論　文

軽量なデータ構造を利用したソフトウェア進化履歴の高速な復元手法

伊藤　薫[†a)]　石尾　隆[†,††]　神田　哲也[†]　井上　克郎[†]

Efficient Method to Recover Software Evolution History with Lightweight Data Structure

Kaoru ITO[†a)], Takashi ISHIO[†,††], Tetsuya KANDA[†], and Katsuro INOUE[†]

あらまし　ひとたびソフトウェアプロダクトがリリースされると，そこから派生した個別のソフトウェアプロダクトが多数開発される．それらのソフトウェアプロダクトはお互いに派生関係をもち，大部分は共通したソースコードをもつ．このようなソースコードの再利用は開発の効率化や品質向上に効果があるが，再利用元の脆弱性や欠陥なども取り込んでしまう問題がある．そのような脆弱性や欠陥の修正を派生関係をもつソフトウェアプロダクト群に対して効果的に行うためには，正確な派生関係の管理が必要である．この問題に対する既存研究として，ソフトウェアプロダクトに含まれるソースファイル同士を比較し，その類似度から派生関係を復元する手法が提案されている．しかしソースファイルの相互比較に多大な時間を費やすため，長期間に渡って開発されているソフトウェアプロダクトの大規模な集合に対しては，実用的な実行時間で分析できない場合がある．そこで本研究では，ソフトウェアプロダクトやソースコードを軽量なデータ構造で表すことで，その類似度計算を高速化し，より大規模なソフトウェアプロダクトの集合からも派生関係を復元する手法を提案する．九つのデータセットを用いた評価実験の結果，提案手法が既存手法と同程度の精度であることと，計算時間については最大で1,848 倍，中央値で 127 倍高速であることを確認した．既存手法では 3 日間でも派生関係を分析できなかった大規模なデータセットでも，提案手法は最短で 8 分程度で分析を完了する．

キーワード　b-bit MinHash，Linear Counting，進化履歴分析，再利用分析

1. まえがき

ソフトウェアプロダクトの開発において，機能が類似する既存のソフトウェアプロダクトを再利用することは一般的である [1]．その際，開発者は再利用したソフトウェアプロダクトに含まれる欠陥の修正や，機能の追加などを開発中のソフトウェアプロダクトに施す．このとき，開発したソフトウェアプロダクトと，そのソフトウェアプロダクトに再利用されたソフトウェアプロダクトは派生関係にある．派生関係をもつソフトウェアプロダクト同士は，多くの場合，共通のプログラム要素を多数含む．

派生関係について正しく管理されていれば，あるソフトウェアプロダクトで欠陥を修正したりソースコードの品質を向上させたりした場合に，そのソフトウェアプロダクトからの派生関係を辿ることで，同様の変更を適用できる共通の要素をもつ他のソフトウェアプロダクトを特定することが容易となる．野中らは，ある企業で開発された組み込みソフトウェアの複数のソフトウェアプロダクトについて保守記録を分析し，ある修正を他のソフトウェアプロダクトにも適用するリアクティブな欠陥修正が段階的に行われていること，そしてそれが全ての欠陥修正の 40% 以上を占めていたことを報告した [2].

しかし，実際には開発者はしばしば派生関係の管理なしにソフトウェアプロダクトを再利用する．Hemel らは Linux カーネルへの欠陥修正がそれぞれの派生ソフトウェアプロダクトごとに個別に行われており，それらの修正が共有されていないことを報告している [3]．このような問題が起きる一つの要因としては，複数の開発組織にまたがるソフトウェアプロダクトの再利用がある．また，Rubin らは，ソースコードの版管理システムが派生関係の管理には機能不足であることを指

† 大阪大学大学院情報科学研究科，吹田市
Graduate School of Information Science and Technology, Osaka University, 1–5 Yamadaoka, Suita-shi, 565–0871 Japan
†† 奈良先端科学技術大学院大学先端科学技術研究科，生駒市
Graduate School of Science and Technology, Nara Institute of Science and Technology, 8916–5 Takayama-cho, Ikoma-shi, 630–0192 Japan
a) E-mail: ito-k@ist.osaka-u.ac.jp
DOI:10.14923/transinfj.2020JDP7080

摘している [4].

派生関係をソフトウェアプロダクトの集合から復元するために，既存研究では，直接の派生関係にあるソフトウェアプロダクトほどソースファイルの類似度が高いと仮定し，ソースファイルの類似度に基づく派生関係の推定が行われている [5]．具体的には，比較するソフトウェアプロダクト間でソースファイルの全ての組について最長一致部分列 (Longest Common Subsequence : LCS) を用いた類似度を計算し，それらを合計することでソフトウェアプロダクト間の類似性を計算する．その値をもとに全域木を作成することで最も類似するソフトウェアプロダクト同士を結び付け，更に，比較する二つのソフトウェアプロダクト間での LCS から得られた差分を元に，ソフトウェアプロダクトの派生順序を判定する．しかし，LCS の計算に基づく類似度の分析に時間がかかるため，大規模なソフトウェアプロダクトの集合には実用的な時間で適用できない．ソースファイル同士を個別に比べるのではなく，ファイル圧縮アルゴリズムを用いて二つのソフトウェアプロダクトに含まれるソースファイル集合に対する情報距離を計算する手法 [6] も提案されているが，ファイルの圧縮処理に時間を要するため，これも同様に大規模なソフトウェアプロダクトの集合には実用的な時間で適用できない．

そこで本研究では，ソフトウェアプロダクト同士の比較に軽量なデータ構造を用いることで高速化を図り，より大規模なデータセットに適用可能にする手法を提案する．軽量なデータ構造には，それぞれ異なる性質をもつ *b*-bit MinHash 法と Linear Counting 法を独立に利用し，それぞれのデータ構造の場合で構築した系統樹の精度と実行性能を既存手法 [5] と比較する．

一方の *b*-bit MinHash 法は，既存研究 [7] でソースファイル単位の比較に適用できることが示されており，本研究ではそれをソフトウェアプロダクト間のソースコードの再利用量の計算に適用する．既存研究では，*b*-bit MinHash 法を用いたソースファイルの比較を，ソフトウェアプロダクト中の一部分と，他方のソフトウェアプロダクト全体を比較するために利用していたが，本研究では，二つのソフトウェアプロダクト全体同士を比較するために利用する．もう一方の Linear Counting 法は，集合を固定長のビット列で表現し，集合に含まれるユニークな要素の量を推定する手法である．一つのソフトウェアプロダクトを一つの集合表現に変換することで，空間計算量を抑えながら，ソフトウェアプロダクト間の共通集合の基数を高速に計算する．これは，圧縮アルゴリズムによって共通要素を取り出す Hayase らの手法 [6] と発想としては近く，圧縮アルゴリズムによって得られる情報距離を，Linear Counting 法で得られるビットベクトルの類似度に置き換えたものといえる．この方法ではソフトウェアプロダクト全体での比較を一度のビット列の比較で行うことができ，既存手法だけでなく *b*-bit MinHash 法と比較しても大きな高速化が見込まれる．ただし，ソフトウェアプロダクトを一つの集合で表した場合，ソースファイル単位での情報が欠けてしまい，精度が低下してしまう可能性がある．本研究では，二つのデータ構造を用いることで，ソフトウェアを表す粒度を変えた場合に，高速化と精度の低下がどのようなトレードオフとなっているかを調査する．

