一般社団法人　電子情報通信学会
THE INSTITUTE OF ELECTRONICS,
INFORMATION AND COMMUNICATION ENGINEERS

信学技報
IEICE Technical Report
SS2021-1,KBSE2021-13(2021-07)

抽象構文木を用いた差分検出手法の活用による Git のファイル追跡機能改善の試み

藤本　章良†　　肥後　芳樹†　　楠本　真二†

† 大阪大学大学院情報科学研究科
〒 565-0871 大阪府吹田市山田丘 1-5
E-mail: †{a-fujimt,higo,kusumoto}@ist.osaka-u.ac.jp

あらまし　ソフトウェア開発においては，バージョン管理システムが頻繁に利用されている．バージョン管理システムの一つである Git は，変更の履歴を個別のファイルごとに辿ることができ，ファイル名が変更された場合でもファイル内容の類似度をもとに変更前のファイルを特定し，追跡を行うことが可能である．ファイル内容の類似度は，全行数のうち変更が行われた行数の割合で計算される．しかし，行単位による比較はソースコードの構造的特徴を考慮しておらず，粒度も粗いため，変更前のファイルの特定を誤ったり追跡が途切れたりする可能性がある．提案手法では，これらの問題の改善に向けて抽象構文木を利用した差分検出手法を利用し類似度を算出する．197 個のオープンソースプロジェクトに対して実験を行った結果，ファイル名変更の検出数が 3.3%増加し，行単位の比較による手法と出力が異なるファイルの変更履歴について平均で 1.37 倍追跡できるコミット数が増加した．また，精度を調査したところ F 値が最大で 0.943 となり，既存手法の最大値である 0.926 を上回った．
キーワード　Git，ファイル追跡，抽象構文木

1. まえがき

ソフトウェア開発においては，バージョン管理システムが頻繁に利用されている [1]．バージョン管理システムを利用することで，過去のバージョンとして記録された状態に戻す，バージョン間の差分を確認する，複数の開発者で並行して開発を進めるといった行為が容易に実現できる．さらに蓄積された変更履歴は，開発者がソフトウェアの振る舞いや変更目的を理解するために役立つ [2]．加えて，研究分野での利用もなされており，バグを含むモジュールの予測 [3] や変更パターンの検出 [4] といった研究が広く行われている．

バージョン管理システムの一つである Git は，リポジトリ全体の変更履歴の他に個別のファイルに対して行われた変更履歴だけを抽出して出力可能である．特定のファイルについての変更履歴を出力するために，ファイル名を利用して対象のファイルが変更されたコミットを特定し履歴を出力する．ファイル名が変更された場合でも変更前のファイルを特定し，それ以前の履歴は変更前のファイル名で追跡する．ファイル名が変更された場合，変更前のファイルを特定するためにファイル内容の比較を行う．変更前にのみ存在するファイルの中から 1 つファイルを選択し，追跡中のファイルと比較する．2 つのファイル間で一致する行数の割合を類似度として算出し，類似度が一定の閾値を超えていればファイル名の変更が行われたとみなす．

しかし，類似度の算出方法には課題が存在する．2 つのファイルを行単位で比較する場合，ソースコードの構造的特徴は考慮されておらず比較の粒度も粗い．そのため，変更前のファイルの特定を誤ったり追跡が途切れてしまったりする可能性がある．例えばファイル名の変更に伴い，クラス名やメソッド名，変数名が変更されただけの場合であっても行全体の変更とみなし，その結果類似度が低下し追跡が途切れるおそれがある．

そこで本研究では，Git のファイル追跡機能の改善のために，抽象構文木を利用した差分検出手法を用いて類似度を算出する手法を提案する．抽象構文木はソースコードの構造そのものを表現しており，これらの差分から類似度を算出することでファイルの追跡を改善できると著者らは考えた．

提案手法の有用性を評価するために，197 個のオープンソースプロジェクトに対して複数の実験を行った．まず，ファイル名変更の検出数を調査したところ，提案手法はファイル名の変更を 3.3%多く検出した．次に，目視で精度を確認したところ，行単位の比較により類似度を算出する手法（以下，既存手法）と比べ再現率が大きく上回った．F 値も既存手法よりも高くなり，閾値を 40%に設定した時 0.943 で最大となり既存手法の最大値である 0.926 を上回った．また，出力される履歴の長さを調査し，既存手法と出力が異なるファイルのみに注目したところ，出力される変更履歴のコミット数が 1.55 倍に増加した．最後に，実行時間を比較すると提案手法は実行時間が 1.71 倍増加した．実行時間短縮のためにヒューリスティックを適用すると，実行時間と追跡精度の間にトレードオフが存在するという結果が得られた．

