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

検出済みクローンペアの推移的関係を利用した未検出クローンペアの補完手法

石村　涼介†　　肥後　芳樹†

† 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: †{r-isimur,higo}@ist.osaka-u.ac.jp

あらまし　コードクローン検出では，メソッドの全組合せに対して類似度を計算すると，計算量が非常に大きくなる．そのため，NIL や NiCad などの検出器では，類似度計算の前にフィルタリングを行い，クローンである可能性が低いペアを候補から除外する．このようなフィルタリングは実行時間の削減に有効である一方，検出すべきコードクローンが誤って除外される可能性がある．本研究では，コードクローン関係の推移律に着目し，フィルタリングによって類似度計算の対象外となったペアを後処理として補完する手法を提案する．提案手法では，メソッドペア (*A*,*B*) および (*B*,*C*) がクローンとして検出されているにもかかわらず，(*A*,*C*) が検出されていない場合，(*A*,*C*) を推移律欠損ペアとして収集する．そして，推移律欠損ペアに対して類似度を計算し，条件を満たすペアを検出結果に追加する．Java の OSS を対象とした実験の結果，提案手法により，小さなオーバーヘッドで Recall を向上できた．具体的には，NIL のデフォルトのパラメータ設定において，NIL 単体で 0.90 以上であった Recall を，提案手法によって更に 0.01〜0.05 向上でき，追加の実行時間は最大 0.3 秒であった．これにより，コードクローン検出における実行時間と検出漏れのトレードオフを緩和できる可能性を示した．

キーワード　コードクローン，推移律，NIL

1. ま え が き

ソースコード中には，同一または類似した処理を行うコード片が複数存在する．このようなコード片はコードクローンと呼ばれる [1]．コードクローンは，コピーアンドペーストによる再利用などによって発生する [2]．コードクローンが存在する場合，あるコード片に不具合が含まれていると，そのクローンにも同様の不具合が含まれている可能性がある [3]．また，あるコード片を修正する際には，対応するクローンも同時に修正する必要がある場合がある．そのため，コードクローンの検出は，ソフトウェアの保守やリファクタリングにおいて有用である [4]．

これまでに，コードクローンを自動的に検出するための多くの手法が提案されている．代表的な検出器として，関数・メソッド単位のコード片を対象にクローンを検出できる NiCad [5] や NIL [6] が挙げられる．これらの検出器は，ソースコードをトークン列などに変換し，メソッド間の類似度を計算することでクローンを検出する．しかし，類似度計算では LCS(Longest Common Subsequence) の算出などを行うため，各ペアに対する計算コストは小さくない．特に，大規模なソフトウェアを対象とする場合，メソッドの組合せ数が増加するため，すべての組合せに対して類似度計算を行うと，計算量が非常に大きくなる．例えば，メソッド数を m とすると，比較対象となるペア数は最大で ${}_mC_2 = m(m-1)/2$ となる．したがって，実用的な時間でコードクローンを検出するためには，比較対象となるペアを効率的にフィルタリングする必要がある．

フィルタリングによる候補ペアの削減においては，実行時間と検出精度の間にトレードオフが存在する．フィルタリング条件を厳しくすると，類似度計算の対象ペア数が少なくなるため，実行時間を短縮できる．しかし，クローンであるペアまで除外される可能性が高くなり，クローンの見逃しが発生しやすくなる．一方で，フィルタリング条件を緩くすると，見逃しの可能性は小さくなるが，類似度計算の対象ペア数が増加し，実行時間が長くなる．このように，既存の検出器では，実行時間の短縮と，クローンの見逃しの抑制との両立が課題となる．

本研究では，この課題に対して，検出済みクローンペアを介して得られる見逃し候補に着目する．メソッドペア (*A*,*B*) および (*B*,*C*) がクローンとして検出されている場合，(*A*,*C*) もクローンである可能性がある．しかし，ペア (*A*,*C*) がフィルタリングによって除外されていた場合，(*A*,*C*) に対する類似度計算が行われず，クローンとして検出されない．本研究では，このようなペアを推移律欠損ペアと呼び，見逃し候補として収集する．そして，推移律欠損ペアに限定して類似度を計算し，条件を満たすペアをクローン候補として補完する手法を提案する．

実験では，Java で記述された OSS を対象として，提案手法の有効性を評価する．具体的には，NIL によって見逃されたコードクローンをどの程度補完できるか，また提案手法によってどの程度の実行時間のオーバーヘッドが発生するかを調査する．

2. 準　　備

2.1 コードクローン

コードクローンとは，ソースコード中に存在する互いに同一または類似したコード片である [1]．コードクローンは，既存のコード片をコピーアンドペーストして再利用した後，一部の修正などによって発生する [2]．コードクローンが存在する場合，あるコード片に含まれる不具合が，そのクローンにも同様に含まれている可能性がある．また，一方のコード片を修正した際に，対応するクローンに対する修正が漏れると，ソフトウェア中に一貫性のない変更が生じる可能性がある．そのため，コードクローンの検出は，ソフトウェアの保守，理解，リファクタリング，および不具合修正の支援において重要である [4], [7]．

