ソフトウェア保守支援のための
高速な類似性分析手法の研究

提出先　大阪大学大学院情報科学研究科
提出年月　2021 年 4 月

伊藤　薫

論文一覧

主要論文

1. Kaoru Ito, Takashi Ishio, Katsuro Inoue. “Web-Service for Finding Cloned Files using *b*-Bit Minwise Hashing”, in Proceedings of the 11th International Workshop on Software Clones (IWSC 2017), pp.1-2, Klagenfurt, Austria, February 2017.（国際会議録）
2. 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”, 電子情報通信学会論文誌, VOL.J103-D, NO.7, pp.542–554, 2020 年 7 月（学術論文）
3. 伊藤薫, 石尾隆, 神田哲也, 井上克郎, “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”, 電子情報通信学会論文誌, VOL.J104-D, No.8, 2021 年 8 月（学術論文）

関連論文

1. Takashi Ishio, Yusuke Sakaguchi, Kaoru Ito, Katsuro Inoue. “Source File Set Search for Clone-and-Own Reuse Analysis”, in Proceedings of the 2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR 2017), pp.257–268, Buenos Aires, Argentina, May 2017.（国際会議録）
2. 坂口 雄亮, 石尾 隆, 伊藤薫, 井上克郎. “ソースファイル群の類似性を用いたソフトウェア再利用元の検索”, 情報処理学会研究報告, Vol.2017-SE-195, 2017 年 3 月.（国内会議録）

内容梗概

ソフトウェア開発において既存のソフトウェアから構成要素を再利用することは一般的に行われている．ソフトウェアを再利用すると，開発効率が向上しより良い品質のソフトウェアが開発できる．しかし，再利用元のソフトウェアに欠陥や脆弱性が見つかった場合，自身のソフトウェアに影響があるか確認し，影響があった場合はその修正を適用する必要がある．

取り込んでしまった欠陥や脆弱性を可及的速やかに取り除くには，ソフトウェアの保守管理作業が重要となる．再利用したソフトウェアに関しての正しい情報が記録されていれば，その情報を元に容易に保守管理作業が可能である．ところが，再利用に関する情報は再利用した際には記録されていても，開発が継続されるにつれ不正確になったり失われてしまうことがある．そのため，限られた情報から現在の状況を推測する必要がある．

開発者はしばしば再利用したソースコードを自身のソフトウェアで利用しやすいように編集する．そのため，単純にソースファイルが一致するソフトウェアを探すだけでは，再利用元のソフトウェアを見つけることは難しい．そこで，ある程度ソースコードに変更が加えられていても問題ないように，その類似性を指標として探索する必要がある．ただし，愚直にソースコードを比較していては，実用的な時間で分析することができない．この問題を解消するため，高速に集合を比較する手法を導入することで，ソースファイル間の関係性を分析する必要がある．

今までにコードクローンと呼ばれる，互いに一致または類似するソースコード断片を分析することで，その出自を分析する手法が多数提案されている．しかし，同一のソフトウェアから複数のソースファイルを再利用することも多く，ソースコード断片だけの解析では対応できない場合がある．そのため，ソフトウェアの保守管理作業を容易にするためには，ソースコード断片に加えてソースファイルやソフトウェアでの類似性を分析する必要がある．

そこで本研究では，(1) あるソースファイルの出自が失われた場合，(2) ソフトウェア中に含まれる再利用したライブラリのバージョンが失われた場合，(3) 既存のソフトウェアとの派生関係の情報が失われた場合の 3 つの状況でソフトウェアの保守管理作業を支援するために，ソースファイルやソフトウェアの類似性を高速に分析する手法を提案する．

上記（1）の状況に対して，ソースファイルに含まれる字句列集合間の類似度を計算することで，ある時点での Debian snapshot archives に含まれる OSS のソースファイル群から最も類似するソースファイルを検索する Web サービスを開発する．類似度の計算に b-bit MinHash 法を導入することで，多数のファイルから類似するものを高速に検索可能なシステムを構築した．

上記（2）の状況に対して，分析対象となるソフトウェアに含まれる，ライブラリから再利用したソースファイル群と，再利用元ライブラリの各バージョンに含まれるソースファイル群を比較することで，ソフトウェアで再利用しているライブラリのバージョンを高速に特定する手法を提案する．この手法は上記（1）で利用した手法の拡張で， b-bit MinHash 法で計算したソースファイル間の類似度を合計することでソフトウェアとライブラリのそれぞれバージョンとのソースファイル群同士がどの程度一致しているか検出し，バージョンを特定する．これにより，高速かつ高精度に再利用しているライブラリのバージョンを特定することが可能となる．提案手法は既存手法と比べて精度と検出速度の両方で向上に成功した．

上記（3）の状況に対して，派生関係を持つことがわかっているソフトウェア集合について，それぞれのソフトウェアを構成するソースファイル群の共通性に基づいて，その進化履歴を高速に再構築する手法を提案する．この手法で言う共通性とは，それぞれのソースファイル群を比較した際に，どの程度違いがないか，つまりどの程度の少ない編集量で集合に含まれるソフトウェア同士の変換が可能かを言う．共通性の計算には b-bit MinHash 法による高速な類似度計算を用いる手法と， Linear Counting 法という集合の基数を高速に計算する手法の二通りの手法を用いて行う．提案手法は既存手法と比べて同程度の精度で高速化に成功し，スケーラビリティが向上した．

本研究の成果を用いることで，開発者はソフトウェア開発において失われたソフトウェアの再利用に関する情報を復元できる．さらに，復元した情報から再利用したソフトウェアについて保守管理作業を効率的に実施できるようになる．

謝辞

本研究を行うにあたり，日頃から様々なご指導を賜りました，大阪大学大学院情報科学研究科コンピュータサイエンス専攻井上克郎教授に，心から感謝申し上げます．

本論文を執筆するにあたり，様々なご指導ご助言を頂きました，大阪大学大学院情報科学研究科コンピュータサイエンス専攻増澤利光教授，ならびに楠本真二教授に感謝申し上げます．

本研究を行うにあたり，研究方針など様々なご指導ご助言を頂きました，奈良先端科学技術大学院大学先端科学技術研究科情報科学領域石尾隆准教授に，心から感謝申し上げます．

本研究を行うにあたり，日々の研究活動のみならず様々なご指導を頂きました，大阪大学大学院情報科学研究科コンピュータサイエンス専攻松下誠准教授，春名修介特任教授に感謝申し上げます．

本研究を行うにあたり，日々の研究活動やレクリエーション，コーディング指南などでご指導ご助力頂いた，大阪大学大学院情報科学研究科コンピュータサイエンス専攻神田哲也助教に多大なる感謝を捧げます．

本研究を行うにあたり，研究活動の中でご助言を頂きました，名古屋大学 大学院情報科学研究科吉田則裕准教授，ならびに奈良先端科学技術大学院大学先端科学技術研究科情報科学領域 Raula Gaikovina Kula 助教に感謝申し上げます．

日々の研究活動の中で，活発な議論を交わし研究を深める一助となりました，大阪大学大学院情報科学研究科コンピュータサイエンス専攻嶋利一真氏，藤原裕士氏ならびに Davide Pizzolotto 氏に感謝します．

最後に，日々の研究生活の中でご助言，ご協力を頂いた，大阪大学大学院情報科学研究科コンピュータサイエンス専攻ソフトウェア工学講座の皆様に厚く御礼申し上げます．

目次

第 1 章　はじめに 1
1.1　ソフトウェアの再利用 1
1.2　ソフトウェアの派生 2
1.3　ソフトウェアの保守作業 3
1.4　本研究の概要 4
1.5　本論文の構成 7

第 2 章　*b*-bit MinHash 法による高速な類似ソースファイル検索サービス 9
2.1　はじめに 9
2.2　再利用に関するソースコードの分析 10
2.2.1　コードクローン検出を用いたツール 10
2.2.2　*b*-bit MinHash 法を用いた類似度計算 11
2.3　ファイルクローンを検出するための WEB サービス 12
2.4　ユースケース 15
2.5　議論 17
2.6　まとめ 19

第 3 章　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 21
3.1　はじめに 21
3.2　背景 23
3.2.1　起源分析 23
3.2.2　類似度計算の高速化手法 24
3.2.3　*b*-bit MinHash 法を用いた類似度計算 24
3.3　提案手法 25
3.3.1　ソースファイル間の類似度計算 27
3.3.2　再利用元バージョンの特定 29
3.4　評価 30
3.4.1　実験対象 30

3.4.2 実験方法 34
3.4.3 提案手法の正確さ 35
3.4.4 提案手法の実行時性能 38
3.5 妥当性への脅威 40
3.5.1 ファイル集合の再利用バージョン判定 40
3.5.2 実験対象の妥当性 40
3.6 まとめ 41

第 4 章 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 43
4.1 はじめに 43
4.2 背景 45
4.2.1 ソフトウェア進化 45
4.2.2 高速な集合の比較手法 46
b-bit MinHash 法を用いた集合の比較 47
Linear Counting 法を用いた集合の比較 47
4.3 提案手法 48
4.3.1 b-bit MinHash 法を用いたソースコードの再利用量の推定 . . 51
4.3.2 Linear Counting 法を用いたソースコードの再利用量の推定 . 52
4.4 評価 52
4.4.1 提案手法の精度 54
4.4.2 提案手法の計算コスト 59
4.4.3 提案手法の実用性 60
4.4.4 大規模なデータセットへの適用 61
4.5 妥当性への脅威 62
4.6 まとめ 62

第 5 章 おわりに 65
5.1 まとめ 65
5.2 今後の研究方針 66

参考文献 67

図目次

1.1 状況 1 に対する手法の概要図 5
1.2 状況 2 に対する手法の概要図 6
1.3 状況 3 に対する手法の概要図 7

2.1 WEB サービスのトップページ 15
2.2 Firefox 45.0b5 に含まれる `pngwrite.c` をクエリとした場合の WEB サービスの検索結果 17

3.1 提案手法の動作例 26
3.2 正確な $J_\tau(f_1, f_2)$ の具体例 28
3.3 b-bit MinHash を用いた類似度計算例 29
3.4 1 ビット MinHash 値と Jaccard 係数の誤差の分布 37

4.1 提案手法の動作例 50
4.2 系統樹における矢印の凡例 56
4.3 Kanda らの手法と提案手法で構築されたデータセット 9 の系統樹 . . 57
4.4 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 58

表目次

2.1 Firefox 45.0b5 に含まれる pngwrite.c をクエリとした場合の類似ファイルのリスト 18

3.1 実験対象のソフトウェア・ライブラリの組 31
3.2 実験対象のソフトウェア・ライブラリを格納するリポジトリ 33
3.3 提案手法の実行結果 35
3.4 ファイル単位での正確さ 35
3.5 ソフトウェア全体とファイル単体での検出結果の違い 36
3.6 提案手法の実行時間 38
3.7 *b*-bit MinHash 不使用の場合を 100% とした提案手法の実行時間 . . 39

4.1 適用対象 52
4.2 適用結果 55
4.3 実行時間の比較 59
4.4 FreeBSD の release ブランチにおける実行時間とその精度 61

第 1 章

はじめに

1.1　ソフトウェアの再利用

ソフトウェア開発において，既存のソフトウェアを再利用して開発することは一般的である．ソフトウェアの再利用とは，既存のソフトウェアを構成する要素を，新しく開発するソフトウェアに転用し再利用することである [25]．再利用可能な構成要素は，以下の 10 個に分類できるとされている [25]．

1. アーキテクチャ（architectures）
2. ソースコード（source code）
3. データ（data）
4. デザイン（design）
5. ドキュメンテーション（documentation）
6. コストの推測（estimates）
7. ヒューマンインターフェース（human interfaces）
8. 企画（plans）
9. 要件（requirements）
10. テストケース（test cases）

この中でも，特にソースコードとドキュメンテーションはソフトウェアのライフサイクルの中でよく再利用されているという調査がある [25]．既存のソフトウェアの構成要素を再利用することは，開発中のソフトウェアの品質や開発効率を向上させることが報告されている [30]．

近年ではソフトウェアの共同開発プラットフォームである GitHub[14] などが普及したことにより，オープンソースでのソフトウェア開発がますます増加している．そのため，開発者にとって再利用可能なソフトウェア資源は膨大で，バイナリ形式だけでなく，ソースファイルそのものをコピーして再利用することも容易である．また，ソースファイルそのものを再利用した場合は，しばしば開発者はソースコードを変更

し，自身のソフトウェアで利用しやすいようにする．

オープンソースソフトウェア（Open Source Software，以降 OSS）は広く利用されているものも多く，欠陥や脆弱性の報告は頻繁に行われ，それらに対応した修正済みのバージョンも早期に公開される．報告された欠陥や脆弱性は放置すると大きなリスクになる．そのため，開発者は再利用元のソフトウェアで見つかった欠陥や脆弱性が，自身のソフトウェアに影響があるか確認し，影響があるのであれば対処する必要がある．

影響のある欠陥や脆弱性を可及的速やかに取り除くには，ソフトウェアの保守管理作業が重要である．近年ではパッケージマネージャを使うことが一般的な開発言語やフレームワークが増えているが，依然として開発者自ら管理する必要がある言語も多い．再利用元のソフトウェアが何であるか，どのバージョンを再利用したのかという情報があれば，保守管理作業は容易に行えるが，開発期間が長期に及んだり，開発担当者が変わったことでそれらの情報は失われてしまうことが報告されている [42].

再利用したソフトウェアに変更を加えていなければ，一致するソースファイルを見つけることでそれらの情報を復元することが可能である．Sasaki らは，ソースファイルの空白とコメントを取り除いた文字列に対する MD5 ハッシュ [34] を利用して，ソースコードが共通するソースファイルを検出する手法を提案した [37]．この手法は，コメント等のプログラムの実行に関係のない編集だけであれば類似するソースファイルを検出できるが，ソースコードが変更されていた場合は検出することができない．そこで，ある程度の編集を許容するように，ソースファイルの類似性から再利用元のソフトウェアを見つける手法が必要である．Kawamitsu らは，字句列の最長一致部分列（Longest Common Subsequence，以降 LCS）からソースファイル間の類似度を定義し，それを用いてライブラリから再利用したあるソースファイルの出自の候補をそのライブラリの全てのバージョンに含まれるソースファイルから検出する手法を開発した [23]．しかし，この類似度の計算には，様々な最適化を施しても合計数千万行程度のリポジトリに対して 4 時間程度時間がかかってしまう．そこで，ソースファイルの中身を直接比較するのではなく，ソースファイル間を効率的に比較し類似性を計算する手法が必要である．

1.2 ソフトウェアの派生

開発されるソフトウェアの領域に関わらず，機能の追加や欠陥修正は必要で，既存のソフトウェアプロダクトをもとに新たなソフトウェアプロダクトを開発することは頻繁に行われている．これらのソフトウェアプロダクトは派生関係を持つ．派生関係を持つソフトウェアプロダクト間では，共通する構成要素も多く，あるソフトウェアプロダクトで発見された欠陥は派生関係にあるソフトウェアプロダクトにも存在する可能性が高い．実際に，野中らが調査したある企業で開発された一連の組込みソフト

ウェアの製品ファミリでは，欠陥修正のうち 40% 程度が派生関係にあるソフトウェアプロダクトで見つかった欠陥の修正を適用した修正だった [46]．そのため，派生関係にあるソフトウェアプロダクトの関係性を把握・管理することはソフトウェアの保守作業において重要である．

そこで，Kanda らは，ソフトウェアプロダクトの集合の派生関係を，それぞれのソフトウェアプロダクトが持つソースファイルの類似性から復元する手法を提案した [22]．この手法では，ソースファイル間の LCS から計算した類似度をもとにどの程度ソフトウェアプロダクト間に共通要素があるか検出し，全域木を構築することでどのようにソフトウェアプロダクトが派生していったのか分析した．しかし，この手法では，様々な最適化を施しても合計 8,000 万行程度のソフトウェアプロダクトの集合に対して 1 日程度時間がかかることを報告しており，さらに規模が大きなソフトウェアプロダクトの集合では，実用的な時間では分析が完了しないことが予想される．そのため，ソフトウェアプロダクトの比較を効率的に行う手法が必要である．

1.3 ソフトウェアの保守作業

ソフトウェアの保守作業に関して，ISO/IEC 規格をもとに JIS 規格が制定されている [47]．これによれば，ソフトウェアの保守は以下の 4 つに目的別に分類される．

適応保守 (adaptive maintenance) 引渡し後，変化した又は変化している環境において，ソフトウェア製品を使用できるように保ち続けるために実施するソフトウェア製品の修正.

是正保守 (corrective maintenance) ソフトウェア製品の引渡し後に発見された問題を訂正するために行う受身の修正 (reactive modification).

緊急保守 (emergency maintenance) 是正保守実施までシステム運用を確保するための，計画外で一時的な修正.

改良保守 (maintenance enhancement) 新しい要求を満たすための既存のソフトウェア製品への修正.

