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

サブルーチン化による集約に適したコードクローン検出手法の提案

川本　琢人†　　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: †{tk-kawam,higo}@ist.osaka-u.ac.jp

あらまし　コードクローンの集約は，ソフトウェアのコード量を削減し，保守に要するコストを減少させる．しかし，コードクローン検出器が検出したコードクローンの集約作業を人の目視によって行うと膨大な時間を要してしまう．コードクローンの集約作業は，集約されるべきコードクローンの選別と，選別されたコードクローンの集約の 2 つの処理に分けられる．本研究は，集約できるコードクローンの選別作業の自動化に注目し，サブルーチン呼び出しへの書き換えによる集約に適したコードクローンの自動検出を目的とする．文の削除及び追加を含むコードクローンを共通したサブルーチンの呼び出しに書き換えるため，サブルーチンに関数オブジェクトを渡す方法を考えた．コードクローンに求められる条件を検討し，サブルーチンの生成に適したコードクローンの検出手法を提案した．検出手法は抽象構文木を使用する方法を用いている．本稿では C#を対象とするコードクローン検出器を実装した．また，多くの文の削除及び追加を許容したコードクローンを検出するコードクローン検出器 NIL と提案手法の検出結果を比較し，提案手法によってサブルーチンの生成に適したコードクローンが検出できているか確認した．
キーワード　コードクローン, メソッドの抽出，サブルーチン

1. はじめに

ソースコード中に存在する，一致あるいは類似したコード片はコードクローンと呼ばれる．不必要に発生したコードクローンは，バグ修正や機能追加のための編集作業に漏れを生じさせる原因となる [1]．不要なコードクローンの適切な除去により，編集作業の漏れを防ぐことが可能になり，ソフトウェアの保守性が高く保たれる．しかし，手作業によるコードクローンの除去作業には手違いが起こりうる．大規模なソフトウェアへの適用を考慮すれば，コードクローンの除去は自動的な処理として行うのが望ましい．

コードクローンを削減する方法の 1 つに，共通した処理を 1 つのサブルーチンに集約するメソッドの抽出がある [2]．共通した処理を 1 つのサブルーチンにまとめれば，コードクローンはサブルーチン呼び出しに書き換えられる．サブルーチンへの集約はソースコードの量を削減し，高い保守性の維持に貢献する [3]．共通した処理が複数箇所に記述されなくなるため，処理に対する変更やバグ修正はサブルーチン定義のただ 1 箇所にのみ施せばよく，複数箇所に変更を加える必要がなくなる．

メソッドの抽出は，コードクローン間で共通するロジックを新たなメソッドに抽出する．この性質上，肥後らは文の追加や削除を含むコードクローンに対しては原則として適用できないとした [2].

引数を持つ関数は，引数に渡す値を変えることでその挙動を様々に変化させられる．引数に関数オブジェクトを渡して，関数内でその関数オブジェクトを呼び出せば，関数呼び出し部からコードを注入して関数のロジックの一部を変化させられる．関数オブジェクトを使用すれば，文の追加や削除，変更を伴うコードクローンからサブルーチンを抽出できるといえる．

本稿では，サブルーチンの抽出によるコードクローンの集約がどのようなコードクローンに対して有効であるか検討し，これに適したコードクローンの検出手法を提案する．

提案手法は抽象構文木を使用する方法を用いて，コードクローンの検出単位を連続した文のまとまりの単位とした．検出対象とするプログラミング言語に C#を選択した．サブルーチンの抽出による集約に適した文の追加や削除を含むコードクローンを検出するため，制御構造の内部の違いを無視して構文木を比較した．

2. 背　　景

この章では，コードクローンについて説明したうえで，サブルーチンの抽出に適したコードクローンが満たすべき条件を検討する．

2.1 コードクローン

コードクローンは，ソースコード中の類似したコード片である．互いに類似するコード片の集合をクローンセットという．クローンセットに含まれるコード片の単位は，メソッド単位や複数行のテキストなど，検出の目的に応じてさまざまに設定される．また，コード片の一致にもさまざまな定義が与えられる．Roy らは，コード片の一致の定義に幅を与えて，コードクローンを次の 4 つに分類した [5].

タイプ 1　空白，タブ文字，改行やコメントなどを除いて一致するコードクローン.

タイプ 2　タイプ 1 の除外対象のほか，リテラル，型，識別子を除いてはじめて一致するコードクローン.

