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

信学技報
TECHNICAL REPORT OF IEICE.

複数プロジェクトから高速にコードクローンを検出するキュレーションの提案

土居　真之† 　肥後　芳樹† 　楠本　真二†

† 大阪大学大学院情報科学研究科，吹田市
E-mail: †{m-doi,higo,kusumoto}@ist.osaka-u.ac.jp

あらまし　コードクローン (互いに類似するコード片) はソフトウェア保守に悪影響を与える要因の 1 つとして挙げられている． コードクローンの検出対象であるソースコードが大規模な場合，検出に必要な時間が増加してしまい検出できなくなることがある． そこで本研究では複数プロジェクトから高速にコードクローンを取得するキュレーションという手法を提案する．提案手法ではまず検出対象である個々のプロジェクトにおいて，プロジェクト内に存在するクローンを検出する．この検出結果から調査の必要がないコードクローンを取り除き，残りのコードクローンから類似するコードクローンをまとめる．これによりプロジェクト間に存在するコードクローンを高速に検出できる．実験では提案手法と CCFinder のコードクローンの検出時間を比較した．その結果，全ての実験対象で提案手法は CCFinder よりも速くコードクローンを検出し，最大で 49 倍速く検出できた．

キーワード　コードクローン検出，ソフトウェア保守，キュレーション

1. ま え が き

コードクローン (以降，クローンと呼ぶ) とは，ソースコード中の同一あるいは類似するコード片のことを指す．クローンはソフトウェアの保守に悪影響を与え [1], [2]，バグが増える要因になるとされている [3], [4]．そのためクローンの検出技術はさかんに研究されており，クローンをソースコード中から自動的に検出するツールが多数開発，公開されてきた [5].

ソースコードの量は増えてきており，大規模なソースコードから効率よくクローンを検出する技術が必要である． 例えば大規模クローン検出により類似したモバイルアプリケーションの検出 [6] やコンポーネントの起源発見 [7] やコード検索 [8] が可能となってきている．しかし大規模なクローン検出にはいくつかの課題がある．例えばクローン検出は対象となるソフトウェア群から特定したクローンをすべてを出力するため，多くのプロジェクトからクローン検出すると出力されるクローンの量は膨大になる．さらに検出対象のプロジェクト群に新たに 1 つプロジェクトを追加したり，検出対象であるプロジェクトの一部が更新されたりした場合，既存の検出器では全プロジェクトからクローンを再検出する必要がある．そのため再検出にも長い時間を要する．これらの問題を解決するためスケーラブルにクローンを検出する手法が必要である．

クローンの検出結果には調査の必要がないクローンが多く含まれていることがある．調査の必要がないクローンとは，ソフトウェア開発・保守を行うにあたってクローン情報を扱う場合にその存在が役に立たないクローンである．例えばリファクタリングする価値がない，あるいはリファクタリングするのが困難なクローンが挙げられる [9]．リファクタリングが困難なクローンの例として言語固有のクローンが挙げられる．特定のプログラミング言語が使用されるとき，開発者は言語の仕様上，類似するコード片を繰り返し書かなければならない場合がある．言語依存のコードクローンはそのようなコード片で構成されているため，言語依存のクローンは調査の必要がなく無視できるとされている [10]．そのため検出結果から調査の必要がないクローンを取り除く手法が多数提案されてきた．例えば，コード片の高凝集および低カップリングから判定する手法 [11] や，コード片に含まれる繰り返し構造の割合で判定する手法 [10]，さらにはこれらのメトリクスを機械学習することで判定する手法 [12] が挙げられる．これらの手法を用いることで，検出結果から調査の必要がないクローンを取り除くことができる．そこで著者らはプロジェクトごとのクローン検出結果から調査の必要がないクローンを取り除くことで，高速に多くのプロジェクトからプロジェクト間クローンを検出できると考えた．

そこで本研究では，複数プロジェクトから高速にクローンを検出する手法を提案する．従来手法では複数プロジェクトから一度にクローンを検出するが，提案手法ではこれを 3 ステップに分けて実行する． (1) 個々のプロジェクトにおいて，プロジェクト内に存在するクローンを検出し， (2) 検出結果から調査の必要ないクローンを取り除き，(3) 得られたクローンから類似するクローンを集約する．これにより提案手法では検出の過程において調査の必要がないクローンが取り除かれるため，プロジェクト間クローンを高速に検出できる．高速化される要因としては以下の 2 つがある．