ソフトウェアの開発において，保守作業がそのうちの 66% を占めることが過去の調査で報告されている [43]．特に，是正保守によるソフトウェアの問題点の修正は，安定したサービスを提供したり，攻撃者からソフトウェアの利用者を保護するために重要である．そのため，効率的に保守作業を行えば，ソフトウェア開発のコストを低減させることができる．そこで，今までに様々なソフトウェアの保守作業を支援する手法が提案されている．中でも，ソフトウェアの保守を困難にすると言われるコードクローンに関して活発に研究がなされてきている [1, 12, 15, 21, 26, 31, 36, 44].

しかしながら，これらの研究では，ソースコードのうちごく限られたソースコード断片同士での類似性を効率的に検出することを主眼においており，1.1 節や 1.2 節で述べたような状況ではあまり有用ではない．そのような状況でソフトウェアの保守作業

を支援するには，ソフトウェアのうち一部がどこから再利用されたのかという情報と，ソフトウェアプロダクトがどのような派生関係を持つのかという情報を提示する必要がある．これらの情報をもとに，開発者は自身が開発した部分に由来しない欠陥や脆弱性を修正する必要があるのか判断することが可能となる．そのため，ソフトウェアの保守作業を支援するためには，それらの情報を適切かつ効率的に提供することが重要である．

1.4 本研究の概要

ここまで述べてきたように，ソフトウェア間のソースコードの類似性を分析し，ソフトウェアの保守作業に有用な情報を提示することは重要である．また，そのために開発された既存手法には，主に実行速度と精度の面で改善の余地がある．そこで，本研究では，ソフトウェア間の分析において，近似的に類似性を計算する手法を導入することで，それらの課題を解決することを目指す．

本研究で主に使用する，類似性を近似計算する手法は *b*-bit MinHash 法である．この手法は，集合間の Jaccard 係数を時間的・空間的に効率よく計算し，Jaccard 係数が大きいほど誤差が少ないという特徴がある．そのため，前節で述べたような，ほとんどが一致しているような再利用における分析に有用である．この手法をソースファイル単位間の類似性とソースファイル集合間の類似性の分析に対して適用することで，既存手法の課題であった速度について解決を目指す．

精度の面では，複数のソースファイルの組について効率的に分析できるため，ファイル集合での分析が容易になり，いくつかの状況ではより精度の向上を見込むことができる．そこで本研究では，近似計算による効率化と分析の精度への影響を確認するため，以下に示す実際のソフトウェア開発において想定される 3 つの状況に対して，保守作業を支援する手法を提案し，評価する．

状況 1 あるソースファイルの出自が失われた場合．
状況 2 ソフトウェア中に含まれる再利用したライブラリのバージョンが失われた場合．
状況 3 既存のソフトウェアとの派生関係の情報が失われた場合．

状況 1：あるソースファイルの出自が失われた場合

状況 1 では，自身のソフトウェアに別のソフトウェアからソースファイルをコピーして再利用しているが，その出自が時間が経つにつれてわからなくなってしまった場合を想定している．この失われた情報を復元するために，OSS に含まれるソースファイルから *b*-bit MinHash 法を利用して高速にクエリとなるソースファイルと類似するものを検索する WEB サービスを提案する．この WEB サービスでは，ソースコード

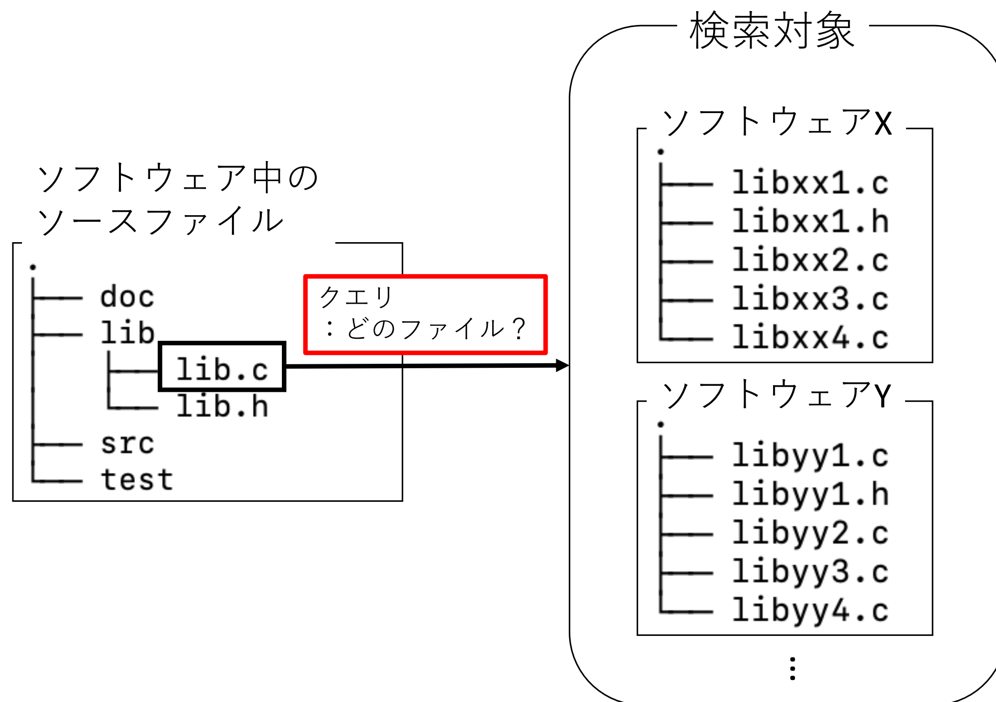


図 1.1: 状況 1 に対する手法の概要図

に変更が加えられていても類似ファイルを検出するため，ソースファイルを字句の集合として扱い，集合間の Jaccard 係数が閾値以上であるソースファイルを，OSS のソースファイルを登録したデータベースから検索する．*b*-bit MinHash 法という局所性鋭敏型ハッシュ（LSH）の一種を用いることで Jaccard 係数を高速に計算し，1,000 万個単位のソースファイルから数百ミリ秒でクエリに類似するソースファイルを抽出できる．

図 1.1 に状況 1 に対する提案手法の概要図を示す．図の左側に開発中のソフトウェアを，右側に検索対象とするオープンソースソフトウェアの集合を示している．開発中のソフトウェアで用いられている，あるオープンソースソフトウェアに由来するソースファイル `lib.c` をクエリとし，検索対象とするオープンソースソフトウェアの集合からクエリに類似するソースファイルを検索する．この際，前述の通りソースファイルを字句の集合と見做して類似度を計算するため，`lib.c` が編集されていたとしても，検索対象から元となったソースファイルを検索できる．

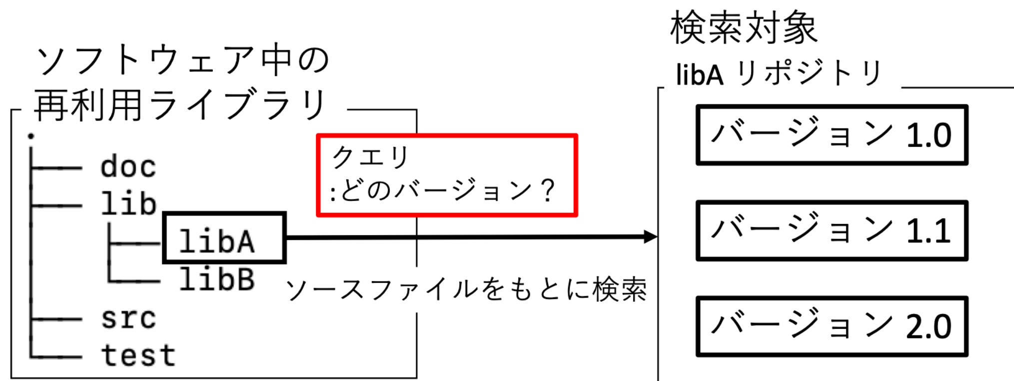


図 1.2: 状況 2 に対する手法の概要図

状況 2：ソフトウェア中に含まれる再利用したライブラリのバージョンが失われた場合

状況 2 では，自身のソフトウェアにおいて，ある機能を実現するためにライブラリを再利用し取り込んだが，時間経過により再利用した際のライブラリのバージョンがわからなくなってしまった場合を想定している．この失われた情報を復元するために，分析対象のソフトウェアに含まれるライブラリから再利用したソースファイル群とライブラリの各バージョンに含まれるソースファイル群とを比較し，再利用元のライブラリのバージョン集合から，分析対象のソフトウェアが再利用しているバージョンを効率的に特定する手法を提案する．この手法では，既存手法 [23] で行われていたソースファイル単位での分析手法を拡張し，ソースファイル集合同士を比較することで再利用元のライブラリのバージョンを特定する．ソースファイル集合同士の比較では，ソースファイル単位の類似度計算に高速な手法を適用することで効率化し，ソースファイル単位だけでなく，集合同士の比較でも既存手法より大幅に高速化・高精度化する．

図 1.2 に状況 2 に対する提案手法の概要図を示す．図の左側に開発中のソフトウェアを，右側に検索対象とするライブラリ libA のリポジトリを示している．開発しているソフトウェアの `libA` ディレクトリに含まれる，あるライブラリ libA から再利用したソースファイル群をクエリとして，ライブラリ libA のリポジトリから，`libA` の元となったライブラリのバージョンを検出する．

状況 3：既存のソフトウェアとの派生関係の情報が失われた場合

状況 3 では，ソフトウェアの開発が進むにつれ，リリースしたソフトウェアプロダクトの集合がどのような派生関係を持つのかわからなくなってしまった場合を想定し

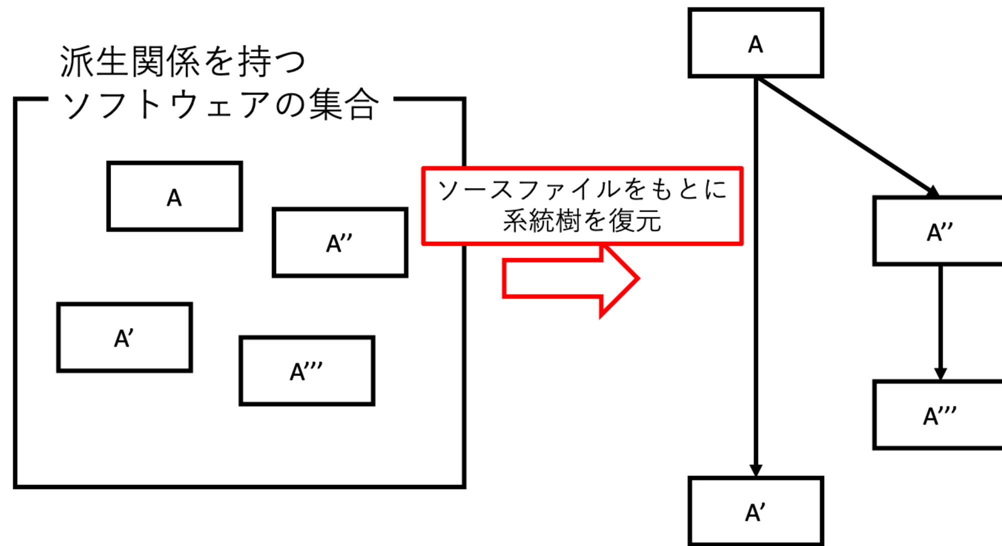


図 1.3: 状況 3 に対する手法の概要図

ている．この失われた情報を復元するために，分析対象とするソフトウェアプロダクトの集合について，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定義し，その値を用いて系統樹を効率的に再構築する手法を提案する．この手法では，既存手法 [22] で行われていた，ソースコードの類似度とその差分による有向全域木の構築手法を拡張し，軽量なデータ構造を適用することで精度を下げる事なく大幅に高速化する．軽量なデータ構造には b-bit MinHash 法と Linear Counting 法をそれぞれ個別に適用し，2 通りの効率化を行う．

図 1.3 に状況 3 に対する提案手法の概要図を示す．図の左側に派生関係を持つソフトウェアの集合を，右側にはそこから構築したそれらの系統樹を示す．ソフトウェア A，A’，A’ ’，A’ ’ ’ が持つソースファイル群の再利用量を用いて，その系統樹を構築する．

1.5 本論文の構成

以降，第 2 章では状況 1 に対して提案する，b-bit MinHash 法を利用したクローンされたファイルを検索するための WEB サービスについて述べる．第 3 章では状況 2 に対して提案する，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について詳細に述べる．第 4 章では状況 3 に対して提案する，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について詳述する．最後に，第 5 章では本論文のまとめと将来の研究方針について述べる．

第 2 章

b-bit MinHash 法による高速な類似ソースファイル検索サービス

本章では，まず初めに軽量な類似度計算手法である b-bit MinHash 法をソースファイルに適用した際の有用性について検証するため，ソフトウェアから再利用されたソースファイルの出自を検索するための WEB サービスを提案した．また，ユースケースでその有用性を確認し，理論的にどの程度の見逃し率や不要な要素が紛れ込んでしまうか検討した．

2.1 はじめに

多くのソフトウェア開発者は，開発の効率とソフトウェアの品質を向上させるために，自分のプロジェクトの一部としてオープンソースソフトウェアを再利用している．ただし，再利用元のソフトウェアに欠陥や脆弱性が発見された場合，その修正を自身のプロジェクトに再利用したソフトウェアにも適用する必要がある．修正の影響範囲は再利用元ソフトウェアのバージョンに基づいて発表されるため，何をどこから再利用したのかという情報はメンテナンスをする際に重要である．しかし，再利用に関する情報は，開発履歴の中に記録されていなかったり，容易に失われてしまったりすることがある [42].

このような情報を復元するためには，コードクローン検出技術が有効である．開発者は，コードクローン検出ツールを用いて，ソースファイルと既存の OSS ファイルとの間のクローンを検出し，その結果をもとに再利用情報を復元できる．この考えに基づいて，我々の研究グループは Ichi Tracker[15] と FC-Finder[37] というツールを開発した．Ichi Tracker はソースコード検索エンジンを利用して，コード断片が OSS においてどのような変遷をたどっているのか分析した．また，FC-Finder は，Unix OS の一つである FreeBSD のソースファイルについて，空白やコメントを除いたソースコードの MD5 ハッシュが一致するソースファイルの集合をファイル単位でのクロー

ン集合として検出した．これらのツールは，OSS に含まれるソースファイルをデータベースや検索システムに蓄積したものを対象としている．

しかし，これらのツールにおいて OSS のデータベースを常に更新しておくのは個々人の開発者にとって作業コストが高い．この問題は，ツールを WEB サービスとして提供することで解消することができる．WEB サービスを利用するだけであれば，ツールの利用者はデータベースの管理をする必要がなく，手元にあるファイルをクエリとして送信するだけで良くなり，利便性を向上させることができる．ただし，クエリとしてファイルを送信できない場合，例えば組織のセキュリティポリシーによりソースファイルの秘匿が義務付けられているような企業内のソフトウェア開発者は，利用することができない．

ソースファイルを送信することなく同一のファイルを検索するサービスとして，Software Heritage プロジェクト [5] がある．Software Heritage プロジェクトでは，ファイルの SHA-1 ハッシュ値を入力として，OSS プロジェクトの中に存在する完全に一致するファイルを報告する簡潔な Web サービスを提供している．このサービスは一定の有用性があるものの，クエリファイルが再利用元のソースファイルから変更されている場合には検出できない．

この問題を解決するために，我々は Cloned File Detector という WEB サービスを提案する．利用者は，ソースファイルから生成されるシグネチャをクエリとして WEB サービスに送信することで，OSS に含まれるクエリと類似するファイルを検索できる．Cloned File Detector は b-bit MinHash 法 [24] を採用しており，MinHash 法 [3] で生成されたファイルのシグネチャを用いて，2 つの文書間の類似度を推定する．ファイルのシグネチャは SHA-1 ハッシュ値，コメント・空白を除いた SHA-1 ハッシュ値，言語，トークン数，b-bit MinHash 法で生成されたビット列で構成される．提案する WEB サービスは OSS に含まれるソースファイルと投稿されたシグネチャの類似度を計算することができるが，WEB サービスはシグネチャから元のファイルの内容を復元することが困難であるため，企業のソフトウェア開発者であっても利用が可能となる．

以降，2.2 節では研究背景について，2.3 節では提案する WEB サービスについて詳しく述べる．また，2.4 節ではサービスのユースケースについて説明し，最後に 2.5 節にまとめを述べる．

2.2 再利用に関するソースコードの分析

2.2.1 コードクローン検出を用いたツール

再利用したソースコードの出自を調べる際，コードクローンを検出することは有効である．これまでにさまざまなコードクローン検出を応用したツールが提案されて

いる．

Ichi Tracker [15] では，コードクローン検出ツールである CCFinder[21] を利用してソースコード断片についてさまざまな OSS においてどのように分布しているのか分析した．このツールでは，ソースコード検索サイトを用いて，クエリとしたあるソースコード断片と，検索結果に現れたソースコード断片に対してコードクローン検出を行い，クローンであると判定された場合にそのソースコード断片を収集する．その後，収集したソースコード断片について，編集時刻を参照することにより，どのような変遷をたどったのか表示する．