コード 1　サブルーチンの生成例

```
static void subroutine_name(argument)
{
    code_fragment
}
```

タイプ 3　タイプ 2 の除外対象のほか，文の変更，挿入，または削除の違いを許容してはじめて一致するコードクローン．

タイプ 4　同様な処理を行うが，構文が異なるコードクローン．

特にタイプ 3 のコードクローンはギャップを含むコードクローンとも呼ぶ．文の変更，挿入，削除によってほかのコードクローンと一致しない部分を指してギャップと呼ぶ．複数行にわたって連続して一致しない部分が続く場合，連なった行全てをまとめて 1 つのギャップと数える．

2.2　サブルーチンの生成が求めるコードクローンの性質

本節では，検出されたコードクローンからサブルーチンの定義を生成する処理に注目し，サブルーチンの抽出による集約に適したコードクローンの条件を検討する．以後，簡単のため，サブルーチンの抽出による集約をサブルーチン化と呼ぶ．

サブルーチンの定義の形式を確認する．あるクローンセットに対して，サブルーチンの定義は含まれるコード片の 1 つを用いてコード 1 の通り記述される．ただし，subroutine_name には適切なサブルーチン名, argument には適切な引数, code_fragment には検出されたコード片が当てはまる．サブルーチン名および引数の決定はサブルーチン化の処理が担うため，ここでは適切な名前および引数が既に当てはめられているとして考える．

サブルーチンが追加されたソースコードが正しく動くためには，少なくとも構文に間違いがあってはならない．クローンセットによって，サブルーチンの定義および呼び出しの生成に際して構文の間違いを生んでしまう場合がある．ただし，構文に間違いが無くとも，名前空間の違いによるコンパイルエラーや実行コンテキストの違いによる実行時の予期せぬ挙動が引き起こされる可能性が考えられる．本稿では構文の間違いを防ぐことにのみ注目する．本節では，サブルーチン化の際に構文の間違いが生じる具体例を挙げて，サブルーチン化に適したクローンセットについて検討する．

まず，構文の間違いを生んでしまうコード片の 1 つ目の例をコード 2 に示す．6 行目の波括弧閉じ ‘}’ には，コード片の中に対応する波括弧開き ‘{’ が存在しない．このコード片をサブルーチン定義に当てはめると構文エラーが発生する．このエラーは，コード片に制御構造の一部しか含まれないことに起因する．サブルーチン化に適したコードクローンは，制御構造をコード片に含むとき，制御構造の全体を含まなければならないといえる．C#では，制御構造は 1 つの文である．コードクローンの検出単位を 1 つ以上の連続した文とすれば，コード片に含まれる制御構造は必ずその全体が含まれるようになる．以降，簡単のため 1 つ以上の連続した文を文のシーケンスと呼ぶ．

つづいて，構文の間違いを生んでしまうコード片の 2 つ目

コード 2　サブルーチン化に適さないコード片の例 1

```
    var target = list[i];
    if (predicate(target))
    {
        function(target);
    }
}
```

コード 3　サブルーチン化に適さないコード片の例 2

```
public int member1;
protected float member2;
private string member3;
```

の例をコード 3 に示す．このコード片はクラスのメンバ変数の宣言部である．このコード片をサブルーチン定義に当てはめると，定義部と呼び出しの双方で構文エラーが発生する．C#はローカル変数の宣言にアクセス修飾子を付けられないため，定義部にエラーが発生する．また，クラス定義内には関数の呼び出しを書けないため，サブルーチンの呼び出し箇所でも構文エラーが発生する．

この例の問題点は，クラス定義内の共通部分に対してサブルーチン化を試みた点である．クラス定義内の共通部分は，メンバ変数の宣言のほか，メンバ関数の宣言も含まれる．メンバ関数の宣言は関数定義の中には書けない．サブルーチンの呼び出し箇所で構文エラーが発生するのはメンバ変数の宣言の場合と同様である．クラス定義直下の共通部分はサブルーチン化するべきではない．ソースコードの他の部分についても考慮すると，サブルーチン化に適したコードクローンはメソッド定義内のコードクローンに限られるといえる．