本研究の主な貢献は以下のとおりである．

- 既存の軽量なデータ構造を用いた高速な集合間の比較技術である *b*-bit MinHash 法と Linear Counting 法をソースコード間の類似度計算に応用し，ソフトウェア全体の再利用量を求める手法を提案した．
- 導入した軽量なデータ構造を用いた高速な集合間の比較手法により処理時間を大幅に高速化した．既存手法 [5] では数分から十数時間かかっていたところ，提案手法では数秒から数分程度になった．
- 高速化の結果，より大規模なデータセットに対して系統樹の復元手法を適用可能になった．UNIX の開発履歴データ [8] は，既存手法では現実的な時間での解析は不可能であったが，提案手法では最短で 8 分程度で解析が可能となった．

以降，**2.** では研究の背景について述べ，**3.** では提案手法を詳述する．**4.** ではデータセットに本手法を適用することで評価を行い，**5.** で妥当性への脅威について述べる．最後に，**6.** でまとめを述べる．

2. 背　　景

2.1 ソフトウェア進化

ソフトウェアは開発が進むにつれ機能追加や修正などにより進化・派生することが知られており，進化履歴からは様々な情報を読み取ることができる．例えば Manabe らは，オープンソースソフトウェアが更新に伴いライセンスを変えていくことを報告している [9]．Hotta らは，ソフトウェアが進化するにつれ，重複している機能と重複していない機能で，変更の頻度は重複している機能の方が多いが，統計的に優位な差はな

いことを報告した [10]．Mondal らは，マイクロクローンと呼ばれる小規模なソースコードの複製について，ソフトウェアが進化する中で必要な修正が見逃される確率を分析，報告した [11]．

ソフトウェアの進化履歴は，このような分析を行うための重要な情報であるが，長期間開発されているソフトウェアについては，版管理システムを導入する以前など，全体の履歴が分からない場合がある．そこで，今までにソフトウェアの進化履歴自体を復元する研究がなされている．Spinellis は，Unix がどのように進化したのか分析するため，24 個のスナップショットと現在利用されている Git リポジトリのデータを組み合わせて，45 年間の Unix システムの開発履歴を単一の Git リポジトリとして構築した [8]．Kanda らは，ソフトウェアプロダクトの集合について，それぞれのソフトウェアプロダクトに含まれるソースファイル間の LCS に基づいてその派生関係を自動で再構築した [5]．Hayase らは，Kanda らの手法をコルモゴロフ複雑性を用いて拡張し，派生関係を再構築した [6]．

既存の進化・派生によって生まれてきたソフトウェアプロダクトをプロダクトラインとして整理することができれば，今後のソフトウェアの進化・派生を計画的に，また効率的に行うことができる．Duszynski らは，ソースコードツリーを様々な粒度で比較することで，派生関係にあるソフトウェアプロダクト同士の共通性について分析する，N-way Diff というツールを提案している [12]．

2.2 高速な集合の比較手法

ソフトウェアの分析においては，ファイル同士の比較を高速に行うことが重要となる．Kawamitsu らはファイル同士の LCS の長さを用いて比較しているが，その計算には両方のファイルの長さの積に比例した時間が必要であり，様々な最適化を行った状態でも，合計数千万行のリポジトリの組の分析に最大で 4 時間程度かかることを報告している [13]．Kanda らはソフトウェアプロダクトの集合に対してその派生関係を自動で復元したが，最適化を行っても LCS を用いた類似度計算は長時間となり，合計 8,000 万行程度のソフトウェアの集合に対して 1 日程度の処理時間がかかることを報告している [5]．

ソフトウェアから類似したコード片の組を検出するコードクローン検出技術では，ソフトウェアを比較する際に，類似度の計算を高速化する方法として，比較すべき候補を事前に効率良く絞り込む手法が適用されている．Jiang らは，ソースコードから作成した木構造の断片について，局所性鋭敏型ハッシュ (LSH) を用いてクラスタリングすることで高速にコードクローンを検出する手法を提案した [14]．また，横井らは，ソースコードから作成した TF-IDF ベクトルについて，cross-polytope LSH を用いてクラスタリングし，コードブロック単位でのコードクローンを検出する手法を提案した [15]．Sajnani らは，転置インデクスを使って比較すべき候補を絞ることで，効率的にコードクローンを検出する SourcererCC を提案した [16]．

しかし，これらの手法は，比較する二つのソースファイル中の互いに類似したソースコード片の量が少ないことを仮定したものであり，派生関係をもつソフトウェアプロダクト同士に含まれるソースファイルのような，大部分が一致しているソースコードの集合の比較には適さない．そのため，小さな差異に鋭敏な高速化手法を用いる必要がある．本研究では，そのような性質をもつ二つの手法，b-bit MinHash 法 [17] という LSH の一種と，アクセスログの解析手法として利用されている Linear Counting 法 [18] により，軽量なデータ構造として集合を表現することでソースコード間の類似度を高速に計算する．これらの手法を用いることで，提案手法は大部分が一致しているようなソースコードの集合を比較し，その差異を分析することを可能としている．

2.2.1 b-bit MinHash 法を用いた集合の比較

集合間の類似度の一つである Jaccard 係数 [19] には，その推定値を高速に求める MinHash 法 [20], [21] が提案されている．また，MinHash 法を拡張し，より効率的な計算量で Jaccard 係数の推定を行う手法として，b-bit MinHash 法が提案されている [17]．Jaccard 係数は集合間でどの程度共通している要素があるのかを計測する指標であり，集合 S_1, S_2 に対して以下の式で表される．

$$J(S_1, S_2) = \frac{|S_1 \cap S_2|}{|S_1 \cup S_2|}$$

MinHash 法は集合に対して複数のハッシュ関数を適用し，固定された大きさのハッシュ値列に変換する．その後，二つのハッシュ値列の要素の一致割合により Jaccard 係数を推定する手法である．b-bit MinHash 法では，MinHash 法で計算したハッシュ値の下位 b ビットのみをハッシュ値として用いる．b-bit MinHash 法のハッシュ関数を $b_i(f)$ とすると，二つの集合 S_1, S_2

Algorithm 1 集合のビットマップ表現の構築

入力
S：基数を求めたい集合
M：ビットマップの大きさ
出力
$B(S) = \{B_i \mid 0 \le i < M\}$：集合 S を表す大きさ M のビットマップ

```
1: Initialize all B_i ← 0
2: for s ∈ S do
3:     h = H(s)
4:     B_h = 1
5: end for
```

に対して $b_i(S_1)$ と $b_i(S_2)$ が一致する確率 $P(S_1, S_2)$ は，集合間の Jaccard 係数と，偶然ハッシュ値の下位 b ビットが一致する確率の和となる．

$b = 1$ のとき，$P(S_1, S_2)$ の近似値として，k 個のハッシュ値の一致割合 $P_o(S_1, S_2)$ を用いることで，以下の式により集合 S_1, S_2 間の Jaccard 係数の推定値 $J_b(S_1, S_2)$ を計算することが可能である．

$$J_b(S_1, S_2) = \left(P_o(S_1, S_2) - \frac{1}{2}\right) \times 2$$

$$P_o(S_1, S_2) = 1 - \frac{1}{k}\sum_{i=1}^{k} \mathrm{XOR}(b_i(S_1), b_i(S_2))$$

言い換えると，比較したいソースファイルをそれぞれ k ビットの列に変換すれば，ファイル間の相互の比較はそれらのビット列の XOR 演算によって高速に実行することができる．詳しくは文献 [7] にて説明している．