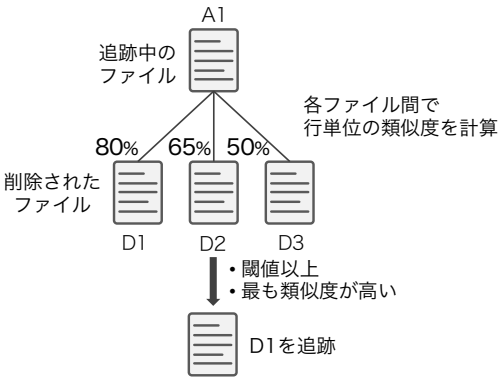

図 1　Git におけるファイル名の変更検出

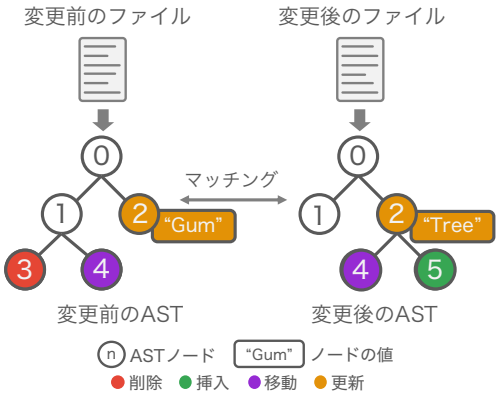

図 2　GumTree の差分検出

2. 準　　備

2.1 Git のファイル追跡

Git では`--follow`オプションを用いることで，特定のファイルを追跡して変更履歴を表示することができ，ファイル名が変更されても追跡可能である．追跡しているファイルのファイル名が変更された場合，変更前のファイルを特定し，それ以前のコミットに対しては変更前のファイル名で追跡を行う．ファイル名が変更されたコミットにおける Git のファイル追跡のプロセスを図 1 に示す．Git では，それぞれのコミットについて削除されたファイルと追加されたファイル，内容が変更されたファイルの情報を記録している．ファイル名の変更を検出するために，追跡中のファイルとそのコミットで削除されたファイル間で，ファイル内容の類似度を計算する．閾値を超えたファイルの組み合わせをファイル名の変更が行われたファイルの組とみなす．閾値を超えたファイルの組が複数存在する場合，最も類似度が高い組をファイル名の変更が行われたファイルの組とみなす．

ある 2 つのファイルの類似度 S は，2 つのファイル内容を行単位で比較し，以下の式で算出される．$length(f)$ は，ファイル f の行数であり，$common(f_1, f_2)$ は一致する行数である．

$$S(f_1, f_2) = \frac{common(f_1, f_2)}{max(length(f_1), length(f_2))} \times 100 \qquad (1)$$

行単位の差分を検出するために，Myers [5] や Histgram，ハッシュを用いたアルゴリズムが利用される．

2.2 抽象構文木

抽象構文木（以下，AST）は，ソースコードを構文解析して得られる木構造のデータである．1 つのソースファイルから 1 つの AST が生成される．AST のエッジは，その両端のノードに直接の親子関係があることを示す．AST の各ノードは以下の 5 つの要素から構成される．

ID　各 AST 内で固有の識別子

親ノード　AST の各ノードは，親ノードへの参照を持つ．ただし，根ノードの親は存在しないので何も保持しない．

子ノード　AST の各ノードは，子ノードへの参照を持つ．ただし，葉ノードの子は存在しないので何も保持しない．

ラベル　if 文や変数宣言といった文法上の型を表す．

値　各ノードが持つラベル以外の情報である．例えば，識別子のノードはメソッド名や変数名を値として保持する．

2.3 GumTree [6]

GumTree は，AST の差分を検出するツールである．GumTree は変更前後のソースファイルを入力として受け取り，それぞれに対応する AST を生成する．2 つの AST の違いを AST のノード単位の編集スクリプトとして出力する．編集スクリプトとは，変更後の AST を得るために変更前の AST に適用された編集操作の列である．GumTree が出力する編集スクリプトは，挿入・削除・移動・更新の 4 つの操作から構成される．