- 複雑な計算を必要とする従来のクローン検出は検出対象を小さく分割して実行される．
- 調査の必要がないクローンはプロジェクト間クローンを生成する前に取り除かれる．

著者らは検出器 CCFinder [13] と CCFinder を用いて実装さ

れた提案手法とで実行時間を比較した．その結果 100K LOC の疑似プロジェクト 128 個からクローンを検出したとき，提案手法は CCFinder 単独でクローンを検出するよりも最大 49 倍も高速に検出できることを確認した．また検出対象の一部が更新され，クローンを再検出する場合の実行時間も比較した．その結果，CCFinder 単独でクローンを再検出した場合と比べて 141 倍短い時間で再検出することを確認した．

2. 準　　備

2.1 コードクローン

あるソースコード中に存在する 2 つのコード片が互いに同一または類似しているとき，2 つのコード片間にはクローン関係が成り立つという．クローン関係は反射律，推移律，対称律が成り立つ同値関係である．

ある 2 つのコード片においてクローン関係が成立するとき，このコード片のペアを**クローンペア**という．またクローン関係の同値類を**クローンセット**という．すなわちクローンセットは，クローン関係がコード片の任意のペア間で成立するコード片の最大セットである．クローンセット内のコード片は**コードクローン**または単に**クローン**という．

2.2 RNR

調査の必要がないクローンを取り除く手法としてメトリクス RNR により取り除く手法がある [10]．$RNR(S)$ はクローンセット S に含まれるコード片がどの程度繰り返し要素を含まないかの割合を表す．f をクローンセット S に含まれているファイルとする．$TOC_{whole}(f)$ はファイル f を構成している字句数，$TOC_{repeated}(f)$ はファイル f を構成している字句のうち，繰り返し要素である字句の数を表すとする．このとき $RNR(S)$ は次式で表される．

$$RNR(S) = 1 - \frac{\sum_{f \in S} TOC_{repeated}(f)}{\sum_{f \in S} TOC_{whole}(f)}$$

例えば，次の字句で構成されているとするファイル f_1, f_2 からクローン検出し RNR を計測することを考える．

$f_1 : a\ b\ c\ a\ b$

$f_2 : c\ c^*\ c^*\ a\ b$

なお “$*$” は繰り返し要素の字句である．

コード片を表すためにラベル $C(f_i, j, k)$ を用いる．$C(f_i, j, k)$ はファイル f_i の j 番目の字句から k 番目までの字句で構成されるコード片を表す．このとき，それぞれのファイルからクローンセット S_1, S_2 が検出される．

$S_1 : C(f_1, 1, 2), C(f_1, 4, 5), C(f_2, 4, 5)$

$S_2 : C(f_2, 1, 2), C(f_2, 2, 3)$

2 つのクローンセットに対して RNR の値を算出すると，

$$RNR(S_1) = 1 - \frac{0+0+0}{2+2+2} = 1.0$$

$$RNR(S_2) = 1 - \frac{1+2}{2+2} = 0.25$$

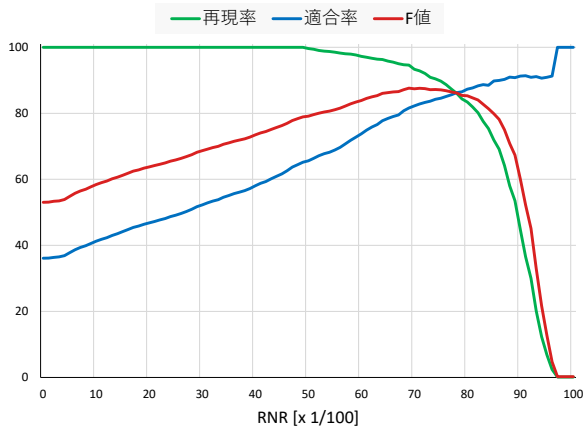


図 1　RNR を用いたフィルタリングによる再現率と適合率及び F 値の推移

となり，S_2 を構成するコード片の大部分が繰り返しの字句であることがわかる．RNR の低いクローンセット中にあるコード片は繰り返し構造が多く，例えば連続した変数宣言，連続したメソッド呼び出し，および switch-case 文の連続のようなコード片である．これらはプログラミング言語の構造によりコードクローンとなる典型的な言語依存のクローンであり調査の必要がないとされている [10]．

