Vol.2013-SE-182 No.29
2013/10/25

ソースコードを構成する処理ブロックの特徴に着目したコードクローン推定技術

垣谷広輝[†1]　安藤優作[†1]　平山雅之[†1]　菊地奈穂美[†2]

本論ではプログラムの保守性と信頼性の観点から開発の早期に，作成途中のソフトウェアなど未完成（コンパイル完了前）コード内のコードクローンを検出する方法を提案する．提案手法ではソースコードの行数や文字数などの形状を表す情報からコードクローンを効率的に検出する．この手法は，ソースコードの形状情報のみを用いるため，未完成プログラム内のコードクローン検出にも有効であり，また解析対象の言語を問わないといった利点を持つ．

Code clone candidate detection technique with source code block feature

KOUKI KAKIYA[†1]　YUSAKU ANDO[†1]　MASAYUKI HIRAYAMA[†1]
NAHOMI KIKUCHI[†2]

This paper proposes a code clone detection technique. In the technique, some information for source code shape image - such as number of lines and number of characters in each code blocks are used and code clone candidate blocks can be identified with high efficiency. This paper shows a basic idea of the technique and also shows a simple case study of the technique. The technique can be applicable to incomplete code as well as complete code. The proposed technique does not use grammatical analysis, so it can be applicable to any code by various program languages.

1. はじめに

近年，システム開発は既存資産に機能を追加する方式が主流である．その中でもソースコードの部分的コピー&ペーストによって発生するコードクローンは，保守性と信頼性の観点から問題となることがある．コード中でバグが存在する部分がコードクローンだった場合，そのクローンの元となったコード断片にもバグがある可能性があり，チェックが必要となったり，場合によっては各コピー元やコピー先の箇所を個別に修正しなければならないといった問題が発生する．結果としてコードクローンの存在は工数増加の原因となる．そのため，コード中のコードクローンを検出し除去することが求められるが，コードの一部を除去することになるため，できるだけ開発の早い段階で確認し，除去する方が効率的である．本研究ではコードクローンを早期に検出する方法を提案する．

コードクローンとは，ソースコード内で既存のコードを引用して作成された類似，または一致する処理が記述されたコード部分のことをいう．コードクローン検出の既存ツール[1][2][3][7][8]の多くでは構文解析を用いて検出を行っている．この方法の場合，対象となるソースコードはプログラム言語の文法に従っていることが必要であり，開発初期の未完成（コンパイル完了前）コードや文法に従っていないバグを持つコードなどにおけるクローンの検出は難しい．この問題を解決し，未完成コードやバグコード中のクローンを検出するために，本研究ではソースコードを処理の塊であるコードブロックに分割し，その形状を用いてコードクローンを効率的に検出する方法を提案する．コードブロックの特徴は文字数と行数であらわされるため，ソースコードの文法的な正しさに関係なく，クローンを発見することが可能になると考えられる．

2. 従来技術

コードクローンを検出する既存ツールの多くは構文解析を使用している．構文解析とは静的なコード解析手法の一つであり，ソースコードを字句解析したものを用いて，ソースコードの文法を判別することが出来る．字句解析はソースコードをその言語において意味のある最小単位である字句(トークンとも言う)に分割することである．その後，構文解析において字句を基にして構文木を作成し，その言語の構文規則にあっているかを確認するといった処理を行う．

例えば，字句解析の例を挙げると C 言語では，字句の種類は，int, double, for, while などの予約語，関数名，変数名などの識別子, =, >, - などの演算子, {, }, ; などの記号，数値，文字，文字列などの定数に分かれる．

次に構文解析では構文規則から字句解析した文を基にして構文木を作成する．例えば，if 文ではまず条件式，次にその条件を満たすときの処理，満たさなかった時の処理，

†1 日本大学
Nihon University
†2 沖電気工業株式会社
Oki Electric Industry Co., Ltd.

といった順番で構成されている. さらに詳細部分を見ると，条件式は比較する対象が 2 つと比較演算子，処理部分は数式や，引数を持つ予約語などといった字句によって構成されている．このような構文規則と字句を照らし合わせることで，コードの処理を文法から知ることができる．

なお，字句解析を行なうプログラムのことをレキシカルアナライザ(lexical analyzer)，前述の様に構文解析を行なうプログラムのことを パーサ(parser)と呼ぶ.

構文解析でクローン検出を行う場合，この構造木を比較することにより，クローンを検出することができる[1][2]. しかし，構文解析が使用できるのはあくまでも文法を基にコードが解析できる場合のみであり，文法的に正しくないバグを含んでいる場合やコードに一行処理を付け加えることで処理が若干変わった場合などではクローンの検出が難しくなる.

既存研究としては，井上らの「AGM アルゴリズムを用いたギャップを含むコードクローン情報の生成」[5]や村上らの「Smith-Waterman アルゴリズムを利用したギャップを含むコードクローン検出」[6]が挙げられる.

井上らの研究ではギャップを含むクローン，つまりコピー&ペースト後に後に文の追加・削除・変更が行われているものを検出する方法を提案している. この手法では AGM アルゴリズムを使用しているが，検出したクローンのうち約 8 割が保守性には関係のない部分であったことが報告されている．その問題点に関しては，宮崎らがクローン検出の際にフィルタリングすることで，保守性に関係のあるクローンのみを抽出した[4]．これらの手法では AGM アルゴリズムを適応させるグラフを作成するために処理が一致するコードクローンの情報を必要としており，そのために構文解析を使用している.

また, 村上らの研究では Smith-Waterman アルゴリズムを利用し，クローンの検出を行っている．この手法は構文解析と字句解析を行い，それをハッシュ値に変換したものを比較している．その手順の中でギャップの位置情報を利用することでギャップを含むクローンの検出を行い，クローンの検出精度を上げている.