図 2 に GumTree の動作を示す．木構造の差分を計算するために変更前後のファイルから生成した AST に対してマッチングを行う．マッチングとは変更前後における AST のノード間に対応づけを行う処理である．マッチングされたノードは，変更の前後で同じノードとして扱われる．マッチングの結果と AST を参照し，Chawathe らのアルゴリズム [7] を用いて変更前の AST に対して操作が行われたノードを得る．この例では，3 番のノードは変更前にのみ存在するため削除，5 番のノードは変更後にのみ存在するため挿入，4 番のノードは親のノードが変化したため移動と出力される．また，2 番のノードは変更前後の両方に存在し親も変わらないが，値が変化したため更新と出力される．

3. 研 究 動 機

図 3 は，AndroidBootStrap プロジェクトで発生した`DimenUtils.java`についての変更を示している．この変更では，プロジェクト構造の変化によりパス名が変更されている．同時にメソッドの追加と Javadoc の追加も行われているが，変更の前後では同一のファイルと考えられる．しかし，この 2 つのファイルの類似度を 2.1 節 (1) 式に基づいて計算すると，25%となる．Git における類似度の閾値のデフォルト値は 50%であり，これらのファイルは同一でないファイルとして扱われるため，この変更以前の履歴を追跡することが不可能となる．

同一のファイルであるにもかかわらず類似度が低下する原因は，行単位による比較ではソースコードの構造を考慮できていないからである．例えば 1 行目を比較すると，パッケージ名の一部のみが変更されているが，パッケージ文全体が削除・挿入されたとして類似度が計算される．同様に削除行 6 行目と挿入行 16 行目を比較するとメソッド名のみが変更されているが，行全体の

```
ebfb0d4753392ad00a5866e368cd0357aec12947 api tidy up
- .../src/main/java/com/beardedhen/androidbootstrap/support/DimenUtils.java
+ .../src/main/java/com/beardedhen/androidbootstrap/utils/DimenUtils.java
```

```
 1     - package com.beardedhen.androidbootstrap.support;
     1 + package com.beardedhen.androidbootstrap.utils;
 2   2   import android.content.Context;
 3   3   import android.content.res.Resources;
 4   4   import android.support.annotation.DimenRes;
     5 + /**
     6 +  * Utils class for resolving color resource values.
     7 +  */
 5   8   public class DimenUtils {
 6     -     public static float textSizeFromDimenResource(Context context,
                                                 @DimenRes int sizeRes) {
     9 +     /**
    10 +      * Resolves a dimension resource that uses scaled pixels
    11 +      *
    12 +      * @param context the current context
    13 +      * @param sizeRes the dimension resource holding an SP value
    14 +      * @return the text size in pixels
    15 +      */
    16 +     public static float pixelsFromSpResource(Context context,
                                                 @DimenRes int sizeRes) {
 7  17           final Resources res = context.getResources();
 8  18           return res.getDimension(sizeRes) / res.getDisplayMetrics().density;
 9  19       }
    20 +     /**
    21 +      * Resolves a dimension resource that uses density-independent pixels
    22 +      *
    23 +      * @param context the current context
    24 +      * @param res     the dimension resource holding a DP value
    25 +      * @return the size in pixels
    26 +      */
    27 +     public static float pixelsFromDpResource(Context context,
                                                 @DimenRes int res) {
    28 +         return context.getResources().getDimension(res);
    29 +     }
    30 +
    31 +
10  32   }
```

図 3　類似度が低く追跡が途切れる変更

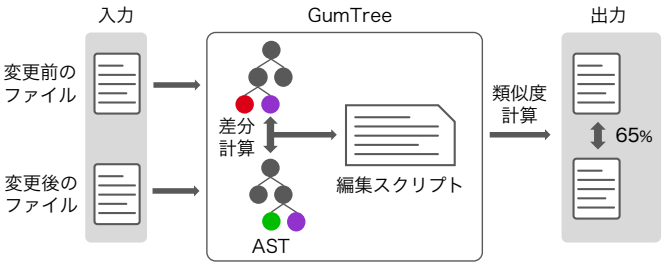

図 4　提案手法の概要

削除と挿入となりこのメソッド全体の 25%が変更されたこととなる．また Javadoc はプログラムの機能を変化させないが，この例では挿入された 24 行のうち Javadoc が 17 行を占めており類似度を低下させる大きな要因となっている．

そこで，ソースコードの構造を考慮した上で類似度の計算を行うことで，追跡の精度が向上するのではないかと著者らは考えた．ソースコードの構造を考慮するため，AST を利用した類似度の計算手法を提案する．AST の利用により，行単位による比較よりも粒度を細かくすることができ，Javadoc やコメントのような類似度を計算する上でノイズとなりうる成分の割合を削減することができる．