2.2.2 Linear Counting 法を用いた集合の比較

Linear Counting 法 [18] は集合の基数を推定する手法の一つである．これは，大まかにいうと，集合 S を表すビットマップ $B(S)$ を構築し，ビットマップのうち 1 であるビットの数から S の基数の近似値を計算する手法である．Algorithm 1 に Linear Counting 法における集合を表すビットマップを構築するアルゴリズムを示す．入力として基数を計算したい集合 S とビットマップの大きさ M を与え，ビットマップ $B(S)$ を返す．S の全ての要素 s について，値域 $[0, M-1]$ の値を返すようなハッシュ関数 H を適用し，得られた $H(s)$ に対応する位置の $B(S)$ のビットを 1 とする．このとき，S の基数 $C(S)$ は M が S の要素数に対して十分に大きければ以下の式で近似計算できる．

$$C(S) = -M \ln\left(\frac{M - \mathrm{Bits}(B(S))}{M}\right)$$

ここで，$\mathrm{Bits}(B(S))$ は，$B(S)$ の 1 であるビットを数える関数である．

Linear Counting 法では集合をビット列で表現するため，和集合や共通集合をビット演算によって計算することができる．二つの集合 S_1, S_2 についてビットマップ $B(S_1), B(S_2)$ を用意したとき，その和集合と共通集合の基数の推定値 $U_l(S_1, S_2), I_l(S_1, S_2)$ は，以下の式で計算できる．

$$U_l(S_1, S_2) = C(S_1 \cup S_2)$$

$$I_l(S_1, S_2) = C(S_1 \cap S_2)$$

ここで，$B(S_1 \cup S_2) = B(S_1) \vee B(S_2)$，$B(S_1 \cap S_2) = B(S_1) \wedge B(S_2)$ で計算できることから，二つの集合から得られたビットマップ同士の AND 演算や OR 演算を行うだけで集合の比較が可能となる．

3. 提案手法

提案手法は，互いに派生関係をもつ n 個のソフトウェアプロダクトの集合 $S = \{s_i | 1 \le i \le n\}$ について，それぞれのソフトウェアプロダクトを構成するソースコードを比較することで系統樹を構築する．ここで，本研究における系統樹とは，ソフトウェアの進化履歴を表す最小全域木とその辺に方向を与えたグラフのことを指す．具体的な手順としては，まず，集合に含まれる全てのソフトウェアプロダクトの組について，既存のソフトウェアプロダクトの構成要素を再利用した量を示す指標（以降，ソースコードの再利用量）を計算する．ある二つのソフトウェアプロダクト s_i と s_j との間のソースコードの再利用量は $S_{\mathrm{reuse}}(s_i, s_j)$ と表現するものとし，その計算には異なる性質をもつ軽量なデータ構造を利用する二つの手法，すなわち b-bit MinHash 法と Linear Counting 法を独立に用い，それぞれについて，後述する方法で値を求める．

次に，ソースコードの再利用量の合計を最大化するように，全てのソフトウェアプロダクトを接続する全域木を構築する．これは，ソフトウェアの派生開発において，開発者がなるべく多くの部分を再利用しようとすると仮定すると，ソースコードの再利用量が大きい木ほど実際の派生関係に近づくと考えられるためである．全域木は，ソフトウェアプロダクトを頂点として，頂点間をソフトウェアプロダクト間のソースコードの再利用量を表現する辺でつないだ完全グラフに対して，Prim の手法 [22] を適用することで構築する．ここで，Prim の手法で用いるソフトウェアプロダクト

s_i, s_j をつなぐ辺の重み $w(s_i, s_j)$ は，以下の式で定義する．

$$w(s_i, s_j) = -S_{\text{reuse}}(s_i, s_j)$$

ソースコードの再利用量の符号が反転しているのは，Prim の手法では辺の重みの合計を最小化する最小全域木を構築するためである．Prim の手法では，木の構築を開始する頂点は任意に選ぶことができるが，提案手法ではソースコードが最も少ないもの（b-bit MinHash 法を用いる場合はソースファイル数が最小のもの，Linear Counting 法を用いる場合はソフトウェアプロダクトを表す集合の要素数が最小のもの）を選択する．

最後に，全域木を構築したあと，木の各辺に対して派生順序を判定する．類似したソフトウェアプロダクトの組 A, B を開発するとき，ソフトウェアプロダクト A に対して追加・編集を行って新しいソフトウェアプロダクト B を作成することの方が，ソフトウェアプロダクト B から機能を削除してソフトウェアプロダクト A を作成することよりも多いと仮定し，派生関係をもつソフトウェアプロダクト間では，派生後の方がソースコードが増加すると考える．しかし，リファクタリングなどにより派生先のソフトウェアプロダクトのソースコード長が派生元よりも減少する場合もあり得る．そのため，単純にソースコード量の大小を比較するだけでは，実際の派生順序とは異なる順序で判定される可能性がある．そこで，b-bit MinHash 法を用いた場合，単純にソフトウェアプロダクト全体で見たトークン数の多寡ではなく，トークン数が増加したソースファイル数の多い方を派生先とする．Linear Counting 法を用いた場合は，ファイルの区切りという情報が使用できず同様の対処が行えないため，単純にビットマップから分かる集合の基数の大きさを比較し，大きい方を派生先とする．最終的に得られる系統樹は，開発者がソースコードをできるだけ多く再利用し，かつ，ソースコードを追加することで開発しようとした場合の派生関係を表す．

図 1 に提案手法の動作例を示す．この図では，派生関係をもつソフトウェアプロダクトの集合 $S = \{s_1, s_2, s_3, s_4\}$ の完全グラフに対して，相互のソースコードの再利用量 $S_{\text{reuse}}(s_i, s_j)$ を計算し，それをもとに辺の重み $w(s_i, s_j)$ と辺を示している．辺のうち赤い矢印は，提案手法によって決定したソフトウェア同士の派生順序を表す．提案手法は，初めに計

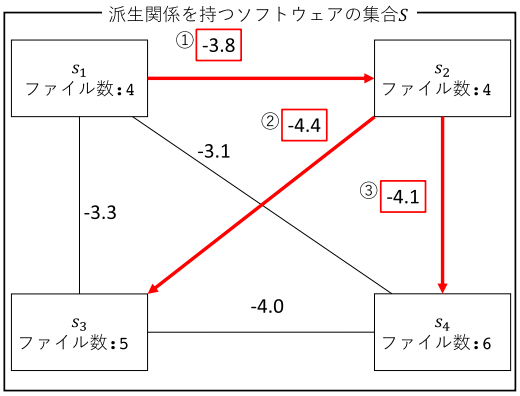


図 1　提案手法の動作例

算した S_{reuse} に基づき，ファイル数が最小の s_1 を開始する頂点として選択して Prim の手法を開始し，s_1 がもつ辺のうち最も重みが小さい辺を選択する．図では $w(s_1, s_2)$ が最も小さいため，s_2 を含む辺を選択する．次に，既に接続済みの s_1, s_2 のいずれかからそれ以外の頂点へと接続される辺の中から，重みが最も小さい辺を選択する．図の場合では，$w(s_2, s_3)$ が最も小さいため，s_2 と s_3 をつなぐ辺を選択する．以降同様に全ての頂点が辺によってつながるまで辺の選択を繰り返し，最終的に，図に示した①，②，③が選択される．その後，選択された全ての辺について，派生順序を後述するソフトウェアプロダクト間のソースコード量の差分から判定する．その結果として，この派生関係の系統樹は $s_1 \rightarrow s_2 \rightarrow s_3$ と，$s_2 \rightarrow s_4$ というように有向グラフで表される．つまり，まず s_1 から s_2 が派生し，その後 s_2 から s_3 と s_4 に分岐して開発されたと判断することができる．