既存研究では調査する必要がないクローンのフィルタリングをこの RNR によって行った際の再現率，適合率，F 値を評価している．各値の定義を下記している．

$$\text{再現率 (\%)} = 100 \times \frac{|S_{practical} \cap S_{filtered}|}{|S_{practical}|}$$

$$\text{適合率 (\%)} = 100 \times \frac{|S_{practical} \cap S_{filtered}|}{|S_{filtered}|}$$

$$F \text{ 値} = \frac{2 \times \text{再現率} \times \text{適合率}}{\text{再現率} + \text{適合率}}$$

ここで $S_{practical}$ は調査が必要なクローンセット，$S_{filtered}$ は RNR によって調査が必要と判断されたクローンセットである．

図 1 は RNR の閾値が 0 から 1.0 の場合の再現率，適合率，および F 値の推移を示す図である．図 1 から RNR の閾値を 0.5 とすることで，調査が必要なクローンを取り除くことなく，65%の調査の必要がないクローンを取り除ける．

3. 提 案 手 法

図 2 に提案手法の概要を示す．提案手法は以下の 3 つのステップで構成される．

Step1　プロジェクトごとにクローンを検出する．

Step2　RNR により調査の必要がないクローンを取り除く．

Step3　クローンセットをキュレーションすることで類似したクローンセットをまとめる．

以降それぞれのステップについて説明する．

3.1 Step1: クローン検出

このステップでは個々のプロジェクトに対してプロジェクト内に存在するクローンを検出する．これによりクローンをより

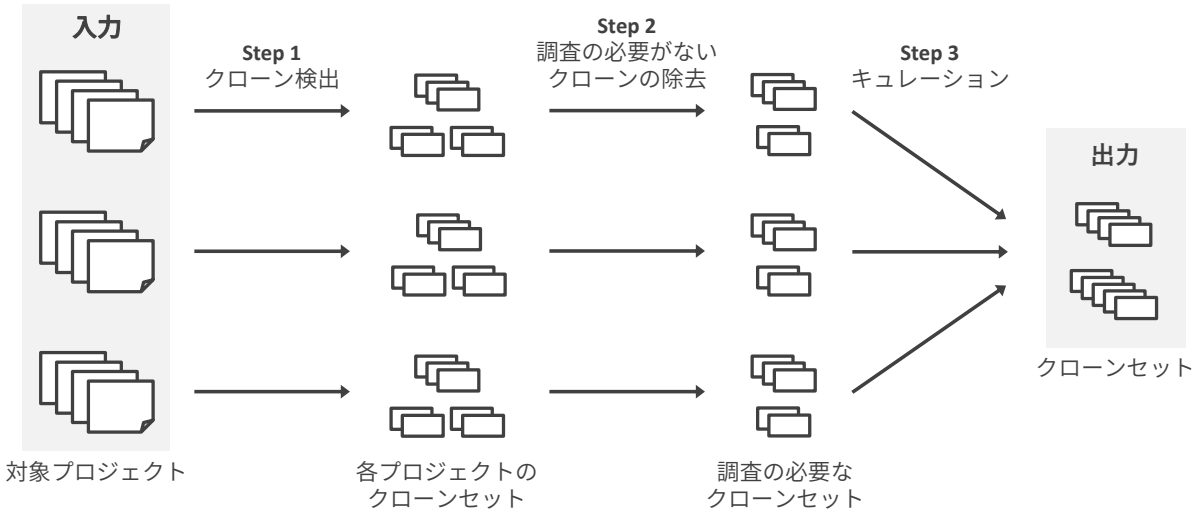


図 2　提案手法の概要

高速に検出できるようになる．高速になるのは 2 つの理由がある．

- 各プロジェクトから個別にクローン検出することにより，一回のクローン検出で検出対象にするソースコードの規模が小さくなるため．
- 個々のプロジェクトでクローン検出を並列実行できるため．

クローンの検出には既存のクローン検出器を用いる．ただし適応できる検出器は 2.2 節で述べた RNR を計測する機能が備わった検出器であり，備わっていない場合は計測する機能を別途追加実装する必要がある．

3.2　Step 2: 調査の必要がないクローンの除去

検出器により検出されたクローンには調査の必要がないクローンが含まれている．そこで各プロジェクトで検出されたクローンから調査の必要がないクローンを RNR によって取り除く．取り除くクローンセットは Higo らの研究に基づいて $RNR < 0.5$ を満たすクローンセットとした [10]．RNR が低いクローンを取り除くのは以下の 2 つの理由がある．