4. 提 案 手 法

提案手法では，Git のファイル追跡プロセスのうち変更前後におけるファイルの類似度の算出方法を改善する．提案手法の概要を図 4 に示す．類似度を算出したい変更前後のファイルを GumTree の入力として与える．GumTree はそれぞれのファイルから AST を生成し，差分を計算する．そして GumTree から出力された編集スクリプトを元に，類似度を算出する．t_1，t_2 をそれぞれ変更前後のファイルから生成した AST とした時，類似度 S を次のように定義する．$treeSize(t)$ は，AST t に含まれるノード数を表す．$editScript(t_1, t_2)$ は，t_1，t_2 を GumTree に与えた時に出力される編集スクリプトである．

$$S(t_1, t_2) = \left(1 - \frac{length(editScript(t_1, t_2))}{treeSize(t_1) + treeSize(t_2)}\right) \times 100 \quad (2)$$

変更前後のファイルの内容が類似しているほど，編集スクリプトが短くなるため，類似度が高くなる．この類似度が閾値を超えたファイルを，ファイル名が変更されたファイルとし追跡を行う．

図 3 の例に対し提案手法を用いて類似度を算出する．変更前後のファイルから生成した AST のノード数はそれぞれ 53 と 103 であり，GumTree の出力する編集スクリプトの長さは 52 である．(2) 式を用いて計算した結果，類似度は 66.7%となりファイル名が変更されたとみなし，追跡を続ける．

編集スクリプトの操作に重みはなく，どの操作が行われても長さは 1 だけ加算される．GumTree の出力する編集スクリプトに含まれる操作のうち，挿入と削除は操作対象が単一の AST ノードであるのに対し，移動は操作対象が AST の部分木である．あるソースコードが挿入または削除された場合，その各ノードに対して操作が行われたと扱われ，編集スクリプトの長さはノードの数になる．一方で，移動は部分木に対しての操作であるため編集スクリプトの長さは 1 である．従って，挿入と削除は移動よりも編集スクリプトが長くなる．ソースコードの削除・挿入よりも移動の方が機能差が小さいと考えられるため，移動が発生した時の類似度が高くなるよう，GumTree 出力した編集スクリプトに操作ごとに重みをつけずそのまま利用する．

なお比較対象のファイル少なくとも一方が，テキストファイルのように AST を生成できないファイルである場合や，文法的エラーにより AST の生成に失敗する場合，既存の行単位による比較から類似度を算出する．

5. 評 価 実 験

提案手法の評価のために，Git の Java 実装である JGit(注1) の一部を改変し，提案手法を実装し実験を行った．GumTree は Java で実装されており API も公開されているため，JGit に組み込むのが容易だったからである．この評価実験では

- ファイル名変更の検出数
- 追跡可能な変更履歴の長さ
- 追跡精度
- 実行時間

の 4 つの項目について，既存手法と提案手法をそれぞれ適用した JGit を用いて比較を行った．

5.1 実 験 対 象

実験対象として，Borges のデータセット [8] から 197 個の Java プロジェクトを選択した．このデータセットは GitHub でスター数の多い 2,279 個のプロジェクトから構成されている．データセットには Java プロジェクトが 202 個含まれているが，GitHub で公開されていないプロジェクト（2 個）と GumTree の実行に失敗するプロジェクト（3 個）を除外した．