上記の手法では，変更されたクローンも検出できるが文法的に正しくないコードに対しては検出ができない．そのため，本手法では文字数などのソースコードの形状面の特徴を用いて文法によらないクローンの検出を目指している.

3. 提案法

3.1 概要

提案法では，ソースコードの行数や文字数といったコードブロックの形状特徴に着目し, 以下の Step1～4 の処理によりコードクローンを検出する.

Step1: ソースコードを構成するブロックに分ける.
Step2: 各ブロックの形状を比較し，クローンの候補ブロックを特定する.
Step3: 上記のクローン候補ブロックを精査し，クローン確率が高いブロックを検出する.
Step4: コード内の特定文字の数を比較し，処理が類似するブロックを検出する.

この提案法の特徴を表 3.1 に示す.

表 3.1　提案手法の特徴

	提案手法
精度	構文解析と比較すると低い可能性が高い
対象コード	文字コードの 一致しているもの HTMLやXMLでも可
対象コードの完成度	未完成でも可
出力結果	コードクローン疑惑のあるものをリスト化

表 3.1 に示すように，提案法ではクローンの抽出に文字数及び行数を使用するため，形状的にクローンの可能性が高い物を発見することはできるが，処理の詳細内容が類似しているかは明確には判別できない．一方で，文字数及び行数のみで測定するため, 様々な言語に応用できる. また，コードクローンの検出精度を上げる方法として，後述の制御文によるブロック分けなど，特定の言語においては，文法を利用した検出方法も併用も可能と考えている．さらに提案法では，構文解析を利用しないため，文法的に完成していない未完成コードにおいても，クローンの検出が可能である.

提案法の持つ上記の特徴を考慮し，提案法ではクローン抽出の結果出力として，クローンの可能性がある処理部分のペアおよびグループをその類似度が高い順にリスト化したものを提示する形をとっている.

3.2 Step1 :ブロック分割

Step1 では，クローンの検出を効率的に行うために予めソースコードを構成している処理の塊をコードブロックとして検出する.

ソースコードを構成するブロックに分ける方法として，（A）改行に着目した方法，と（B）制御文に着目した方法の 2 つを場合により使い分けることとする.

（A）改行によるブロック分割と比較

ソースコードでは多くの場合，一処理を構成するコードブロック部分は改行文字により分けられている．そのため，改行文字を用いてブロックとして分割する方法を採用する．具体的な方法としては，ソースコードの中で空白改行を見つけた場合，そこをブロックの区切りとする．これをコードの終わりまで続けることで，一つのコードを複数のブロックの塊に分割する．なお，2 行以下のブロックは printf 文やプロトタイプ宣言により作られたブロックである可能性が高いため，ブロック分けは行うがクローンブロックの候補とはしないものとする．図 3.1 が実際に改行を境にブロック分けしたものとなる．

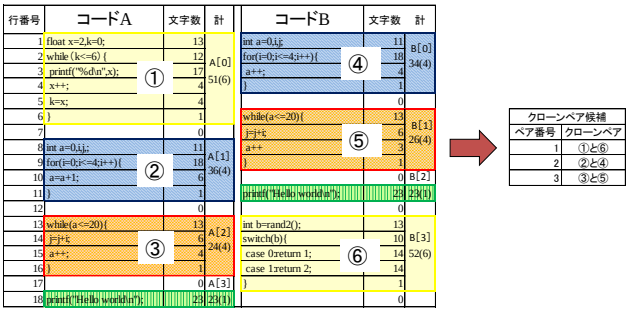

図 3.1　ブロック抽出(改行)

図 3.1 ではコード A(左側)，コード B(右側)それぞれにブロックが 3 つずつ検出されている．各ブロックについてはブロック①は総文字数 51 行数 6, ブロック②は総文字数 36 行数 4，ブロック③は総文字数 24 行数 4，ブロック④は総文字数 34 行数 4，ブロック⑤は総文字数 23 行数 4，ブロック⑥は総文字数 52 行数 5 となる．なお，空白文字(スペース)やコメント文についてもその処理ブロックの特徴として考え，文字数にカウントする．

（B）制御文によるブロック分割と比較

上記（A）の手法で採用した改行に着目したブロック分割手法は改行がなければ適用できない．改行を用いたブロック分割が行われない場合には繰り返し文や分岐文などの制御文を認識し，ブロック分割する方法を提案する．制御文の書き方はプログラム言語によって若干の差異がある．ここでは，C や Java で書かれたプログラムを念頭に置き，それらの制御文では必ず”{”,”}” の 2 つがペアとして使用されていることを利用する．よって,”{”,”}” を区切りとして検出し，ブロック分割を行う．実際にブロック分割したものが図 3.2 となる．図 3.2 のブロック抽出では制御文によるブロック分けのため，変数宣言などの行を除外してブロックとしている．そのため，ブロック①は総文字数 38 行数 5，ブロック②は総文字数 25 行数 3，ブロック④は総文字数 23 行数 3, ブロック⑥は総文字数 39 行数 4 となる．ブロック③と⑤については，空白改行に挟まれる部分にちょうど制御文が収まる形になっているため，改行を用いたブロック分割でも制御文を用いたブロック分割でも，どちらの方法でも同じようにブロックを検出することができる．

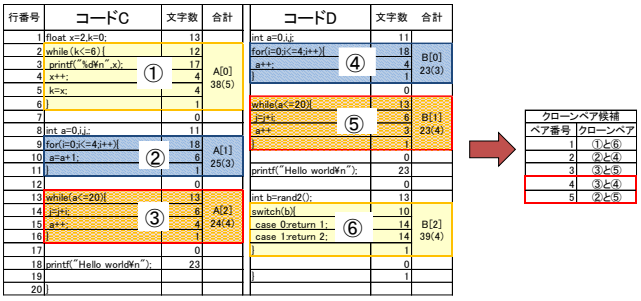

図 3.2　ブロック抽出(制御文)

