類似度の推定値が検索結果に表れる類似度の推定値の中での最高値になるかどうかを調査した．結果の理解のために，2 つの基準から結果を 4 つに分類した．

基準 1　再利用元のバージョンの類似度の推定値が，検索結果に表れる類似度の推定値の中での最高値になる
基準 2　検索結果に表れたファイルの中で，再利用元のバージョンの類似度の真の値が，検索結果に表れた類似度の真の値の中での最高値になる

基準 2 の対象が検索結果に表れたファイルに限られているのは，時間の制約上データベースすべてのファイルに対し 197 件のクエリとの類似度の真の値を求めることができないためである．この分類結果を**表 2** に示す．

クエリとなる対象の 197 件のうち 182 件は基準 1 を満たす，つまりクエリと記録されている再利用元との類似度の推定値が，検索結果に表れる類似度の推定値の最高値と等しくなった．基準 1 を満たすクエリについて，各クエリの検索結果中で類似度の推定値が最高値をとるファイル数は，最低 1 件，最高 56 件，平均 11 件であった．さらに，基準 1 を満たすクエリの各検索結果に含まれる類似度の推定値が最高値をとるファイルを対象に，コメント，フォーマットの違いを無視して重複を取り除いた数を求めた．ただし，重複の検出には md5 チェックサムを用いた．その結果，重複を取り除いたファイル数は，最低 1 件，最高 18 件，平均 2.3 件となった．

基準 1 を満たさない 15 件のクエリについては，検索結果ごとに類似度の推定値を降順に並べたとすると，記録されているバージョンの順位は，2 位から 14 位，平均して 5.3 位であった．

基準 2 を満たしたクエリは 197 件中 187 件であった．残りの 10 件は，再利用元のバージョンとクエリとの類似度が真の値でも最高値をとらないものである．したがって，

表 3　検索クエリとして用いるソースファイル
Table 3　Source files as search queries

ID	ファイル名	ベクトルの一致数	検索結果の件数	LOC	サイズ [B]
0	mozpngconf.h	0	0	525	27,513
1	pngwrite.c	14,842	503	1,590	51,254
2	pngwtran.c	26,355	453	572	17,279
3	pngrtran.c	41,179	818	4,296	147,369

類似度に真の値ではなく推定値を用いたことによる影響の大きさは，基準 2 を満たす 187 件中のクエリのうち基準 1 を満たさない 10 件，5.3%であると考えられる．

5.3　パフォーマンスの評価

提案手法のパフォーマンスについて評価した．実行環境の CPU は，Intel(R) Xeon(R) CPU E5-2620 が 2 プロセッサであり，メモリは 64GB，OS は Windows(R) 7 Professional である．また，データベースや検索クエリのファイルはすべて SSD に保存されている．

評価に使用するデータベースは，5.2 と同様である．

5.3.1　登録に対する評価

データベースは 200 プロジェクトから合計 567113 件のファイルを登録したものである．登録のために Git のリポジトリから一度すべてのファイルを取り出してディレクトリに保存し，その後データベースへの登録を行った．取り出したファイルすべてをディレクトリから登録するために要した時間は 230 分であった．

5.3.2　検索に対する評価

検索のクエリとして，v8monkey のコミット 3a04be0690dff135ec42784557fedbf6c572cd22 の modules/libimg/png 以下に含まれている 4 ファイルを使用した．**表 3** にそれらについて示す．検索に使用するベクトルのデータベース中で一致する数が多いほど実行時間がかかることが予想されるため，その数が固まらないように 5.2 で用いたクエリの中から選定した．また，ベクトルの一致数が 5.2 で用いたクエリの中で最大になるものを含めた．

検索を行った際にデータベースの内容の一部を OS がメモリにキャッシュするため，後続のクエリの速度が先行するクエリの検索の影響を受ける．このことを考慮し，4 つのファイルの検索の順序を全通り，つまり $4! = 24$ 通りの順序で検索を行い，その実行時間を測定した．24 通りのそれぞれの検索の間で OS のキャッシュをクリアし，同一ファイルの検索がキャッシュに影響されないようにした．

まず検索クエリとなるソースファイルをディスクから読み出し，MinHash のベクトルに変換するまでの時間について集計した．その結果，中央値は ID0 から 3 までそれぞれ 47ms, 94ms, 63ms, 219ms であった．