5.2 ファイル名変更の検出数

類似度の閾値を変化させ，ファイル名変更の検出数の変化を既

(注1)：https://www.eclipse.org/jgit/

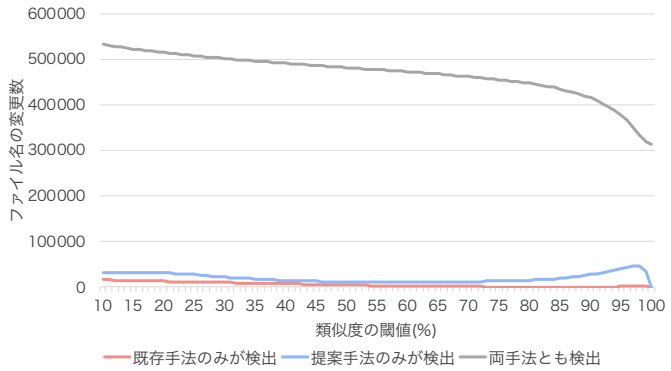

図 5　手法別によるファイル名変更の検出数の合計

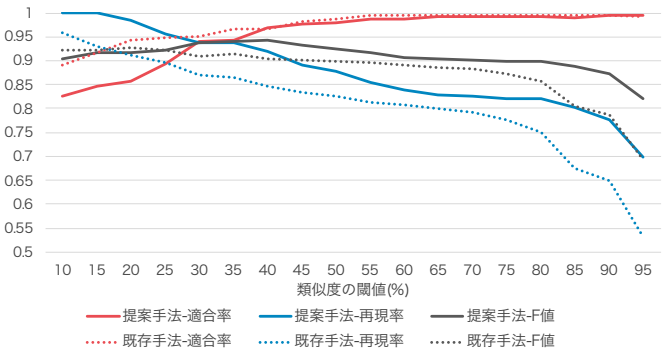

図 6　手法別による適合率・再現率・F 値

存手法と提案手法の間で比較する．各プロジェクトの全てのコミットに対し，ファイル名変更の検出を実行する．ファイル名の変更数を計測し，全プロジェクトで合計した結果が図 5 である．

両手法ともに検出するファイル名の変更が大部分を占めているが，提案手法のみが検出するファイル名の変更は，既存手法のみが検出する変更数を上回っており，平均で 3.3%，最大で 13.1% 上回っている．特に，閾値が 90%以上の場合，提案手法のみが検出するファイル名の変更が増加することがわかる．変数名やメソッド名のみの変更のような僅かな変更に対して，既存手法では行全体の変更となり一致部分が少なくなる一方で，提案手法では対象の AST ノードだけに対する変更となるため算出される類似度が高い値となりやすい．そのため，90%以上の高い閾値において，提案手法のみが検出できるファイル名の変更が増加したと考えられる．

5.3　追跡精度

各手法によって出力された履歴が実際に正しいかを調査する．ファイル名の変更がない場合は追跡が正しく行われるのは自明であるため，変更履歴中のファイル名の変更を正しく検出できるかを確認する．ファイル名の変更検出のための閾値を変えて，適合率，再現率，F 値がどのように変化するかを調査する．

まず，各プロジェクトから 1 つのファイルを無作為に選択し，それらのファイルについて正しい変更履歴のデータセットを作成する．しかしファイル名の変更時に，削除されたファイルと追加されたファイルの全ての組み合わせを確認し，正解のデータセットを作成することは現実的でない．そこで以下の手順で正解のデータセットを作成した．

1. `jgit log -p -U 15 -M -Mscore 10 --follow` *`path`*(注2) を実行し，出力された履歴を目視で確認する．検出したファイル名の変更のうち，正しく検出した個数を計測する．閾値を 10% まで下げ，できるだけ多くの変更を確認する．
2. 2 人の著者の間で独立に 1. の手順を行う．
3. 2. の結果を照合し，一致しなかったファイルの変更履歴については，著者らで議論を行い判断を行う．

HEAD コミットにおいて Java ファイルが含まれていない 6 個のプロジェクトは除外し，合計で 191 個のプロジェクトを対象とした．手順 1. では，提案手法を実装した JGit を利用した．また，手順 3. でファイル名の変更の数が一致しなかったプロジェクトは 5 つあり，著者らの間で合意をとった．

適合率，再現率を算出するために，各手法が出力した履歴のうちファイル名変更の検出数と，上記手順で作成したデータセットのファイル名の変更数を比較する．各手法のファイル名変更の検出数は

```
jgit log -p -U 15 -M -Mscore threshold --follow path
  | grep "^rename from|^copy from"
  | wc -l
```

で計測する．

ファイル名変更の検出に必要な類似度の閾値を 5%ごとに変更し，それぞれの場合の適合率，再現率および F 値を求めた結果が図 6 である．

適合率に注目すると，閾値 40%以上では既存手法と提案手法はほとんど同じ値を取っている．一方，閾値 40%以下の場合提案手法が既存手法を下回っていることから，低い閾値の場合に限り提案手法は必要以上に追跡する可能性がある．続いて再現率に注目すると，どの閾値を設定しても提案手法が既存手法を上回っている．すなわち，提案手法は既存手法よりも長くファイルの追跡が可能であり，類似度を高く設定しても追跡が途切れることが少ないといえる．また，提案手法の F 値は閾値 40%の時に 0.943 で最大となり，全ての閾値における既存手法の F 値を上回っている．以上の結果より，追跡精度は提案手法が優れているといえる．