これまでに，コードクローンを自動的に検出するための多くの手法が提案されている．代表的な検出手法には，テキストベース，トークンベース，構文木ベース，プログラム依存グラフベース，メトリクスベース，およびこれらを組み合わせたハイブリッドな手法がある [4]．テキストベースの手法は，ソースコードを文字列または行の系列として扱い，コード片間の文字列類似度に基づいてクローンを検出する．トークンベースの手法は，ソースコードを字句解析によってトークン列に変換し，トークン列間の類似性を比較する．例えば，CCFinder [7] はトークンベースの代表的な検出器である．

近年では，機械学習や深層学習を用いたコードクローン検出も提案されている．例えば，ASTNN [8] は抽象構文木に基づくコード表現を学習し，コード分類やコードクローン検出に利用している．また，CodeBERT [9] のような事前学習済みモデルを用いて，コード片の意味的な類似性を捉えようとする研究も行われている．

コードクローン検出器は，コード片のペアに対して類似度を計算し，その値がしきい値以上である場合にクローンとして判定することが多い．このとき，あるコード片のペア (A,B) がクローンであり，さらに (B,C) がクローンである場合に，(A,C) もクローンであるかという推移的な性質を考えることができる．

しかし，コードクローン関係が一般に推移律を満たすかどうかは自明ではない．例えば，(A,B) および (B,C) の類似度がしきい値以上である場合でも，(A,C) の類似度はしきい値を下回る可能性がある．

一方で，検出結果の中には，推移的な関係からクローンである可能性が示唆されるにもかかわらず，検出されていないペアが存在する可能性がある．そこで本研究では，既存の検出器によって得られたクローンペア集合から推移律欠損ペア (A,C) を収集し，実際にクローンとして補完可能であるかを調査する．

2.2 NIL

NIL は，大規模なソースコード集合から large-variance clone を検出することを目的としたトークンベースのコードクローン検出器である [6]．large-variance clone とは，コピーアンドペーストされたコード片に対して，多数の文の追加や削除が複数箇所に分散して行われたクローンである．このようなクローンでは，コード片全体としては大きな差異が存在する一方で，変更されていない部分ではトークンの順序が保存されていることが多い．NIL はこの性質に着目し，N-gram，転置インデックス，および LCS を組み合わせることで，大規模なソースコードに対して効率的にクローンを検出する．

NIL では，クローン検出の単位としてメソッドに相当するコードブロックを用いる．具体的には，入力された Java ソースコードから，波括弧で囲まれたコードブロックを抽出し，トークン列に変換する．NIL は言語仕様に基づく厳密な字句解析は行わず，演算子や波括弧などの記号，空白，改行を区切りとしてコードブロックを単純に分割する．そのため，高速にトークン列を生成できる．

2.2.1 前処理 (フィルタリング)

前処理では，本処理の計算量を削減するために，共通する N-gram の割合が低いペアを候補から除外する．N-gram の長さはパラメータ n で指定し，デフォルト設定では $n = 5$ である．また，フィルタリングしきい値はパラメータ θ で指定し，デフォルト設定では $\theta = 0.1$ である．

具体的には，NIL ではメソッド A と B に対するフィルタリング用の類似度を次式により計算する．

$$filtration\_sim(A,B) = \frac{common\_ngrams(A,B)}{\min(ngrams(A), ngrams(B))}$$

ここで，$common\_ngrams(A,B)$ は A と B が共有する N-gram 数を表し，$ngrams(A)$ および $ngrams(B)$ はそれぞれのコード片から生成される N-gram 数を表す．この類似度がフィルタリングしきい値 θ を下回る候補は，クローンである可能性が低いとして除外される．

2.2.2 本処理 (類似度の計算)

本処理では，フィルタリングをパスしたペアに対して，トークン列間の LCS に基づく類似度を計算する．large-variance clone では，文の追加や削除によってコード片の長さが大きく異なる場合でも，元のコード片に由来する多くのトークンの順序が保存されていることがある．そのため，NIL では 2 つのトークン列の LCS 長を用いて，類似度を次のように計算する．

$$verification\_sim(c_1,c_2) = \frac{lcs(c_1,c_2)}{\min(|c_1|,|c_2|)}$$

ここで，$lcs(c_1,c_2)$ は c_1 と c_2 のトークン列間の LCS 長を表し，$|c_1|$ および $|c_2|$ はそれぞれのトークン列の長さを表す．この類似度がしきい値 δ 以上である場合，NIL はそのペアをクローンペアとして出力する．デフォルト設定では $\delta = 0.7$ である．NIL では，長さが大きく異なる large-variance clone を検出するため，分母に 2 つのトークン列長の最小値を用いている．

3. 提 案 手 法

提案手法を図 1 に示す．本手法は，既存のコードクローン検出器の検出結果を利用する後処理である．本手法の入力はクローンペアの集合であり，出力もクローンペアの集合である．図 1 の例では，NIL によるクローン検出の見逃しを補完する．