3.3 ブロック比較

3.3.1 Step2：クローン候補ブロックの特定

Step1 のブロック分割で抜き出された各ブロックについて, ブロック内の総文字数, 行数を用いて形状評価を行い，各ブロック同士での類似度を比較する．これにより，Step2 ではクローンの候補となるブロックを形状から特定する．類似度として，二つのブロックを比較した際の総文字数，行数の差を求める．この差が総文字数±30%，行数±10%以内に収まっている場合は二つのブロックは形状が似ていると判定する．

例えば, 図 3.1 の左上の③ブロックでは総文字数 24 文字，行数 4 行という情報があることになる．これに対して，コード B を構成する④~⑥の各ブロックと比べると⑤ブロックが総文字数 26 文字，行数 4 行であり，行数±30%，総文字数±10%の差に収まっており，③と⑤がクローン候補のペアとして（ペア番号 3）抽出できる．このようにして，図 3.1 に示すようにクローン候補のペアが検出できる．一方で，図 3.2 は制御文に着目したブロック分割を例示したものであるが，クローン候補のペア番号 4 と 5 は図 3.1 の改行を使用した手法では検出されなかったクローン候補である．

3.3.2 Step3：クローン候補ブロックの精査

Step2 のクローンブロック候補の特定により抽出されたクローンブロック候補を対象に，より詳細な特徴の比較を行うことでクローンブロックの特定を行う．詳細な特徴比較の方法として Step3 ではブロック構成行ごとの文字数を評価し，クローンブロック候補の特定を行う．

Step2 でクローンである可能性が高いと判定されたブロックを対象にして，1 行あたりの文字数をそれぞれ比較する．これにより，Step2 で抽出されたものの中で構成が全く違っているにも関わらず，偶然一致してしまったものをクローンブロックとしないようにすることができる. 図 3.3 では 1 行につき配列を 1 つ用意し，その配列番号が同じも

の同士を比較する．例えば A[0]と B[0]，A[1]と B[1] といった形で中身を比較する． 全配列の一致率を平均して±60%を超えた場合はクローンである可能性が非常に高い組として検出する．検出した行は図 3.3(a)に示すように，一致したものに色を付けて表している．今回使用した例ではコード B の方で a++の後にセミコロンがないため，文法としては誤ったものとなっており，文字数も変化しているが，他の行に対する配列の比較は一致している．そのため，変更されたクローンである確率が高いと認識できる．

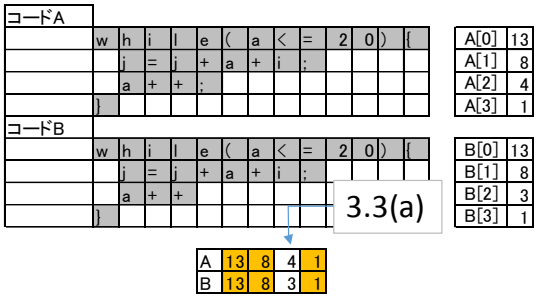

図 3.3　1 行ごとの文字数比較(行数一致)

ここで，ブロックの行数が違っていた場合には同じ行番号同士で比較するパターンだけでなく，行をずらした形でのパターンの比較も行う．その中で一致率が 60%以上のパターンが見つかれば，その二つのブロックをクローンブロックであると判定する．図 3.4 に行数が異なる場合の 1 行ごとの文字数比較を行った例を示す．図 3.4(a)が比較結果である．この場合はどちらのパターンも一致率が 60%未満のため，クローンではないと判定する．

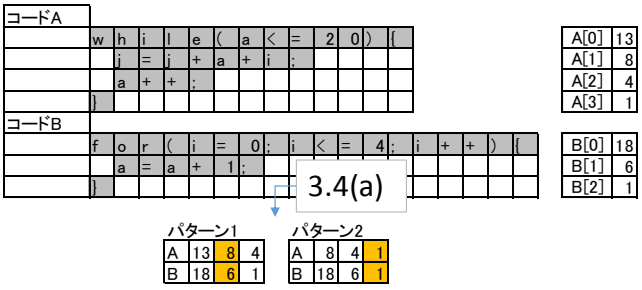

図 3.4　1 行ごとの文字数比較(行数不一致)

クローンブロック候補の精査で用いた一致率の計算についてここで示しておく．図 3.5 に，図 3.3 で使用した文字数比較の一致率算出を示す．行番号ごとにそれぞれの文字数差と許容誤差を比較し，比較対象がその範囲内であれば一致とする．

ここで，対象となる両コードの一行当たりの文字数を，それぞれ平均した値の±10%を許容誤差とする．もし，コード A の文字数が 12, コード B の文字数が 18 であるなら，（12+18）÷2×0.1＝1.5 となり，四捨五入し，±2 文字を許容誤差とする．

実際に一致率を算出する例として，図 3.5 を示す．これは図 3.3 の比較を表として表したものである．図 3.5 の行番号 1 を見ると，コード A，B ともに文字数は 13 文字である．そのため，文字誤差は 0，許容誤差が±1 なので，行番号 1 は一致と判定される．

一致した行が全行番号に対してどの程度あるかを一致率として求める．図 3.5 の例ではブロック同士の一致率は 4 行中 3 行が一致しているため 75%となり，閾値とした±60%以内に収まっている．従って，この二つのブロックはクローンである確率が高いと判定される．また，図 3.4 のパターンに分けた形式でも考え方は同様に行い，一致率を算出する．