5.4　追跡可能な変更履歴の長さ

この実験では，`--follow` オプションを利用してファイルの変更履歴を追跡した時，出力されるコミット数を既存手法と提案手法とで比較する．各プロジェクトに含まれる全ての Java ファイルから取得した変更履歴について，1 ファイルあたりの変更履歴におけるコミット数の平均値をプロジェクトごとに算出し，箱ひげ図として図 7 に表す．プロジェクトに含まれるファイルは，HEAD コミットに存在しているファイルだけではなく，過去に存在する全てのファイルを対象とした．また類似度の閾値は，追跡精度の実験の結果から提案手法において最も F 値が高くなる 40%とした．

それぞれの手法を比較すると，提案手法の方がわずかではあるが長く履歴を追跡していることがわかる．中央値を比較すると，既存手法は 8.37，提案手法は 8.49 であり全体として追跡で

(注2)：`-Mscore` は閾値を変更するために実装したオプションである．

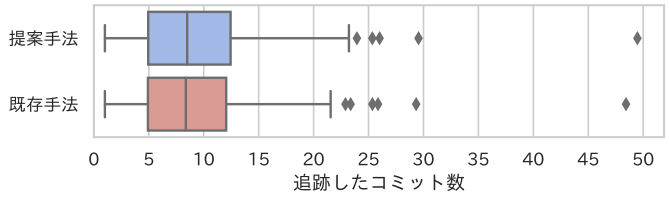

図 7　手法別による追跡できる履歴の長さ

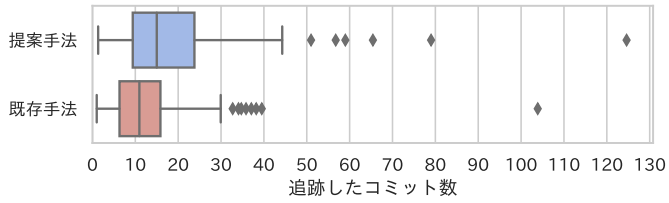

図 8　手法によって出力が異なるファイルのみに注目した時に追跡できる履歴の長さ

きる履歴のコミット数が 1.5%増加した．

次に，提案手法と既存手法で出力する履歴が異なったファイルのみに注目し，各プロジェクトごとに 1 ファイルあたりのコミット数の平均値を算出した場合の箱ひげ図が図 8 である．プロジェクトに含まれるファイルのうち，提案手法と既存手法の出力が異なるファイルを含んでいたプロジェクトは，合計で 150 個であった．出力が異なるファイルのみを比較した場合，提案手法が既存手法よりもかなり長く変更履歴を追跡していることが明らかとなった．中央値を比較すると，1.37 倍追跡可能なコミット数が増加している．これは，既存手法では途切れてしまっていた変更履歴の追跡が，提案手法によってより長く追跡可能になったことが要因であると考察される．

また，既存手法の方が 1 ファイルあたりのコミット数の平均が長いプロジェクトは合計で 12 個あった．それらのプロジェクトについて比較したところ，コミット数の差は最大で 7 であり平均で 1.27 であった．一方で，提案手法がより長く追跡できるプロジェクトは 138 個であり，長さの差の平均は 6.56 であった．以上のことから，提案手法は既存手法よりも長く変更履歴を追跡することが可能であり，既存手法の方が提案手法よりも長く履歴を追跡できるプロジェクトは存在するが，その場合におけるコミット数の差は大きくないことが明らかとなった．

5.5 実行時間

手法の違いによって，履歴の追跡に必要な時間がどの程度変化するかを調査する．各プロジェクトから 1 つのファイルを選択し，それぞれの手法を用いて以下のコマンドを実行した時，ファイルの履歴の追跡にかかる時間を測定する．

```
jgit log -M -Mscore 40 --follow path
```

類似度の閾値は，履歴の長さの調査時と同じく 40%とした．また 5.4 節の実験と同様に，HEAD コミットに Java ファイルが含まれる 191 個のプロジェクトが対象である．

実験では，CPU: Ryzen Threadripper 3960X(24C/48T)，メモリ: 128GB を搭載した Ubuntu 上で行った．また，JGit に対してファイル追跡の最適化とマルチスレッドへの対応を行なっている．