本手法は，以下の手順で実行する．

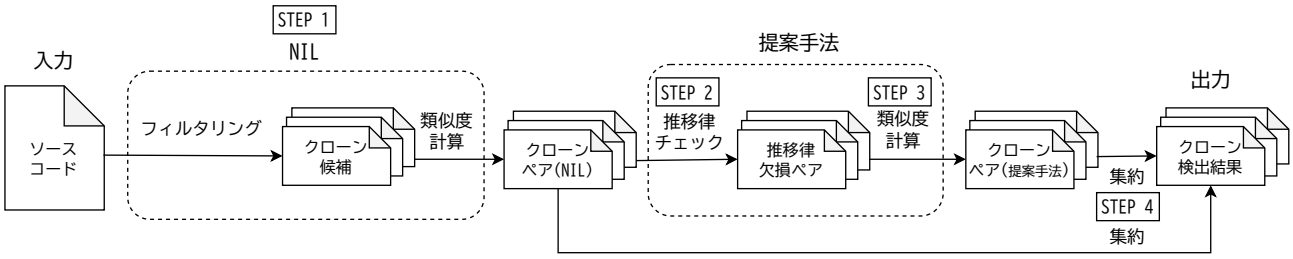

図 1　提 案 手 法

STEP 1：NIL によるクローンペアの検出

ソースコードを NIL に入力し，コードクローンを検出する．NIL では，類似度を計算する前に，フィルタリングによってクローン候補を削減する．その後，クローン候補に対して類似度を計算し，類似度がしきい値以上であるペアをクローンペアとして出力する．

STEP 2：推移律欠損ペアの収集

NIL が検出したクローンペアを用いて，推移律欠損ペアを収集する．具体的には，メソッドペア (A,B) および (B,C) がクローンとして検出されているにもかかわらず，(A,C) が検出されていない場合，ペア (A,C) を推移律欠損ペアとみなす．

STEP 3：推移律欠損ペアに対する類似度計算

収集した推移律欠損ペアに対して類似度を計算する．計算方法は，NIL で用いられる類似度計算と同様である．類似度がしきい値以上である推移律欠損ペアをクローンペアとして出力する．

STEP 4：検出結果の集約

NIL の出力と提案手法の出力の和集合をとり，最終的なクローン検出結果とする．ここで，提案手法は NIL で検出されていない推移律欠損ペアのみを出力するため，NIL の出力との共通部分は原理的に存在しない．

なお，本手法は NIL の検出結果を入力とする後処理であるから，NIL 本体のフィルタリング処理や類似度計算処理は変更しない．また，推移律欠損ペアに限定して類似度を計算するため，追加の計算量を抑えながら，フィルタリングによって除外されたクローンを補完できると考えられる．

本手法は，フィルタリング条件の厳しさに伴う実行時間と見逃しのトレードオフを緩和することを目的とする．本手法によって，フィルタリング条件をある程度厳しく設定して実行時間を短縮しつつ，見逃しを補うことが期待できる．

4. 実 験 設 計

本実験では，提案手法が NIL によるコードクローン検出結果の見逃しをどの程度補完できるか調査する．また，補完による検出精度の向上と，追加計算コストとの関係を評価する．

4.1 RQ(Research Question)

本実験では，以下の 2 つの RQ を設定する．

RQ1：提案手法は NIL によって見逃されたコードクローンをどの程度補完できるか．

RQ2：提案手法の適用によってどの程度のオーバーヘッドが発生するか．

表 1　実験対象となるソフトウェア．クローンペア数および検出時間は，NIL においてフィルタリングを一切実行しない場合の値．

ソフトウェア	LOC	全ペア数	クローンペア数 (フィルタなし)	検出時間 [s] (フィルタなし)
Maven 3.9.16	55,978	1,175,811	7,217	2.5
Ant 1.10.17	114,196	3,918,600	1,435	5.1
Tomcat 10.1.55	266,124	21,968,506	17,775	17.8
Lucene 10.4.0	491,545	71,862,066	39,067	43.5

4.2 評 価 指 標

RQ1 では Recall を評価する．まず，NIL においてフィルタリングを一切実行しない場合に得られる検出結果を基準集合 G とする．次に，各パラメータ設定において，NIL 単体で得られるクローンペア集合を D_{NIL} とし，提案手法によって追加されるクローンペア集合を D_{ADD} とする．このとき，提案手法適用後の最終的なクローンペア集合 D を

$$D = D_{\mathrm{NIL}} \cup D_{\mathrm{ADD}}$$

で計算し，Recall を次式により計算する．

$$Recall = \frac{|D \cap G|}{|G|}$$

Recall が高いほど，フィルタリングによる検出漏れが少ないことを意味する．

RQ2 では実行時間を評価する．具体的には，各パラメータ設定における NIL 単体の実行時間と，提案手法によるオーバーヘッドを計測する．

4.3 対象ソフトウェア

表 1 に実験対象となるソフトウェアを示す．LOC はテストコードを除去した後のソースコード行数であり，空白行およびコメントのみからなる行は含まない．全ペア数はメソッド数を m としたときの ${}_mC_2 = m(m-1)/2$ により算出した．クローンペア数および検出時間は，NIL においてフィルタリングを一切実行しない場合の値である．