- 調査の必要がないクローンを調査せずに済む．
- 計算時間が短縮できる．

したがってクローンセット間の類似度を計測が必要なクローンセットが少なくなり，より高速に検出できる．

3.3　Step 3: キュレーション

個々のプロジェクトから得られたクローンセットに対し，類似するクローンセットをまとめる．クローンセットのペア $< S_x, S_y >$ が類似しているかの判定は閾値 θ を用いて下式を満たすかどうかで判定する．

$$|N(S_x) \cap N(S_y)| \geq \lceil \theta \cdot max(|N(S_x)|, |N(S_y)|) \rceil$$

max 関数は引数のうち最も大きい値を出力する関数であり，$N(S_x)$ はクローンセット S_x 内のコード片に含まれる字句列から生成された N-gram の集合，$|N(S_x)|$ は $N(S_x)$ の要素数である．$N(S_x)$ と $N(S_x)$ を基にクローンセット間の類似度を計算することで，類似した構造のコード片をもつクローンセット間の類似度が高くなるように計算できる．

4.　実　　装

4.1　クローン検出器

提案手法ではクローンの検出を既存のクローン検出器によって行う．提案手法の実装する際，利用するクローン検出器として CCFinder [13] を用いた．CCFinder を用いる理由としては RNR の計測機能が既に備わっていることがあげられる．

4.2　クローンセット間の類似度計測

類似するクローンセットを特定するためにはクローンセット内に存在するコード片の字句列から N-gram を生成する必要がある．デフォルトの設定では N-gram のサイズを 5 で生成する [14]．また字句列内に含まれる識別子とリテラルは正規化し，それぞれ特定の文字列に変換する．例えば識別子である `index` や `getOrder` は全て `identifier` に，文字列は全て `literal` に変換する．

4.3　経験則の導入

本研究ではより高速に類似度計算を行うため大規模にクローンを検出する検出器 SourcererCC [15] で用いられている以下の 2 つの経験則を導入した．

共通要素によるフィルタリング　あるクローンセット内に存在する N-gram の集合と他のクローンセットの集合に共通する N-gram が存在しないクローンセットの組は類似度計算の対象から取り除く．

位置フィルタリング　クローンセット内の N-gram 集合を N-gram の出現数順に並べたときの位置を基にクローンセットが類似しうるかどうかを判定する．類似する可能性のないクローンセットは類似度計算の対象から取り除く．

以降，各経験則について詳細を述べる．

4.3.1　共通要素によるフィルタリング

共通要素によるフィルタリングでは，クローンセット間に共通する N-gram が存在するクローンセットのみを類似度計算の対象とすることで計算量を削減する．具体的には以下の特性を用いる．

特性 1：あるクローンセット S_x および S_y はそれぞれ n_x, n_y 個の N-gram で構成されており，$N(S_x)$ と $N(S_y)$ は事前定義された順序で並び替えられているとする．ここで自然数 i に対して $|N(S_x) \cap N(S_y)| \geqq i$ であるとき，$N(S_x)$ および $N(S_y)$ の最初の $max(n_x, n_y) - i + 1$ 個は，少なくとも 1 つ一致する必要がある．

例えば S_x, S_y が $N(S_x) = \{a, \mathbf{b}, c, d, e\}$，$N(S_y) = \{\mathbf{b}, c, d, e, f\}$ の 5 つの N-gram で構成されているとする．類似度の閾値 θ を 0.8 とすると，2 つのクローンセットは少なくとも $\lceil 0.8 \cdot 5 \rceil = 4$ 個の N-gram が一致する必要がある．特性 1 により，S_x と S_y が類似するには最初の $max(n_x, n_y) - i + 1 = 5 - 4 + 1 = 2$ 個の N-gram で構成される部分集合が少なくとも 1 つの N-gram 一致する必要がある．S_x と S_y の場合 **b** が両方の部分集合で共通しているため，S_x と S_y は類似していると判定される．また共通の N-gram が存在しない場合は S_x と S_y は類似していないと判断できる．すなわち特性 1 により，S_x と S_y のすべての N-gram を互いに比較する代わりに，最初の $max(n_x, n_y) - i + 1$ 個の N-gram で構成される部分集合のみを比較して，S_x と S_y が類似しているかを推測できる．