提案手法では，ソフトウェアプロダクト間のソースコードの再利用量を軽量なデータ構造を利用した計算手法を導入することで，スケーラビリティを向上する．採用する手法は，b-bit MinHash 法と Linear Counting 法である．以降，それぞれの手法の適用方法を説明する．

3.1 b-bit MinHash 法を用いたソースコードの再利用量の推定

b-bit MinHash 法は，著者らの以前の研究と同様にファイル間の類似度の計算に利用する [7]．ソースファイルをそれぞれ字句列の 3-gram 多重集合とみなし，それらの類似度として b-bit MinHash 法で計算した Jaccard 係数の推定値を用いる．あるソースファイル

の字句列を f とし，その 3-gram 多重集合を $\tau(f)$ とすると，3-gram 多重集合で表されたソースファイル f_1 とソースファイル f_2 との間の Jaccard 係数 $J_\tau(f_1, f_2)$ は以下の式で定義される．

$$J_\tau(f_1, f_2) = \frac{|\tau(f_1) \cap \tau(f_2)|}{|\tau(f_1) \cup \tau(f_2)|}$$

b-bit Minhash 法では，$b = 1$ とし，ハッシュ関数を k 個用いることで，$J_\tau(f_1, f_2)$ の推定値を k ビットのビットベクトルに対する XOR 演算を行うだけで求めることができる．この $J_\tau(f_1, f_2)$ の推定値を，ファイル単位の類似度 $\mathrm{sim}(f_1, f_2)$ として用いるものとして，ソフトウェア間のソースコードの再利用量 $S_{\mathrm{reuse}}(s_i, s_j)$ を以下の式で計算する．

$$S_{\mathrm{reuse}}(s_i, s_j) = \max\left(\sum_{f \in s_i} S(f, s_j), \sum_{f \in s_j} S(f, s_i)\right)$$

$$S(q, s) = \begin{cases} S_{\max}(q, s), & \text{if } S_{\max}(q, s) \geq \theta \\ 0, & \text{otherwise} \end{cases}$$

$$S_{\max}(q, s) = \max_{f \in \mathrm{Files}(s)} \mathrm{sim}(q, f)$$

ここで $\mathrm{Files}(s)$ は s に含まれる全てのソースファイルを表す．$S(q, s)$ はソースファイル q の内容を作る素材として使えそうなソフトウェア s のファイルの類似度（ただししきい値 θ 以下の値は無視したもの）に相当し，全てのソースファイルに関して類似度を合計したものがソースコードの再利用量 $S_{\mathrm{reuse}}(s_i, s_j)$ となる．ただし，ソースファイル数の差などにより s_i に含まれる s_j からのソースコードの再利用量と s_j に含まれる s_i からの再利用量が異なる値となる場合があるため，$S_{\mathrm{reuse}}(s_i, s_j)$ は二つの値のより大きいほうを選択する．

3.2 Linear Counting 法を用いたソースコードの再利用量の推定

Linear Counting 法を用いる場合は，ソフトウェアプロダクト一つにつき全てのソースファイルの行を表現したビットマップを一つ構築し，**2.2.2** で説明した $I_l(S_1, S_2)$ を用いて，以下の式でソースコードの再利用量を計算する．

$$S_{\mathrm{reuse}}(s_i, s_j) = I_l(\mathrm{lines}(s_i), \mathrm{lines}(s_j))$$

ここで，$\mathrm{lines}(s)$ はソフトウェアプロダクト s に含まれる全てのソースファイルのユニークな行（ただし行頭及び行末の空白を除いたもの）の集合を表す．ソースコードの再利用量は，一方のソフトウェアプロダクトから他方を作る際に書き換えが不要なソースコードの行数に相当する．ソースコードからビットマップへの変換では，インデントや改行文字の変更の影響を受けないように各行に対して行頭及び行末の空白・タブ文字・改行文字を除去した後，UTF-8 でのバイト列表現に対して MurmurHash3 [23] の 32 ビットのハッシュ値を求めた．ハッシュ値をビットマップのサイズ M で割った余りが，Algorithm 1 における $H(s)$ の値であり，ビットマップ中での対応するビットの番号である．

4. 評　　価

提案手法の有効性を評価するために，派生関係が既知のデータセットに対して，ソースコードから派生関係を復元する実験を行う．提案手法は軽量なデータ構造として b-bit MinHash 法と Linear Counting 法を用いた場合を定義したため，それらと既存手法の性能を比較する．

適用対象は，既存研究 [5] で用いられたものと同一である．適用対象それぞれの特徴を以下に述べる．

（1） PostgreSQL の諸バージョンのうち分岐が生じないもので構成されたソフトウェアプロダクトの集合．

（2） PostgreSQL のうちバージョン 8 系列全てのソフトウェアプロダクトの集合．

（3） PostgreSQL のうちバージョン 8 系列で一定期間ごとでサンプリングしたソフトウェアプロダクトの集合．

（4） PostgreSQL のうちバージョン 8 系列でそれぞれのマイナーバージョンで最新から幾つか遡って構築したソフトウェアプロダクトの集合．

（5） FFmpeg とその分岐である Libav を含むソフトウェアプロダクトの集合．FFmpeg のあるバージョンから Libav に分岐する．

（6） BSD 系列の複数の OS で構成されたソフトウェアプロダクトの集合．合流や分岐があり，閉路が存在するため木構造ではない．そのため提案手法で得られる派生関係の数は実際より少ないと予想される．

（7） Groovy の諸バージョンで構成されたソフトウェアプロダクトの集合．規模が小さい．

（8） Apache hibernate の全てのソフトウェアプロダクトの集合．規模が大きい．

（9） OpenJDK 6 の全てのソフトウェアの集合．JDK 7 を実装した後に，それを用いて部分的な機能を

表 1　適 用 対 象

No.	名称	開発言語	ソフトウェア数	平均ファイル数	平均コード行数
1	Pgsql-minor	C	14	643	180,233
2	Pgsql8-all	C	144	767	586,708
3	Pgsql8-latest	C	38	781	55,095
4	Pgsql8-annually	C	25	766	28,383
5	Ffmpeg	C	16	991	317,124
6	BSD	C	18	1,014	559,132
7	Groovy	Java	37	942	178,751
8	hibernate	Java	62	4,401	522,181
9	OpenJDK6	Java	16	7,060	2,392,471

もつ JDK 6 の実装が行われているため，系統樹の構築が難しいと予想される．

ただし，これらはデータセット構築時に参照した際の特徴である．表 1 に適用対象のデータセットとその諸情報を載せる．

提案手法は Java 11 で実装した．b-bit MinHash 法で用いるハッシュ関数の数は $k = 2048$ であり，ビット数は $b = 1$ である．Linear Counting 法で用いるビットマップのサイズは，128 M ビットとする．実験に用いる計算機環境の OS は Oracle Linux Server release 7.9，CPU は Intel Xeon E5-2690 v4，RAM は DDR4-2400 ECC Memory 512 GB，ストレージは SAS 接続の 1 TB の HDD，Java の実行環境は OpenJDK 11 である．