本実験では，各ソフトウェアを個別に解析対象とし，同一ソフトウェア内のクローンのみを検出した．したがって，異なるソフトウェアに属するメソッド間のクローン検出は行っていない．

4.4 実 験 手 順

NIL では N-gram 長を表すパラメータ n と，フィルタリング

しきい値 θ を設定できる．ここで，パラメータ n が小さいほどフィルタリングの通過条件が緩くなることを意味する．また，パラメータ θ が小さいほどフィルタリングの通過条件が緩くなることを意味する．

本実験では，$n = 1, 3, 5, 7, 9$ および $\theta = 0, 0.1, 0.2, \ldots, 1.0$ の組合せ (n,θ) について実験を行った．具体的には，各パラメータ設定に対して以下の操作を実施した．

（1） NIL を実行してクローンペアを取得する．
（2） NIL の出力結果から推移律欠損ペアを収集する．
（3） 推移律欠損ペアに対して類似度を計算する．
（4） 類似度がしきい値以上のペアを検出結果へ追加する．

なお，NIL では LCS 類似度のしきい値 δ も変更可能であるが，本実験ではデフォルトの δ=0.7 に固定した．

5. 実験結果

5.1 提案手法の補完性能 (RQ1)

表 2〜表 5 に，各ソフトウェアに対する Recall の結果を示す．各セルの 1 行目は提案手法適用後の Recall，2 行目は提案手法による Recall の増分である．

実験結果より，フィルタリング条件の厳しさに応じて，以下の 3 つの傾向が見られた．

傾向 1：フィルタリング条件が緩い場合は Recall が高いが，増分は小さい (水色)

水色で示した部分は，フィルタリング条件が緩く，NIL 単体でも 0.80 以上の Recall を達成した設定を表す．これらの設定では，提案手法による Recall の増分は小さい傾向があった．例えば，NIL のデフォルト設定である $(n,\theta) = (5, 0.1)$ では，いずれのソフトウェアにおいても提案手法適用後の Recall は 0.92 以上であり，提案手法による増分は 0.01〜0.05 であった．これは，元々フィルタリングによる検出漏れが少ないため，提案手法によって補完可能なクローンペアの数も限定的であったためと考えられる．このような設定では，提案手法による改善幅は大きくないものの，元々高かった Recall をさらに向上できている．

傾向 2：フィルタリング条件が厳しい場合は Recall が低く，増分も小さい (灰色)

灰色で示した部分は，フィルタリング条件が厳しく，NIL 単体での Recall が 0.10 以下の設定を表す．これらの設定では，提案手法による Recall の増分は小さい傾向があった．例えば，$(n,\theta) = (9, 1.0)$ では，いずれのソフトウェアでも Recall がほぼ 0 となった．これは，フィルタリングによって NIL が検出するクローンペアが大きく減少し，推移律欠損ペアを収集するための手がかりが不足したためと考えられる．提案手法は既存の検出結果をもとに補完候補を収集する後処理であるため，NIL の検出結果が極端に少ない場合には，補完可能なペアも少なくなる．

傾向 3：傾向 1 と傾向 2 の境界では Recall の増分が大きい (オレンジ)

オレンジで示した部分は，傾向 1 と傾向 2 の境界に位置する設定を表す．この部分では，NIL 単体の Recall は低い一方で，提案手法による Recall の増分が大きい設定が確認された．例えば，表 2 における $(n,\theta) = (5, 0.3)$ では，提案手法によって Recall が 0.19 から 0.88 まで向上した．また，表 4 における $(n,\theta) = (7, 0.1)$ では，提案手法によって Recall が 0.39 から 0.92 まで向上した．このような設定では，フィルタリングによって多くのクローンペアが除外されているものの，検出済みのクローンペアから推移律欠損ペアを収集することで，除外されたペアの多くを補完できたと考えられる．

以上の結果は，提案手法が特にフィルタリングによる検出漏れが発生しやすい境界的な設定において有効であることを示している．

表 2　RQ1 の実験結果 (Maven)．各セルの 1 行目は提案手法適用後の Recall，2 行目は提案手法による Recall の増分を表す．傾向 1(NIL 単体の Recall が 0.80 以上) のセルは水色，傾向 2(NIL 単体の Recall が 0.10 以下) のセルは灰色，傾向 3(それ以外) のセルはオレンジで示す．

$n\backslash\theta$	0	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1.0
1	1.00 +0.00	1.00 +0.00	1.00 +0.00	1.00 +0.00	1.00 +0.00	1.00 +0.00	1.00 +0.01	0.95 +0.01	0.31 +0.16	0.06 +0.01	0.01 +0.00
3	1.00 +0.00	1.00 +0.00	1.00 +0.00	0.99 +0.04	0.90 +0.01	0.29 +0.16	0.09 +0.01	0.06 +0.01	0.05 +0.01	0.01 +0.00	0.00 +0.00
5	1.00 +0.00	0.99 +0.02	0.90 +0.01	0.88 +0.69	0.11 +0.01	0.08 +0.01	0.06 +0.01	0.04 +0.00	0.04 +0.00	0.01 +0.00	0.00 +0.00
7	0.99 +0.00	0.90 +0.01	0.30 +0.15	0.11 +0.01	0.09 +0.01	0.06 +0.01	0.05 +0.01	0.04 +0.00	0.01 +0.00	0.00 +0.00	0.00 +0.00
9	0.90 +0.01	0.88 +0.69	0.13 +0.03	0.09 +0.01	0.06 +0.01	0.05 +0.01	0.04 +0.01	0.03 +0.00	0.01 +0.00	0.00 +0.00	0.00 +0.00