FCFinder[37] では, ファイル単位でのクローンを検出するツールを提案し, FreeBSD で使用可能なソフトウェアパッケージの集合である FreeBSd Ports Collection[33] に含まれるソースコードのうち，どれくらい同一のソースファイルが用いられているか調べた．このツールでは，C 言語のソースファイルに関して，ソースコードの字句分割を行い，ある一定以上の数の字句を含む場合に，その字句列からコメントや空白を省いて MD5 ハッシュ [34] を計算し，その値が一致するソースファイル全てをファイルクローンの集合とする．FreeBSD Ports Collection に対する適用実験では，ファイルクローンの集合の要素数に対するファイルクローン集合の数や，異なるパッケージ間のファイルクローン数に対してファイルクローンが検出されたプロジェクトの数が，冪乗則に従うことを確認した．ただし例外が存在し，そのような場合はパッケージ間で機械的にコピーされたようなファイルが大量にあることが確認できた．

2.2.2 *b*-bit MinHash 法を用いた類似度計算

類似度の計算そのものを高速化する手法として，類似度の 1 つである Jaccard 係数 [18] には，その推定値を高速に求める MinHash 法 [2, 3] が提案されている．Jaccard 係数は集合間の類似性を計測する指標であり，以下の式で表される．

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

ここで S_1, S_2 は，それぞれファイルの内容を表現した集合である．MinHash 法は，任意のハッシュ関数 h_i が与えられたとき，以下のように集合 S からハッシュ値が最小である要素を取り出す関数 $m_i(S)$ を使用する．

$$m_i(S) = \min_{s \in S} \ h_i(s)$$

2 つの集合 S_1, S_2 に対して $m_i(S_1), m_i(S_2)$ を計算すると，$m_i(S_1) = m_i(S_2)$ となる確率が $J(S_1, S_2)$ に一致する．そのため，複数のハッシュ関数を用意すれば，これらのハッシュ値の一致割合から，Jaccard 係数の推定値を求めることができる．

MinHash 法を省メモリで実現する方法として *b*-bit MinHash 法 [24] が提案されている．この手法では，MinHash 法で計算したハッシュ値 $m_i(S)$ の下位 b ビットのみ

をハッシュ値として用いる．たとえば $b = 1$ のとき，b-bit MinHash 法のハッシュ関数 $b_i(f)$ は以下の式で表される．

$$b_i(S) = \mathrm{LSB}(m_i(S))$$

ここでの LSB は Least Significant Bit を表す．

2 つの集合 S_1, S_2 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．ハッシュ値の下位 b ビットが偶然一致する確率は $\frac{1}{2^b}$ であるので，$P(S_1, S_2)$ は 以下の式で表される．

$$P(S_1, S_2) = \frac{1}{2^b} + \left(1 - \frac{1}{2^b}\right) \times J(S_1, S_2)$$

最も計算量が少なくなるのが $b = 1$ のときであり，k 個のハッシュ関数を用いても k ビットの記憶域で済む．$P(S_1, S_2)$ の近似値として，k 個のハッシュ値の一致割合 $P_o(S_1, S_2)$ を用いて，前述の式を変形した以下の式により集合 S_1, S_2 間の Jaccard 係数の推定値 $J_e(S_1, S_2)$ を計算することが可能である．

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

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

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．

2.3 ファイルクローンを検出するための WEB サービス

提案手法ではあるクエリファイル q に類似するファイルをデータベースから検出する．本手法では，ソースファイル間の類似度にソースファイルの trigram 多重集合に対する Jaccard 係数を利用する．これは，文字列の一致を推定するのに用いられている [40]．ソースファイル f に対する trigram の多重集合を $\tau(f)$ とすると，本研究におけるファイル f_1, f_2 間の類似度は 2.2.2 項の式を用いて，以下の式で定義される．

$$\mathrm{sim}(f_1, f_2) = J(\tau(f_1), \tau(f_2))$$

trigram を構成する要素はソースコード中の字句で，字句解析を用いてコメントと空白を除いて字句列を抽出する．

厳密に類似度を計算する場合，WEB サービスの利用者はソースファイルをサーバに送信する必要がある．そこで，企業の開発者が公開できないソースファイルをクエリにできるように，提案手法では類似度の計算にファイルの内容が不要な b-bit MinHash 法 [24] を利用する．大まかにいうと，この手法はソースファイルに含まれ

る trigram から計算した b ビットのハッシュ値を k 個用意した，$b \times k$ ビットのビット列同士の排他的論理和で 1 となるビットの数，つまりハミング距離を計算することで，ファイル間の類似度を推定する．時間計算量と空間計算量から計算されるコストが最小となるのは $b = 1$ のときであると論文中で述べられている．そこで，今回の実装では $b = 1, k = 2048$ とした．言い換えると，ソースファイルから 2048 個の trigram をランダムに取り出し，それを 2048 ビットのファイルシグネチャとする．ファイル q から得られたファイルシグネチャ $s(q)$ は以下に示す式で q とデータベースに含まれるファイル f との類似度を推定する．

$$\mathrm{sim}_e(q, f) = 1 - \frac{d(s(q), s(f))}{k} \times 2$$

ここで，$s(f)$ はファイル f のファイルシグネチャで，$d(s(q), s(f))$ はシグネチャ間のハミング距離を表す．また，比較する 2 つのファイルについて，SHA-1 ハッシュ値が一致した場合は $\mathrm{sim}_e(q, f)$ を 1 として扱い，$\mathrm{sim}_e(q, f)$ が負の値になった場合は 0 として扱う．ただし，この推定類似度には誤差が存在する．事前に調査したところ，99.9% のファイルの組み合わせについて，絶対誤差は 0.05 よりも小さかった．

データベースからクエリ q に類似するファイルのリストを抽出する際，以下の式で絞り込みを行う．

$$\mathrm{CFD}(s(q)) = \{f | \mathrm{sim}_e(s(q), s(f)) \geq \theta\}$$

これは推定類似度 sim_e について，閾値 θ 以上であったファイル f の集合を表す．

Algorithm 1 にデータベースからクエリに類似するファイルを検索するアルゴリズムを示す．アルゴリズムでは，クエリファイル q とデータベース D，類似度の閾値 θ を入力として，q に類似するファイルの集合 R を出力する．計算量を減らすため，まずはじめに SHA1(q) により計算したそれぞれのファイルの SHA-1 ハッシュ値を比較し，一致している場合に類似度の計算をせず D に含まれるファイル f を R に加える．次に，$N(q)$ により，空白とコメントを除いてファイルを正規化し，その SHA-1 ハッシュ値が一致する場合に f を R に加える．この際，$\mathrm{sim}_e(q, f)$ は完全に一致する場合 3.0 に，正規化して一致する場合は 2.0 とする．これは，最後に類似度順に並び替えた際に，より一致しているものを上位に表示するためである．最後に，$\mathrm{sim}_e(q, f)$ を計算し，θ 以上である f を R に加える．ただし，比較を行うファイルは，同じプログラミング言語で記述されたソースファイル同士のみとする．本研究では，ある程度類似度が高いファイルにクローン元となったファイルが存在すると仮定し，$\theta = 0.7$ としている．

また，データベースから検出したファイルは，$\mathrm{sim}_e(q, f)$ の値で降順に並べ替える．その際，SHA-1 ハッシュ値で一致した場合の 3.0，正規化して一致する場合の 2.0 はともに 1.0 として表示する．ただし，同じ値の場合はそのファイルを含んでいる OSS 中でのファイルパスをアルファベット順に並べている．ファイルの SHA-1 ハッシュ

Algorithm 1 WEB サービスの検索アルゴリズム

入力

q：検索対象のソースファイル

D：データベースに含まれるソースファイルの集合

θ：類似度の閾値

出力

R：q に類似する D に含まれるソースファイルの集合

```
1:  Initialize R ← φ
2:  for f ∈ D do
3:      if f is same file extension to q then
4:          if SHA1(f) = SHA1(q) then
5:              sim_e(q, f) ← 3.0
6:              R ← f
7:          else if SHA1(N(f)) = SHA1(N(q)) then
8:              sim_e(q, f) ← 2.0
9:              R ← f
10:         else if sim_e(q, f) >= θ then
11:             R ← f
12:         end if
13:     end if
14: end for
```

値や正規化したソースコードの SHA-1 ハッシュ値が同一であるファイルが見つかった場合は，ハイライトして表示する．

今回用意したデータベースは，the Snapshot Archive of Debian GNU/Linux[10] で公開されている 2016 年 8 月でのスナップショットを元に構築した．このアーカイブは 2005 年から現在までの，Debian 向けにリリースされたパッケージの全てのソースコードを含んでいる．Debian のメンテナは自身で用意したパッチを適用することがあるので，データベースの構築時には “`*.orig.*`” というパターンにマッチするようなアーカイブファイルのみを抽出した．

さまざまなプログラミング言語が OSS の開発では利用されているが，現状の実装では Java と C/C++ ファイルにのみ対応している．しかし，ソースコードの字句解析には ANTLR を利用しており，またファイルのシグネチャを計算する際には言語に依存しないハッシュ関数を用いているため，容易に対応言語を拡張できる．

この手法は，ある種の Type 3 コードクローン検出手法とも言え，Type 2 コードクローンのみを取り出すことはできない．しかしながら，検索結果には双方が含まれて

Cloned File Detector

Cloned File Detector is a web service to search a file cloned from OSS projects. This service employs b-bit minwise hashing technique that estimates similarity of source files using only their hash signatures.

Three steps to use the service:

1. Download our offline hash computation tool. It is based on JavaSE 8 and ANTLRv4. (The source code will be available soon.)
2. Execute the tool for your source code. The tool translates C, C++ and Java files into file signatures.
```
java -jar cfd-client.jar yoursource.c
```
3. Submit a hash signature (a single line generated by the tool) to the following form. The web service reports files in our database that are likely similar to the query file.

File Signature Input result of cfd-client.jar

Submit

If you would like to try the web-service without installing the tool, please use one of the following lines created from several versions of libpng library.

File	Signature (including File SHA1, File SHA1 w/o white space and comments, File Extension (.c/.cpp/.java), #Tokens, b-bit Minwise Hash Signature)
pngwrite.c in Firefox 45.0b5	c441996243c9cf6b0032b5a1437626995b6f8ffd,1391a0b7dc4c1e0ed6aae2521aa24c3e402ccf07,2,8415,hURH
png.c in libpng-1.6.21.tar.gz	a154546e50871bc29160c8ff49b5b2859527f9e2,8d54538be5d1aa5d962901432a759ff28d1f0e7b,2,17429,lWVs
png.c in chromium-50.0.2661.75	9ab8cb7c37233c641a813c39e9ac5c9c1c046465,bd7161742c4eb42e093d74c3cffb8d9a2892732b,2,4092,z7X

Note: The current version of this web service uses a database of OSS projects including 573,000 files, due to the limited capacity of our web server. We are planning to move the service to a new server to provide a larger database.

Our web service also accepts a source file. If it is provided, our server returns actual similarity.

Upload File Select File Browse

File Type C

Submit

図 2.1: WEB サービスのトップページ

いる．

2.4 ユースケース

開発した WEB サービスのデモを公開している[*1]．サーバーの利用可能な資源の都合上，デモでは用意したデータセットのうち，573,000 個のファイルのみをデータベースに格納している．また，ファイルのシグネチャを計算するツールをデモページで配布している．そこから得られるファイルのシグネチャを入力することで，そのファイルに類似したファイルのリストを WEB サービスは表示する．

図 2.1 に WEB サービスのトップページのスクリーンショットを示す．一番上のブロックでは，簡単な WEB サービスの概要を記載し，配布しているシグネチャの計算

[*1] https://sel.ist.osaka-u.ac.jp/webapps/ClonedFileDetector/

ツールの使い方の説明をしている．その下のブロックにシンプルな検索フォームが記載している．テキストボックスに計算したシグネチャをコピーし，Submit ボタンを押すことで，検索を実行する．また，検索フォームの下にはサンプルのための OSS 中のいくつかのファイルのシグネチャを表にして示し，自分でファイルシグネチャを計算することなく WEB サービスを試すことができるようになっている．一番下のブロックでは，ファイルのシグネチャを計算せず，直接ファイルをアップロードすることでデータベースから類似ファイルを検索するフォームを記載している．この場合，アップロードするファイルの言語を指定する必要がある．

図 2.2 に開発した WEB サービスの出力例の一部を示す．ただし，この例ではセータセット全てを含むデータベースを利用している．クエリとして入力するファイルは，ウェブブラウザアプリケーションである Firefox のバージョン 45.0b5 に含まれる `pngwrite.c` で，このファイルは画像形式の一つである PNG 形式を取り扱うためのライブラリ libpng の，バージョン 1.6.21 の同名ファイルから再利用されている．WEB サービスはクエリと類似するファイルを検索し，その SHA-1 ハッシュ値，空白とコメントを除いた SHA-1 ハッシュ値，ファイル名，類似度の推定値をそれぞれ表示している．ただし，2 つの SHA-1 ハッシュ値は全てを表示すると大きく幅をとってしまうため上位 8 文字のみを表示し，全体はマウスホバーをすることで確認できるようにした．

また，クエリファイルと検出された類似ファイル間の類似度によって，列をハイライトする．ファイルの SHA-1 ハッシュ値が一致している場合，つまり Algorithm 1 における類似度が 3.0 の場合は濃い緑としている．コメントと空白を除いた SHA-1 ハッシュ値が一致している場合，つまり Algorithm 1 における類似度が 2.0 の場合は薄い緑としている．それ以外の場合は，視認性を高めるため，白色と薄灰色の交互としている．

この図を見ると，9 つの類似ファイルがデータベースから見つかったことがわかる．個別に確認すると，まず 1 つ目のファイルは SHA-1 ハッシュ値が一致しており，内容が同一であるとわかる．ファイル名からも，データベース中に完全に一致したファイルがあると確認できる．次に，2 つ目のファイルは，コメントと空白を除いて同様のファイルであるとわかる．ファイル名から，クエリファイルが含まれているバージョンよりも新しい Firefox のバージョンで，コード上は同一のファイルが使われているとわかる．3 つ目以降のファイルはソースコードに若干の違いが存在する．3 つ目は Firefox と同じく Mozila が開発しているメーラーの Thunderbird のあるバージョンで，クエリファイルの含まれる Firefox のバージョンに最も近いリリース日の Thunderbird のバージョンだった．4 つ目と 5 つ目は異なるソフトウェアで用いられている同じライブラリから再利用された同名ファイルである．最後に，6 つ目のファイルは Firefox のリポジトリにおいて記録されていたライブラリの再利用元のバージョンに含まれるファイルである．ただし，結果の表では表示されていないが，SHA-1

No.	File SHA-1	File SHA-...	File Name	Similarity
1	c4419962...	304cd806...	firefox-45.0b5/media/libpng/pngwrite.c	1.000000
2	3c27631a...	304cd806...	firefox-47.0b5/media/libpng/pngwrite.c	1.000000
3	4410037a...	3fee91a3e...	thunderbird-44.0b1/mozilla/media/libpng/pngwrite.c	0.999026
4	96b86f7d8...	e47a2fbd3...	stella-4.7/src/libpng/pngwrite.c	0.974437
5	f1faf25c36...	0e3bdb81...	qtbase-opensource-src-5.6.0/src/3rdparty/libpng/pngwrite.c	0.974190
6	d5d6ff676...	0e3bdb81...	texlive-bin-2015.20160213.39691/libs/libpng/libpng-src/pngwrite.c	0.974190
7	bc1ad57e...	3f63477c2...	libpng-1.6.22/pngwrite.c	0.913829
8	2abbd0b8...	d78c1833...	mozilla-release/media/libpng/pngwrite.c	0.829895
9	afb7c2516...	fac9ae132...	mozilla-beta/media/libpng/pngwrite.c	0.824893

図 2.2: Firefox 45.0b5 に含まれる `pngwrite.c` をクエリとした場合の WEB サービスの検索結果

ハッシュ値が同一のファイルについては一つのみを表に記載し，残りはファイル名にマウスオーバーすることで表示される．ここでは，texlive 2015 においては libpng 16.21 の同名ファイルが変更されずに用いられていたため，このような表示となった．コメントと空白なしの SHA-1 ハッシュ値を見ると，5 つ目と 6 つ目は同じであるとわかる．これは，ソースコードは同一のファイルであったことを示している．

また，それらのファイルについて，厳密な類似度を計算し，推定類似度の誤差がどの程度なのか表 2.1 に示す．比較がしやすいように表には検索結果と同様の項目も記載する．表から分かる通り，類似度が高いほど誤差が少ないことが分かる．

厳密な類似度の値はクエリの内容が必要なため，WEB サービスからは直接提供しない．ただし，利用者は検索結果のファイルをダウンロードすることで，WEB サービスにアップロードすることなく検索結果との実際の類似度を計算することができる．加えて，クエリと完全に一致するファイルがデータベースに見つかったとしても，WEB サービスはクエリの内容を知ることは難しい．

2.5 議論

今回利用した b-bit MinHash 法で推定した Jaccard 係数には誤差がある．その影響で，検索時の閾値より小さいファイルを誤って検索結果に含めてしまう可能性がある．