2.3　メソッドの抽出との比較によるサブルーチン化の検討

肥後らは，共通した処理からメソッドを抽出する場合はタイプ 1 およびタイプ 2 のコードクローンが適切であるとしている [2]．この場合，サブルーチンの引数では値を参照，格納するポインタを受け渡す．一方，サブルーチンの引数に関数オブジェクトを渡すサブルーチン化では，対象のコードクローンはタイプ 3 のコードクローンの一部まで広がる．引数から関数オブジェクトを受け取るサブルーチンへの集約に適したコードクローンの例をコード 4 に示す．この例では ‘functionA’ の呼び出し部分と ‘functionB’,‘functionC’ の呼び出し部分は共通していない．Roy らの分類によるとこれはタイプ 3 のコードクローンである．ただし，文の削除および追加は if 文の内部でのみ見られ，if 文の条件式や if 文の外側には見られない．この例の場合はコード 4 に示したサブルーチン呼び出しへ構文の間違いなしに置き換えられる．相違した部分を関数オブジェクトの呼び出しに書き換えると，サブルーチンは双方ともに代替できる．関数オブジェクトを引数に用いた集約のごく簡単な例をコード 5 に示す．

タイプ 3 のコードクローンであっても，サブルーチン化に適さない場合がある．サブルーチン化に適さないコードクロー

コード 4　タイプ 3 のコードクローンの例

```
if (predicate(obj))
{
    functionA(obj);
}
```

```
if (predicate(obj))
{
    functionB(obj);
    functionC(obj);
}
```

コード 5　サブルーチン定義の例

```
static public void subroutine(ObjectClass obj,
      Action<ObjectClass> action)
{
    if (predicate(obj))
    {
        action.Invoke(obj);
    }
}
```

コード 6　タイプ 3 だがサブルーチン化に適さないコードクローンの例

```
if (predicate(obj))
{
    function(obj);
}
```

```
if (predicate(obj))
{
    for(int i = 0; i < N; i++)
    {
        function(obj);
    }
}
```

ンの例をコード 6 に示す．互いに一致している部分をハイライトして示している．このコードクローンは，両コード片の最長共通部分列である．一致が検出された各 4 行のうち，3 行目と 4 行目は構造として異なる．3 行目の関数呼び出し文は，一方は if 文，もう一方は for 文の内部直下に位置している．4 行目の波括弧閉じ '}' は，一方は if 文，もう一方は for 文の括弧閉じである．これはタイプ 3 のコードクローンであるが，サブルーチン化には適さない．共通部分をサブルーチンにまとめづらいコードクローンを検出対象から除外するため，共通部分のギャップは制御構造の内部に限って認めるのが望ましいといえる．

2.4　サブルーチン化に適したコードクローン

前節までで述べたサブルーチン化に適したコードクローンに求められる条件を次にまとめる．

（1）　コードクローンは，文のシーケンス単位で検出されるべきである．

（2）　コードクローンは，メソッド定義の中からのみ検出されるべきである．

（3）　タイプ 1,2 または制御構造の内部に限ってギャップを許容したコードクローンが検出されるべきである．

既存のコードクローン検出手法が検出するコードクローンに，これら 3 つの条件が当てはまるか確認する．CCAligner は 1 つのギャップを許容するコードクローンを検出する [6]．CCAligner は，2 つの制御構造が含まれ，その制御構造の内部がいずれも他と一致しないコードクローンを検出しない．LVMapper および NIL は複数のギャップを許容するコードクローンを検出する [4], [8]．NiCad はタイプ 1，2，および 3 に類される，文の挿入および削除，変更までを許容したコードクローンを，互いのコード片の最長共通部分列を求めて検出する [7]．しかしいずれの検出器の場合でも，検出されるコードクローンのギャップの位置を制御構造の内部に限定しない．また，検出にあたって，検出単位のうち一致した行数の割合に閾値を設けている．そのため，ギャップの割合が大きくなると，一致の割合が閾値を超えなくなる場合がある．サブルーチン化に適したコードクローンはギャップの割合の大きさに決まった上限を必要としない．

本稿では，3 つの条件を満たすサブルーチン化に適したコードクローンの検出手法を提案する．

3.　アイデアと提案手法および実装

3.1　抽象構文木を用いたコードクローン検出手法

サブルーチン化に適したコードクローンは，文のシーケンス単位のコード片であり，制御構造の内部に限ってギャップを許容する．文の単位でコードクローンを検出する方法には，抽象構文木を用いる方法がある [1]．抽象構文木を使用したコードクローンの検出手法に Baxter の方法がある [9]．Baxter の方法は次の 3 つの処理からなる．