提案手法をデータセットに対して適用して得られた系統樹がもつ辺と，実際の系統樹がもつ辺の一致率で精度を評価する．本実験では，手法の最終的な出力結果である系統樹の有向辺の精度に加えて，辺を無向辺とした場合の精度も評価対象とする．これは，無向辺の精度が提案手法の再利用量の定義の妥当性を，有向辺の精度が派生関係の向きの決定方法の妥当性を表す指標となるためである．有向辺とする場合は，辺の両端の頂点と辺の向きが実際の系統樹のものと一致しているとき正解とみなす．無向辺とする場合では，その辺がもつ両端の頂点が実際の系統樹のものと一致していれば正解とみなす．提案手法の比較対象は，既存手法の結果 [5] である．

4.1　提案手法の精度

提案手法と既存手法の結果を表 2 に示す．表 2 中の左から 3 列目まではデータセットの番号，実際の系統樹がもつ辺の数，既存手法と提案手法で構築した系統樹がもつ辺の数を記載している．以降の列は，既存手法と提案手法で構築した系統樹の無向辺の場合と有向辺の場合の，実際の系統樹の辺との一致数とその割合である．全てのデータセットの辺を合計した場合，b-bit MinHash 法の無向辺の精度は 88.1%，有向辺の精度は 80.3% であった．Linear Counting 法の無向辺の精度は 88.1%，有向辺の精度は 78.7% であった．既存手法 [5] の無向辺の精度は 88.9%，有向辺は 82.5% であり，提案手法は b-bit MinHash 法と Linear Counting 法のいずれも，既存手法より若干低い精度となった．全てのデータセットを平均した場合については，無向辺に関しては既存手法が最も結果が良く，次に同順で Linear Counting 法，b-bit MinHash 法が並んだ．有向辺については b-bit MinHash 法，既存手法，Linear Counting 法の順だった．

無向辺の精度に関しては，既存手法と比べて b-bit MinHash 法と Linear Counting 法のどちらの場合でも，ほとんど差はなかった．これは，提案手法での再利用量の定義が既存手法と同等の妥当性があることを示している．有向辺の精度に関しては，大抵のデータセットで既存手法と同等だったが，一部のデータセットで低下していた．これは，提案手法の派生順序の判定方法が既存手法よりも劣ることを示している．ただし，b-bit MinHash 法を利用する場合はデータセット 7 とデータセット 9 について既存手法よりも精度が高く，適用対象によっては提案手法の方が適している場合があると考えられる．以降，有向辺の精度に影響した提案手法と既存手法の特性について述べる．

b-bt MinHash 法を用いた提案手法では，「派生先のソフトウェアプロダクトに含まれる一つのソースファイルは，派生元のソフトウェアプロダクトから最も似ている一つのソースファイルを再利用して作成された」というモデルを想定しており，ソースコード量が増加したソースファイルの数が多い方を派生先としている．既存手法ではそのようなモデルは想定しておらず，全ての類似するソースファイルの組について組のうち片方を基準とした際のソースコードの増加量を計算し，その値のソフトウェアプロダクト間全体での合

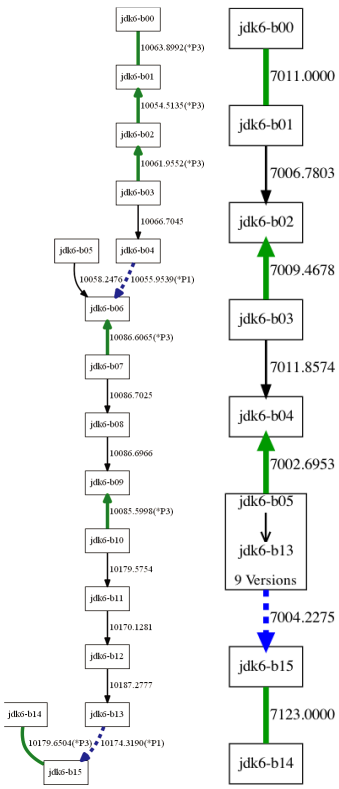
表 2　適 用 結 果

No.	全体	辺の数	一致数 既存手法 無向辺		既存手法 有向辺		b-bit MinHash 法 無向辺		b-bit MinHash 法 有向辺		Linear Counting 法 無向辺		Linear Counting 法 有向辺	
1	13	13	13	100.0%	13	100.0%	13	100.0%	13	100.0%	13	100.0%	13	100.0%
2	143	143	137	95.8%	132	92.3%	135	94.4%	124	86.7%	135	94.4%	128	89.5%
3	37	37	30	81.1%	30	81.1%	31	83.8%	30	81.1%	31	83.8%	29	78.4%
4	24	24	20	83.3%	20	83.3%	20	83.3%	20	83.3%	20	83.3%	20	83.3%
5	15	15	14	93.3%	14	93.3%	14	93.3%	14	93.3%	14	93.3%	14	93.3%
6	17	15	11	64.7%	11	64.7%	10	58.8%	10	58.8%	10	58.8%	10	58.8%
7	36	36	30	83.3%	24	66.7%	28	77.8%	25	69.4%	28	77.8%	22	61.1%
8	61	61	53	86.9%	46	75.4%	53	86.9%	43	70.5%	53	86.9%	42	68.9%
9	15	15	13	86.7%	7	46.7%	14	93.3%	11	73.3%	14	93.3%	6	40.0%
平均	40.1	39.9	35.7	86.1%	33.1	78.3%	35.3	85.7%	32.2	79.6%	35.3	85.7%	31.6	74.8%
全体	361	359	321	88.9%	298	82.5%	318	88.1%	290	80.3%	318	88.1%	284	78.7%

計が大きい方を派生先としている．つまり，二つのソフトウェアプロダクト間でのソースコードの増加量を調べる際，*b*-bit MinHash 法を利用する提案手法では，派生元とするソフトウェアプロダクトに含まれるソースファイルについて 1 対 1 の対応関係から差分を考えるが，既存手法では 1 対多の対応関係から差分を考えている．そのため，しきい値よりも大きな類似度をもつようなファイルの全ての組に対して類似度が加算されていくので，類似したソースファイルの組の個数が指標に影響を与えてしまう．

図 2 に，*b*-bit MinHash 法を利用した提案手法と既存手法とでそれぞれ構築したデータセット 9 の系統樹を示す．正しく系統樹を構築できた部分については，図を小さくするため，誤りが存在しない隣接頂点と辺を一つの頂点にまとめて表し，実際の系統樹と異なる辺について誤りを強調している．緑色の矢印は順序関係が間違っている場合の辺，青色の破線の矢印はバージョンを飛ばしてつながった辺を表す．また，辺に鏃がない場合は順序関係が定義できないことを表し，これはソースコードが変動したソースファイルが存在しないことを示す．既存手法で構築された系統樹は，既存手法の Online Appendix(注1) に掲載されている内容を引用した．実際の系統樹は，名前の末尾の数字の昇順に枝分かれなくつながっている．

図 2(a) と図 2(b) の双方で，jdk6-b00 と jdk6-b01 の組と，jdk6-b14 と jdk6-b15 の組がそれぞれソースコードレベルで一致していることが分かる．また，図 2(b) においては，全体のうち半分以上の頂点を占める jdk6-b05 から jdk-b13 までは順序関係の判定も正確にできており，図中では短縮して表現してある．図 2(a) と比較すると，図 2(b) では派生順序の判定を誤った辺と接続を誤った辺がともに少なかった．データセット 9 では，ソフトウェアプロダクト内で互いに類似したファイルの組が 3,000 以上あったため，既存手法における