表 2.1: Firefox 45.0b5 に含まれる `pngwrite.c` をクエリとした場合の類似ファイルのリスト

順位	SHA-1	空白とコメントを除いた SHA-1	ファイル名	推定類似度	厳密な類似度
1	c4419962...	304cd806...	firefox-45.0b5/media/libpng/pngwrite.c	1.000000	1.000000
2	3c27631a...	304cd806...	firefox-47.0b5/media/libpng/pngwrite.c	1.000000	1.000000
3	4410037a...	3fee91a3...	thunderbird-44.0b1/mozilla/media/libpng/pngwrite.c	0.999023	0.999026
4	f1faf25c...	0e3bdb81...	qtbase-opensource-src-5.6.0/src/3rdparty/libpng/pngwrite.c	0.977539	0.974437
5	d5d6ff67...	0e3bdb81...	texlive-bin-2015.20160213.39691/libs/libpng/libpng-src/pngwrite.c libpng-1.6.21/pngwrite.c	0.977539	0.974190
6	96b86f7d...	e47a2fbd...	stella-4.7/src/libpng/pngwrite.c	0.977539	0.974190
7	bc1ad57e...	3f63477c...	libpng-1.6.22/pngwrite.c	0.914062	0.913829

そこで，b-bit MinHash 法における類似度の分散から不要なファイルの混入確率を検討する．元論文 [24] から，ソースファイル f_1, f_2 間の推定類似度の分散 $V(f_1, f_2)$ は以下の式で表される．

$$V(f_1, f_2) = \frac{\mathrm{sim}(f_1, f_2)\,(1 - \mathrm{sim}(f_1, f_2))}{k}$$

ただし，$k = 2048$ とする．今回の実装では閾値を 0.7 としたので，$\mathrm{sim}(f_1, f_2) = 0.7$ のときの推定類似度の分散は，$0.7(1 - 0.7)/2048 = 0.000103$ である．このとき標準偏差は $\sigma = \sqrt{V(f_1, f_2)} = 0.010126$ である．

論文から推定類似度の分布は二項分布であることがわかり，正規分布に近似できるとすると，それぞれの信頼区間の信頼度から，99.99% が 0.7 ± 0.040504 の範囲に含まれる．累積分布確率を用いて，不要なファイルが検索結果に含まれる確率を考えると，ある類似度 x までの累積分布関数 $\mathrm{CDF}(x)$ は誤差関数 $\mathrm{erf}(x)$ を用いて，以下の式で表される．

$$\mathrm{CDF}(x) = \frac{1}{2}\left(1 + \mathrm{erf}\left(\frac{(x - \mathrm{sim}(f_1, f_2))^2}{\sqrt{2V(f_1, f_2)}}\right)\right)$$

この式から累積確率を計算すると，類似度 0.75 以上の全てのファイルを検索結果に含めるために閾値を 0.7 とした時，100% の確率で漏らさず検索することができることがわかる．反対に，0.7 より小さい類似度のファイルが 0.7 以上であると推定される確率は，0.409% 程度である．これらから，今回の実装において，不要なファイルが検索結果に含まれる確率は十分小さく，また必要なファイルは十分に含まれていることがわかる．

2.6 まとめ

クエリとしてファイルシグネチャを受け取り，データベース中から類似するファイルを検索する，Cloned File Detector という WEB サービスを開発した．このサービスでは時間計算量と空間計算量がともに効率的な b-bit MinHash 法を類似度計算に採用することで，高速に巨大なデータベースから類似ファイルを検索することを可能にした．この手法で計算した類似度には誤差があるが，設定した閾値では十分に類似するファイルを検索することができ，かつ不要なファイルが表示される可能性は少ない．また，ソースファイルのアップロードが必要ないため，ソースファイルから生成したシグネチャのみをクエリとすることで，ソースファイルを公開出来ない企業の開発者であっても本サービスを利用可能である．

将来的な発展として，ソースコードのホスティングサイトである GitHub などから自動でファイルを収集し，データベースを継続的に拡張することが挙げられる．また，対応するプログラミング言語を増やすことで，より検索する対象を拡張することが挙げられる．

第 3 章

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

本章では，第 2 章で有用性を確認したソースファイルにおける *b*-bit MinHash 法の適用について，それを拡張してソースファイル集合の類似性を検出する手法に応用する．この手法では，ある再利用したソースファイル集合に対して，再利用元のソースファイル集合を検出する．また，拡張した手法について評価実験を行い，その有効性を確認した．

3.1 はじめに

ソフトウェア開発の現場において，オープンソースソフトウェア（Open Source Software，以降 OSS）を再利用することが一般的に行われている [8]．OSS の再利用は，独自に開発した場合と比べて品質や信頼性の向上，機能の開発期間の短縮などの効果をもたらす [30]．

ソフトウェア開発者は，しばしば，OSS として作られたライブラリのソースファイルを再利用して自身のソフトウェアに取り込む．ソースファイルは，常にそのまま使われるわけではなく，たとえば Mozilla が開発している Gecko Engine では，ファイル圧縮に関するライブラリである zlib をコピーしたのち，独自のマクロを使用するよう zconf.h ファイルに記述を追加している．また，Google の開発するスマートフォン用 OS である Android のバージョン 8.0.0 は 画像ファイルを扱うライブラリである libpng を再利用しており，ファイルの 1 つ pngpread.c にバグ修正のパッチを適用した状態で使用している．

再利用したライブラリのバージョン番号は，ソフトウェアの保守において重要な情報である．再利用したライブラリに含まれる脆弱性や欠陥の影響を回避するために，

開発者はライブラリを新しいバージョンに更新するか，修正パッチを適用しなければならない [4]．脆弱性や欠陥の影響範囲はバージョン番号によって記述されることから，影響の有無を把握し適切な更新作業を実施するために，正しいバージョン番号を把握することが必要となる．

しかしながら，プロジェクトの開発期間が長くなるにつれ，ソフトウェアをどこから再利用したのか，ソフトウェアのどのバージョンを再利用したのかという情報が失われてしまうことがある [42]．全てのソースファイルについて内容を比較すると高コストだが，ソースファイルに編集が加えられていなければ，SHA-1 などのハッシュ値を各ソースファイルに対して計算することで，ソフトウェア中のソースファイルと一致するソースファイルを持つライブラリのバージョンを簡単に検出できる．しかし，先に紹介した例のように編集が加わっている場合には，単純な同一性の判定だけではバージョン情報を復元することができない．

本研究では，分析対象ソフトウェアのソースファイル集合とライブラリのバージョンごとのファイル集合を比較することで，分析対象のソフトウェアが利用しているライブラリのバージョンを検出する手法を提案する．本手法は，既存手法 [23] で用いたファイル間の類似度の考え方をファイル集合に拡張し，ファイル集合間で最も似ているファイルの組における類似度の合計を集合間の類似度とすることで，ファイル集合全体として最も類似するライブラリのバージョン 1 つを自動的に検出する．また，局所性鋭敏ハッシュ（Locality Sensitive Hashing，以降 LSH）の 1 つである b-bit MinHash 法を用いたファイル比較により，ファイル間の類似度の高速な計算を実現する．この手法を用いれば，ソフトウェア開発者は再利用したライブラリの更新が必要かどうか，ソースコードからバージョン番号を高速に復元し，判断できるようになる．

本論文は著者らの発表内容 [16] を拡張したものである．発表 [16] では b-bit MinHash による類似度の推定値を真の類似度の計算の枝刈りのためだけに使用していたが，本論文では類似度の推定値をそのまま合算することでバージョンの検出に用いる．また，ライブラリの名称はディレクトリ名等から判断ができる状況を想定し，既存手法 [23] との比較実験を追加した．本論文の貢献は以下のとおりである．

- b-bit MinHash による類似度の推定値を直接合算することでソースファイル集合の再利用元を高速かつ正確に検出する手法を提案した．
- 提案手法が実用的であることをオープンソースソフトウェアにおける実際の再利用の事例を用いた評価実験で確認した．ソフトウェアのすべてのバージョンについてライブラリのどのバージョンを再利用しているかを調べる場合，平均で 104 秒程度の処理時間で，99.3% の割合で再利用しているライブラリのバージョンを正しく判定することができた．従来手法 [23] では最大で数時間の計算が必要であったが，それを大幅に短縮し，同時に正確さも向上している．

以降，2 章では研究の背景について述べ，3 章では提案手法を詳しく説明する．4 章

では提案手法の評価を行い，5 章で妥当性への脅威について述べる．最後に，6 章でまとめを述べる．

3.2 背景

3.2.1 起源分析

起源分析とは，ソフトウェア内に存在するソースコードから，その作成に使われた（コピー元となった）他のソフトウェアやライブラリなどの起源を分析する手法である．コピーされたソースコードは，編集されていなければその起源を辿ることは容易いが，その後それぞれのソフトウェアに合わせて手を加える場合があり，ソースファイルの類似性に着目した分析が行われている．

プロジェクト間でのソースファイルの再利用を検出するために，コードクローン検出と呼ばれる互いに同一または類似するコード片の組を検出する手法が適用されている．German らは，複数のオープンソースプロジェクト間のコードクローンについて調査し，互いにどの程度コードクローンを持つのか分析することで，そのコードクローンの起源となるプロジェクトを特定した [27]．Inoue らは，インターネット上の OSS 中から，対象とするコード片に対するコードクローンを検出し，それらを時系列に並べることでその利用の変遷を表示する Ichi Tracker を提案した [15]．また，Sasaki らは，動作に影響のないコメントや空白を削除した場合のコードクローンに着目し，BSD 系 OS の一種である FreeBSD で利用されている OSS 同士でどのようにソースコードが再利用されているか調査した [37]．Lopes らは，版管理システムの一種である Git のオンラインホスティングサービスである GitHub 上で管理されているオープンソースプロジェクトについて，プロジェクト間でどのくらい同一のソースファイルが使われているか，どのようにコードクローンが分布しているか分析した [**?**]．

ソースファイルに含まれるバグや脆弱性の問題を調査するためには，ソフトウェアだけでなく，どのバージョンを使っているかまで特定することが必要である．そこで Kawamitsu らは，分析対象のソフトウェアとライブラリのリポジトリを入力として，ソフトウェアのリビジョンごとにソースファイルがどのライブラリのバージョンから再利用されているか最長一致部分列を元に分析する手法を提案した [23]．Ito らは，GNU/Linux OS の一種である Debian が公開しているパッケージ集合をデータベース化し， 入力されたソースファイルと最も類似するソースファイルを検索するウェブサービスを提案した [17]．これらの手法はファイルごとに再利用元の候補となるバージョンを報告するが，複数のバージョンに同一のファイルが含まれることもあるため，ソフトウェアが再利用したライブラリ全体としてのバージョンは使用者が判断しなければならない．

著者らは，入力をファイル単位からソースファイルの集合へと拡張し，データベース

中から最も多く類似するソースファイルを含む OSS を検索する手法を提案した [16]．Jewmaidang らは，ソフトウェアのリポジトリとライブラリのリポジトリを比較することで，ソフトウェアが利用しているライブラリのバージョンの変遷を可視化した [19]．

3.2.2 類似度計算の高速化手法

起源分析においては，ファイル同士の類似度を高速に計算することが重要となる．Kawamitsu ら [23] は類似度としてファイル同士の最長一致部分列の長さを使用しているが，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．

類似度の計算を高速化する方法として，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，比較すべき候補を事前に効率よ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，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 [20] 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 [44]．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 [36]．しかし，これらの手法は，互いに類似したソースコード片が少ないことを仮定したものであり，起源分析のように，多数の類似ファイルから最も類似したファイルを選択したいという目的には適さない．

3.2.3 b-bit MinHash 法を用いた類似度計算

類似度の計算そのものを高速化する手法として，類似度の 1 つである Jaccard 係数 [18] には，その推定値を高速に求める MinHash 法 [2, 3] が提案されている．Jaccard 係数は集合間の類似性を計測する指標であり，以下の式で表される．

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

ここで S_1, S_2 は，それぞれファイルの内容を表現した集合である．MinHash 法は，任意のハッシュ関数 h_i が与えられたとき，以下のように集合 S からハッシュ値が最小である要素を取り出す関数 $m_i(S)$ を使用する．

$$m_i(S) = \min_{s \in S} \ h_i(s)$$

2 つの集合 S_1, S_2 に対して $m_i(S_1), m_i(S_2)$ を計算すると，$m_i(S_1) = m_i(S_2)$ となる確率が $J(S_1, S_2)$ に一致する．そのため，複数のハッシュ関数を用意すれば，これら

のハッシュ値の一致割合から，Jaccard 係数の推定値を求めることができる．

MinHash 法を省メモリで実現する方法として b-bit MinHash 法 [24] が提案されている．この手法では，MinHash 法で計算したハッシュ値 $m_i(S)$ の下位 b ビットのみをハッシュ値として用いる．たとえば $b = 1$ のとき，b-bit MinHash 法のハッシュ関数 $b_i(f)$ は以下の式で表される．

$$b_i(S) = \mathrm{LSB}(m_i(S))$$

ここでの LSB は Least Significant Bit を表す．

2 つの集合 S_1, S_2 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．ハッシュ値の下位 b ビットが偶然一致する確率は $\frac{1}{2^b}$ であるので，$P(S_1, S_2)$ は 以下の式で表される．

$$P(S_1, S_2) = \frac{1}{2^b} + \left(1 - \frac{1}{2^b}\right) \times J(S_1, S_2)$$

最も計算量が少なくなるのが $b = 1$ のときであり，k 個のハッシュ関数を用いても k ビットの記憶域で済む．$P(S_1, S_2)$ の近似値として，k 個のハッシュ値の一致割合 $P_o(S_1, S_2)$ を用いて，前述の式を変形した以下の式により集合 S_1, S_2 間の Jaccard 係数の推定値 $J_e(S_1, S_2)$ を計算することが可能である．

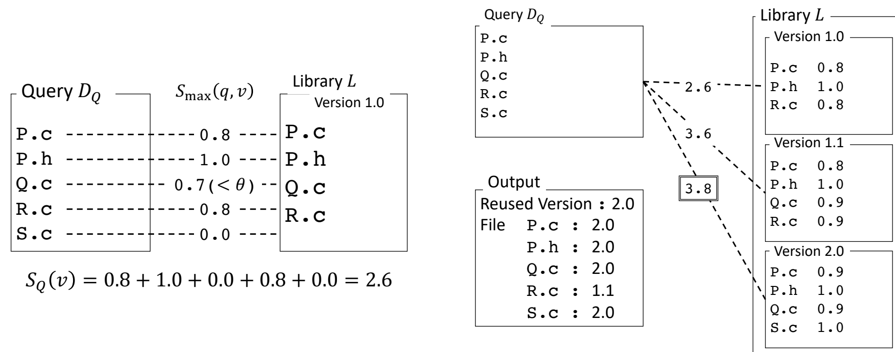
$$J_e(S_1, S_2) = \left(P_o(S_1, S_2) - \frac{1}{2}\right) \times 2$$

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

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．なお，これは第 2.2 節において説明した手法と同様のものである．

3.3 提案手法

提案手法は，分析対象のソフトウェアにおいてライブラリから再利用したファイルを格納しているディレクトリ D_Q とそのライブラリの版管理システムのリポジトリ L を入力とし，D_Q に再利用されているライブラリのバージョン r_Q と，ディレクトリ内のファイル $q \in D_Q$ それぞれに最も類似したライブラリのバージョン $r[q]$ を出力する．入力をディレクトリとしたのは，再利用したライブラリのファイル群が 1 つのディレクトリにまとめて配置されることが多いためである．再利用されたバージョンの情報を D_Q 全体に対して 1 つ出力すると同時にファイルごとにも個別情報を出力するのは，全体としては古いバージョンのライブラリを利用している場合でも，たとえば特定のファイルだけがセキュリティパッチの適用で新しいバージョンのものに差



(a) ある 1 つのバージョンとの比較（類似度の閾値 $\theta = 0.8$）

(b) ライブラリの全バージョンとの比較とその結果

図 3.1: 提案手法の動作例

し替えられているといった状況を提示できるようにするためである．なお，ここでのバージョンとは，開発者がライブラリを対外的に公開する，版管理システムの中でタグと呼ばれるラベルが紐づけられているリビジョンのことである．

提案手法は，D_Q に含まれるソースファイルと，ライブラリの各バージョン $v \in L$ に含まれるファイル群を比較し，最も類似したファイルを含むようなバージョンを特定する．具体的には，ファイル間の類似度 $\mathrm{sim}(q, f)$ が与えられたとき，ライブラリのあるバージョン v に対する入力ファイル群の類似度の合計値 $S_Q(v)$ が最大となるような v を求める．入力ファイルの 1 つ q に対して，バージョン v の中で最も類似しているファイルの類似度 $S(q, v)$ は，以下の式で定義される．ただし，ファイル間の類似度は，ある閾値 θ 以上の値の場合のみ記録し，θ 未満の場合は 0 とする．