次に，データベースからの検索にかかる時間を**図 5** に示す．横軸は ID を，縦軸は実行時間をミリ秒で表している．ここで，ID は 4 で除した値が表 3 に示す ID を，剰余がそのクエリが何番目に検索が行われたかを示す．例えば，ID が 4 の倍数になっているものはキャッシュをクリアしてから最初に検索された場合の結果を示しており，剰余が 3 である場合は他の 3 クエリを検索した後に該当するクエリを検索した場合の結果を示している．

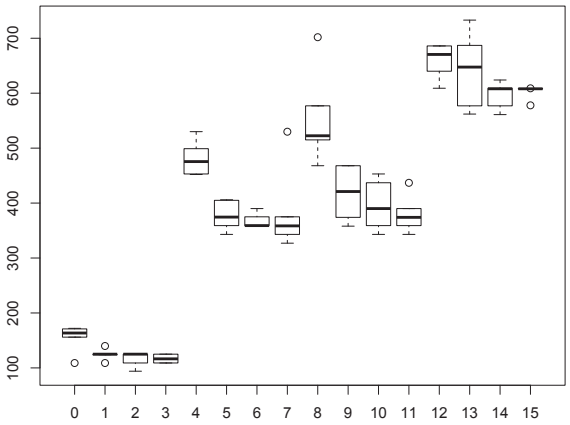

図 5　データベースからの検索の実行時間
Fig. 5　Execution time of searching from the database

6.　妥当性への脅威

本研究の実装では 64bit のハッシュ長を用い，ハッシュ関数の振る舞いが完全にランダムにふるまうことを仮定し，その上でハッシュ値の衝突する確率が非常に低いとして衝突した場合の影響を考慮していない．しかしながら，実際に衝突が発生したのか，また衝突が発生していた場合の結果への影響についての検証をしていない．また，ハッシュ関数の偏りがケーススタディの結果に影響を与えた可能性がある．

ケーススタディで検索のクエリとして用いたプロジェクトは，1 プロジェクトのみである．このため，他のプロジェクトや，他の言語が利用されているプロジェクトに対し，この手法を適用した場合の有効性は本ケーススタディで確認した有効性とは異なる可能性がある．

5.2 で結果を分類するために設けた基準 2 において時間の制約上，類似度を求める対象を検索結果に表れたファイルのみに限定した．しかしながら，検索結果に表れなかったファイルの中に，クエリとの類似度が検索結果に表れたファイル以上になるものが存在する可能性がある．

7.　まとめ

本研究では，検索のクエリとして与えられたソースコードのファイルに対し，大量のソースコードのファイルの中

からソースコードの内容がクエリに類似するものを検索する手法を提案した．locality-sensitive hashing を用いることにより，高速な検索を実現した．ケーススタディでは，別コミットに含まれるファイルを合わせた延べ 197 ファイルを対象とし検索を行い，そのうちの全 197 件について，コミットメッセージに記録されていた再利用元を検索することができた．また，その中でも 182 件については記録されていた再利用元の類似度の推定値が，検索結果中の推定値の最高値となった．さらに，検索時間については 1 件につき 1 秒以内で検索を完了した．

提案手法は，検索のために 1 つのソースファイルを指定して検索を行う必要がある．今後の課題としては，プロジェクト全体のソースコードを与え，その中から再利用とそうでないものを判別することがあげられる．また，提案手法では 1 つのファイルについての情報しか利用していないが，入力として複数ファイルを与え，それらの情報を合わせて利用することで手法の精度の向上が期待できる．

謝辞 本研究は JSPS 科研費 25220003 の助成を受けたものです．

参考文献

[1] Brinza, D., Schultz, M., Tesler, G. and Bafna, V.: RAPID detection of gene-gene interactions in genome-wide association studies, *Bioinformatics*, Vol. 26, No. 22, pp. 2856–2862 (2010).

[2] Broder, A. Z.: On the resemblance and containment of documents, *Compression and Complexity of Sequences 1997. Proceedings*, pp. 21–29 (1997).

[3] Das, A. S., Datar, M., Garg, A. and Rajaram, S.: Google News Personalization: Scalable Online Collaborative Filtering, *Proceedings of the 16th International Conference on World Wide Web*, WWW '07, New York, NY, USA, ACM, pp. 271–280 (2007).