(a) 既存手法の場合 [5]　(b) *b*-bit MinHash 法を利用した提案手法の場合

図 2　既存手法と提案手法で構築されたデータセット 9 の系統樹

(注1)：https://sel.ist.osaka-u.ac.jp/pret/

再利用量の指標に当たる数値が大きくなり，計算結果に誤りが生じたと考えられる．これらの類似ファイルの組は，Java において，同一のインタフェースを実装したクラスなどで類似したファイルが多数作られやすいことが原因であると考えられる．

一方，*b*-bit MinHash 法で有向辺の精度が低下する場合もあった．データセット 2 では，プロダクト内で相互に類似するソースファイルの組が少なく，既存手法が用いるソースコードの編集量に基づく派生順序の判定が開発者の実際の作業量に近い値を算出していた可能性がある．このデータセット 2 に対して *b*-bit MinHash 法を利用した提案手法の結果を図 3 に示す．この図にある矢印の形状などは全て図 2 の説明に準じ，誤りのない区間に関する短縮表現については，別系統への分岐箇所のみ独立した頂点として表現した．加えて，赤色の破線の矢印は系統間の接続を誤った場合の辺であり，黒色の破線の矢印は完全な誤りの辺をそれぞれ表す．図 3 を見ると，既存手法で構築された系統樹と同様，大まかに六つの系統があることと，それぞれの系統間を誤った辺で結んでいることが分かる．また，提案手法は既存手法と比較して派生順序の判定に関して誤りが増えていた．ファイルの一部を別ファイルに切り出して追記を行うなど，全体で見ればソースコードの量は増えたとしても，ソースファイルごとに見た際にはソースコード量が減るような編集が，提案手法でのこのような判定の誤りの原因の一つと考えられる．ただし，個別の系統ごとでは，一つの辺を除いて既存手法と同様に構築できており，既存手法と同様に利用者はもっともらしい派生順序を検討できる．

Linear Counting 法を利用した提案手法では，表 2 にあるように，無向辺の精度は *b*-bit MinHash 法を利用した提案手法と同じ値で，有向辺の精度が既存手法と同等か少し劣る結果となった．これは，Linear Counting 法を利用する提案手法の場合は派生順序の判定にユニークなソースコード行の集合の大きさだけを用いており，これが実際の開発におけるソースコードの再利用方法をうまく反映していないことが原因であると考えられる．Linear Counting 法を用いた提案手法の精度を高めるために，軽量なデータ構造を用いて有向辺の向きを正しく算出するための指標を考えることが，今後の課題の一つである．

4.2 提案手法の計算コスト

提案手法では，ソースコードを表す軽量なデータ構造を用いることで，既存手法と比べて効率的に系統樹を構築することができる．類似度の計算について考えると，既存手法ではソースコードをトークン列に分割し，二つのソースコードつまりトークン列同士の LCS を得る．このとき，類似度計算の時間計算量はトークン列の長さを N として，効率的に実装しても $O(N^2)$ である．対して，*b*-bit MinHash 法では，一つのソースファイルをビット列で表し，ビット列同士の XOR 演算により類似度を計算可能であるため，その時間計算量は CPU 命令の XOR 演算を用いれば $O(1)$ である．また，Linear Counting 法においても，ソースファイルはビット列で表され，類似度の計算には OR 演算と AND 演算を用いるのみなので，$O(1)$ の時間計算量で済む．

表 3　実行時間の比較

No.	既存手法 実行時間 (A) [s]	b-bit MinHash 法 実行時間 (B) [s]	倍率 (A/B)	Linear Counting 法 実行時間 (C) [s]	倍率 (A/C)
1	198.383	5.587	35.507	2.521	78.677
2	27,083.988	35.105	771.523	101.395	267.113
3	1,925.312	7.843	245.488	10.079	191.019
4	687.191	5.930	115.883	5.648	121.677
5	303.399	3.225	94.071	2.223	136.512
6	238.807	7.850	30.421	3.550	67.276
7	527.101	7.584	69.503	4.009	131.477
8	31,272.477	179.789	173.940	16.926	1,847.569
9	4,350.722	89.724	48.490	13.651	318.716
平均	–	–	176.092	–	351.115

これを定量的に評価するため，既存手法と提案手法の実行時間を計測し，比較する．既存手法と提案手法について，それぞれ 6 回実行し実行時間の平均値を得る．既存手法の実行にはその著者からソースコードの提供を受けて行った．既存手法では適宜並列化処理が施されており，利用するスレッド数を指定できる．比較のため，提案手法にも並列化処理を実装し，同一スレッド数の実行で比較を行う．そのスレッド数は 16 とした．実行時間の計測では，提案手法については Java のシステムメソッドの一つである nanoTime メソッドを，既存手法については bash の time コマンドを用いる．その結果を表 3 に示す．

b-bit MinHash 法では既存手法に対して最大で 772 倍，平均で 176 倍高速だった．また，Linear Counting 法では最大で 1,848 倍，平均で 351 倍と更に高速だった．この結果から，提案手法は既存手法と比べて非常に高速であると分かった．