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

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

ここで $\mathrm{Files}(v)$ は，バージョン v に含まれるファイルの集合である．このとき，$S_Q(v)$ の定義は以下の通りである．

$$S_Q(v) = \sum_{q \in D_q} S(q,v)$$

つまり，$S_Q(v)$ は $S(q, v)$ を合計することで，D_Q に使われているファイルのどれだけの内容が v に由来するものかを表す．

図 3.1 に提案手法の動作例を示す．この図は，Query D_Q として与えられた 5 個のファイルに対して，Library L の 3 つのバージョンのうちどれを再利用したのか特定しようとした場合を示している．図 3.1a は D_Q と L のある 1 つのバージョンとを比較する際の動作例を示している．D_Q に含まれる全てのソースファイルについて，

$S_{\max}(q,v)$ を計算する．この例では類似度の閾値を $\theta = 0.8$ としているため，Q.c では $S_{\max}(q,v) = 0.0$ となる．そのため，全体での類似度は $S_Q(v) = 2.6$ となる．この手順を L の全てのバージョンにおいて行った結果が図 3.1b である．$S_Q(v)$ が最大となるのは，Version 2.0 のときであり，これを D_Q が再利用しているライブラリのバージョンであると出力する．また，それぞれのソースファイルごとに，最も類似度が高いソースファイルが含まれるライブラリのバージョンも個別に出力する．この出力結果の場合，D_Q はライブラリ L の Version 2.0 を再利用している可能性が高いが，R.c という 1 ファイルのみ異なるバージョンが混ざった状態であることが判断できる．この情報から，開発者はライブラリの更新やパッチの適用などの活動を行うことができるようになる．

$S_Q(v)$ を計算するためには，入力されたディレクトリ D_Q と，ライブラリ L に所属するファイルの総当たり比較が必要となる．そこで，本研究では高速化のために b-bit MinHash 法を用いた類似度計算を採用する．以降，本手法でのファイル間での類似度の定義と，具体的な計算手順を示す．

3.3.1 ソースファイル間の類似度計算

本研究では，ソースファイルをそれぞれ字句の trigram 多重集合とみなし，それらの類似度として Jaccard 係数の推定値を b-bit MinHash 法で計算する．あるソースファイルの字句集合を f とし，その trigram 多重集合を $\tau(f)$ とすると，trigram 多重集合を用いたファイル f_1 とファイル 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．

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

$$\tau(f) = \{\langle t_i, t_{i+1}, t_{i+2} \rangle \mid 1 \leq i \leq |f| + 2\}$$

ただし，$\langle t_i, t_{i+1}, t_{i+2} \rangle$ は f 中の連続する 3 つの字句の組である．$\tau(f)$ を構築する際，f は先頭と末尾に空要素を 2 つずつ加えることで，すべての $\tau(f)$ の要素は少なくとも 1 つの空でない字句を持つ組になるようにする．字句解析はプログラミング言語ごとに異なるが，本論文の実験対象となる C 言語用の解析では，コメントと空白は削除するが，プリプロセッサのための指令（たとえば #include など）はそれぞれ個別のトークンとして保持するようにした．なお，識別子などは特に正規化等を行わず，そのまま保持している．これらの情報はバージョンの特定に有用であることが報告されているためである [23]．

しかしながら，正確に $J_\tau(f_1, f_2)$ を計算する場合，あらかじめ集合ごとに要素を特定の順序で整列しておいたとして，$\tau(f_1)$, $\tau(f_2)$ の大きさの和に比例した時間が必要となる．そこで，これを軽量に計算するため，本研究では b-bit MinHash 法を用いて類似度を計算する．$b = 1$ とし，ハッシュ関数を k 個用いることで，k ビットのビッ

```
Example code:
  a: while ((*dst++ = *src++) != '\0');
  b: while (*dst++ = *src++);
```

$\tau(a)$	$\tau(b)$
⟨_, _, while⟩,⟨_, while, (⟩, ⟨while, (, (⟩^{u},⟨(, (, * ⟩^{u}, ⟨(, *, dst⟩,⟨*, dst, ++⟩, ⟨dst, ++, =⟩,⟨++, =, *⟩, ⟨=, *, src⟩,⟨*, src, ++⟩, ⟨src, ++,)⟩, ⟨++,), !=⟩^{u},⟨), !=, '\0'⟩^{u}, ⟨!=, '\0',)⟩^{u},⟨'\0',), ;⟩^{u}, ⟨), ;, _⟩,⟨;, _, _⟩	⟨_, _, while⟩,⟨_, while, (⟩, ⟨while, (, *⟩^{u}, ⟨(, *, dst⟩,⟨*, dst, ++⟩, ⟨dst, ++, =⟩,⟨++, =, *⟩, ⟨=, *, src⟩,⟨*, src. ++⟩, ⟨src, ++,)⟩, ⟨++,), ;⟩^{u}, ⟨), ;, _⟩,⟨;, _, _⟩

$$J_\tau(a,b) = \frac{|\tau(a) \cap \tau(b)|}{|\tau(a) \cup \tau(b)|} = \frac{11}{19} = 0.579$$

図 3.2: 正確な $J_\tau(f_1, f_2)$ の具体例

トベクトルに対する XOR 演算を行うだけで，ファイル f_1, f_2 の組に対する類似度 $\mathrm{sim}(f_1, f_2) = J_e(\tau(f_1), \tau(f_2))$ を求める．

本研究では具体的なハッシュ関数の実装として，$\tau(f)$ における trigram τ の n 回目の出現に対して，以下の関数を適用する．*[1]

$$h_i(\tau, n) = \mathrm{SHA1}(\tau) \times n \times r_i$$

ここで SHA1(τ) は trigram τ を構成する 3 つのトークンを連結したバイト列に対する SHA-1 ハッシュ値であり，r_i は xorshift[29] による擬似乱数（ただし最小ビットは 1 に固定したもの）である．n と r_i で SHA-1 ハッシュの順序を並べ替え，$\tau(f)$ の全要素に対して最小となる値を保存し，得られたハッシュ値の下位 b ビットを得る．

b-bit MinHash 法によって得られる類似度 $\mathrm{sim}(f_1, f_2)$ は推定値であり，$J_\tau(f_1, f_2)$ に対して二項分布する [24]．その分散は $J_\tau(f_1, f_2)$ に応じて変化し，$J_\tau(f_1, f_2)$ がより高いほど小さくなり，1 のとき 0 になる．言い換えると，ある 2 つのソースコード間の $J_\tau(f_1, f_2)$ と $\mathrm{sim}(f_1, f_2)$ の差分は 0 を中心とした二項分布となり，$J_\tau(f_1, f_2)$ が大きいほどその差分も小さい．このことから，十分な要素数を持つソースファイル集合同士であれば，$J_\tau(f_1, f_2)$ を定義通りに計算した場合と比べても，十分正確に類似度の合計値を求めることが期待できる．

図 3.2 に定義通りに $J_\tau(f_1, f_2)$ を計算する具体例を，図 3.3 に b-bit MinHash 法を用いて類似度の推定値を計算する具体例を示す．図 3.2 中の ⟨ A, B, C ⟩ は連続する字句の組を表し，_ は空要素を表す．また，⟨ A, B, C ⟩^{u} となっている要素は，その

*[1] 実装は https://github.com/NAIST-SE/CodeHash で公開している

入力	b_1	b_2	b_3	b_4	b_5	b_6	b_7	b_8	b_9	b_{10}
$\tau(a)$	0	1	1	0	0	0	1	1	0	1
$\tau(b)$	0	1	1	0	0	1	0	1	0	1

$$\mathrm{sim}(a,b) = \left(\frac{8}{10} - \frac{1}{2}\right) \times 2 = 0.6$$

図 3.3: b-bit MinHash を用いた類似度計算例

集合に固有の要素である．ここから $J_\tau(a,b) = \frac{11}{19} = 0.579$ が得られる．図 3.3 は 10 個のハッシュ関数を用いて b-bit MinHash 法を適用した場合を示す．それぞれのソースコードの trigram 多重集合に対して 10 個のハッシュ関数 b_i を適用し，図に示すようなハッシュ値の列が得られたとすると，一致しているハッシュ値は 8 つあるので，$\mathrm{sim}(a,b) = (\frac{8}{10} - \frac{1}{2}) \times 2 = 0.6$ となる．この例では 0.021 の誤差が生じているが，実際の適用ではハッシュ関数の個数を増やすことで，より正確に類似度を計算する．

3.3.2 再利用元バージョンの特定

Algorithm 2 に提案手法の疑似コードを示す．入力は再利用元ライブラリのバージョンを調査したいファイル集合 D_Q，そのソフトウェアが再利用しているライブラリのリポジトリ L である．似ていないファイルの類似度がノイズとして加わることを防ぐため，類似度が閾値 θ 以上のファイルの組だけを扱う．また，b-bit MinHash 法の誤差を許容する範囲として m を導入している．加えて，計算コストを削減するために，ソフトウェアとライブラリそれぞれに含まれるファイルについて，事前に b-bit MinHash 法のためのハッシュ列やファイルの SHA-1 ハッシュ値を計算しておく．

アルゴリズムは 2 行目から 20 行目までが，ライブラリ L のすべてのバージョン $v \in L$ に含まれるファイル f について，入力されたファイル $q \in D_Q$ と比較するループ構造となっている．計算を高速化するため，まず 5 行目でファイルが同一の拡張子を持つ（同一のプログラミング言語である）ことを確認し，次いで 6 行目でファイルの内容が完全一致するかどうかを SHA-1 ハッシュによって確認する．ファイルが完全一致していれば，字句解析などを一切行うことなく，類似度を 1.0 とする．完全一致でないファイルに対しては，9 行目で trigram 多重集合の大きさの比較を行い，大きさがあまりに異なる（類似度が決して閾値 θ を超えない）ファイルについては比較を行わないようにする．この条件式は，2 つの集合の大きさの比が閾値 θ よりも小さい場合，Jaccard 係数が以下のように θ を上回ることがないことを利用している．

$$J_\tau(q,f) \leq \frac{\min(|\tau(q)|, |\tau(f)|)}{\max(|\tau(q)|, |\tau(f)|)} < \theta$$

類似度が閾値 θ を超える可能性があるファイルの組に対してのみ，10 行目で b-bit MinHash 法による類似度の計算を行う．$\mathrm{sim}(q, f)$ は Jaccard 係数を中心として二項分布するため，$\mathrm{sim}(q, f)$ が θ を超えないソースファイル間であっても，$J_\tau(q, f)$ は θ を超えている場合がある．そこで，マージン m だけ閾値を引き下げることで，類似しているソースファイルの組み合わせが見落とされないようにする．

類似度の計算が完了すると，アルゴリズムの 21 行目でバージョンごとに類似度の合計を行い，最も類似度の合計値が高かったライブラリのバージョンを出力 r_Q として特定する．また，22 行目で，D_Q に含まれたファイル q それぞれに対して最も類似しているファイルを保有するバージョン $r[q]$ を求める．なお，同一のファイルを含むライブラリのバージョンが複数存在する場合があるが，r_Q の出力では，類似度が最大の候補が複数ある場合には最新のバージョンを報告する．$r[q]$ の候補が複数ある場合，その中に r_Q が含まれている場合は r_Q を使用し，そうでない場合は最新版を報告する．

3.4　評価

本研究では提案手法の妥当性を以下の基準で評価する．

1. ライブラリのバージョンを検出する正確さ
2. ライブラリのバージョン検出に要する時間

ファイル単位で再利用元のバージョンを検出する Kawamitsu らの手法 [23] を基準としてみた場合，提案手法の特徴は，再利用されたファイル集合全体での類似度を基にバージョンを検出すること，b-bit MinHash 法による高速な類似度の計算を導入したことの 2 点にある．そこで，正確な Jaccard 係数の計算を用いてファイル単位で再利用元バージョンを検出する場合を基準として，ファイル集合全体での類似度を基にバージョンを検出する効果と，Jaccard 係数を b-bit MinHash 法で計算した効果を計測する．

3.4.1　実験対象

実験対象として 5 つのソフトウェアと 4 つのライブラリのプロジェクトを使用する．表 3.1 にこれらのソフトウェアと利用されているライブラリの組み合わせを示す．実験対象ソフトウェアは，再利用ライブラリのバージョンを記録していること，長期間保守されていること，規模が異なること，開発コミュニティが異なることを基準として選択した．5 つのソフトウェアは全て libpng と zlib を利用しており，合計で 13 組の再利用関係が含まれている．

表 3.2 にこれらのソフトウェアおよびプロジェクトの開発状況の情報を示す．ID が

表 3.1: 実験対象のソフトウェア・ライブラリの組

ソフトウェア	libpng	curl	ogg	zlib
android	✓	✓	✓	✓
apitrace	✓			✓
fs2open	✓			✓
gecko-dev	✓		✓	✓
v8monkey	✓			✓

1 から 8 のプロジェクトはライブラリを利用しているソフトウェアで，ID が 9 から 12 のプロジェクトはライブラリである．全てのプロジェクトは分散版管理システムの Git で管理されており，表中のリポジトリ URL からクローンした．android は 機能部位単位で Git リポジトリを分けて管理しているため，元になったライブラリに対応するリポジトリを実験対象としている．コミット数はリポジトリをクローンした時点での最新のリビジョンまでを数えたもので，LOC は最新のリビジョンでの総ソースコード行数（Lines of Code）である．バージョン数はタグとして付けられたバージョンの個数である．同一リビジョンに複数のバージョンが紐づけられている（同一内容のソースコードが複数のバージョンとしてリリースされた）場合は，それらをまとめて 1 つのバージョンとして扱う．

Algorithm 2 再利用しているライブラリのバージョンの検出

入力

D_Q : 解析対象ソフトウェアのあるバージョンでの再利用ライブラリが含まれるディレクトリ

L : ソフトウェアが利用しているライブラリのリポジトリ

θ, m : 類似度の閾値

出力

r_Q : D_Q中のソースファイルに最も類似する L のバージョン

$r[q]$: D_Q中のソースファイル q に最も類似するファイルを持つ L のバージョン

1: Initialize $S(q, v) \leftarrow 0$ for all possible $q \in D_Q$ and $v \in L$
2: **for** $v \in L$ **do**
3: **for** $f \in \mathrm{Files}(v)$ **do**
4: **for** $q \in D_Q$ **do**
5: **if** f is same file extension to q **then**
6: **if** $\mathrm{SHA1}(f) = \mathrm{SHA1}(q)$ **then**
7: $S(q, v) \leftarrow 1.0$
8: **else**
9: **if** $\frac{\min(|\tau(q)|,|\tau(f)|)}{\max(|\tau(q)|,|\tau(f)|)} \geq \theta$ **then**
10: **if** $\mathrm{sim}(q, f) \geq \theta - m$ **then**
11: **if** $\mathrm{sim}(q, f) \geq S(q, v)$ **then**
12: $S(q, v) \leftarrow \mathrm{sim}(q, f)$
13: **end if**
14: **end if**
15: **end if**
16: **end if**
17: **end if**
18: **end for**
19: **end for**
20: **end for**
21: $r_Q \leftarrow \arg\max_{v \in L} \sum_{q \in D_q} S(q, v)$
22: $r[q] \leftarrow \arg\max_{v \in L} S(q, v)$ **for each** $q \in D_Q$

表 3.2: 実験対象のソフトウェア・ライブラリを格納するリポジトリ

ID	リポジトリ名	リポジトリ URL	期間	コミット数	LOC	バージョン数
1	android(libpng)	https://android.googlesource.com/platform/external/libpng	2008/10-2018/4	378	88401	46
2	android(curl)	https://android.googlesource.com/platform/external/curl	2010/3-2018/6	205	208530	22
3	android(ogg)	https://android.googlesource.com/platform/external/libogg	2012/6-2017/10	21	3283	9
4	android(zlib)	https://android.googlesource.com/platform/external/zlib	2008/10-2018/1	259	38057	35
5	apitrace	https://github.com/apitrace/apitrace.git	2008/7-2018/2	4264	413558	9
6	fs2open	https://github.com/scp-fs2open/fs2open.github.com.git	2002/1-2018/3	16526	776585	319
7	gecko-dev	https://github.com/mozilla/gecko-dev.git	1998/3-2018/4	634881	18053155	98
8	v8monkey	https://github.com/zpao/v8monkey.git	2007/3-2012/2	87432	6604218	72
9	libpng	git://git.code.sf.net/p/libpng/code	2009/4-2018/3	5956	87744	1557
10	curl	https://github.com/curl/curl.git	1999/12-2018/7	23312	196098	179
11	ogg	https://github.com/xiph/ogg.git	2000/9-2018/4	497	3830	11
12	zlib	https://github.com/madler/zlib.git	2011/9-2017/1	419	35609	72

3.4.2 実験方法

提案手法を Java で実装し，表 3.1 のソフトウェア・ライブラリの組に対して実行する．実験対象ソフトウェアの各バージョンに対して，ライブラリから再利用したファイルを格納しているディレクトリ（D_Q）をディレクトリ名に基づいて特定し，ライブラリのリポジトリ（L）と合わせて実装プログラムに与えて，バージョン番号を取得する．提案手法における類似度の閾値は，$\theta = 0.9$，$m = 0.1$ とし，b-bit MinHash 法のビット数とハッシュ関数の個数は $b = 1$，$k = 2048$ とした．