行番号	1	2	3	4
コードA	13	8	4	1
コードB	13	8	3	1
文字数差	0	0	1	0
許容誤差	±1	0	0	0
一致判定	○	○	×	○

$$一致率=\frac{一致行数}{全行数}=\frac{3}{4}=0.75=75\%$$

○:一致　×:不一致

図 3.5　行数比較の例

3.4 Step4：ブロック特徴評価

さらに，クローン判定を詳細に行うため，Step4 ではソースコード中の特定文字を利用して判定した処理の特徴からブロックの比較を行う．

ある言語における特定の処理にはよく使われる文字が存在する．その文字を特定文字として測定することで，ブロックの処理的類似点を見つけることが可能になる．

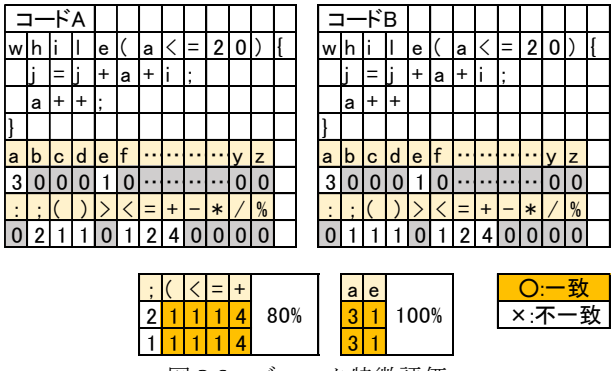

図 3.6　ブロック特徴評価

図 3.6 は C 言語の繰り返し文における文字毎の使用文字数をカウントした例である．コード A に対し，コード B では計算式などの終端に記述する‘;’が一つ不足している．

ブロックの特徴評価については主に‘+’などの演算子や‘{’などの記号といったアルファベット以外の文字に着目する．図 3.6 の場合，処理は同一ではあるが，片方が文法的に正しくないといったパターンになっている．

このとき，それぞれの処理における演算子及び記号の各

種毎の文字数を調べると‘;’以外の‘(’,‘)’,‘<’,‘=’,‘+’については数が一致している．このことから特定の文字の数が同じものについては類似した処理となる可能性があると考えられる．例の場合は 2 つとも加算処理が多いため，‘+’の演算子の数が多く検出されている．

また，ある言語においては特定のアルファベットを使用する文，C 言語の場合は while などがあるため，特定処理同士を比較する場合，特定アルファベットの文字数も一致すると考えられる．

しかし，演算子や記号は図 3.6 のように記述ミスにより，アルファベットは変数や文章などの自由に定義できる文字列の存在により，文字数が一致しなくなってしまう可能性がある．そのため，ここでは比較できる文字数のうち，ある一定の割合が一致すれば，同じ処理である，すなわちクローンである可能性が高いと判定する．

4. ケーススタディ

4.1 ケーススタディの対象

本手法が実際のソースコードにも適用可能であることを確認するため，机上実験を行った．クローン検出の対象となるコードは異なる 2 つのコードを使用する．本実験ではオープンソースである BorlandC++Compiler 5.5 のソースコード[9]にある accum.cpp(図 4.1)と adj_diff.cpp(図 4.2) を使用する．

4.2 Step1：ブロック分割

まず，コードを空白改行によってブロック分割する．図 4.3 は accum.cpp の一部を抜粋したものである．

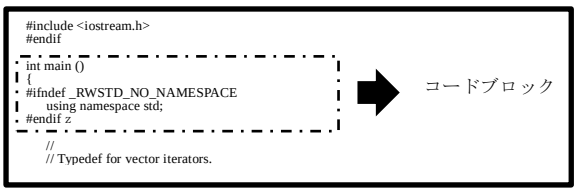

図 4.3　ブロック分割

図 4.3 は図 4.1 に示した accum.cpp の 51 行目の#endif の次の空白改行と 57 行目の#endif の次の空白改行で区切られたブロックを抜き出したものである．コードの始めから走査していき，空白改行を発見した場合，そこまでのコードをブロック 1 とし，次の行をブロック 2 の開始行とする．

```
#include "stlexam.h"
#pragma hdrstop
/**************************************************************************
 *
 * accum.cpp - Example program for accumulate. See Class Reference Section
 *
 **************************************************************************
 *
 * (c) Copyright 1994, 1998 Rogue Wave Software, Inc.
 * ALL RIGHTS RESERVED
 *
 * The software and information contained herein are proprietary to, and
 * comprise valuable trade secrets of, Rogue Wave Software, Inc., which
 * intends to preserve as trade secrets such software and information.
 * This software is furnished pursuant to a written license agreement and
 * may be used, copied, transmitted, and stored only in accordance with
 * the terms of such license and with the inclusion of the above copyright
 * notice.  This software and information or any other copies thereof may
 * not be provided or otherwise made available to any other person.
 *
 * Notwithstanding any other lease or license that may pertain to, or
 * accompany the delivery of, this computer software and information, the
 * rights of the Government regarding its use, reproduction and disclosure
 * are as set forth in Section 52.227-19 of the FARS Computer
 * Software-Restricted Rights clause.
 *
 * Use, duplication, or disclosure by the Government is subject to
 * restrictions as set forth in subparagraph (c)(1)(ii) of the Rights in
 * Technical Data and Computer Software clause at DFARS 252.227-7013.
 * Contractor/Manufacturer is Rogue Wave Software, Inc.,
 * P.O. Box 2328, Corvallis, Oregon 97339.
 *
 * This computer software and information is distributed with "restricted
 * rights."  Use, duplication or disclosure is subject to restrictions as
 * set forth in NASA FAR SUP 18-52.227-79 (April 1985) "Commercial
 * Computer Software-Restricted Rights (April 1985)."  If the Clause at
 * 18-52.227-74 "Rights in Data General" is specified in the contract,
 * then the "Alternate III" clause applies.
 *
 **************************************************************************/

#include <compnent.h>
#include <numeric>        // For accumulate.
#include <vector>         // For vector.
#include <functional> // For multiplies.

#ifdef _RW_STD_IOSTREAM
#include <iostream>
#else
#include <iostream.h>
#endif

int main ()
{
#ifndef _RWSTD_NO_NAMESPACE
    using namespace std;
#endif

    //
    // Typedef for vector iterators.
    //
    typedef vector<int,allocator<int> >::iterator iterator;
    //
    // Initialize a vector using an array of integers.
    //
    int d1[10] = {1,2,3,4,5,6,7,8,9,10};
    vector<int,allocator<int> > v1(d1+0, d1+10);
    //
    // Accumulate sums and products.
    //
    int sum  = accumulate(v1.begin(), v1.end(), 0);
    int prod = accumulate(v1.begin(), v1.end(), 1, multiplies<int>());
    //
    // Output the results.
    //
    cout << "For the series: ";
    for(iterator i = v1.begin(); i != v1.end(); i++)
            cout << *i << " ";

    cout << " where N = 10." << endl;
    cout << "The sum = (N*N + N)/2 = " << sum << endl;
    cout << "The product = N! = " << prod << endl;

    return 0;
}
```