基本アルゴリズム

構文木の小さな部分木同士を比較し，互いに一致する部分木に分類する．

シーケンス検出アルゴリズム

互いに一致する部分木を含む文のシーケンスが一致するか調べる．部分木を含む木のシーケンスの全通りについて，互いに長さが一致するシーケンスの一致を調べる．

類似したコードクローンを発見するアルゴリズム

検出したいクローンセットは完全に一致するコードクローンの集合に限らないため，差異を含む類似したコードクローンを発見する．互いに異なるが，それぞれ一致する部分木を持つ木同士を一致しているとみなし，類似したコードクローンを発見する．

本稿では，基本アルゴリズムとシーケンス検出アルゴリズムを用いる．Baxter の方法は，大量の構文木のマッチングを高速化するために木および木のシーケンスのハッシュ値を使用する．このため，Baxter の方法は構文木からハッシュ値を求める関数が必要である．

Baxter は，意図的にハッシュ値の衝突を引き起こす関数を選択すると，検出されるコードクローンの性質を特徴づけられると述べている．制御構造の内部が異なっていてもコードクローンとして検出するため，本稿では制御構造の内部の違いを一切無視する関数を選択する．この関数の選択によって，連続した文からなるシーケンス単位で検出しながらも，限られた範囲でギャップを許容できる．

3.2 ハッシュ関数による検出結果の変化

ハッシュ関数が制御構造の内部を無視した値を返すためにもたらす検出結果の変化について，具体的な検出結果を挙げて説明する．

C#で使用できる制御構造の 1 つに do 文（do statement）がある．do 文は繰り返し構造の 1 つであり，1 回以上繰り返される文は文のシーケンスである．

do 文を含むコードの例をコード 7 に示す．各例の 5 行目が do 文内の文のシーケンスである．この 2 つのコードに対して，制御構造の内部を考慮したハッシュ関数でコードクローン検出をしたところ，基本アルゴリズムによってそれぞれ 7 行目の条件文 'rc == -1' が検出されたのみであった．一方，制御構造の内部を無視したハッシュ関数でコードクローン検出をしたところ，1 行目から 9 行目までのシーケンスが検出され，制御構造の内部は依然としてコードクローンとして検出されていない．この検出結果は，2 つのコード片は do 文の内部を除いて一致していることを明らかにしている．

コードクローン検出にあたって，どの制御構造の内部を無視するかを事前に決定しておく必要がある．内部を無視する制御構造の選択は，対象とするプログラミング言語に依存する．本稿では，if 文，for 文，foreach 文，while 文，do 文の 4 つの制御構造の内部を無視するハッシュ関数を使用した．

Baxter の方法で制御構造の内部を無視するハッシュ関数を使用すると，制御構造の内部にも一致部分を含むコードクローンは，制御構造の文を含むコードクローンと，制御構造の内部の文からなるコードクローンの 2 つに分かれる．制御構造の内外でコードクローンが分かれてしまう例を図 2 に示す．コードクローン A1，A2，A3 は同じクローンセットに含まれる互いに一致したコード片である．コード片に含まれる 2 つの if 文は，その内部を無視して一致している．コードクローン A1，A2 は上方の if 文の内部にコードクローン B1，B2 をそれぞれ含む．また，コードクローン A3 は下方の if 文の内部にコードクローン B3 を含む．コードクローン B1，B2，B3 は同じクローンセットに含まれる．ここで，A1 と B1 を結合して作られるコード片と A2，B2 を結合して作られるコード片は互いに一致し，この 2 つのコード片も同じクローンセットに含まれるコードクローンであるといえる．検出された 2 つのクローンセットはそのままに，分かれて検出されたクローンセットを 1 つにまとめたクローンセットを検出結果にさらに追加する処理が必要である．この処理について，次節で詳しく述べる．

3.3 2 つに分割されたクローンセットの統合

ある制御構造に注目したとき，制御構造の文をシーケンスに含むコードクローンを親，内部のシーケンスの部分を含むコードクローンを子と呼び，この親と子には親子関係があるという．親子関係にある 2 つに分かれたクローンセットが検出された場合，1 つにまとめられたクローンセットを求める必要がある．1 つにまとめられたクローンセットを求める処理を親子関係の解決と呼ぶ．また，Baxter の方法による検出結果に，1 つにまとめられたクローンセットすべてを加えた集合を親子関係を解決したクローンセットの集合と呼ぶ．