検出結果の正確さは，実験対象ソフトウェアの各バージョンに対して，検出結果がソフトウェア側の記録と一致した割合によって求める．ソフトウェア側の記録は，リポジトリ中のコミットメッセージやドキュメントファイルの中に記録されていた再利用ライブラリのバージョン番号を手作業で検出したものである．この手作業での分析では，ライブラリからコピーされたファイルは利用せず，実験対象ソフトウェアで作成された記録のみを用いてバージョンの検出を行った．バージョン番号の識別には Git リポジトリにおけるタグを用いたが，gecko-dev と ogg の組においては，再利用しているバージョンの記録に ogg 側の Subversion リポジトリのリビジョン番号が使用されていたため，それらを手作業で Git リポジトリのバージョン番号に読み替えた．

ファイル単位で再利用元バージョンを検出する場合と比較するため，提案手法のファイル単位での出力（$r[q]$）についても，ファイル単位でソフトウェア側の記録との一致率から正確さを求める．Kawamitsu らの手法 [23] に従ってファイル単位で最も類似したライブラリのバージョンが複数検出された場合は，ソフトウェア側の記録に該当するバージョンを含むとき記録と一致するとみなす．提案手法は，既存手法と比べると，ファイル集合全体での集計結果を用いて，特定の 1 バージョンだけに候補を絞った出力を行っているとみなすことができるため，出力されるバージョン数も比較を行う．

提案手法の実行時性能の評価は，実装プログラムをそれぞれ 10 回ずつ実行し，その実行時間を計測することで行う．入力されたソースファイル群の読み込みと比較に大きな時間を要するため，ソフトウェアの読み込み，ライブラリの読み込み，ソースファイルの比較のステップと全体での時間を分けて記録する．計測に用いる計算機環境の OS は Oracle Linux 4.1.12-112.16.7.el7uek.x86_64，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 1TB のもの，Java の実行環境は OpenJDK 8 である．時間の計測は，Java のシステムメソッドの 1 つである nanoTime メソッドの呼び出しを実装プログラム中に記述することで行う．また，マルチスレッド処理は使用せず，単一のスレッドで処理を行う．

b-bit MinHash 法がもたらす効果を評価するため，評価実験では Jaccard 係数を b-bit MinHash 法を用いて近似計算した場合と，ファイルの内容から正確に計算した

表 3.3: 提案手法の実行結果

ソフトウェア	ライブラリ	バージョン数	一致する数		一致しない数		記録なし	
android(libpng)	libpng	46	46	100.0%	0	0.0%	0	0.0%
android(curl)	curl	22	19	86.4%	3	13.6%	0	0.0%
android(ogg)	ogg	9	9	100.0%	0	0.0%	0	0.0%
android(zlib)	zlib	35	35	100.0%	0	0.0%	0	0.0%
apitrace	libpng	9	8	88.9%	0	0.0%	1	11.1%
	zlib	9	9	100.0%	0	0.0%	0	0.0%
fs2open	libpng	293	293	100.0%	0	0.0%	0	0.0%
	zlib	293	293	100.0%	0	0.0%	0	0.0%
gecko-dev	libpng	98	98	100.0%	0	0.0%	0	0.0%
	ogg	98	94	95.9%	0	0.0%	4	4.1%
	zlib	98	98	100.0%	0	0.0%	0	0.0%
v8monkey	libpng	72	72	100.0%	0	0.0%	0	0.0%
	zlib	72	72	100.0%	0	0.0%	0	0.0%
全体		1,154	1,146	99.3%	3	0.3%	5	0.4%

表 3.4: ファイル単位での正確さ

ソフトウェア	ライブラリ	提案手法	既存手法
android	libpng	100.0%	98.4%
	curl	86.4%	86.4%
	ogg	100.0%	100.0%
	zlib	100.0%	99.6%
apitrace	libpng	90.7%	90.7%
	zlib	100.0%	100.0%
fs2open	libpng	100.0%	100.0%
	zlib	100.0%	100.0%
gecko-dev	libpng	100.0%	95.4%
	ogg	96.3%	95.5%
	zlib	100.0%	100.0%
v8monkey	libpng	100.0%	86.4%
	zlib	100.0%	100.0%
全体		95.6%	94.7%

場合の 2 通りを実行する．

3.4.3 提案手法の正確さ

表 3.3 にソフトウェアのバージョンごとの正確さを計測した結果を示す．fs2open のバージョン数が表 3.2 のバージョン数と異なる値だが，これは一部のバージョンで

表 3.5: ソフトウェア全体とファイル単体での検出結果の違い

ソフトウェア	ライブラリ	バージョン数	ファイル数	不一致数	[%]	バージョンあたりの不一致数
android(libpng)	libpng	46	3,211	50	1.56	1.09
android(curl)	curl	22	13,137	273	2.08	12.41
android(ogg)	ogg	9	63	0	0.00	0
android(zlib)	zlib	35	2,960	11	0.37	0.31
apitrace	libpng	9	194	0	0.00	0
	zlib	9	240	0	0.00	0
fs2open	libpng	319	6,446	0	0.00	0
	zlib	319	6,445	0	0.00	0
gecko-dev	libpng	98	2,520	116	4.60	1.18
	ogg	98	536	4	0.75	0.04
	zlib	98	2,646	1	0.04	0.01
v8monkey	libpng	72	1,586	215	13.56	2.99
	zlib	72	1,756	0	0.00	0
全体		1,206	41,740	670	1.61	0.56

ライブラリが利用されていなかったためである．検出結果と記録はソフトウェアごとでは 86.4% から 100.0% が，全体では 99.3% が一致し，ソフトウェアとライブラリの組のうち，13 組中 10 組については検出結果と記録が全て一致した．記録と一致しなかった android における curl 再利用の 3 つのバージョンについては，再利用したライブラリのドキュメントファイルと検出結果のバージョンが一致したことから，ソフトウェア側の記録が間違っている可能性がある．また，再利用したバージョンの記録がない場合についても，apitrace の 1 例ではライブラリのドキュメントファイルに記述されているバージョン情報と提案手法の検出結果が一致している．これらのことから，提案手法は再利用しているライブラリのバージョンを高い精度で検出することが可能である．

表 3.4 に提案手法のすべてのファイルに対して求めた全体での正確さを示す．既存手法，すなわち最も類似したファイルを選択する方式の正確さは 94.7% であり，提案手法は 95.6% でそれを上回った．また，既存手法ではファイルごとの情報だけでは単独のバージョンに絞り込むことはできず，最頻値で 4 バージョン，中央値で 23 バージョン，最大で 1612 バージョンを列挙した．提案手法は，全体での結果と各ファイルについて個別のバージョンが一致する場合はただ 1 つ，一致しない場合でも 2 つのバージョンだけに絞り込んでおり，それでも正確さで既存手法を上回っていることから，既存手法よりも簡潔な出力で，正確さを達成している．

表 3.5 に，提案手法におけるファイル単位での検出結果がソフトウェア全体の検出結果と異なったファイルの数を示す．表中のファイル数の列は，ソフトウェア中のライブラリのディレクトリに含まれるソースファイルの数を全てのバージョンで足し合わせた値である．提案手法は 1.61% のファイルに対して，ファイル集合全体でのバージョンとは異なるバージョンを出力した．それらのファイルを目視で分析したところ，

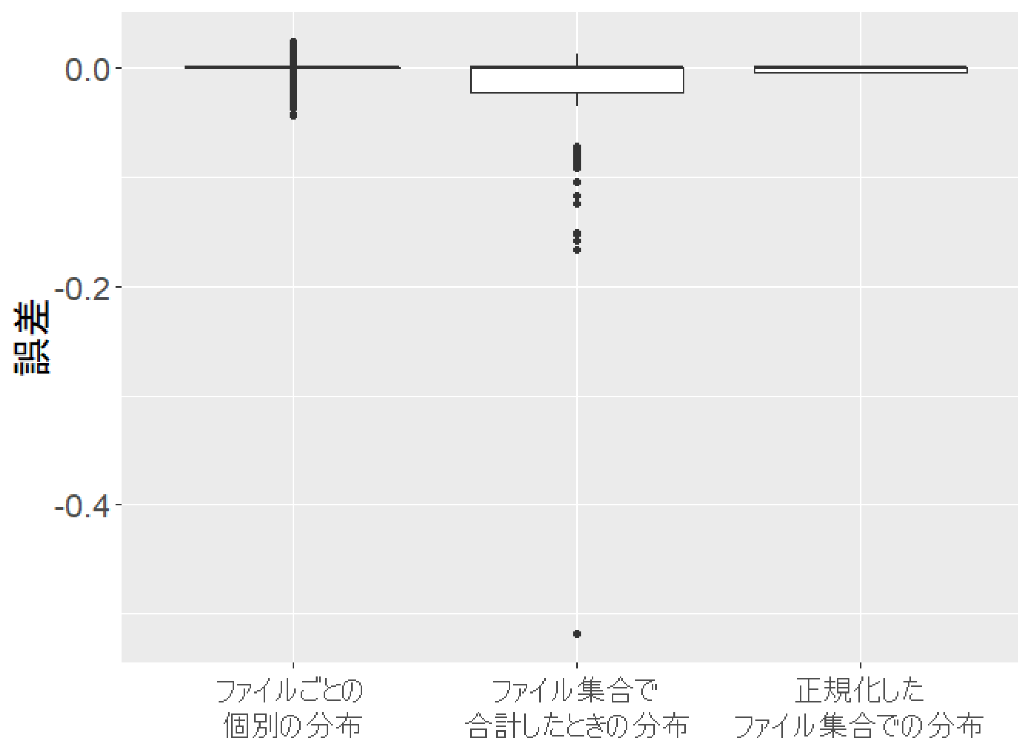


図 3.4: 1 ビット MinHash 値と Jaccard 係数の誤差の分布

以下のような事例が含まれていた．

- 利用しているライブラリの新しいバージョンでは削除されたファイル．新しいバージョンのライブラリを上書きコピーした結果，ライブラリ側で削除されたファイルだけが上書きされずに残ったと考えられる．
- ライブラリ側でバグや脆弱性の修正版等がリリースされる前に，修正パッチをソフトウェア側で独自に適用したファイル．そのファイルのみ，ライブラリの新しいバージョンとして検出された．
- ソフトウェア側で独自に変更を加えたファイル．たとえば条件式の中で関数を呼ぶのではなく変数に代入してから条件式で参照するなどのコーディングスタイルの統一を実施したものがあった．変更内容が，他のバージョンの内容と偶然類似していた場合に，異なるバージョンとして報告された．

提案手法は全体として最も類似したバージョンを報告するため，そのバージョンを基準に差分を分析することで，これらの影響を迅速に把握することができた．これはファイル単位で検出を行っていた既存手法では実施できなかった分析である．

図 3.4 に，提案手法（b-bit MinHash 法）で計算した類似度と，正確な Jaccard 係数の誤差の分布を示す．左端の箱ひげ図が提案手法の実行中に計算されたすべての類

表 3.6: 提案手法の実行時間

ソフトウェア	ライブラリ	全実行時間 t_1 [ms]	ソフトウェア 読み込み [ms]	ライブラリ 読み込み [ms]	比較 [ms]	比較回数
android	libpng	242,371	4,571	223,418	14,381	2.2×10^8
	curl	95,864	9,064	45,515	41,287	6.7×10^8
	ogg	981	601	357	22	3.2×10^3
	zlib	7,250	2,275	4,342	632	1.2×10^7
apitrace	libpng	214,544	1,480	212,289	773	1.5×10^7
	zlib	6,156	928	5,147	80	1.0×10^6
fs2open	libpng	279,272	2,012	210,973	66,286	5.0×10^8
	zlib	9,045	1,335	5,064	2,645	2.8×10^7
gecko	libpng	235,119	12,740	208,702	13,675	1.8×10^8
	ogg	9,988	9,593	312	82	3.1×10^4
	zlib	17,526	10,507	4,946	2,072	2.7×10^7
v8monkey	libpng	231,638	7,603	214,594	9,440	1.7×10^7
	zlib	12,141	6,007	5,048	1,085	1.2×10^8
平均		104,761	5,286	87,747	11,727	1.4×10^8

似度の値の誤差の分布である．b-bit MinHash 法はファイルの内容が同一（Jaccard 係数 1）であるとき差分は原理的に 0 となるので，ばらつきは非常に小さい．中央の箱ひげ図がこれらの類似度をファイル集合ごとに合計したときの誤差であり，最大でも 0.52 である．右端の箱ひげ図のようにファイルの個数で正規化すると，最大で 0.006 と，ソフトウェア全体の類似度に対する比率としてはきわめて小さな誤差であり，実際に本実験ではバージョンの検出には影響せず，提案手法は Jaccard 係数を正確に計算した場合と同一の結果を出力した．

3.4.4 提案手法の実行時性能

表 3.6，3.7 に提案手法の実行時間を計測した結果を示す．左から全体での実行時間と，実験対象ソフトウェアのソースファイルの読み込みに要した時間，ライブラリのソースファイルの読み込みに要した時間，比較に要した時間を示す．3 つの段階以外にも出力の集計等の処理が含まれるため，3 つの段階の合計時間は全体の合計時間とは一致しない．ソフトウェアの読み込み時間は実装の都合上全てのバージョンを読み込んだ時間である．読み込み時間はバージョン数と全バージョンを通してのユニークなファイル数が多いほど増大する．表 3.2 を見るとバージョン数はライブラリの方が多い傾向にあるため，ソフトウェアよりもライブラリの方が読み込み時間が長くなる傾向にある．

表 3.6 の「比較回数」の列は，提案手法における類似度の計算回数（Algorithm 1 における 10 行目の実行回数）を示している．類似度の計算回数は最大で 6.7×10^8 回行われているにも関わらず比較時間は平均で 11 秒程度である．これは，提案手法における類似度の計算が一度のビット演算と算術演算だけであり，時間計算量が $O(1)$ と

表 3.7: *b*-bit MinHash 不使用の場合を 100% とした提案手法の実行時間

ソフトウェア	ライブラリ	類似度を正確に計算する場合との比較 正確に計算した場合の 実行時間全体 t_2 [ms]	t_1/t_2	使用メモリサイズ [MB] Jaccard 係数	b-bit MinHash
android	libpng	4,170,739	5.8%	944.46	4.32
	curl	3,220,414	3.0%	226.96	3.12
	ogg	1,044	94.0%	1.51	0.01
	zlib	105,303	6.9%	24.47	0.36
apitrace	libpng	545,570	39.3%	946.01	4.34
	zlib	16,602	37.1%	23.69	0.35
fs2open	libpng	11,871,778	2.4%	945.39	4.33
	zlib	300,693	3.0%	22.99	0.34
gecko	libpng	4,294,906	5.5%	958.16	4.40
	ogg	10,478	95.3%	23.08	0.34
	zlib	278,052	6.3%	1.71	0.01
v8monkey	libpng	3,000,782	7.7%	22.30	0.33
	zlib	178,727	6.8%	950.12	4.36
平均		2,153,468	24.1%	391.60	2.05

なっているためである．

表 3.7 の右から 3 列目に，*b*-bit MinHash 法を利用しなかった場合（Jaccard 係数を直接計算した場合）の実行時間全体を 100% としたときの提案手法の実行時間を示す．ソフトウェアおよびライブラリ読み込みに要する時間はどちらの方法でも変わらず，実行時間の短縮は比較に要する計算時間の差によるものである．また，右から 4 列目に，正確に Jaccard 係数を計算した場合の全体の実行時間を示す．これらから，比較回数が少ないものについては実行時間の短縮の効果が小さいが，平均でも 24.2% の実行時間で計算を完了できるようになったことがわかる．ライブラリが ogg の場合はほとんど改善がみられないが，これは ogg が Jaccard 係数を計算したとしてもほとんど時間がかからないほど小規模で比較回数が少なく，読み込みにかかる時間の方が支配的なためである．

表 3.7 の右端の 2 列は，Jaccard 係数を計算する場合の集合表現を構築する際のメモリ使用量と *b*-bit MinHash 法を使用した場合のメモリ使用量の概算値である．Jaccard 係数を計算する場合は 1 ファイルの 1 要素ごとに 64 ビット整数を 1 つ使用する，*b*-bit MinHash の場合は 1 ファイルに対し 2048 ビット使用するとみなし，Java のデータ構造によるオーバーヘッドを無視して算出した値である．結果として，メモリの使用量は Jaccard 係数を計算する場合の 1% 未満となり，大規模なライブラリのリポジトリに対する解析でも，通常の計算機で十分に取り扱うことができると考えられる．

上記の結果から，提案手法は既存手法よりも高速かつ省メモリである．また，ライブラリの読み込みには時間がかかるが，*b*-bit MinHash の計算結果をキャッシュとして保管しておけばその影響は小さくすることが可能である．提案手法を使用して，様々