提案手法は，空間計算量に関しても効率的である．既存手法の実装では LCS を総当たりで効率良く計算するために，分析対象の全てのソースファイルの文

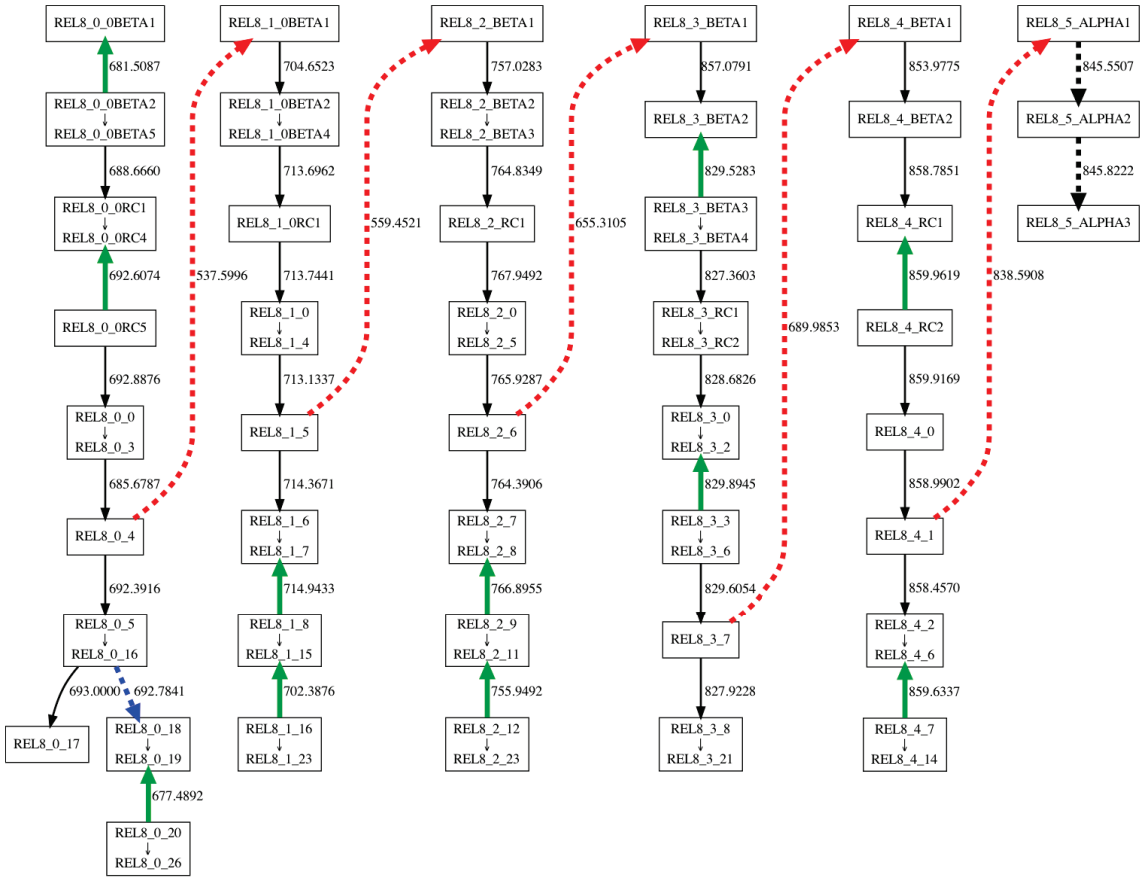


図 3　*b*-bit MinHash 法を用いた提案手法で構築されたデータセット 2 の系統樹

字列表現をメモリに保持していた．これに対し，*b*-bit MinHash 法であれば全てのソースファイルの数 n 個分の 2048 ビットのビット列つまり $2048n$ ビット，Linear Counting 法であれば 1 ソフトウェア当り 128 M ビットで済む．そのため，提案手法は既存手法と比べて空間計算量は大幅に小さく，特に Linear Counting 法ではソフトウェアの大きさによらないため，空間計算量は比較するソフトウェアそれぞれが大きい集合ほど相対的に低くなる．

4.3　提案手法の実用性

実用性という観点では，既存手法と同様に，系統樹の大まかな構造を把握する分には十分な情報がある．例えば，提案手法では，利用する軽量なデータ構造がどちらの場合でもデータセット 6 は精度が 60% を切っているが，これは図 3 の場合と同様に系統間の派生時期が誤って判定されていたためで，大まかにどのような系統があるのかは正しく認識されていた．そのため，利用者は，誤検出が起きやすい系統間の派生時期以外の情報については，系統樹から読み取ることができる．また，派生順序が誤って判定されている辺があったとしても，無向辺として見た場合おおよそ正確に系統樹は構築できたので，その頂点からたどって近い位置に存在する頂点間の順序がもっともらしければ，正しい派生順序を推測することが可能である．そのため，利用者が解釈することで実用上はある程度の誤った派生順序の判定に対応可能である．

系統樹の構築時にソースコードは増加するという仮定に反するような事例，例えばリファクタリングを行ったりデッドコードを削除した場合は，派生関係の順序が実際とは反対に判定されてしまう可能性がある．また，二つのソフトウェアプロダクト間でソースファイルに関する変更がなかった場合，派生順序やその後どちらから派生して開発が続いたのか系統樹に反映できない．これらの制限の存在は，既存手法と同様であ

る．これらの場合は，各ソフトウェアプロダクトのタイムスタンプなどを参照することで，正しい派生順序を把握することになる．ただし，ソースファイルに関する変更がなかった場合に関しては，その情報自体も利用者に有用であると考えている．

ソフトウェアプロダクトが二つの系統に別れ，その後再び合流するような閉路をもつ派生関係があった場合は，全域木という閉路をもたない表現の都合上，全ての派生関係を系統樹に反映することはできない．そのため，閉路にも対応したネットワーク構造を応用するなど，派生関係により適した表現方法を検討することが今後の課題の一つである．

以上のように，得られる系統樹の質という点では，どちらの軽量なデータ構造を利用した場合でも，提案手法には既存手法と同様の実用性がある．それに加えて，提案手法は既存手法と比べて実行時間が大幅に短縮されているため，適用可能なデータセットの規模の点で実用性が向上している．

4.4 大規模なデータセットへの適用

スケーラビリティの向上を確認するため，より大規模なデータセットに既存手法と提案手法をそれぞれ適用する．適用対象は，Spinellis が構築したデータセット [8] のうち，FreeBSD の release ブランチのみを抽出したものとする．対象ソフトウェアプロダクト集合の大きさは 74 で，ソフトウェアプロダクト一つ当りの平均コード行数は 6,523,017.1 行，平均ソースファイル数は 11,334 個である．データセット全体としてみた場合，既存手法を適用したデータセットのうち最も規模が大きいものと比較して，コード行数で 5.71 倍，ファイル数で 3.07 倍である．FreeBSD の開発リポジトリに含まれる，bsd-family-tree [24] を参考にして構築した系統樹と，それぞれの手法で構築した系統樹を比較し，その精度を評価する．また，それぞれ実行にかかった時間を計測し，精度と実行時間の二つの尺度で比較する．その結果を表 4 に示す．

既存手法では 3 日間かけても処理は終了しなかった．ただし，74 個のソフトウェアプロダクトのうち 8 個まで処理できていた．*b*-bit MinHash 法では，実行時間が約 15 分，構築した系統樹の精度は有向辺，無向辺のいずれも 78.1% だった．Linear Counting 法では，実行時間が約 8 分で終了しており，無向辺の精度は *b*-bit MinHash 法と同じ 78.1%，有向辺の精度は 75.3% だった．この結果から，提案手法は既存手法と比べて大幅にスケーラビリティが向上していることが分かる．

表 4　FreeBSD の release ブランチにおける実行時間とその精度

手法	精度（無向辺）	精度（有向辺）	実行時間 [s]
既存手法	-	-	3 日で 8 個処理
b-bit MinHash 法	78.1%	78.1%	921.358
Linear Counting 法	78.1%	75.3%	504.062

5. 妥当性への脅威

本研究では，既存研究と同一のデータセットと，今回新たに用意したデータセットを用いて評価を行った．そのため，既存のデータセットについては既存研究と同様の妥当性への脅威が存在する．つまり，実験対象がよく管理されている OSS に限られている点，幾つかのパラメータが一通りしか試されていない点である．

新しく用意したデータセットについて，正解の系統樹を定める際に参照した bsd-family-tree は，FreeBSD のバージョンごとの派生関係を表している．FreeBSD はバージョンを Git のブランチとして管理しており，実験で利用したソースコードはその時点での各ブランチの最新のものを利用した．それらのソースコードは，開発者が実際に分岐のための作業を行った時点のソースコードよりも更に様々な編集が行われており，提案手法の実験結果は，それらの編集の影響を受けている．Git 上の全ての更新履歴を個別のバージョンとみなして提案手法を使えば更に正確さは向上する可能性があるが，提案手法の実際の利用状況ではそこまでの履歴が残っていないと考え，評価実験では使用していない．

6. むすび

本研究では，互いに派生関係にあるソフトウェアプロダクトの集合を入力として，ソフトウェアプロダクト間の再利用量をもとに高速にそれらの系統樹を構築する手法を提案した．また，既存手法では時間のかかっていた類似度計算に，軽量なデータ構造でソフトウェアプロダクトを表し，近似値を用いる高速な計算手法を適用することで処理の短縮を行い，大幅にスケーラビリティを向上させた．既存手法では 3 日間処理しても 8 個のソフトウェアプロダクトしか処理できなかったところ，74 個全てを処理しても最短で 8 分程度で実行可能だった．提案手法を用いることで，開発者は短時間で大規模なソフトウェアプロダクト集合の派生関係を構築し，その情報を利用してメンテナンスすることができる．