図 4.1　accum.cpp

```
#include "stlexam.h"
#pragma hdrstop
/**************************************************************************
 *
 * adj_diff.cpp - Example program for adjacent_difference.
 *                See Class Reference Section
 *
 **************************************************************************
 *
 * (c) Copyright 1994, 1998 Rogue Wave Software, Inc.
 * ALL RIGHTS RESERVED
 *
 * The software and information contained herein are proprietary to, and
 * comprise valuable trade secrets of, Rogue Wave Software, Inc., which
 * intends to preserve as trade secrets such software and information.
 * This software is furnished pursuant to a written license agreement and
 * may be used, copied, transmitted, and stored only in accordance with
 * the terms of such license and with the inclusion of the above copyright
 * notice.  This software and information or any other copies thereof may
 * not be provided or otherwise made available to any other person.
 *
 * Notwithstanding any other lease or license that may pertain to, or
 * accompany the delivery of, this computer software and information, the
 * rights of the Government regarding its use, reproduction and disclosure
 * are as set forth in Section 52.227-19 of the FARS Computer
 * Software-Restricted Rights clause.
 *
 * Use, duplication, or disclosure by the Government is subject to
 * restrictions as set forth in subparagraph (c)(1)(ii) of the Rights in
 * Technical Data and Computer Software clause at DFARS 252.227-7013.
 * Contractor/Manufacturer is Rogue Wave Software, Inc.,
 * P.O. Box 2328, Corvallis, Oregon 97339.
 *
 * This computer software and information is distributed with "restricted
 * rights."  Use, duplication or disclosure is subject to restrictions as
 * set forth in NASA FAR SUP 18-52.227-79 (April 1985) "Commercial
 * Computer Software-Restricted Rights (April 1985)."  If the Clause at
 * 18-52.227-74 "Rights in Data General" is specified in the contract,
 * then the "Alternate III" clause applies.
 *
 **************************************************************************/

#include <numeric>          // For adjacent_difference.
#include <vector>           // For vector.
#include <functional>       // For multiplies.

#ifdef _RW_STD_IOSTREAM
#include <iostream>
#else
#include <iostream.h>
#endif

int main ()
{
#ifndef _RWSTD_NO_NAMESPACE
    using namespace std;
#endif
    //
    // Initialize a vector of ints from an array.
    //
    int arr[10] = {1,1,2,3,5,8,13,21,34,55};
    vector<int,allocator<int> > v(arr+0, arr+10);
    //
    // Two uninitialized vectors for storing results.
    //
    vector<int,allocator<int> > diffs(10), prods(10);
    //
    // Calculate difference(s) using default operator (minus).
    //
    adjacent_difference(v.begin(),v.end(),diffs.begin());
    //
    // Calculate difference(s) using the times operator.
    //
    adjacent_difference(v.begin(), v.end(), prods.begin(), multiplies<int>());
    //
    // Output the results.
    //
    cout << "For the vector: " << endl << "          ";
    copy(v.begin(),v.end(),ostream_iterator<int,char,char_traits<char> >(cout," "));
    cout << endl << endl;

    cout << "The differences between adjacent elements are: " << endl << "        ";
    copy(diffs.begin(),diffs.end(), ostream_iterator<int,char,char_traits<char> >(cout," "));
    cout << endl << endl;

    cout << "The products of adjacent elements are: " << endl << "          ";
    copy(prods.begin(),prods.end(), ostream_iterator<int,char,char_traits<char> >(cout," "));
    cout << endl;

    return 0;
}
```

図 4.2　adj_diff.cpp

accum			
ブロックNO	行数	総文字数	行番号
A[1]	40	2093	1～40
A[2]	4	145	42～45
A[3]	5	85	47～51
A[4]	5	78	53～57
A[5]	20	593	59～78
A[6]	3	144	80～82

adj_diff			
ブロックNO	行数	総文字数	行番号
B[1]	41	2129	1～41
B[2]	3	140	43～45
B[3]	5	89	47～51
B[4]	28	848	53～80
B[5]	3	201	82～84
B[6]	3	186	86～88

図 4.4　ブロック分割結果

これを繰り返すことでコードのブロック分割を行う.このようにしてブロック分割した結果を図 4.4 に示す.