表 3　RQ1 の実験結果 (Ant)

$n\backslash\theta$	0	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1.0
1	1.00 +0.00	1.00 +0.00	1.00 +0.00	1.00 +0.00	1.00 +0.00	1.00 +0.00	0.98 +0.01	0.92 +0.04	0.76 +0.07	0.51 +0.17	0.06 +0.00
3	1.00 +0.00	0.99 +0.00	0.96 +0.00	0.92 +0.03	0.82 +0.03	0.71 +0.08	0.57 +0.14	0.37 +0.11	0.14 +0.01	0.07 +0.00	0.03 +0.00
5	0.96 +0.00	0.92 +0.02	0.83 +0.04	0.73 +0.09	0.62 +0.12	0.46 +0.09	0.31 +0.09	0.15 +0.01	0.07 +0.00	0.04 +0.00	0.03 +0.00
7	0.88 +0.01	0.80 +0.04	0.72 +0.08	0.60 +0.11	0.47 +0.11	0.33 +0.06	0.24 +0.08	0.10 +0.01	0.05 +0.00	0.04 +0.00	0.03 +0.00
9	0.77 +0.04	0.71 +0.07	0.60 +0.10	0.45 +0.08	0.33 +0.05	0.26 +0.08	0.15 +0.04	0.06 +0.00	0.04 +0.00	0.03 +0.00	0.03 +0.00

表 4　RQ1 の実験結果 (Tomcat)

$n\backslash\theta$	0	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1.0
1	1.00 +0.00	1.00 +0.00	1.00 +0.00	1.00 +0.00	1.00 +0.00	1.00 +0.00	1.00 +0.01	0.98 +0.01	0.92 +0.07	0.52 +0.36	0.01 +0.00
3	1.00 +0.00	1.00 +0.00	1.00 +0.00	0.98 +0.05	0.91 +0.46	0.43 +0.21	0.14 +0.04	0.08 +0.02	0.03 +0.00	0.01 +0.00	0.00 +0.00
5	1.00 +0.00	0.98 +0.01	0.93 +0.11	0.24 +0.04	0.17 +0.04	0.11 +0.03	0.06 +0.01	0.03 +0.00	0.01 +0.00	0.01 +0.00	0.00 +0.00
7	0.95 +0.01	0.92 +0.53	0.24 +0.06	0.16 +0.03	0.10 +0.02	0.06 +0.01	0.03 +0.01	0.02 +0.00	0.01 +0.00	0.00 +0.00	0.00 +0.00
9	0.28 +0.04	0.23 +0.05	0.16 +0.03	0.10 +0.02	0.06 +0.01	0.04 +0.00	0.03 +0.00	0.01 +0.00	0.01 +0.00	0.00 +0.00	0.00 +0.00

表 5　RQ1 の実験結果 (Lucene)

$n\backslash\theta$	0	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1.0
1	1.00 +0.00	1.00 +0.00	1.00 +0.00	1.00 +0.00	1.00 +0.00	1.00 +0.00	1.00 +0.01	0.95 +0.05	0.85 +0.17	0.49 +0.21	0.11 +0.03
3	1.00 +0.00	1.00 +0.00	0.99 +0.02	0.96 +0.05	0.89 +0.11	0.74 +0.18	0.54 +0.16	0.31 +0.09	0.18 +0.04	0.09 +0.01	0.04 +0.00
5	0.99 +0.02	0.95 +0.05	0.88 +0.09	0.75 +0.12	0.59 +0.14	0.45 +0.12	0.32 +0.09	0.16 +0.03	0.10 +0.01	0.06 +0.01	0.04 +0.00
7	0.93 +0.04	0.87 +0.09	0.71 +0.12	0.54 +0.11	0.43 +0.11	0.32 +0.08	0.24 +0.10	0.11 +0.02	0.08 +0.01	0.05 +0.00	0.04 +0.00
9	0.84 +0.07	0.71 +0.11	0.57 +0.12	0.43 +0.09	0.34 +0.07	0.26 +0.06	0.20 +0.09	0.09 +0.01	0.07 +0.01	0.04 +0.00	0.04 +0.00

5.2 提案手法のオーバーヘッド (RQ2)