4.3.2 位置フィルタリング

共通要素によるフィルタリングは比較する N-gram 集合の要素数が異なると誤検出する場合がある．例えば $N(S_x) = \{a, \mathbf{b}, c, d\}$ と $N(S_y) = \{\mathbf{b}, c, d, e, f\}$ であるクローンセット S_x, S_y が類似しているか判定すると仮定する．S_x と S_y が類似していると判定されるためには 4 個の N-gram が一致する必要があるが，2 つのクローンセットには共通の N-gram が 3 つしかないため，S_x と S_y は類似したクローンセットではない．しかし最初の $max(n_x, n_y) - i + 1 = 2$ 個の N-gram で構成される部分集合は共通して **b** が存在する．その結果，特性 1 が満たされ類似したクローンセットの候補と誤検出されてしまう．

この誤検出を位置フィルタリングによって取り除く．位置フィルタリングは特性 1 で並び替えた N-gram の順序を利用して，2 つのクローンセットに共通しうる N-gram の個数の上限を求める．$N(S_x)$ と $N(S_y)$ で一致する **b** の位置から $N(S_x)$ と $N(S_y)$ で比較していない N-gram の個数の最小値が取得できる．すなわち **b** 以降に存在する N-gram は $N(S_x)$ の場合 2 個 (c, d)，$N(S_y)$ の場合 4 個 (c, d, e, f) 存在するので，$N(S_x)$ と $N(S_y)$ で比較していない N-gram の個数の最小値は 2 である．この値と既に一致した N-gram の個数の和は 2 つのクローンセットに共通しうる N-gram のまでを上限である．$N(S_x)$ と $N(S_y)$ で一致しているのは **b** だけなので S_x と S_y の共通しうる N-gram の個数の上限の和は $min\,(2, 4) + 1 = 3$ 個となる．ここで類似していると判定されるために必要な 4 個よりも小さいため，S_x と S_y のペアは類似するクローンセットの候補からを取り除かれる．この特性は次のように定義される．

特性 2：あるクローンセット S_x と S_y に対し，$N(S_x)$，$N(S_y)$ を事前定義された順序で既に並び替えられているとする．また $N(S_x)$ の i 番目の N-gram が $N(S_y)$ にも存在するとき，$N(S_x)$ の 1 番目から i 番目までの部分集合を $N_{left}(S_x)$，$i + 1$ 番目以降の部分集合を $N_{right}(S_x)$ と表す．ここで S_x と S_y が閾値 θ で類似しているならば，$N(S_x)$ と $N(S_y)$ とで共通して存在する N-gram は全て，$min\,(|N_{right}(S_x)|, |N_{right}(S_y)|) + |N_{left}(S_x) \cap N_{left}(S_y)|$ のとる値が類似していると判定されるために必要な $\lceil \theta \cdot max(|N(S_x)|, |N(S_y)|) \rceil$ 以上となる．

5. Research Questions

本研究では 2 つの Research Question (RQ) を設定し提案手法を評価実験した．

5.1 RQ1: 提案手法によりどの程度検出速度が向上するか

この RQ ではまず BigCloneBench データセット [16] からサイズの等しい擬似プロジェクトを作成する．次に提案手法によるクローン検出と CCFinder による検出の実行時間を比較する．提案手法では個々の疑似プロジェクト内クローンを CCFinder によって検出する．そしてプロジェクト間クローンはキュレーションによって生成される．後者の CCFinder による検出では CCFinder を用いてプロジェクト間クローンを擬似プロジェクト全体から検出する．

5.2 RQ2: 検出対象が更新された場合どの程度高速に検出できるか

この RQ ではプロジェクトのソースコードが更新され，プロジェクト間クローンを再度取得する必要があると想定する．提案手法を用いる場合，個々のプロジェクト内クローンの検出結果を用いるため更新されたプロジェクトからのみ再度クローン検出をすればよい．そして更新されたプロジェクトの検出結果と他のプロジェクトの検出結果からキュレーションを行う．提案手法を用いない場合，対象プロジェクト全体に対して再度 CCFinder を実行する必要がある．この評価では提案手法と CCFinder の 2 種類の検出に要する実行時間を比較する．