[4] Davies, J., German, D. M., Godfrey, M. W. and Hindle, A.: Software Bertillonage: Finding the Provenance of an Entity, *Proceedings of the 8th Working Conference on Mining Software Repositories*, pp. 183–192 (2011).

[5] Davies, J., German, D. M., Godfrey, M. W. and Hindle, A.: Software Bertillonage: Determining the provenance of software development artifacts, *Empirical Software Engineering*, Vol. 18, pp. 1195–1237 (2013).

[6] Heinemann, L., Deissenboeck, F., Gleirscher, M., Hummel, B. and Irlbeck, M.: On the Extent and Nature of Software Reuse in Open Source Java Projects, *Proceedings of the 12th International Conference on Software Reuse*, Lecture Notes in Computer Science, Vol. 6727, pp. 207–222 (2011).

[7] Indyk, P. and Motwani, R.: Approximate Nearest Neighbors: Towards Removing the Curse of Dimensionality, *Proceedings of the Thirtieth Annual ACM Symposium on Theory of Computing*, STOC '98, New York, NY, USA, ACM, pp. 604–613 (1998).

[8] Inoue, K., Sasaki, Y., Xia, P. and Manabe, Y.: Where does this code come from and where does it go? – Integrated code history tracker for open source systems –, *Proceedings of the 34th International Conference on Software Engineering*, pp. 331–341 (2012).

[9] Jiang, L., Misherghi, G., Su, Z. and Glondu, S.: DECKARD: Scalable and Accurate Tree-Based Detection of Code Clones, *Proceedings of the 29th International Conference on Software Engineering*, ICSE '07, Washington, DC, USA, IEEE Computer Society, pp. 96–105 (2007).

[10] Jing, Y. and Baluja, S.: VisualRank: Applying PageRank to Large-Scale Image Search, *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, Vol. 30, No. 11, pp. 1877–1890 (2008).

[11] Kanda, T., Ishio, T. and Inoue, K.: Extraction of product evolution tree from source code of product variants, *Proceedings of the 17th International Software Product Line Conference*, Tokyo, Japan, ACM, pp. 141–150 (2013).

[12] Kawamitsu, N., Ishio, T., Kanda, T., Kula, R. G., De Roover, C. and Inoue, K.: Identifying Source Code Reuse across Repositories using LCS-based Source Code Similarity, *Proceedings of the 14th International Working Conference on Source Code Analysis and Manipulation*, pp. 305–314 (2014).

[13] Leskovec, J., Rajaraman, A. and Ullman, J. D.: *Mining of Massive Datasets*, chapter 3, Cambridge University Press (2014).

[14] Manku, G. S., Jain, A. and Das Sarma, A.: Detecting Near-duplicates for Web Crawling, *Proceedings of the 16th International Conference on World Wide Web*, WWW '07, New York, NY, USA, ACM, pp. 141–150 (2007).

[15] Mohagheghi, P., Conradi, R., Killi, O. M. and Schwarz, H.: An empirical study of software reuse vs. defect-density and stability, *Proceedings of the 26th International Conference on Software Engineering*, pp. 282–291 (2004).

[16] Moses S., C.: Similarity Estimation Techniques from Rounding Algorithms, *Proceedings of the Thiry-fourth Annual ACM Symposium on Theory of Computing*, STOC '02, New York, NY, USA, ACM, pp. 380–388 (2002).

[17] Qiu, J., Su, X. and Ma, P.: Library Functions Identification in Binary Code by Using Graph Isomorphism Testings, *Proceedings of the 22nd IEEE International Conference on Software Analysis, Evolution, and Reengineering*, pp. 261–270 (2015).

[18] Rubin, J., Czarnecki, K. and Chechik, M.: Managing Cloned Variants: A Framework and Experience, *Proceedings of the 17th International Software Product Line Conference*, pp. 101–110 (2013).

[19] Sæbjørnsen, A., Willcock, J., Panas, T., Quinlan, D. and Su, Z.: Detecting Code Clones in Binary Executables, *Proceedings of the 18th ACM International Symposium on Software Testing and Analysis*, ACM, pp. 117–128 (2009).

[20] Xia, P., Matsushita, M., Yoshida, N. and Inoue, K.: Studying Reuse of Out-dated Third-party Code in Open Source Projects, *JSSST Computer Software*, Vol. 30, No. 4, pp. 98–104 (2013).

[21] 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).