ブロック分割した結果，accum.cpp は A[1]から A[6]，adj_diff.cpp は B[1]から B[6]と，ともに 6 つのブロックに分割された．図 4.3 でブロック分割の例に挙げたブロックは A[4]のブロックであり，このブロックは accum.cpp の 53 行目から 57 行目に位置する行数 5，総文字数 78 のブロックである．なお，今回のケーススタディでは各行の改行文字はカウント(2 文字)，ファイル終端の EOF(End of File)はカウントしていない.

ブロックNO	行数	文字数	判定
A[1]	40	2093	
B[1]	41	2129	○
誤差[%]	2.439	1.6909	
A[1]	40	2093	
B[2]	3	140	×
誤差[%]	92.5	93.311	
A[1]	40	2093	
B[3]	5	89	×
誤差[%]	87.5	95.748	
A[1]	40	2093	
B[4]	28	848	×
誤差[%]	30	59.484	
A[1]	40	2093	
B[5]	3	198	×
誤差[%]	92.5	90.54	
A[1]	40	2093	
B[6]	3	186	×
誤差[%]	92.5	91.113	
A[2]	4	145	
B[2]	3	140	○
誤差[%]	25	3.4483	
A[2]	4	145	
B[3]	5	89	×
誤差[%]	20	38.621	
A[2]	4	145	
B[4]	28	848	×
誤差[%]	85.714	82.901	
A[2]	4	145	
B[5]	3	198	○
誤差[%]	25	26.768	
A[2]	4	145	
B[6]	3	186	○
誤差[%]	25	22.043	

ブロックNO	行数	文字数	判定
A[3]	5	85	
B[3]	5	89	○
誤差[%]	0	4.4944	
A[3]	5	85	
B[4]	28	848	×
誤差[%]	82.143	89.976	
A[3]	5	85	
B[5]	3	198	×
誤差[%]	40	57.071	
A[3]	5	85	
B[6]	3	186	×
誤差[%]	40	54.301	
A[4]	5	78	
B[4]	28	848	×
誤差[%]	82.143	90.802	
A[4]	5	78	
B[5]	3	198	×
誤差[%]	40	60.606	
A[4]	5	78	
B[6]	3	186	×
誤差[%]	40	58.065	
A[5]	20	593	
B[5]	3	198	×
誤差[%]	85	66.61	
A[5]	20	593	
B[6]	3	186	×
誤差[%]	85	68.634	
A[6]	3	144	
B[6]	3	186	○
誤差[%]	0	22.581	

○:クローン候補とする
×:クローン候補ではない

図 4.5　候補ブロック検出結果

4.3 ブロック比較

4.3.1 Step2：候補ブロックの特定

Step1 で分割したブロックについて文字数･行数の比較を行う．ここでは，比較する 2 つのブロックの文字数差が±30%，かつ行数差が±25%の場合，クローンの候補ブロックとする．文字数差・行数差の計算としては比較する数値の大きい方を分母，小さい方を分子として計算を行い，100 を乗じることでパーセント表示に直す．ここで採用した文字数差と行数差の基準は，机上実験のための暫定値である．図 4.5 に比較結果を示す.

ここでは accum.cpp の A[1]から A[6]の 6 ブロックと，adj_diff.cpp の B[1]から B[6]の 6 ブロックをペアにして比較した．その結果，文字数，行数の差が基準内に収まるブロックのペア 6 個(図 4.5 中 ○判定のもの）を抽出できた.

4.3.2 Step3：候補ブロックの精査

次に Step2 で抽出したクローン候補ブロックの 6 つのペアに対して，行ごとの比較を行う．なお，ここではクローン判定に利用する一致率（3.3.2 参照）の基準値は 60%を採用し，結果を図 4.6 に示す.

このうち，A[3]と B[3]の比較では全 5 行中の 5 行目のみ文字数が一致しなかったが，この場合の一致率は 4/5=80%となり基準を満たしている．一方，A[6]と B[6]のペアでは文字数が一致する行がないためクローンの可能性は低いと判定する．また，比較するブロックの行数が異なる場合には，比較する行をずらして，複数回ブロック比較を行い一致率を求める．例えば，A[1]と B[1]のペアでは，40 行と 41 行の比較のため，まず，A[1]の 1 行目～40 行目と B[1]の 1 行目～40 行目を比較する．次いで A[1]の 1 行目～40 行目と B[1]の 2 行目～41 行目を比較し，どちらかの比較で一致率が基準値を満たした場合にクローンの可能性が高いと判定する.

Step3 での比較を行った結果，Step2 で抽出した 6 つのペアの中から,(A[1],B[1])，(A[3],B[3])の 2 組がクローンの可能性が高いと判定できた.

4.4 Step4：特定文字

最後にコードの特徴を反映する特定文字を使用したブロックの比較を行う．この工程は Step2 においてクローン候補となったペアに対して行い，使用する特定文字は暫定的に‘w’，‘<’，‘>’，‘=’，‘.’の五種類とした．文字ごとに数の比較を行い，一致する文字の種類数を計測する．一致した種類数を比較した文字の全種類数で割ることで一致率を求め，一致率が 80%を超えたものはクローンの可能性が高いと判定する．なお，今回は下記の理由から，前述の 5 つの文字を判定対象として利用した.