表 6〜表 9 に実行時間の結果を示す．各セルの 1 行目は提案手法適用後の実行時間，2 行目は提案手法によるオーバーヘッドである．

実験結果より，提案手法によるオーバーヘッドは，NIL 単体の実行時間に対して小さいことが確認された．Maven および Ant では，すべての設定においてオーバーヘッドは 0.2 秒以下であった．また，Tomcat および Lucene のように対象ソフトウェアの規模が大きい場合でも，オーバーヘッドは限定的であり，最大でも Lucene の $(n, \theta) = (1, 0.6)$ における 3.7 秒であった．したがって，提案手法は，推移律欠損ペアに対して追加で類似度計算を行うものの，実行時間に与える影響は比較的小さいといえる．

また，フィルタリング条件の厳しさに応じて，以下の 2 つの傾向が見られた．

傾向 4：フィルタリング条件を緩くすると実行時間は長くなる

例えば, 表 9 における $(n, \theta) = (1, 0)$ では, 提案手法適用後の実行時間は 47.0 秒であり，デフォルト設定である $(n, \theta) = (5, 0.1)$ の 4.4 秒と比較して 10 倍以上の時間を要している．これは，フィルタリング条件が緩い場合には類似度計算の対象となるペア数が増加するためであると考えられる．このような設定では Recall は高いものの，実行時間の観点からは必ずしも効率的ではない．

傾向 5：フィルタリング条件を厳しくすると実行時間は短くなる

例えば，表 9 では，デフォルト設定である $(n, \theta) = (5, 0.1)$ における実行時間が 4.4 秒であるのに対し，$(n, \theta) = (5, 0.3)$ では 4.1 秒，$(n, \theta) = (7, 0.1)$ では 4.2 秒，$(n, \theta) = (9, 0.1)$ では 4.0 秒であった．また，表 8 では，$(n, \theta) = (5, 0.1)$ の実行時間が 3.2 秒であるのに対し，$(n, \theta) = (7, 0.1)$ では 2.8 秒であった．このように，フィルタリング条件をデフォルト設定より厳しくすることで，実行時間を同程度またはそれ以下に抑えられる場合がある．

6. 考　　察

6.1 提案手法が有効に機能する条件

RQ1 の結果より，フィルタリング条件が中程度に厳しい設定 (傾向 3) において，提案手法の効果が大きいことが分かった．例えば，表 4 における $(n, \theta) = (7, 0.1)$ では Recall が 0.39 から 0.92 へ向上しており，デフォルト設定 $(n, \theta) = (5, 0.1)$ における 0.98 に近い Recall を維持できた．これらの結果は，提案手法が，フィルタリングによる検出漏れを補完する後処理として有効であることを示している．

一方で，フィルタリング条件が緩い場合 (傾向 1)，NIL 単体でも高い Recall を達成しているため，提案手法によって補完可能なクローンペアは少ない．また，条件を厳しくした場合 (傾向 2) には，NIL によって検出されるクローンペア自体が少なくなり，推移律欠損ペアを収集するための手がかりが不足する．そのため，提案手法を適用しても Recall はほとんど向上しなかった．

したがって，提案手法は任意のフィルタリング条件で有効ではなく，検出結果に一定数のクローンペアが含まれている場合に効果を発揮するといえる．

表 6　RQ2 の実験結果 (Maven)．各セルの 1 行目は提案手法適用後の実行時間 [s]，2 行目は提案手法によるオーバーヘッド [s] を表す．セルの色は，実行時間が表中の最大値に近いほど濃くなる．

$n\backslash\theta$	0	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1.0
1	2.7 +0.1	2.6 +0.1	2.4 +0.1	2.4 +0.1	2.1 +0.1	2.2 +0.1	2.2 +0.1	2.1 +0.1	1.8 +0.0	1.9 +0.0	1.9 +0.0
3	1.9 +0.1	2.0 +0.2	1.9 +0.2	1.8 +0.0	1.7 +0.0	1.5 +0.0	1.7 +0.0	1.7 +0.0	1.6 +0.0	1.5 +0.0	1.6 +0.0
5	2.1 +0.1	2.1 +0.2	1.8 +0.0	1.6 +0.0	1.7 +0.0	1.6 +0.0	1.7 +0.0	1.6 +0.0	1.7 +0.0	1.6 +0.0	1.6 +0.0
7	2.1 +0.1	1.9 +0.0	1.6 +0.0	1.7 +0.0	1.6 +0.0	1.7 +0.0	1.7 +0.0	1.6 +0.0	1.5 +0.0	1.8 +0.0	2.0 +0.0
9	1.8 +0.1	1.6 +0.0	1.7 +0.0	1.5 +0.0	1.7 +0.0	1.7 +0.0	1.6 +0.0	1.7 +0.0	1.6 +0.0	1.6 +0.0	1.6 +0.0

表 7　RQ2 の実験結果 (Ant)