なプロジェクトに対して自動でライブラリの利用状況を監視するようなシステムの構築も十分可能であると考えている．

3.5 妥当性への脅威

3.5.1 ファイル集合の再利用バージョン判定

本研究ではソフトウェア全体での再利用バージョンを決定する際，ファイル単位の類似度を合計する方法を用いた．本研究で用いた b-bit MinHash 法で計算した類似度は，文献 [24] によると誤差が 0 を中心に二項分布するため，類似度を合計することで，十分にファイル数が多い場合には正負の誤差が相殺され、平均で 0 に近づくことを期待した．評価実験においても，図 3.4 に示したように誤差を小さく抑えることができた．

ファイル単位の類似度を集計する方法はこれに限定されているわけではなく，たとえばファイル単位でのバージョンの多数決を取るという方法も考えられる．この 2 つの類似度の集計方法は，類似度が以下の条件を満たす場合に，異なる結果を出力する．

$$|S(q, v_1) > S(q, v_2)| > |S(q, v_1) < S(q, v_2)|$$
$$S_Q(v_1) < S_Q(v_2)$$

このような状況は，ライブラリの再利用方法によって，v_1，v_2 のどちらが正解の場合にも起こりうる．類似度の合計を採用する場合では，再利用したファイルの 1 つを大幅に編集した結果，その類似度の差が支配的になり，実際に再利用したバージョンの過半数のファイルが完全一致であっても，間違ったバージョンを検出してしまう可能性がある．多数決を採用する場合では，再利用したファイルの過半数に対して編集した結果，実際に再利用したバージョンとの類似度が低下すると，他のファイルの類似度の値に関わらず誤ったバージョンを検出してしまう．類似度の集計方法の違いが結果に影響する可能性はあるが，評価実験ではこのような状況は発生しなかった．ライブラリの再利用に際してファイルの修正は基本的には軽微であり，表 3.5 に示したとおり，すべてのファイルのうち 1.61% 以外はソフトウェア全体の検出結果のバージョンと最も類似していた．

3.5.2 実験対象の妥当性

実験対象のソフトウェアには，利用ライブラリのバージョンを適切に記録しているものだけを使用している．そのため，評価実験の結果には，それらのソフトウェアの特徴が影響している可能性がある．ただし，対象ソフトウェアは異なるコミュニティで開発されているものを選択したため，開発者の再利用方法に関する偏りの影響は少ないと考える．

実験対象の再利用状況を確認すると，ライブラリに含まれるすべてのソースファイルを再利用する場合がほとんどだった．libpng については Android のみテストに使われるソースファイルなどは再利用していなかった．このようなライブラリの一部分のみを再利用する場合としては以下のものが考えられる．

1. ライブラリ内の一部のファイルのみが再利用された場合．たとえばテストファイルの除去が行われるなど．
2. ファイルごとに異なるバージョンのライブラリからの再利用が行われた場合．たとえば修正パッチを最新版から取り込んだ場合など．
3. あるファイルの一部分のみが再利用された場合．たとえば特定の関数内のアルゴリズムだけが流用される場合など．

1 つ目の場合については，本手法は再利用バージョンの判定に閾値を超えるような類似度を持つソースファイルを用いるので，改変が大きくない限り再利用バージョンの検出は可能である．2 つ目の場合については，提案手法の章で述べたように，ファイル単位での再利用バージョンも出力するため，開発者はそれぞれのファイルについて対応方法を検討することが可能である．加えて，この事例は 3.4.3 に挙げたとおり，実験対象としたソフトウェアについても一部のファイルについて確認されており，提案手法によって簡単に確認できている．3 つ目の場合については，入力されたディレクトリ中の全てのファイルについて $S(q, v)$ が閾値を下回るのであれば提案手法で検出することができない．しかし，そもそも提案手法の適用対象は再利用したライブラリのみが含まれるディレクトリであるため考慮しない．このような場合はコードクローン検出手法など，異なる分析手法を用いる必要がある．

3.6 まとめ

本研究ではソフトウェアのファイル集合とライブラリのリポジトリを入力として，分析対象のソフトウェアが再利用しているライブラリのバージョンを検出する手法を提案した．ファイル単位の類似度をソースファイル集合で合計することで，99.3% の正確さで全体としてどのバージョンを再利用しているかを検出し，個別にどのファイルが変更されているかを分析可能とした．また，b-bit MinHash 法を採用することで，既存手法よりも高速かつ省メモリな計算を達成した．提案手法を用いることで，ソフトウェア開発者は再利用しているライブラリのバージョンを非常に高精度に特定することができ，その情報を用いてライブラリの更新などについて判断を下すことができるようになる．

今後の課題として，再利用されたファイルに施された修正を，自動的に既知のパッチ等と照合し，再利用されたファイルを効果的に分析する手法の確立が挙げられる．また，あるソフトウェアに使用されているすべてのソースファイルから，再利用ライ

ブラリを自動的に検出する技術への拡張も，今後の課題である．

第 4 章

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

本章では，第 3 章で確認したソースファイル集合へ b-bit MinHash 法を利用した類似性検出手法を拡張して，ソフトウェアプロダクト集合の派生関係を復元する．また，第 3 章での場合と比べてソースファイル集合間の比較回数が増加するため，更なるスケーラビリティの向上を目指し，Linear Counting 法による手法を提案する．また，評価実験によりこれらの手法の有効性を検証した．

4.1 はじめに

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 [6]．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．派生関係を持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持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 [46].

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 [12]．このような問題が起きる 1 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摘している [35]．

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，Kanda らの研究では，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 [22]．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つけ，さらに，比較する 2 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 2 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 [11] 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持つ 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 Kanda らの手法 [22] と比較する．

一方の b-bit MinHash 法は，著者らの研究 [45] 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．以前の研究では，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，2 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．1 つのソフトウェアプロダクトを 1 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 [11] 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，Kanda らの手法だけでなく *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 1 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，2 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通りである．

- 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．Kanda らの手法 [22] 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 [39] は，Kanda らの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 章では研究の背景について述べ，3 章では提案手法を詳述する．4 章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 章で妥当性への脅威について述べる．最後に，6 章でまとめを述べる．

4.2 背景

4.2.1 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．たとえば 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 [28]．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 [13]．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 [31]．

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，

全体の履歴がわ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 [39]．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 [22]．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 [11].

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 [7].

4.2.2　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 [23]．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 [22].

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よ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 [20]．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 [44]．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 [36].

しかし，これらの手法は，比較する 2 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，派生関係を持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，そのような性質を持つ 2 つの手法，b-bit MinHash 法 [24] 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 [41] 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．

b-bit MinHash 法を用いた集合の比較

集合間の類似度の 1 つである Jaccard 係数 [18] には，その推定値を高速に求める MinHash 法 [2, 3] 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 [24]．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，2 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，2 つの集合 S_1, S_2 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，ハッシュ値の下位 b ビットが偶然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 [45] にて説明している．また，これは 2.2 節と 3.2 節において説明した手法と同様のものである．

Linear Counting 法を用いた集合の比較

Linear Counting 法 [41] は集合の基数を推定する手法の 1 つである．これは，集合 S を表すビットマップ $B(S)$ を構築し，そのビットマップのうち 1 であるビットの数

Algorithm 3 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 $B_i \leftarrow 0$
2: **for** $s \in S$ **do**
3: $\quad h = H(s)$
4: $\quad B_h = 1$
5: **end for**

から S の基数の近似値を計算する手法である．Algorithm 3 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．2 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，2 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．

4.3 提案手法

提案手法は，互いに派生関係を持つ 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．具体的な手順は以下の通りである．

1. 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．
2. ある 2 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持つ軽量なデータ構造を利用する
3. 2 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，すべ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，開発者ができるだけ多くの部分を再利用すると仮定すると，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 [32] 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 を繋ぐ辺の重み $w(s_i, s_j)$ は，以下の式で定義する．

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

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，派生関係を持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うる．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

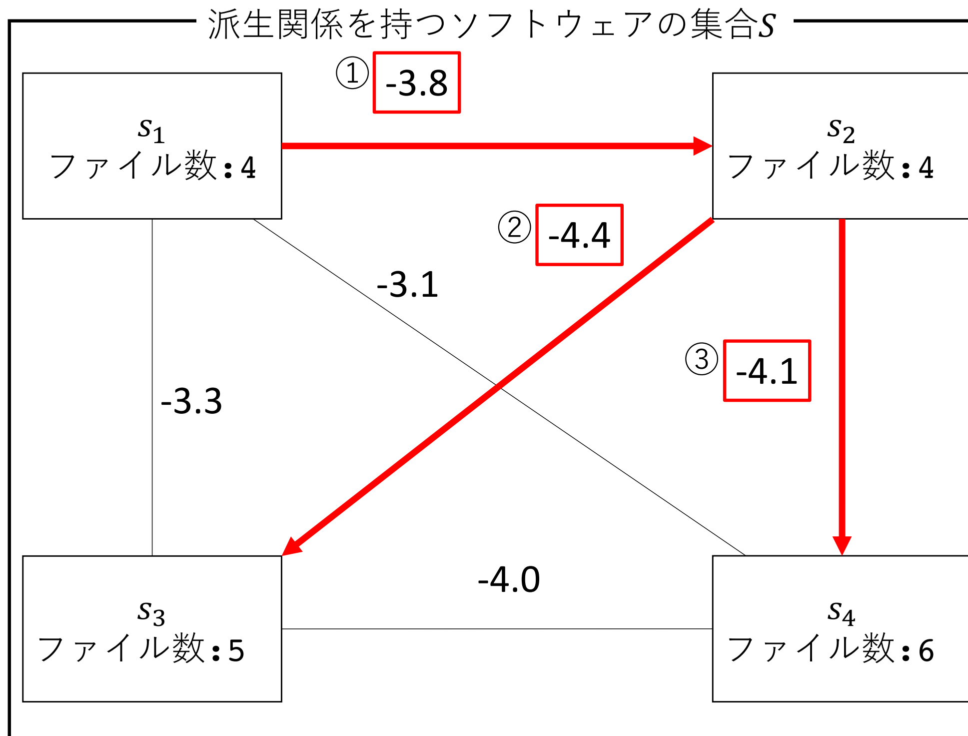


図 4.1: 提案手法の動作例

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わ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 4.1 に提案手法の動作例を示す．この図では，派生関係を持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 が持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 を繋ぐ辺を選択する．以降同様に全ての頂点が辺によって繋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 $s_1 \to s_2 \to s_3$ と，$s_2 \to s_4$ 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．

4.3.1 *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 [45]．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（ただし閾値 θ 以下の値は無視したもの）に相当し，すべ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 2 つの値のより大きいほうを選択する．なお，ここでの $\mathrm{sim}(q, f)$ は 3.3 節における $\mathrm{sim}(f_1, f_2)$ と同様のものである．

表 4.1: 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

4.3.2 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 1 つにつきすべてのソースファイルの行を表現したビットマップを 1 つ構築し，4.2.2 項で説明した $I_l(S_1, S_2)$ 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および行末の空白・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[38] 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 3 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．

4.4 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 Kanda らの手法の性能を比較する．

適用対象は，Kanda らの研究 [22] 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いく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持つ 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 4.1 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 1TB 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，その辺が持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，Kanda らの手法の結果 [22] である．

4.4.1 提案手法の精度

提案手法と Kanda らの手法の結果を表 4.2 に示す．表 4.2 中の左から 3 列目まではデータセットの番号，実際の系統樹が持つ辺の数，Kanda らの手法と提案手法で構築した系統樹が持つ辺の数を記載している．以降の列は，Kanda らの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．すべ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．Kanda らの手法 [22] 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，Kanda らの手法より若干低い精度となった．すべ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 Kanda らの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，Kanda らの手法，Linear Counting 法の順だった．

無向辺の精度に関しては，Kanda らの手法と比べて *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 Kanda らの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 Kanda らの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 Kanda らの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 Kanda らの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 Kanda らの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 1 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 1 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．Kanda らの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計が大きい方を派生先としている．つまり，2 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，Kanda らの手法では 1 対多の対応関係から差分を考えている．そのため，閾値よりも大きな類似度を持つようなファイルのすべ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．

表 4.2: 適用結果

No.	全体	辺の数	一致数											
			Kanda らの手法 [22]				b-bit MinHash 法				Linear Counting 法			
			無向辺		有向辺		無向辺		有向辺		無向辺		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%

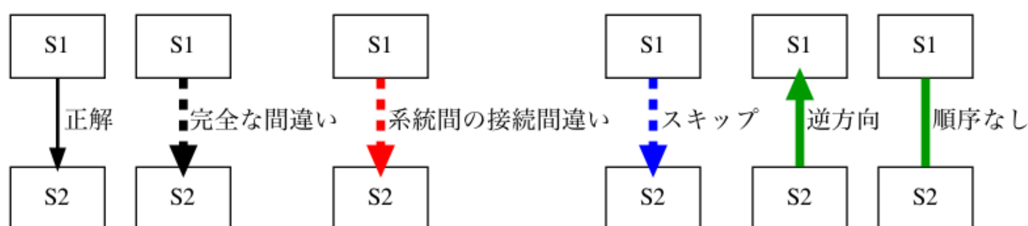


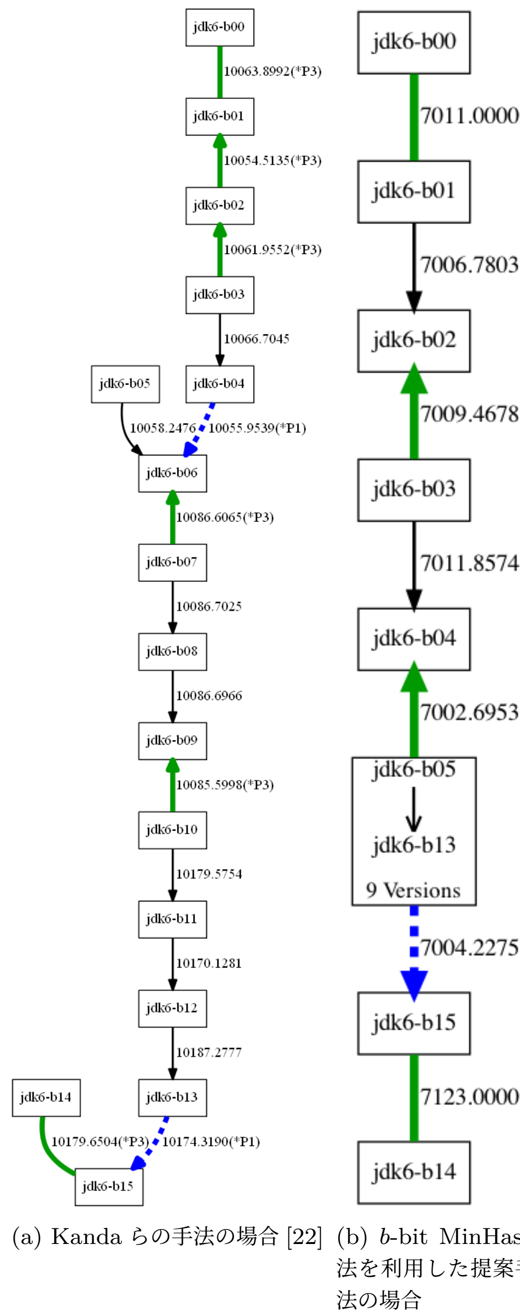
図 4.2: 系統樹における矢印の凡例

図 4.3 に，*b*-bit MinHash 法を利用した提案手法と Kanda らの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 1 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．系統樹における枝の表記の凡例を図 4.2 に示す．線が直線であれば，派生関係自体は正しく，破線の場合は派生関係が間違っている．破線のうち，黒で示したものは完全な間違いで，赤で示したものは系統間の接続間違いである．また，青で示したものはバージョンを飛ばして繋がっていることを，緑は順序の判定が間違っていることを表している．特に，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．Kanda らの手法で構築された系統樹は，Kanda らの手法の Online Appendix[*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繋がっている．

図 4.3a と図 4.3b 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わかる．また，図 4.3b 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 4.3a と比較すると，図 4.3b 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，Kanda らの手法における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，同一のインター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，Kanda らの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 4.4 に示す．この図にある矢印の形状などは全て図 4.2

[*1] https://sel.ist.osaka-u.ac.jp/pret/



(a) Kanda らの手法の場合 [22]　(b) *b*-bit MinHash 法を利用した提案手法の場合

図 4.3: Kanda らの手法と提案手法で構築されたデータセット 9 の系統樹

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．図 4.4 を見ると，Kanda らの手法で構築された系統樹と同様，大まかに 6 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わかる．また，提案手法は Kanda らの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，