3.4 提 案 手 法

提案するコードクローン検出器は，複数ファイルからなるソースコードを入力とし，クローンセットの集合を出力する．提案手法の概観を図 1 に示す．提案手法は構文解析と Baxter の方法，親子関係の解決の 3 つの処理からなる．まず，構文解析によってソースコードから構文木を得る．つづいて，コードクローンを連続した文からなるシーケンスの単位で検出するため，構文木を比較して検出する Baxter の方法 [9] を用いる．Baxter の方法の検出結果に親子関係の解決を施して，サブルーチン化に適したクローンセットの集合を出力する．

親子関係の解決のアルゴリズムを次に示す．ここで，簡単のため，本来検出されるべき 1 つのコードクローンをまとめられたコードクローン，これを含むクローンセットをまとめられたクローンセットと呼ぶ．

- 注目している親子関係について，子の，親のシーケンス上の位置を n とする．親を含むクローンセットから，シーケンスの n 番目の制御構造の内部に子のクローンセットが含むコードクローンが存在するコードクローンを抽出する．
- 抽出されたコードクローンを，それぞれの子のコードクローンとまとめると，その集合はまとめられたクローンセットである．

クローンセットの集合に含まれる全ての親子関係のうち，その子が親となる親子関係を持たない親子関係から順にこの方法によってまとめられたクローンセットを求める処理を親子関係の解決と呼ぶ．この処理は Baxter の方法による検出結果に，求められたクローンセットを重複なく加えたクローンセットの集合を出力する．この集合を親子関係を解決したクローンセットの集合と呼ぶ．親子関係を解決したクローンセットの集合は，サブルーチン化に適したクローンセットの集合である．Baxter の方法による検出結果から得られた，子の一致を無視したクローンセットは，子の一致を無視してサブルーチン化する際にサブルーチン定義にまとめる共通部分を表しているため，このクローンセットは破棄せず残す．

3.5 提案手法の実装

提案手法の実装には C#を使用した．この実装は 1 つ以上のソースファイルを入力で受け取る．入力のソースファイルは C#で書かれたソースファイルを対象とする．構文解析器は Microsoft の提供する C#のコンパイラライブラリ Roslyn を使用した [11].

4. 提案手法の確認と考察

4.1 提案手法の確認

提案手法を用いてソースコードからコードクローンを検出

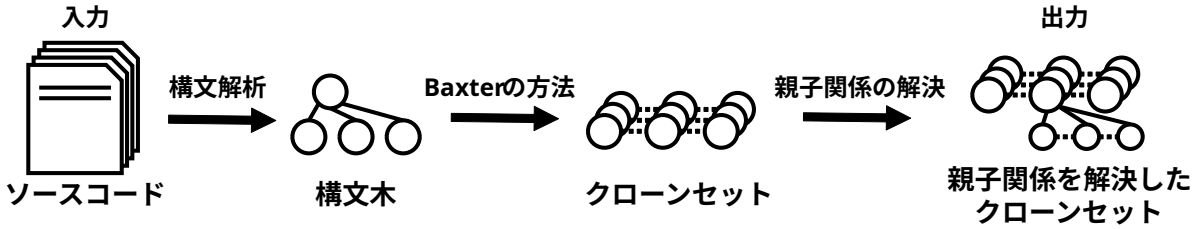

図 1　提案手法の概観

コード 7　制御構造の内部を無視すると検出できるコードクローン

```
int rc;

do
{
    rc = operation(arg1, arg2, arg3, arg4);
}
while (rc == -1);

return rc;
```

```
int rc;

do
{
    rc = operation(arg1, arg2, arg3);
}
while (rc == -1);

return rc;
```

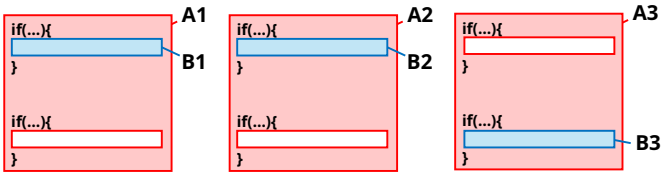

図 2　2 つに分かれたコードクローンの例

し，サブルーチン化に適したコードクローンが検出されているか確認する．既存のコードクローン検出器 NIL と提案手法の両方で同じソースコードに対してコードクローンを検出する．両検出器による検出結果を，互いに他方の検出器でも検出されたか否かで分類する．NIL はコードクローンの最小行数を 6 行，一致する構文要素の数を 10 に設定している．