$n\backslash\theta$	0	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1.0
1	4.9 +0.0	4.6 +0.0	3.5 +0.0	2.8 +0.0	2.5 +0.0	2.3 +0.0	2.3 +0.0	2.2 +0.0	2.2 +0.0	2.3 +0.0	2.3 +0.0
3	2.6 +0.0	2.1 +0.0	2.0 +0.0	1.9 +0.0	1.9 +0.0	1.9 +0.0	1.9 +0.0	2.0 +0.0	2.0 +0.0	2.1 +0.0	2.0 +0.0
5	2.3 +0.0	2.0 +0.0	2.0 +0.0	1.9 +0.0	1.9 +0.0	1.9 +0.0	2.1 +0.0	1.9 +0.0	1.8 +0.0	1.9 +0.0	1.9 +0.0
7	2.2 +0.0	2.1 +0.0	1.9 +0.0	2.0 +0.0	2.0 +0.0	2.0 +0.0	1.8 +0.0	1.9 +0.0	1.9 +0.0	2.0 +0.0	1.9 +0.0
9	2.0 +0.0	1.9 +0.0	1.9 +0.0	1.9 +0.0	2.0 +0.0	1.8 +0.0	1.9 +0.0	1.9 +0.0	1.8 +0.0	1.9 +0.0	1.8 +0.0

表 8　RQ2 の実験結果 (Tomcat)

$n\backslash\theta$	0	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1.0
1	16.6 +0.7	15.6 +0.5	11.5 +0.5	7.0 +0.4	5.1 +0.5	4.2 +0.5	4.0 +0.6	4.1 +0.6	4.0 +0.5	3.2 +0.0	3.3 +0.0
3	5.6 +0.3	3.5 +0.2	3.2 +0.3	3.1 +0.2	2.8 +0.0	2.8 +0.0	3.0 +0.0	2.9 +0.0	2.8 +0.0	2.7 +0.0	2.7 +0.0
5	3.6 +0.4	3.2 +0.3	2.9 +0.2	2.6 +0.0	2.8 +0.0	2.7 +0.0	2.6 +0.0	2.6 +0.0	2.6 +0.0	2.5 +0.0	2.6 +0.0
7	3.2 +0.3	2.8 +0.0	2.6 +0.0	2.9 +0.0	2.7 +0.0	2.6 +0.0	2.7 +0.0	2.5 +0.0	2.6 +0.0	2.8 +0.0	2.6 +0.0
9	2.7 +0.0	2.8 +0.0	2.7 +0.0	2.7 +0.0	2.7 +0.0	2.7 +0.0	2.7 +0.0	2.7 +0.0	2.6 +0.0	2.5 +0.0	2.5 +0.0

表 9　RQ2 の実験結果 (Lucene)

$n\backslash\theta$	0	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1.0
1	47.0 +2.3	41.2 +2.2	28.1 +2.9	16.0 +2.5	11.1 +2.6	9.6 +3.2	9.4 +3.7	9.3 +3.6	5.3 +0.3	5.3 +0.0	5.2 +0.0
3	12.4 +0.3	4.8 +0.3	4.8 +0.3	4.1 +0.3	4.0 +0.2	4.1 +0.1	4.2 +0.0	4.0 +0.0	4.0 +0.0	3.6 +0.0	4.1 +0.0
5	5.4 +0.3	4.4 +0.2	4.5 +0.2	4.1 +0.1	3.9 +0.1	3.6 +0.0	4.2 +0.0	3.5 +0.0	3.9 +0.0	3.8 +0.0	3.8 +0.0
7	4.5 +0.2	4.2 +0.1	4.0 +0.1	3.7 +0.1	3.6 +0.0	3.9 +0.0	4.0 +0.0	3.8 +0.0	4.0 +0.0	3.3 +0.0	4.2 +0.0
9	4.3 +0.2	4.0 +0.1	3.8 +0.1	4.1 +0.0	3.9 +0.0	3.7 +0.0	3.8 +0.0	3.7 +0.0	3.8 +0.0	3.8 +0.0	4.0 +0.0

6.2 実行時間と Recall のトレードオフの緩和

RQ2 の結果より，提案手法によるオーバーヘッドは全体として小さいことが確認された．提案手法では，全てのコード片ペアに対して類似度を計算するのではなく，推移律欠損ペアに限定して LCS 類似度を計算する．そのため，追加の計算量を抑えつつ，フィルタリングによって除外されたクローンペアの一部を補完できたと考えられる．

通常，フィルタリング条件を厳しくすると，実行時間は短縮される (傾向 5) が，その一方で検出漏れが増加し，Recall は

低下する．これに対して提案手法を適用することで，厳しいフィルタリング条件を用いた場合でも，Recall の低下を一定程度抑えられる場合があった．

例えば，表 4 における $(n, \theta) = (7, 0.1)$ では，Recall 0.92 を達成しつつ，実行時間は 2.8 秒であり，デフォルト設定 $(n, \theta) = (5, 0.1)$ の 3.2 秒より短かった．

以上より，提案手法は，高い Recall のみを優先する場合だけでなく，実行時間を抑えながら十分な Recall を確保したい場合にも有用である．特に，大規模なソフトウェアを対象とする場合，フィルタリングを強めたうえで提案手法を適用することにより，コードクローン検出における実行時間と Recall のトレードオフを緩和できる可能性がある．

6.3 提案手法の限界

第一に，提案手法は初期の検出結果に依存する．NIL が検出したクローンペアをもとに推移律欠損ペアを収集するため，NIL 単体でほとんどクローンペアが検出されない設定では有効に機能しづらい．