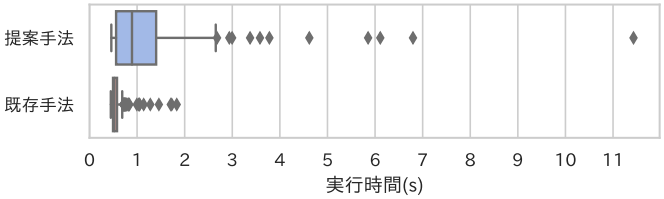

図 9　それぞれの手法による実行時間の比較

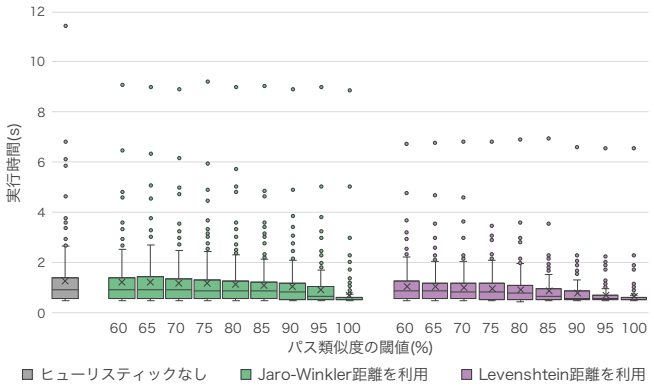

図 10　ヒューリスティックを利用した場合の実行時間の比較

実験結果を図 9 に示す．それぞれの手法について各ファイルの追跡にかかった時間を測定し，箱ひげ図として表現している．提案手法では AST を構築し差分を検出しているため計算量が増加し，全体に既存手法よりも追跡に必要な時間が増加している．中央値を比較したところ，提案手法は既存手法の 1.71 倍の時間を要していた．

続いて，提案手法の適用時に計算時間を削減するためのヒューリスティックを利用した場合，どの程度効果があるか実験を行った．計算時間と追跡精度について調査する．2.1 節で述べたようにファイル名の変更を検出するために，削除されたファイルと追跡中のファイルの間で類似度の計算を行うが，類似性の低いファイル同士の比較も多く含まれている．そこで，ファイルのパス名に着目する．経験的にパスの類似度が高ければ類似したファイルであり，パスの類似度が低ければ無関係なファイルであると考えられる．ファイル間のパス名の類似度がある閾値を超えていれば，類似している可能性の高いファイルとみなし，AST を用いてファイル内容の類似度を算出する．これにより，ファイル内容の比較回数を削減し実行時間の短縮を図る．

パスの類似度は，Levenshtein 距離 [9] と Jaro-Winkler 距離 [10] の 2 通りの計算手法を用いて算出する．距離を [0,1] の間で正規化し，値が高いほど類似性が高いとみなす．ファイル内容の比較を行うために必要なパスの類似度の閾値を 5%ずつ変化させ，それぞれの結果を比較する．また，提案手法のみにヒューリスティックを適用し実験を行った．その結果が図 10 である．

どちらの計算手法においても，パスの類似度の閾値を高く設定するほど実行時間が短縮していることがわかる．このことから，狙い通りファイル内容の比較を行う組み合わせが限定できていると考えられる．また Jaro-Winkler 距離を利用する手法と Levenshtein 距離を利用する手法を比較すると，同じ閾値の場合は Levenshtein 距離を利用した方が実行時間は短縮される．

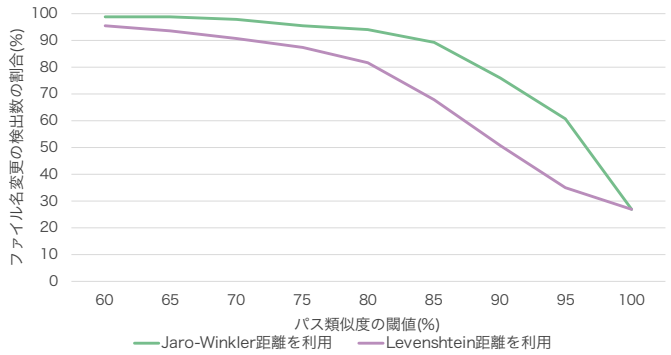

図 11　ヒューリスティック適用時に検出するファイル名の変更数の割合

精度についても調査を行った．パスの類似度の閾値を変化させ，ファイル名の変更をどの程度検出できるかをヒューリスティック適用前と比較する．その結果が図 11 である．図 11 中の縦軸はヒューリスティック適用前に検出したファイル名の変更の個数とヒューリスティック適用後の検出数の割合である．この割合から，Levenshtein 距離を利用する手法の場合早い段階から検出数が低下することが明らかになった．また Jaro-Winkler 距離を利用する場合，パスの類似度の閾値が 85%まではファイル名変更の検出の割合が約 90%と高いが，そこから類似度を高くすると検出の割合は急激に低下している．

実行時間の計測結果（図 10）とファイル名変更の検出の割合（図 11）を同時に比較すると，実行時間と検出精度にはトレードオフの関係があることが明らかになった．どちらの手法が特に優れていることはなく，例えば Levenshtein 距離を利用する手法は実行速度は速いがその分精度は劣る．