また，提案手法で利用した二つの手法について，実験結果から，空間コストがより軽量な Linear Counting 法は特に大規模なデータで有用であり，大幅な高速化に対して精度の低下は軽微であることが分かった．加えて，データセットがある程度の規模であれば *b*-bit MinHash 法が計算コストと精度に関してともに優れていると分かった．

本研究では OSS のみを対象としたが，同一の組織内のみで開発・利用されるようなソフトウェアも世の中に多数存在するため，企業で開発されたソフトウェアプロダクト集合に対して適用することが今後の課題として挙げられる．また，本手法で構築した系統樹から，派生関係において隣接するソフトウェアプロダクト同士で共通するソースコードについて，一つのソフトウェアプロダクトに対して行われた変更を他のソフトウェアプロダクトにも効果的に適用する手法の開発も挙げられる．

謝辞 本研究は科学技術研究費 JP18H04094，JP18H03221，JP19K20239 の助成を受けて行われた．

文　献

[1] Y. Dubinsky, J. Rubin, T. Berger, S. Duszynski, M. Becker, and K. Czarnecki, “An exploratory study of cloning in industrial software product lines,” Proc. 17th European Conf. on Software Maintenance and Reengineering, pp.25–34, March 2013.

[2] 野中　誠，桜庭恒一郎，舟越和己，“組込みソフトウェア製品ファミリにおける是正保守の予備的分析，” 情処学研報，第 2009-SE-166 巻，pp.1–8, Nov. 2009.

[3] A. Hemel and R. Koschke, “Reverse engineering variability in source code using clone detection: A case study for linux variants of consumer electronic devices,” Proc. 19th IEEE Working Conf. on Reverse Engineering, pp.357–366, 2012.

[4] J. Rubin, A. Kirshin, G. Botterweck, and M. Chechik, “Managing forked product variants,” Proc. 16th Int. Software Product Line Conf., pp.156–160, Sept. 2012.

[5] T. Kanda, T. Ishio, and K. Inoue, “Approximating the evolution history of software from source code,” IEICE Trans. Inf. and Syst., vol.E98-D, no.6, pp.1185–1193, 2015.

[6] Y. Hayase, T. Kanda, and T. Ishio, “Estimating product evolution graph using kolmogorov complexity,” Proc. 14th Int. Work. on Principles of Software Evolution (IWPSE), p.66–72, Aug. 2015.

[7] 伊藤　薫，石尾　隆，神田哲也，井上克郎，“軽量な類似度計算によるプロジェクト間のソースファイル集合の再利用検出，” 信学論（D），vol.J103-D, no.7, pp.542–554, July 2020.

[8] D. Spinellis, “A repository of unix history and evolution,” Empirical Software Engineering, vol.22, pp.1372–1404, Aug. 2016.

[9] Y. Manabe, Y. Hayase, and K. Inoue, “Evolutional analysis of licenses in foss,” Proc. Joint ERCIM Work. on Software Evolution (EVOL) and Int. Work. on Principles of Software Evolution (IWPSE), pp.83–87, Sept. 2010.

[10] K. Hotta, Y. Sano, Y. Higo, and S. Kusumoto, “Is duplicate code more frequently modified than non-duplicate code in software evolution? an empirical study on open source software,” Proc. Joint ERCIM Work. on Software Evolution (EVOL) and Int. Work. on Principles of Software Evolution (IWPSE), pp.73–82, Sept. 2010.

[11] M. Mondal, B. Roy, C.K. Roy, and K.A. Schneider, “Investigating near-miss micro-clones in evolving software,” Proc. 28th Int. Conf. Program Comprehension, pp.208–218, June 2020.

[12] S. Duszynski, V. Tenev, and M. Becker, “N-way diff: Set-based comparison of software variants,” Proc. 8th Work. Conf. Software Visualization (VISSOFT), pp.72–83, Sept. 2020.

[13] N. Kawamitsu, T. Ishio, T. Kanda, R.G. Kula, C.D. Roover, and K. Inoue, “Identifying source code reuse across repositories using lcs-based source code similarity,” Proc. 2014 IEEE 14th Int. Working Conf. Source Code Analysis and Manipulation, pp.305–314, Sept. 2014.

[14] L. Jiang, G. Misherghi, Z. Su, and S. Glondu, “Deckard: Scalable and accurate tree-based detection of code clones,” Proc. 29th Int. Conf. Software Engineering, pp.96–105, May 2007.

[15] 横井一輝，崔　恩瀞，吉田則裕，井上克郎，“情報検索技術に基づく細粒度ブロッククローン検出，” コンピュータ ソフトウェア，vol.35, no.4, pp.16–36, 2018.

[16] H. Sajnani, V. Saini, J. Svajlenko, C.K. Roy, and C.V. Lopes, “SourcererCC: Scaling code clone detection to big-code,” Proc. 38th Int. Conf. Software Engineering, pp.1157–1168, May 2016.

[17] P. Li and C. König, “b-bit minwise hashing,” Proc. 19th Int. Conf. World Wide Web, pp.671–680, April 2010.

[18] K.-Y. Whang, B.T. Vander-Zanden, and H.M. Taylor, “A linear-time probabilistic counting algorithm for database applications,” ACM Trans. Database Syst., vol.15, no.2, p.208–229, June 1990.

[19] P. Jaccard, “The distribution of the flora in the alpine zone.1,” New Phytologist, vol.11, no.2, pp.37–50, 1912.

[20] A. Broder, “On the resemblance and containment of documents,” Proc. Compression and Complexity of Sequences 1997, pp.21–29, June 1997.

[21] M.S. Charikar, “Similarity estimation techniques from rounding algorithms,” Proc. 34th Annual ACM Symposium on Theory of Computing, pp.380–388, May 2002.

[22] R.C. Prim, “Shortest connection networks and some generalizations,” The Bell System Technical Journal, vol.36, no.6, pp.1389–1401, 1957.

[23] Y. Seeley and A. Gaul, “Murmurhash3,” https://github.com/yonik/java_util. referenced in Oct. 2019.

[24] FreeBSD.org, “bsd-family-tree,” svnweb.freebsd.org. https://svnweb. freebsd.org/base/release/12.1.0/share/misc/bsd-family-tree?view=markup, Updated in Nov. 4 2019.

（2020 年 12 月 28 日受付，2021 年 3 月 19 日再受付，
4 月 19 日早期公開）

伊藤　　薫

2016 年兵庫県立大学工学部卒，2018 年大阪大学大学院情報科学研究科博士前期課程了．現在大阪大学大学院情報科学研究科博士後期課程に在学中．ソフトウェアの再利用分析に関する研究に従事．

石尾　　隆　（正員）

2003 年大阪大学大学院基礎工学研究科博士前期課程了．2006 年同大学大学院情報科学研究科博士後期課程了．同年日本学術振興会特別研究員 (PD)．2007 年大阪大学大学院情報科学研究科助教．2017 年奈良先端科学技術大学院大学情報科学研究科准教授．2018 年同大学先端科学研究科准教授．博士（情報科学）．プログラム解析，プログラム理解に関する研究に従事．

神田　哲也

2016 年大阪大学大学院情報科学研究科博士後期課程了．同年奈良先端科学技術大学院大学博士研究員．2017 年大阪大学大学院情報科学研究科特任助教．2018 年より同研究科助教．博士（情報科学）．ソフトウェア進化，ソースコード解析に関する研究に従事．

井上　克郎　（正員：フェロー）

1984 年大阪大学大学院基礎工学研究科博士後期課程了（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 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等のプログラム分析や再利用技術の研究に従事．