提案手法は検出するコードクローンの最短行数および一致する構文木のノードの最低数に下限を設けない．検出されたコードクローンをその行数と一致するノードの数でさらに分類して，検出に最適な最短行数や最小ノード数のパラメータを検討する．

提案手法と NIL はコードクローンの検出単位が異なる．提案手法は文のシーケンス単位でコードクローンを検出する．一方 NIL はメソッド単位で検出する．検出の単位が異なるため，NIL がコードクローンを検出したメソッド内で提案手法がコードクローンを検出した場合，NIL と提案手法はそのメソッドで同じくコードクローンを検出したとする．NIL は 2 つのコードクローンからなるクローンペアの集合を出力するため，NIL で検出され提案手法に一致があるクローンペアの数と提案手法で検出されかつ NIL の検出結果と一致するクローンセットの数は異なる．

この確認のために，GitHub 上の C#で書かれたリポジトリ 49 件をコードクローン検出の対象として使用した．総ファイル数は 10,676 個，ソースコードの総行数は 1,332,110 行であった．

4.2　検 出 結 果

分類ごとに検出されたクローンセット，クローンペアの数を表 1 に示す．NIL が検出したクローンペアのうち約 77% が提案手法でも同様に検出できた．提案手法が検出したクローンセットのうち，NIL でも同様にクローンペアが検出されたのは約 13% であった．

4.3　考　　察

提案手法が検出し，NIL が検出しないコードクローンの例をコード 8 に示す．このコードクローンは一致したノードの数は 24，コードクローンの行数は 12 行である．一行ずつ比較すると，完全に一致する行はハイライトされた 7 行のみである．他の行は if 文の内部が異なっていたり，改行位置の違い，条件式の違いを含んでおり，完全には一致しない．行ごとに文字列を比較する方法では，完全に一致しない行が増えると，コードクローンが検出されづらくなる．抽象構文木を使う方法を選択したことで，改行位置の違いによって検出ができなくなる問題が回避される．加えて，制御構造の内部を無視すれば，このコード片は互いに一致する．特に 1 行目は返り値に関する記述が異なっているが，この部分を関数オブジェクトの呼び出し結果に書き換えてサブルーチン化を図る方法も考えられる．この記述が仮に双方で大きく異なる場合は関数オブジェクトの呼び出しに書き換える方法はより有効である．この結果から，提案手法はコード片間の違いの影響を受けずに，サブルーチン化に適したコードクローンを検出できていると確認できる．

表 1　分類された検出結果

分類	クローンセット (ペア) の数
NIL で検出，提案手法に一致なし	5,783
NIL で検出，提案手法に一致あり	19,867
提案手法で検出，NIL に一致なし	122,287
提案手法で検出，NIL に一致あり	16,283

コード 8　提案手法が検出し，NIL が検出しないコードクローンの例

```
if (list1 == null) return list2 == null;
if (list2 == null) return false;
int n = list1.Count; if (list2.Count != n) return false;
for (int i = 0; i < n; i++)
{
    TypeNode t1 = list1[i];
    TypeNode t2 = list2[i];
    if (null == (object)t1 || null == (object)t2) return
        false;
    if (t1 == t2) continue;
    if (!t1.IsEquivalentTo(t2)) return false;
}
return true;
```

```
if (list1 == null) return list2 == null || list2.Count ==
    0;
if (list2 == null) return false;
int n = list1.Count;
if (n != list2.Count) return false;
for (int i = 0; i < n; i++)
{
    TypeNode tp1 = list1[i];
    TypeNode tp2 = list2[i];
    if (tp1 == null || tp2 == null) return false;
    if (tp1 != tp2 && !tp1.IsEquivalentTo(tp2)) return
        false;
}
return true;
```