6. 実 験 対 象

実験には BigCloneBench データセット [16] を用いた．BigCloneBench データセットは IJaDataset 2.0[(注1)] に存在する 25,000 の Java システムのデータセットから作成されている．BigCloneBench データセットには，800 万個の手動検証済みクローンペアを含む 290 万個のファイルが含まれており，大規模クローン検出やクローン検索における評価に用いられている．実験ではこの BigCloneBench データセットから構築した 14 個の疑似プロジェクト群を用いる．疑似プロジェクト群の構築方法の概要を図 3 に示す．まず BigCloneBench データセットから総行数が目的の行数を超えるまでランダムにファイルを取り出し，取り出されたファイル群を 1 つの疑似プロジェクトとする．これを繰り返し 10K LOC と 100K LOC の疑似プロジェクトをそれぞれ 128 個作成した．また対象プロジェクトの数を変更してクローンを検出するために，10K LOC グループおよび 100K LOC グループのそれぞれからランダムに疑似プロジェクトを抽出し，疑似プロジェクト群を 7 種類構築した．各疑似プロジェクト群に含まれる疑似プロジェクトの数は $2^i, i \in [1, 7]$

(注1)：https://sites.google.com/site/asegsecold/projects/seclone

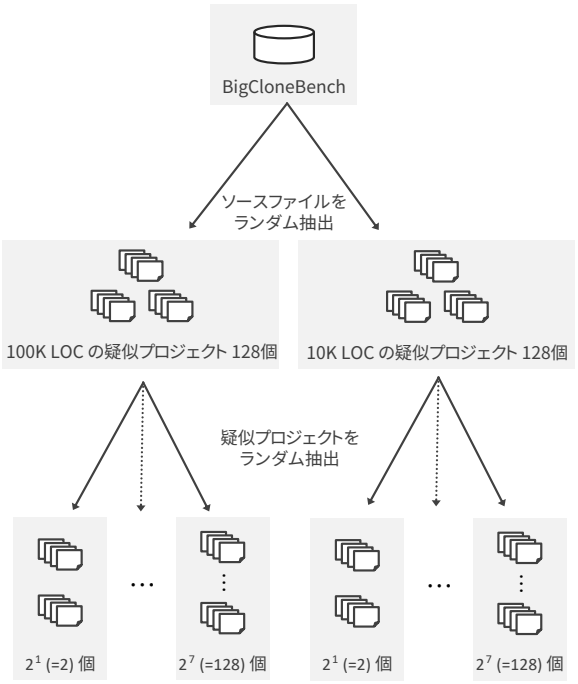


図 3　疑似プロジェクト群の構築方法

である．

7. 実　　験

提案手法の評価は 2.9GHz のプロセッサ，32 GB の RAM，512 GB の SSD を搭載した Windows 10 を用いた．

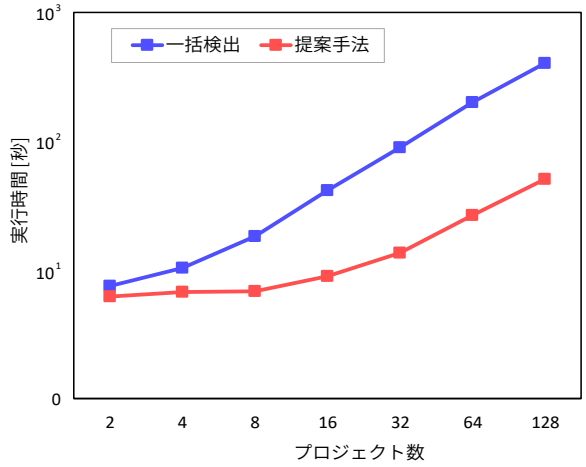
7.1 RQ1: 提案手法によりどの程度検出速度が向上するか

提案手法によりクローンを高速に検出できることを調査するために，CCFinder で複数プロジェクトから一括でクローンを検出した場合と，提案手法によりクローン検出した場合とでの検出時間を比較した．検出に用いた CCFinder の設定はデフォルトとし，提案手法の設定は類似度計算の閾値はデフォルトの 0.8 を用いた．実験対象には 6. 章で述べた疑似プロジェクト群を用いた．