第二に，提案手法で補完できるのは，検出済みクローンペアを介して推移律欠損ペアとして表現できるペアに限られる．そのため，フィルタリングによって除外されたクローンペアであっても，それを結び付ける中間的なコード片が検出結果中に存在しない場合には，推移律欠損ペアとして収集できない．

以上より，提案手法は小さな追加コストで Recall を向上させる有効な後処理である一方，初期検出結果に依存するという性質を持つ．そのため，提案手法を適用する際には，フィルタリング条件を過度に厳しくするのではなく，検出済みクローンペアが十分に残る範囲で設定することが重要である．

7. 妥当性への脅威

7.1 基準集合に関する妥当性への脅威

本実験では，NIL においてフィルタリングを一切実行しない場合の検出結果を基準集合とし，Recall を計算した．この基準集合を用いることで，フィルタリングによって除外されたクローンペアを提案手法がどの程度補完できるかを評価できる．しかし，この基準集合は NIL の類似度計算に基づく検出結果であり，実際の正解集合そのものではない．そのため，基準集合に含まれる全てのペアが真のクローンであるとは限らず，また，基準集合に含まれない真のクローンが存在する可能性もある．

7.2 パラメータ設定に関する妥当性への脅威

本実験では，N-gram 長 n およびフィルタリングしきい値 θ を変化させて評価を行った．一方で，NIL における LCS 類似度のしきい値 δ はデフォルト値である 0.7 に固定した．そのため，δ を変更した場合にも同様の傾向が得られるかは明らかではない．

7.3 適用対象に関する妥当性への脅威

本研究では，NIL を対象として提案手法を評価したが，NIL 以外の検出器に対する評価は行っていない．NiCad のようにコード片間の類似度に基づいてクローンを検出する手法には適用しやすいと考えられる一方で，コード片をベクトル化して分類する手法や，学習済みモデルによってクローン判定を行う手法に対しては，そのまま適用できるとは限らない．

7.4 実行時間計測に関する妥当性への脅威

本実験では，提案手法のオーバーヘッドを実行時間によって評価した．しかし，実行時間は実験環境の性能，OS の状態，他プロセスの影響などによって変動する可能性がある．そのため，得られた実行時間は，本実験環境における値であり，他の環境でも同じ値が得られるとは限らない．

8. あとがき

本研究では，コードクローン検出におけるフィルタリングによる見逃しを補完するため，推移律欠損ペアに着目した後処理を提案した．Java の OSS を対象とした実験の結果，提案手法により小さなオーバーヘッドで Recall を向上できる場合があることを確認した．特に，フィルタリング条件が中程度に厳しく，検出済みクローンペアが一定数残っている設定において有効であった．今後は，人手評価によって補完されたクローンペアの有用性を確認するとともに，NIL 以外の検出器や他のプログラミング言語への適用可能性を調査する．

謝辞　本研究は JSPS 科研費 24H00692, 25K03102, 26K02889, 26H02500 の助成を受けたものです．

文　献

[1] 肥後芳樹，吉田則裕，“コードクローンを対象としたリファクタリング，” コンピュータソフトウェア，vol.28，no.4，pp.43–56，2011．

[2] 神谷年洋，肥後芳樹，吉田則裕，“コードクローン検出技術の展開，” コンピュータソフトウェア，vol.28，no.3，pp.29–42，2011．

[3] K. Inoue, Y. Higo, N. Yoshida, E. Choi, S. Kusumoto, K. Kim, W. Park, and E. Lee, “Experience of finding inconsistently-changed bugs in code clones of mobile software,” Proceedings of the 6th International Workshop on Software Clones, pp.94–95, IWSC 2012, Zurich, Switzerland, 2012.

[4] C.K. Roy, J.R. Cordy, and R. Koschke, “Comparison and evaluation of code clone detection techniques and tools: A qualitative approach,” Science of Computer Programming, vol.74, no.7, pp.470–495, 2009.

[5] C.K. Roy and J.R. Cordy, “NICAD: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization,” Proceedings of the 16th IEEE International Conference on Program Comprehension, pp.172–181, ICPC 2008, 2008.

[6] T. Nakagawa, Y. Higo, and S. Kusumoto, “NIL: Large-scale detection of large-variance clones,” Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, pp.830–841, ESEC/FSE 2021, 2021.

[7] T. Kamiya, S. Kusumoto, and K. Inoue, “CCFinder: A multilinguistic token-based code clone detection system for large scale source code,” IEEE Transactions on Software Engineering, vol.28, no.7, pp.654–670, 2002.

[8] J. Zhang, X. Wang, H. Zhang, H. Sun, K. Wang, and X. Liu, “A novel neural source code representation based on abstract syntax tree,” Proceedings of the 41st International Conference on Software Engineering, pp.783–794, ICSE 2019, 2019.

[9] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, and M. Zhou, “CodeBERT: A pre-trained model for programming and natural languages,” Findings of the Association for Computational Linguistics: EMNLP 2020, pp.1536–1547, Association for Computational Linguistics, 2020.