6. 妥当性の脅威

提案手法は特定のプログラミング言語に依存しない．したがって GumTree を適用可能なプログラミング言語全てに対して適用可能である．評価実験では Java ファイルのみを対象として実験を行ったが，他の言語のファイルを対象とした場合に同様の結果が得られるとは限らない．

追跡精度の調査において，Git の追跡結果を利用してファイル名変更の正解のデータセットを用意した．本来であれば，ファイル名の変更が行われたコミットで全てのファイルの組み合わせを確認し，最も適した組み合わせを正解とする必要がある．したがって，作成したデータセットが精度評価に影響を与えている可能性がある．

7. あとがき

本研究では，Git のファイル追跡改善のために AST を用いた差分検出手法を利用して類似度を算出する手法を提案した．提案手法では，GumTree を利用し出力された編集スクリプトから類似度を算出する．評価実験として，197 個の Java プロジェクトを対象に，ファイル名変更の検出数・追跡可能な変更履歴の長さ・追跡精度・実行時間について調査を行った．その結果，提案手法は既存手法よりも多くファイル名の変更および変更履歴を追跡することができ，より高い F 値が得られた．また，実行時間は増加することがわかった．今後の課題として以下が挙げられる．

類似度の算出手法の検討

提案手法では，AST のノード数と編集スクリプトの割合から類似度を算出する．さらに多くの変更を検出できる，または追跡の精度が高くなるような算出方法を検討することが大きな課題である．例えば，編集操作あるいは AST のノードの種類によって重み付けを行うことにより，ファイル追跡をさらに改善できる可能性がある．

類似度算出の高速化

実験結果から提案手法は既存手法よりも実行時間が長くなることが判明した．AST の差分検出に多くの時間がかかっているため，この部分の最適化が課題となる．類似度の算出手法の検討と合わせ，計算に必要な項目を AST から効率的に抽出することで高速化できると考えられる．

Git の履歴を利用する研究・ツールへの適用

Git の変更履歴を利用する研究およびツールに対し，提案手法を適用するとより良い結果が得られる可能性がある．例として開発履歴に基づいてバグの発生箇所を予測するツールに適用する場合，より高い精度で予測できることが期待される．

謝辞　本研究は，日本学術振興会科学研究費補助金基盤研究（B）（課題番号：20H04166）の助成を得て行われた．

文　献

[1] B. De Alwis and J. Sillito, “Why are software projects moving from centralized to decentralized version control systems?,” ICSE Workshop on Cooperative and Human Aspects on Software Engineering, pp.36–39, 2009.

[2] L. Hattori, M. D’Ambros, M. Lanza, and M. Lungu, “Software evolution comprehension: Replay to the rescue,” IEEE 19th International Conference on Program Comprehension, pp.161–170, 2011.

[3] H. Hata, O. Mizuno, and T. Kikuno, “Bug prediction based on fine-grained module histories,” Proceedings of the 34th international conference on software engineering, pp.200–210, 2012.

[4] S. Negara, M. Codoban, D. Dig, and R.E. Johnson, “Mining fine-grained code changes to detect unknown change patterns,” Proceedings of the 36th International Conference on Software Engineering, pp.803–813, 2014.

[5] E.W. Myers, “Ano (nd) difference algorithm and its variations,” Algorithmica, vol.1, no.1-4, pp.251–266, 1986.

[6] J.-R. Falleri, F. Morandat, X. Blanc, M. Martinez, and M. Monperrus, “Fine-grained and accurate source code differencing,” Proceedings of the 29th ACM/IEEE international conference on Automated software engineering, pp.313–324, 2014.

[7] S.S. Chawathe, A. Rajaraman, H. Garcia-Molina, and J. Widom, “Change detection in hierarchically structured information,” Acm Sigmod Record, vol.25, no.2, pp.493–504, 1996.

[8] H. Borges, A. Hora, and M.T. Valente, “Understanding the factors that impact the popularity of github repositories,” IEEE International Conference on Software Maintenance and Evolution, pp.334–344, 2016.

[9] V.I. Levenshtein, “Binary codes capable of correcting deletions, insertions, and reversals,” Soviet physics doklady, vol.10Soviet Union, pp.707–710 1966.

[10] M.A. Jaro, “Advances in record-linkage methodology as applied to matching the 1985 census of tampa, florida,” Journal of the American Statistical Association, vol.84, no.406, pp.414–420, 1989.