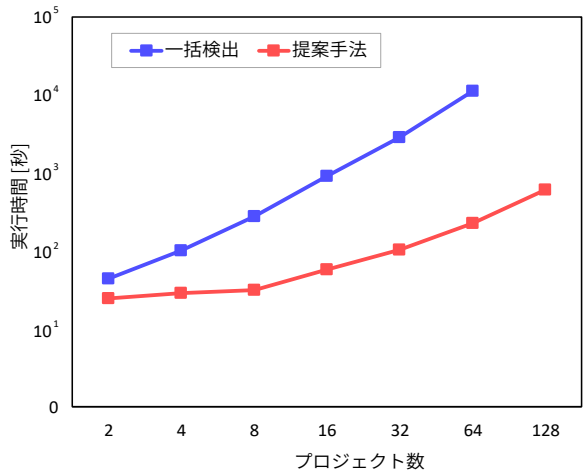
各疑似プロジェクト群からクローンの検出に要した時間を計測した結果を図 4 に示す．10K LOC の場合は 2 プロジェクトのとき最小の 1.20 倍，128 プロジェクトのとき最大の 8.04 倍の検出速度で検出した．プロジェクト数の増加に対し一括で検出した場合にかかった時間と提案手法による検出速度の比は単調増加している．これは提案手法が検出対象全体からのプロジェクト間クローンではなく，個々のプロジェクトからプロジェクト内クローンを検出するためであると考えられる．

100K LOC の場合は 2 プロジェクトのとき最小の 1.79 倍，64 プロジェクトのとき最大の 48.9 倍の検出速度でクローンを検出した．10K LOC と比較すると 100K LOC の方がより早くクローン検出できていることから，検出対象が大きくかつ多くのプロジェクトからクローンを検出するときに提案手法が有効であると考えられる．

また CCFinder はメモリ不足エラーのため 100K LOC の 128 プロジェクトからクローン検出できなかった．これは膨大なソースコードからクローンを検出するには膨大なメモリが必要なためである．一方，提案手法は 100K LOC の 128 プロジェクトからクローンを検出できた．これはキュレーションの対象となるクローンセットを削減するために，調査の必要がないクローンセットを取り除いたためだと考えられる．



(a) 10K LOC の疑似プロジェクトに対するクローン検出に要する時間



(b) 100K LOC の疑似プロジェクトに対するクローン検出に要する時間

図 4　各プロジェクト群に対するクローン検出の実行時間

よって RQ1 の答えとしては，提案されたアプローチが CCFinder よりも約 49 倍速くクローンを検出できる．さらに提案手法ではメモリ不足のため CCFinder でクローンを検出できないサイズでもクローンを検出できる．

7.2 RQ2: 検出対象が更新された場合どの程度高速に検出できるか

この評価では次の状況を想定する．個々のプロジェクトは異なるタイミングに 1 回更新される．開発者はプロジェクトが更新された直後にプロジェクト間クローンをできるだけ高速に検出したいと考えている．この仮定の下で，100K LOC の疑似プロジェクトが 64 個存在する疑似プロジェクト群を用いて逐次更新時のクローン検出を行った．したがって，CCFinder でプロジェクト間クローンを検出する場合，64 個のプロジェクト全体からのクローン検出が 64 回実行される．一方，提案手法を適用する場合，更新されたプロジェクトからのクローン検出と 64 プロジェクトのキュレーションが 64 回実行される．それぞ

れの検出の実行時間を計測した．

実験結果を表 1 に示す．CCFinder は検出に平均で 11,207 秒要した．一方で提案手法は平均 79 秒でクローンを検出し，CCFinder で全プロジェクトからクローン検出するよりも約 141 倍早くクローンを検出した．さらに提案手法におけるクローン検出とキュレーションの実行時間の比率を調査したところ，クローン検出には平均 31 秒，キュレーションには平均 48 秒要していた．そのため実行時間の約 40%がクローン検出に費やされていた．

よって RQ2 に対する答えは，提案手法は CCFinder を使用して 64 の擬似プロジェクト全体からプロジェクト間クローンを検出するのに 141 倍早く検出できる．

8. 妥当性への脅威

8.1 プロジェクト間にのみ存在するクローン

提案手法では検出対象である個々のプロジェクトからクローンを検出し，それをキュレーションすることでプロジェクト間クローンを検出する．そのため，プロジェクト間クローンとして検出されるためには，プロジェクトそれぞれの検出でクローンとして検出される必要がある．したがって提案手法はプロジェクト間にのみ存在するクローンは検出できないが，プロジェクト内にクローンとして存在するコード片は高速に検出が可能である．

8.2 クローン検出器

評価実験ではトークン単位のクローン検出器として CCFinder を用いて実験した．しかしクローン検出器は CCFinder 以外にも多数存在する．そういった他のクローン検出器でも高速にクローンを検出できるとは限らない．