- w　コード中で繰り返し文として利用される while の中で利用されるが,その他の一般的な処理中ではあまり利用される機会が少ない特徴的な文字である.
- <,>　不等号は条件分岐中の比較演算やポインタのメンバ指定などの際に利用され，コード内の処理に依存

ブロックNO	行数	A[1]文字数	B[1]文字数	判定	結果
A[1]:B[1] パターン1	1	22	23	×	×
	2	17	18	×	
	3	77	78	×	
	4	4	4	○	
	5	76	61	×	
	6	4	47	×	
	7	78	4	×	
	8	4	78	×	
	9	55	4	×	
	10	24	55	×	
	11	4	24	×	
	12	74	4	×	
	13	73	74	×	
	14	72	73	×	
	15	75	72	×	
	16	73	75	×	
	17	76	73	×	
	18	75	76	×	
	19	69	75	×	
	20	4	69	×	
	21	71	4	×	
	22	75	71	×	
	23	76	75	×	
	24	63	76	×	
	25	39	63	×	
	26	4	39	×	
	27	68	4	×	
	28	74	68	×	
	29	71	74	×	
	30	58	71	×	
	31	44	58	×	
	32	4	44	×	
	33	75	4	×	
	34	75	75	○	
	35	68	75	×	
	36	73	68	×	
	37	72	73	×	
	38	45	72	×	
	39	4	45	×	
	40	78	4	×	
	41		78		

ブロックNO	行数	A[1]文字数	B[1]文字数	判定	結果
A[1]:B[1] パターン2	1		23		○
	2	22	18	×	
	3	17	78	×	
	4	77	4	×	
	5	4	47	×	
	6	76	69	×	
	7	4	4	○	
	8	78	78	○	
	9	4	4	○	
	10	55	55	○	
	11	24	24	○	
	12	4	4	○	
	13	74	74	○	
	14	73	73	○	
	15	72	72	○	
	16	75	75	○	
	17	73	73	○	
	18	76	76	○	
	19	75	75	○	
	20	69	69	○	
	21	4	4	○	
	22	71	71	○	
	23	75	75	○	
	24	76	76	○	
	25	63	63	○	
	26	39	39	○	
	27	4	4	○	
	28	68	68	○	
	29	74	74	○	
	30	71	71	○	
	31	58	58	○	
	32	44	44	○	
	33	4	4	○	
	34	75	75	○	
	35	75	75	○	
	36	68	68	○	
	37	73	73	○	
	38	72	72	○	
	39	45	45	○	
	40	4	4	○	
	41	78	78	○	

判定
○:一致
×:不一致

結果
○:クローンと判定する
×:クローンではないと判定

ブロックNO	行数	A[2]文字数	B[2]文字数	判定	結果
A[2]:B[2] パターン1	1	23	54	×	×
	2	42	41	×	
	3	38	45	×	
	4	42			

ブロックNO	行数	A[2]文字数	B[2]文字数	判定	結果
A[2]:B[2] パターン2	1	23			×
	2	42	54	×	
	3	38	41	×	
	4	42	45	×	

ブロックNO	行数	A[2]文字数	B[5]文字数	判定	結果
A[2]:B[5] パターン1	1	23	81	×	×
	2	42	94	×	
	3	38	26	×	
	4	42			
	4	42			

ブロックNO	行数	A[2]文字数	B[5]文字数	判定	結果
A[2]:B[5] パターン2	1	23			×
	2	42	81	×	
	3	38	94	×	
	4	42	26	×	
	4	42			

ブロックNO	行数	A[2]文字数	B[5]文字数	判定	結果
A[2]:B[5] パターン3	1	23			×
	2	42			
	3	38	81	×	
	4	42	94	×	
	4	42	26	×	

ブロックNO	行数	A[2]文字数	B[6]文字数	判定	結果
A[2]:B[6] パターン1	1	23	74	×	×
	2	42	94	×	
	3	38	18	×	
	4	42			
	4	42			

ブロックNO	行数	A[2]文字数	B[6]文字数	判定	結果
A[2]:B[6] パターン2	1	23			×
	2	42	74	×	
	3	38	94	×	
	4	42	18	×	
	4	42			

ブロックNO	行数	A[2]文字数	B[6]文字数	判定	結果
A[2]:B[6] パターン3	1	23			×
	2	42			
	3	38	74	×	
	4	42	94	×	
	4	42	18	×	

ブロックNO	行数	A[3]文字数	B[3]文字数	判定	結果
A[3]:B[3]	1	25	25	○	○
	2	21	21	○	
	3	7	7	○	
	4	23	23	○	
	5	9	13	×	

ブロックNO	行数	A[6]文字数	B[6]文字数	判定	結果
A[6]:B[6]	1	38	74	×	×
	2	55	94	×	
	3	51	18	×	

図 4.6 候補ブロック精査結果

する文字である.

= 変数宣言,代入処理や比較演算などで利用され,コード内の処理に依存する文字である.

. ピリオドはincludeファイルの記述や構造体などを利用する際に出現する文字である.

図 4.7 に比較結果のまとめを示す. (A[1],B[1]), (A[3],B[3])の 2 組は，上記の 5 つの特殊文字の出現数がすべて同じであり，一致率が基準を超え，クローンの可能性が高いと判定できる.

一方，(A[2],B[2])，(A[2],B[5])，(A[2],B[6])の各組は一部の文字の出現数は一致するものの，全 5 文字を考えると一致率は基準値を下回っている．また，(A[6],B[6])の組は 5 つすべての文字で出現数が異なっており,これらの 4 つの組はクローンではない可能性が高いと判断できる．なお,ここでクローンの可能性が高いと判定した(A[1],B[1]), (A[3],B[3])の組は Step3 の結果とも一致している.