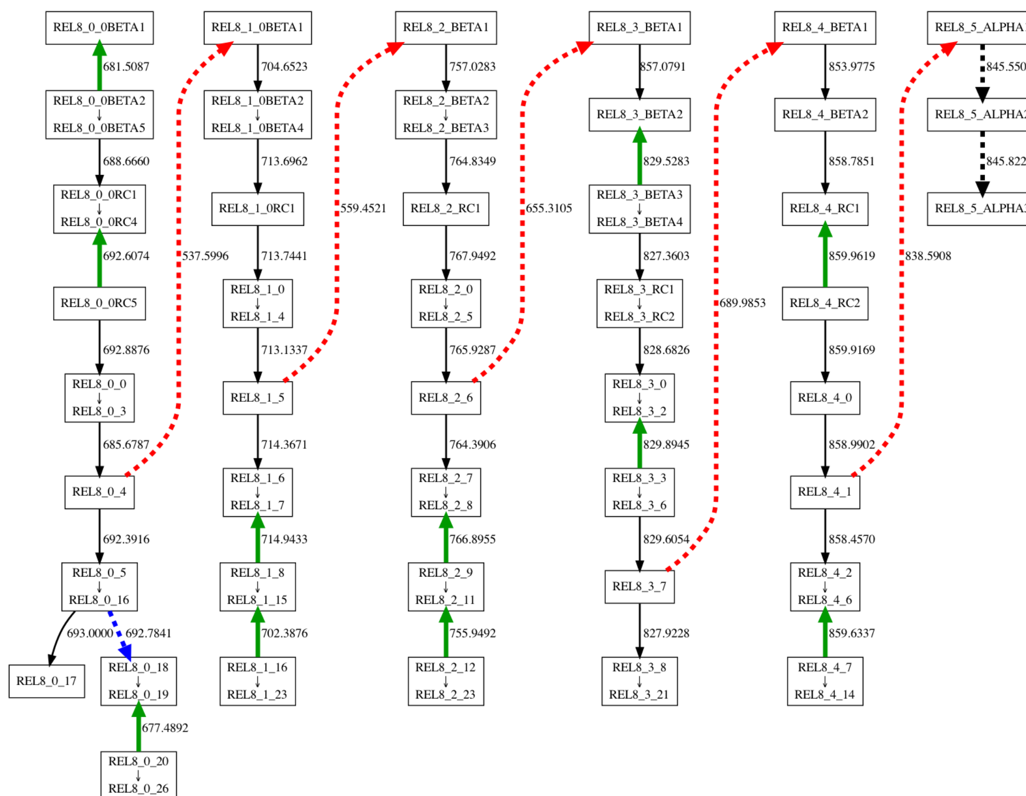


図 4.4: b-bit MinHash 法を用いた提案手法で構築されたデータセット 2 の系統樹

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 1 つと考えられる．ただし，個別の系統ごとでは，1 つの辺を除いて Kanda らの手法と同様に構築できており，Kanda らの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 4.2 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 Kanda らの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 1 つである．

表 4.3: 実行時間の比較

No.	Kanda らの手法 [22]	b-bit MinHash 法		Linear Counting 法	
	実行時間	実行時間	倍率	実行時間	倍率
	(A) [s]	(B) [s]	(A/B)	(C) [s]	(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

4.4.2 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，Kanda らの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，Kanda らの手法ではソースコードをトークン列に分割し，2 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，1 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．

これを定量的に評価するため，Kanda らの手法と提案手法の実行時間を計測し，比較する．Kanda らの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．Kanda らの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．Kanda らの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 1 つである nanoTime メソッドを，Kanda らの手法については bash の time コマンドを用いる．その結果を表 4.3 に示す．

b-bit MinHash 法では Kanda らの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 倍とさらに高速だった．この結果から，提案手法は Kanda らの手法と比べて非常に高速であるとわかった．

提案手法は，空間計算量に関しても効率的である．Kanda らの手法の実装では LCS を総当たりで効率よく計算するために，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 ソフトウェアあたり 128M ビットで済む．そのため，提案手法は Kanda らの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．

これらのことから，Kanda らの手法を含めて，構築した系統樹の精度と実行時性能はトレードオフの関係にあるとわかる．高速化に対して軽微な精度の低下から，特に大規模な適用対象に最も有効なのは Linear Counting 法であり，ある程度の規模であれば b-bit MinHash 法が十分高速かつほとんど精度が低下しないため最も有効である．

4.4.3 提案手法の実用性

実用性という観点では，Kanda らの手法と同様に，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 4.4 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，2 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，Kanda らの手法と同様である．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

表 4.4: 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]
Kanda らの手法	-	-	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

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 2 つの系統に別れ，その後再び合流するような閉路を持つ派生関係があった場合は，全域木という閉路を持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 1 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 Kanda らの手法と同様の実用性がある．それに加えて，提案手法は Kanda らの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．

4.4.4 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 Kanda らの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 [39] 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 1 つあた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，Kanda らの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[9] 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 2 つの尺度で比較する．その結果を表 4.4 に示す．

Kanda らの手法では 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 Kanda らの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わかる．

4.5　妥当性への脅威

本研究では，Kanda らの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，Kanda らののデータセットについては Kanda らの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，いく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さら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 上のすべ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さら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．

4.6　まとめ

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，Kanda らの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．Kanda らの手法では 3 日間処理しても 8 つ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 2 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わかった．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わかった．

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，1 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．

第 5 章

おわりに

5.1　まとめ

本論文では，ソフトウェアの保守管理作業が困難である 3 つの状況に対して，保守管理作業を支援する手法をそれぞれ述べた．1 つ目の，あるソースファイルの出自が失われた場合には，*b*-bit MinHash 法を利用したクローンされたファイルを検出するための WEB サービスを提案し，開発者は OSS のソースファイルをクローンして作成した自身のソフトウェアのソースファイルを公開することなく出自を調べることが可能となった．2 つ目の，ソフトウェア中に含まれる再利用したライブラリのバージョンが失われた場合では，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手法を提案し，開発者はソースコードのみからライブラリのどのバージョンを再利用したのか高速かつ正確に検出することが可能となった．3 つ目の，既存のソフトウェアとの派生関係の情報が失われた場合では，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を提案し，開発者は高速にソフトウェアプロダクトの集合からその派生関係を再構築することが可能となった．

b-bit MinHash 法を利用したクローンされたファイルを検出するための WEB サービスでは，OSS のオペレーティングシステムである Debian で利用可能なソフトウェアパッケージから 3,000 万個以上のソースファイルの情報を含むデータベースを構築し，その中からクエリとしたソースファイルと類似するソースファイルをソースファイルを表すシグネチャを用いて検出した．また，*b*-bit MinHash 法の誤差がどの程度結果に影響するのか検討した．これにより，既存研究の課題であったデータベースの集中管理を実現し，シグネチャを用いた検索によって高速かつソースファイルを公開できない開発者も利用可能な WEB サービスを開発した．

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手法では，開発中のソフトウェアで再利用しているライブラリのソースファイル群が，そのライブラリのどのバージョンから再利用したものであるのかを，軽量な類似度計算手法である *b*-bit MinHash 法を用いて高速かつ正確に検出した．また，ファイル単位で

も再利用元バージョンを検出することで，個別にどのファイルが変更されているかを分析可能とした．これにより，既存研究よりも高速かつ高精度にソースファイル集合の再利用情報を復元することが可能となった，

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では，*b*-bit MinHash 法と Linear Counting 法の 2 つの軽量なデータ構造を用いる手法を利用して，派生関係を持つソフトウェアプロダクトの集合からその進化履歴を高速に復元した．これにより，既存研究では実用的な時間で分析できなかった規模のソフトウェアプロダクト集合を対象に精度を落とすことなく進化履歴の復元が可能となった．

5.2 今後の研究方針

b-bit MinHash 法を利用したクローンされたファイルを検出するための WEB サービスには，自動でデータベースに新たな OSS のソースファイルを追加する機構を組み込むことや，対応する言語を増やすことで検索対象を拡張することが挙げられる．

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手法を応用して，自動で既知のパッチ等と照合し，再利用されたファイルを分析する手法の開発が挙げられる．また，ソースコードのみを情報源として，自身のプロジェクトに含まれる再利用しているライブラリに由来する既知の脆弱性を，自動で検出し通知する手法の開発も挙げられる．

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では，既存手法よりも大幅に向上したスケーラビリティを活かして，複雑なネットワーク構造の構築手法を応用した閉路を含むような進化履歴に対応する手法の開発が挙げられる．

参考文献

[1] Ira D. Baxter, Andrew Yahin, Leonardo Moura, Marcelo Sant'Anna, and Lorraine Bier. Clone detection using abstract syntax trees. In *Proceedings of the International Conference on Software Maintenance*, pp. 368–377, November 1998.

[2] Andrei Z. Broder. On the resemblance and containment of documents. In *Proceedings of the Compression and Complexity of Sequences 1997*, pp. 21–29, June 1997.

[3] Moses S. Charikar. Similarity estimation techniques from rounding algorithms. In *Proceedings of the 34th Annual ACM Symposium on Theory of Computing*, pp. 380–388, May 2002.

[4] Lucian Constantin. Developers often unwittingly use components that contain flaws. iTWorld.com. https://www.itworld.com/article/2936575/software-applications-have-on-average-24-vulnerabilities-inherited-from-buggy-components.html.

[5] Roberto Di Cosmo and Stefano Zacchiroli. Software heritage. www.softwareheritage.org.

[6] Yael Dubinsky, Julia Rubin, Thorsten Berger, Slawomir Duszynski, Martin Becker, and Krzysztof Czarnecki. An exploratory study of cloning in industrial software product lines. In *Proceedings of the 2013 17th European Conference on Software Maintenance and Reengineering*, pp. 25–34, March 2013.

[7] Slawomir Duszynski, Vasil Tenev, and Martine Becker. N-way diff: Set-based comparison of software variants. In *Proceedings of the 8th Working Conference on Software Visualization (VISSOFT)*, pp. 72–83, September 2020.

[8] Christof Ebert. Open source software in industry. *IEEE Software*, Vol. 25, pp. 52–53, May 2008.

[9] 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 November 4 2019.

[10] Debian GNU/Linux. The snapshot archive. snapshot.debian.org. http://

snapshot.debian.org/, Accessed August 19, 2016.

[11] Yasuhiro Hayase, Tetsuya Kanda, and Takashi Ishio. Estimating product evolution graph using kolmogorov complexity. In *Proceedings of the 14th International Workshop on Principles of Software Evolution*, pp. 66–72, August 2015.

[12] Armijn Hemel and Rainer Koschke. Reverse Engineering Variability in Source Code Using Clone Detection: A Case Study for Linux Variants of Consumer Electronic Devices. In *Proceedings of the 19th IEEE Working Conference on Reverse Engineering*, pp. 357–366, October 2012.

[13] Keisuke Hotta, Yukiko Sano, Yoshiki Higo, and Shinji Kusumoto. Is duplicate code more frequently modified than non-duplicate code in software evolution? an empirical study on open source software. In *Proceedings of the Joint ERCIM Workshop on Software Evolution and International Workshop on Principles of Software Evolution*, pp. 73—-82, September 2010.

[14] GitHub Inc. Github. https://github.com.

[15] Katsuro Inoue, Yusuke Sasaki, Pei Xia, and Yuki Manabe. Where does this code come from and where does it go? - integrated code history tracker for open source systems -. In *Proceedings of the 34th International Conference on Software Engineering*, pp. 331–341, June 2012.

[16] Takashi Ishio, Yusuke Sakaguchi, Kaoru Ito, and Katsuro Inoue. Source file set search for clone-and-own reuse analysis. In *Proceedings of the 2017 IEEE/ACM 14th International Conference on Mining Software Repositories*, pp. 257–268, May 2017.

[17] Kaoru Ito, Takashi Ishio, and Katsuro Inoue. Web-service for finding cloned files using b-bit minwise hashing. In *Proceedings of the 2017 IEEE 11th International Workshop on Software Clones*, pp. 1–2, February 2017.

[18] Paul Jaccard. The distribution of the flora in the alpine zone.1. *New Phytologist*, Vol. 11, No. 2, pp. 37–50, 1912.

[19] Kanyakorn Jewmaidang, Takashi Ishio, Akinori Ihara, Kenichi Matsumoto, and Pattara Leelaprute. Extraction of library update history using source code reuse detection. *IEICE Transactions on Information and Systems*, Vol. 101-D, No. 3, pp. 799–802, March 2018.

[20] Lingxiao Jiang, Ghassan Misherghi, Zhendong Su, and Stephane Glondu. Deckard: Scalable and accurate tree-based detection of code clones. In *Proceedings of the 29th International Conference on Software Engineering*, pp. 96–105, May 2007.

[21] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. Ccfinder: a multi-

linguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering*, Vol. 28, No. 7, pp. 654–670, 2002.

[22] Testuya Kanda, Takashi Ishio, and Katsuro Inoue. Approximating the evolution history of software from source code. *IEICE Transactions on Information and Systems*, Vol. E98-D, No. 6, pp. 1185–1193, 2015.

[23] Naohiro Kawamitsu, Takashi Ishio, Tetsuya Kanda, Raula G. Kula, Coen De Roover, and Katsuro Inoue. Identifying source code reuse across repositories using lcs-based source code similarity. In *Proceedings of the 2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation*, pp. 305–314, September 2014.

[24] Ping Li and Christian König. b-bit minwise hashing. In *Proceedings of the 19th International Conference on World Wide Web*, pp. 671–680, April 2010.

[25] Wayn C. Lim. *Managing Software Reuse.* Prentice Hall, 1998.

[26] Cristina V. Lopes, Petr Maj, Pedro Martins, Vaibhav Saini, Di Yang, Jakub Zitny, Hitesh Sajnani, and Jan Vitek. Déjàvu: A map of code duplicates on github. *Proceedings of ACM Program. Lang.*, Vol. 1, No. OOPSLA, pp. 84:1–84:28, October 2017.

[27] Daniel M. German, Massimiliano Di Penta, Yann-Gael Gueheneuc, and Giuliano Antoniol. Code siblings: Technical and legal implications of copying code between applications. In *Proceedings of the 2009 6th IEEE International Working Conference on Mining Software Repositories*, pp. 81–90, May 2009.

[28] Yuki Manabe, Yasuhiro Hayase, and Katuro Inoue. Evolutional analysis of licenses in foss. In *Proceedings of the Joint ERCIM Workshop on Software Evolution and International Workshop on Principles of Software Evolution*, pp. 83–87, September 2010.

[29] George Marsaglia. Xorshift RNGs. *Journal of Statistical Software*, Vol. 8, No. 14, pp. 1–6, 2003.

[30] Parastoo Mohagheghi, Reidar Conradi, Ole M Killi, and Henrik Schwarz. An empirical study of software reuse vs. defect-density and stability. In *Proceedings of the 26th International Conference on Software Engineering*, pp. 282–292, May 2004.

[31] Manishankar Mondal, Banani Roy, Chanchal K. Roy, and Kevin A. Schneider. Investigating near-miss micro-clones in evolving software. In *Proceedings of the 28th International Conference on Program Comprehension*, pp. 208—218, June 2020.

[32] R. C. Prim. Shortest connection networks and some generalizations. *The Bell System Technical Journal*, Vol. 36, No. 6, pp. 1389–1401, 1957.

[33] FreeBSD Project. Freebsd ports collection. https://www.freebsd.org/ports/.

[34] R. Rivest. The md5 message-digest algorithm. RFC 1321 (Informational), Internet Engineering Task Force. http://www.ietf.org/rfc/rfc1321.txt.

[35] Julia Rubin, Andrei Kirshin, Goetz Botterweck, and Marsha Chechik. Managing forked product variants. In *Proceedings of the 16th International Software Product Line Conference*, pp. 156–160, September 2012.

[36] Hitesh Sajnani, Vaibhav Saini, Jeffrey Svajlenko, Chanchal K. Roy, and Cristina V. Lopes. SourcererCC: Scaling code clone detection to big-code. In *Proceedings of the 38th International Conference on Software Engineering*, pp. 1157–1168, May 2016.

[37] Yusuke Sasaki, Tetsuo Yamamoto, Yasuhiro Hayase, and Katsuro Inoue. Finding file clones in freebsd ports collection. In *Proceedings of the 7th IEEE Working Conference on Mining Software Repositories*, pp. 102–105, May 2010.

[38] Yonik Seeley and Andrew Gaul. Murmurhash3. https://github.com/yonik/java_util. referenced in Octover 2019.

[39] Diomidis Spinellis. A repository of unix history and evolution. *Empirical Software Engineering*, Vol. 22, pp. 1372–1404, August 2016.

[40] Esko Ukkonen. Approximate string-matching with q-grams and maximal matches. *Theoretical Computer Science*, Vol. 92, No. 1, pp. 191 – 211, 1992.

[41] Kyu-Young Whang, Brad T. Vander-Zanden, and Howard M. Taylor. A linear-time probabilistic counting algorithm for database applications. *ACM Transactions on Database Systems*, Vol. 15, No. 2, pp. 208–229, June 1990.

[42] Pei Xia, Makoto Matsushita, Norihiro Yoshida, and Katsuro Inoue. Studying reuse of out-dated third-party code in open source projects. *Computer Software*, Vol. 30, No. 4, pp. 98–104, 2013.

[43] Stephen W. L. Yip and Tom Lam. A software maintenance survey. In *Proceedings of the 1st Asia-Pacific Software Engineering Conference*, pp. 70–79, 1994.

[44] 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.

[45] 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. 電子情報通信学会論文誌, Vol. J103-D, No. 7, pp. 542–554, 7 月 2020.

[46] 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. 情報処理学会研究報告, 第 2009-SE-166 巻, pp. 1–8, 11 月

2009.

[47] 日本工業規格. ソフトウェア技術 – ソフトウェアライフサイクルプロセス – 保守. JIS X 0161:2008.