8.3 実 験 対 象

評価実験には BigCloneBench から構築した疑似プロジェクト群を対象にした．ただし他のデータセットや実プロジェクトに対して提案手法を適応した場合，同程度高速に検出できるとは限らない．

9. あ と が き

本研究では個々のプロジェクトから得られるクローンの検出結果を用いることでクローンを高速に検出する手法を提案した．提案手法の性能を評価した結果，既存の検出器を提案手法に適応することで最大で 49 倍の速度でクローンを検出できた．また検出対象が逐次更新された場合は，検出にかかる時間を 141 倍まで早く検出できた．

今後の展開としては CCFinder 以外の検出器でも提案手法を適応できるように改良していくことを考えている．

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：17H01725) の助成を得て行われた．

表 1 対象プロジェクトの逐次更新時における平均検出時間

	一括検出	提案手法
平均検出時間	11,207 秒	79 秒

文 献

[1] A. Lozano and M. Wermelinger, “Assessing the effect of clones on changeability,” 2008 IEEE International Conference on Software Maintenance, pp.227–236, 2008.

[2] M. Mondal, M.S. Rahman, R.K. Saha, C.K. Roy, J. Krinke, and K.A. Schneider, “An empirical study of the impacts of clones in software maintenance,” 2011 IEEE 19th International Conference on Program Comprehension, pp.242–245, 2011.

[3] F. Rahman, C. Bird, and P. Devanbu, “Clones: What is that smell?,” Conference on Mining Software Repositories (MSR 2010), pp.72–81, May 2010.

[4] T. Zhang and M. Kim, “Automated transplantation and differential testing for clones,” Proceedings of the 39th International Conference on Software Engineering, pp.665–676, ICSE ’17, 2017.

[5] A. Sheneamer and J. Kalita, “A survey of software clone detection techniques,” International Journal of Computer Applications, vol.137, no.10, pp.1–21, 2016.

[6] K. Chen, P. Liu, and Y. Zhang, “Achieving accuracy and scalability simultaneously in detecting application clones on android markets,” Proceedings of the 36th International Conference on Software Engineering, pp.175–186, ICSE 2014, 2014.

[7] J. Davies, D.M. German, M.W. Godfrey, and A. Hindle, “Software bertillonage: finding the provenance of an entity,” Proceedings of the 8th working conference on mining software repositories (MSR), pp.183–192, 2011.

[8] C. Ragkhitwetsagul and J. Krinke, “Siamese: scalable and incremental code clone search via multiple code representations,” Empirical Software Engineering, pp.1–49, 2019.

[9] M. Kim, V. Sazawal, D. Notkin, and G. Murphy, “An empirical study of code clone genealogies,” ACM SIGSOFT Software Engineering Notes, vol.30, pp.187–196, 2005.

[10] Y. Higo, T. Kamiya, S. Kusumoto, and K. Inoue, “Method and implementation for investigating code clones in a software system,” Information and Software Technology, vol.49, no.9, pp.985–998, 2007.

[11] A. Goto, N. Yoshida, M. Ioka, E. Choi, and K. Inoue, “How to extract differences from similar programs? a cohesion metric approach,” 2013 7th International Workshop on Software Clones (IWSC), pp.23–29, May 2013.

[12] R. Yue, Z. Gao, N. Meng, Y. Xiong, X. Wang, and J.D. Morgenthaler, “Automatic clone recommendation for refactoring based on the present and the past,” 2018 IEEE International Conference on Software Maintenance and Evolution (ICSME), pp.115–126, 2018.

[13] 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.

[14] G. Myles and C. Collberg, “K-gram based software birthmarks,” Proceedings of the 2005 ACM Symposium on Applied Computing, pp.314–318, SAC ’05, 2005.

[15] H. Sajnani, V. Saini, J. Svajlenko, C.K. Roy, and C.V. Lopes, “Sourcerercc: scaling code clone detection to big-code,” 2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE), pp.1157–1168, 2016.

[16] J. Svajlenko, J.F. Islam, I. Keivanloo, C.K. Roy, and M.M. Mia, “Towards a big data curated benchmark of inter-project code clones,” In Proceedings of the Early Research Achievements track of the 30th International Conference on Software Maintenance and Evolution (ICSME 2014), pp.476–480, 2014.