ブロックNO	特定文字				
	w	<	>	=	.
A[1]	22	0	0	0	21
B[1]	22	0	0	0	21
判定	○	○	○	○	○
一致率	100%		結果		○

	w	<	>	=	.
A[2]	0	4	4	0	4
B[2]	0	3	3	0	3
判定	○	×	×	○	×
一致率	40%		結果		×

	w	<	>	=	.
A[2]	0	4	4	0	4
B[5]	1	12	2	0	2
判定	×	×	×	○	×
一致率	20%		結果		×

ブロックNO	特定文字				
	w	<	>	=	.
A[2]	0	4	4	0	4
B[6]	0	10	2	0	2
判定	○	×	×	○	×
一致率	40%		結果		×

	w	<	>	=	.
A[3]	1	2	2	0	1
B[3]	1	2	2	0	1
判定	○	○	○	○	○
一致率	100%		結果		○

	w	<	>	=	.
A[6]	1	16	0	4	1
B[6]	0	10	2	0	2
判定	×	×	×	×	×
一致率	0%		結果		×

判定
○:一致
×:不一致

結果
○:クローンと判定する
×:クローンではないと判定

図 4.7 特定文字を使ったクローン検出結果

4.5 考察

(1) 提案手法の妥当性について

4 章のケーススタディにおいてクローンの可能性が高いブロックの組と判定されたものは(A[1],B[1])，(A[3],B[3])の 2 組となった．これらの組について実際のコードを人が見て比較したところ，(A[1],B[1])の組ではブロック内の 1 行の記述が変更されているのみ，(A[3],B[3])の組は処理内容が全く同一であることが確認できた．これより提案法を用いてクローンが確実に検出できていることが確認できた．

また，提案法の Step2 でクローン候補に挙がったブロックの組の中で，(A[2],B[5])，(A[2],B[6])，(A[6],B[6])の組については実際に人が見た場合，異なる処理であると判別できた．これらの組については，提案法の Step3 以降の候補ブロックの精査によって，クローンでないブロックの組み合わせとして確実に除外することができている．

一方で，(A[2],B[2])の組については提案法ではクローンと判定できなかったが,実際に人が見て比較した場合，クローンと判定できることがわかった．この組が検出されなかった原因としては空白文字及びコメント文の変更により，行単位での文字数が大幅に変動したことがあげられる．

(2) 判定の基準値について

提案法の Step2 ではブロックの文字数，行数についてはそれぞれ 30%,25%の差異を判定基準とした．また Step3 ではブロック内で文字数が一致する行の割合を用い一致率 60%を基準とした．また Step4 ではブロック内の特殊文字の一致率を用い 80%を基準とした．ケーススタディで用いたこれらの基準値は暫定値であるため,これらの基準値を多少変更することで，コードクローン候補やその可能性が大きく変わってしまうことが考えられる．このため，これらの基準値については，さらなる実験などでより精度の高い値を求めるとともに，実際の利用シーンでは利用者側の判断でチューニングしながら最適値を決めていくなどの方策も検討する必要がある．

(3) Step4 で利用する文字種類について

Step4 ではブロックの特徴を端的に表す文字を指標として用いる．ケーススタディでは w,<,>,=,. の 5 文字を採用したが, その他の文字も指標候補として考えることもできる．このため，一般的なあるいは特徴的なプログラムコード内でどのような文字が多く利用されているかについて，プレ実験などを行い採用する文字を決定する必要がある．

5. まとめ

本稿では，ソースコードを構成するブロック単位における文字数や行数などの形状情報や各ブロックの特徴を表す特定文字などの評価指標を用いてコードクローンを検出する方法を提案した. この方法は構文解析等を用いないため，文法的な誤りが含まれる開発途中のソースコード中でもコードクローンを検出でき，そういったクローンの検出に有用である可能性が高い．

また，4 章ではケーススタディを紹介した．ここでは実際に作成されたコードに対して本手法を適用した．コードをブロック分割し，その中からクローンとなるものだけを抽出することができた．今回の方法では，形状からのクローン候補抽出のための文字数や行数，特殊文字数などをインデックスとして用いており，ケーススタディではこれらの基準値は暫定の値を用いている．このためより高い精度でクローン候補判定するための基準値などについて，今後実験検討を重ねて決定していく必要がある．

また，今回は手法検討のためのケーススタディを中心に紹介したが，この先，提案法をベースにコードクローン検出ツールを開発し，様々な実プロジェクトのソースコードなどを対象に本手法の有効性を確認していきたい．

参考文献

1) Lutz Prechelt, Guido Malpohl and Michael Philippsen : Finding Plagiarisms among a Set of Programs with JPlag, Journal of Universal Computer Science (2002).
2) Toshihiro Kamiya, Shinji Kusumoto and Katsuro Inoue : CCFinder: A Multi-Linguistic Token-based Code Clone Detection System for Large Scale Source Code, IEEE Trans. Software Engineering, Vol.28, No.7, pp.654-670 (2002).
3) 肥後芳樹，吉田則裕，楠本真二，井上克郎 ： 産学連携に基づいたコードクローン可視化手法の改良と実装，情報処理学会論文誌, Vol.48, No.2, pp.811-822 (2007).
4) 宮崎宏海 ： ギャップを含むコードクローンのフィルタリング手法の提案，大阪大学大学院 修士学位論文 (2009).
5) 肥後芳樹，植田泰士，楠本真二，井上克郎 : AGM アルゴリズムを用いたギャップを含むコードクローン情報の生成，電子情報通信学会 技術研究報告, SS2007-48, Vol.107, No.392, pp.61-66 (2007).
6) 村上寛明，堀田圭佑，肥後芳樹，井垣宏，楠本真二 : Smith - Waterman アルゴリズムを利用したギャップを含むコードクローン検出，情報処理学会 ソフトウェアエンジニアリングシンポジウム 2013 (2013).
7) 肥後芳樹，楠本真二，井上克郎 ： コードクローン検出とその関連技術，電子情報通信学会論文誌 D, Vol.J91-D, No.6, pp.1465-1481 (2008).
8) コードクローン分析に基づくソフトウェア開発・保守支援に関する研究成果報告書 (大阪大学), 独立行政法人情報処理推進機構 2012 年度ソフトウェア工学分野の先導的研究支援事業 (2013) . URL http://www.ipa.go.jp/files/000026805.pdf
9) Borland C++ Compiler 5.5, URL http://www.embarcadero.com/jp