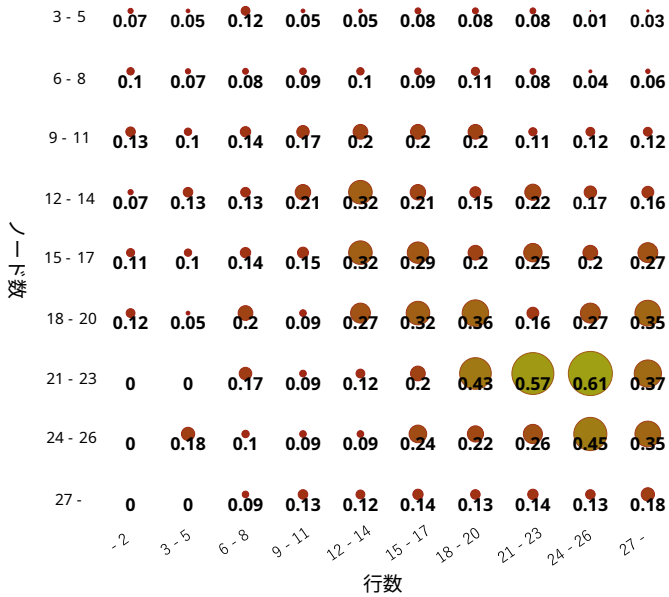

図 3　提案手法の検出結果が NIL の検出結果と一致する割合

提案手法が検出したコードクローンのうち，NIL の検出結果と一致したコードクローンの割合を図 3 に示す．行数およびノード数ごとでコードクローンを分類したうえで，検出されたコードクローンのうち，NIL の検出結果と一致したコードクローンが占める割合を円の大きさで示している．

行数が 8 行以下あるいはノード数が 8 個以下のコードクローンは NIL の検出結果と一致する割合が最大でも 0.2 にとどまる．提案手法が検出するコードクローンの最低行数は 9 行以上，最低ノード数は 9 個以上に設定することが望ましいと考えられる．

5. 妥当性への脅威

NIL と提案手法が検出するコードクローンの単位が異なっており，結果の比較が難しい．比較の際は，提案手法は同メソッド内のコードクローンを検出するが，NIL は同メソッド内のコードクローンを検出しないことに注意が必要である．

6. おわりに

本研究では，サブルーチン化に適したコードクローンの検出手法を提案した．提案手法に則って試作した検出器と既存のコードクローン検出器 NIL と検出結果を比較し，提案手法がサブルーチン化に適したコードクローンを検出することを確認した．今後は，検出されたコードクローンからサブルーチン定義を生成し，コード片をサブルーチン呼び出しに書き換える処理の自動化にも着手する．

謝辞　本研究は JSPS 科研費 24H00692, 21K18302, 21H04877, 23K24823，22K11985 の助成を受けた．

文　献

[1] 井上克郎，神谷年洋，楠本真二．コードクローン検出法．コンピュータソフトウェア，Vol.18，No.5，pp. 529–536，2001.

[2] 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. 443–456, 2011.

[3] Yoshida, N., Ishizu, T., Edwards III, B. and Inoue, K., How slim will my system be? estimating refactored code size by merging clones. Proceedings of the 26th Conference on Program Comprehension, pp. 352 – 360, 2018.

[4] T Nakagawa and 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, 2021.

[5] C. Roy and J. Cordy. A Survey on Software Clone Detection Research. School of Computing TR No.2007–541, 2007.

[6] P. Wang, J. Svajlenko, Y. Wu, Y. Xu and C. K. Roy, CCAligner: A Token Based Large-Gap Clone Detector. 2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE), Gothenburg, pp. 1066-1077, 2018. keywords: Cloning;Tools;Detectors;Software;Computer science;Software engineering;Indexes;Clone Detection;Large-gap Clone;Evaluation,

[7] J. R. Cordy and C. K. Roy, The NiCad Clone Detector. 2011 IEEE 19th International Conference on Program Comprehension, Kingston, ON, pp. 219-220, 2011.

[8] M. Wu, P. Wang, K. Yin, H. Cheng, Y. Xu and C. K. Roy, LVMapper: A Large-Variance Clone Detector Using Sequencing Alignment Approach. IEEE Access, vol. 8, pp. 27986-27997, 2020.

[9] I. D. Baxter, A. Yahin, L. Moura, M. Sant' Anna, and L. Bier. Clone Detection Using Abstract Syntax Trees. Proc. of International Conference on Software Maintenance, pp. 368–377 1998.

[10] Semantic Designs, http://www.semanticdesigns.com/Products/Clone/. 2024 年 9 月 20 日閲覧

[11] Microsoft, Roslyn. 2024 年 9 月 15 日閲覧. https://learn.microsoft.com/en-us/dotnet/csharp/roslyn-sdk/