DEIM Forum 2011 D3-3

複数観点を用いた木類似度算出法によるソースコードの類似構造の抽出

池田　健人† 　小林　隆志†† 　波多野賢治††† 　深川　大路†††

† 同志社大学大学院文化情報学研究科　〒 610–0394 京都府京田辺市多々羅都谷 1–3
†† 名古屋大学大学院情報科学研究科　〒 464–8601 愛知県名古屋市千種区不老町
††† 同志社大学文化情報学部　〒 610–0394 京都府京田辺市多々羅都谷 1–3
E-mail: †ikeda@ilab.doshisha.ac.jp, ††tkobaya@is.nagoya-u.ac.jp,
†††{khatano,dfukagaw}@mail.doshisha.ac.jp

あらまし　本稿では，木構造の複数特徴量と編集操作を用いた類似度算出法のソースコードへの適用を行う．ソースコードの情報は単純な文字列の比較では抽出することの出来ない，入れ子構造による木の構造を保持している．そのため，その構造を反映させた類似度算出法が求められている．そこで，Java のソースコードを XML 形式のリポジトリである JX-model を用いて表現することにより，その XML を対象として木構造の複数の観点を用いた類似度を算出する．JX-model を用いて抽象構文木を生成することにより，我々の提案する木構造類似度算出法で木の構造の類似を見ることでソースコードが持つ構造の類似を抽出する．

キーワード　木構造類似度，Tree Edit Distance，XML，ソースコード

Similar Structure Extraction Method for Source Code based on the Tree Similarity Degree on Several Aspects

Kento IKEDA†, Takashi KOBAYASHI††, Kenji HATANO†††, and Daiji FUKAGAWA†††

† Graduate School of Culture and Information Science, Doshisha University
Tatara-Miyakodani 1–3, Kyotanabe, Kyoto, 610–0394 Japan
†† Graduate School of Information Science, Nagoya University
Furo-Cho, Chikusa-ku, Nagoya, Aichi, 464–8601 Japan
††† Faculty of Culture and Information Science, Doshisha University
Tatara-Miyakodani 1–3, Kyotanabe, Kyoto, 610–0394 Japan
E-mail: †ikeda@ilab.doshisha.ac.jp, ††tkobaya@is.nagoya-u.ac.jp,
†††{khatano,dfukagaw}@mail.doshisha.ac.jp

1. はじめに

近年，計算機の性能向上や一般ユーザへの普及により，個人が大量のデータを扱う機会が増加した．その中でも，木構造データは階層構造を用いることにより多様な表現が出来るため，様々なデータで用いられており，今後もその利用は増え続けると考えられる．そのため，大量にある木構造データの中からユーザが望むデータを的確に検索するための類似度算出法が求められているため，我々はこれまでに木構造が持つ複数の特徴量を考慮した木構造類似度算出法の提案を行ってきた．

本稿ではその類似度算出法の Java ソースコードへの適用を行う．ソースコードの類似を測る際には，ソースコードを文字列として捉え，その差を見るだけではソースコードが有する構造の類似を加味することが出来ない．そこで，ソースコードはプログラミング言語の文法規則に基づく構文木として捉えることが出来るため，その木構造を対象として類似度を比較することでソースコードの構造の類似も加味した類似度が算出出来ると考えられる．そのため，木構造データに含まれる複数の特徴を反映させた類似度算出法をソースコードへ適用することにより，ソースコードに存在する類似構造の抽出を行う．用いる類似度算出法では，パラメータを調整することにより，各特徴の観点が類似度へ与える影響を変化させることが可能であり，ソースコードを対象とした場合にそのパラメータによって算出される類似度とその際に抽出されるソースコードの構造の類似の考察を行う．

2. 関 連 研 究

本節では，ソースコードの類似度算出法及び木構造類似度算出法に関する関連研究の説明を行う．

2.1 ソースコードの類似度算出法

ソースコードの類似を測る際の比較的単純な手法として，最長共通部分列を発見するアルゴリズムである LCS (Longest Common Subsequence) [10] に基づく差分抽出ツール diff [1] を用いる手段がある．この手法ではソースコードを各行ごとの文字列として扱い，行単位の比較を行い，一致する度合いから類似性を判断する．

また，ソースコードの一致個所を特定するコードクローン検出の分野では，diff 同様に行単位でのコードクローンの検出 [3], [5] のほかに，字句単位での検出手法 [6], [8] が提案されている．これらの行単位及び字句単位による手法では，一致する文字列の割合を求めるだけであり，ソースコードの特徴を利用していない．

ソースコードの特徴を利用する類似度としては，プログラム中の字句の類似性に加え，プログラム構造の複雑度であるサイクロマチックス数をメトリクスとして加えた手法が提案されている [12]．

ソースコードの構造的特徴を積極的に用いる類似度判定としては，プログラム構文の木構造である抽象構文木 (Abstract Syntax Tree) を用いたコードクローン抽出技術が提案されている [2], [4]．これらの抽象構文木を用いる手法では，ソースコードの構文解析を行った木構造を対象としているため，行単位や字句単位では加味出来なかったソースコードの構造を加味したクローンコードの類似性判定が可能である．

本稿ではソースコードに限らない木構造データ全般を対象とした類似度算出法を定義し，抽象構文木に適用することで，ソースコードの類似度を求める．抽象構文木の情報は，構文木情報をマークアップするためのモデルである JX-model [9] を用いる．コードクローンの分野では，ソースコード中に含まれる一致するコード片の抽出を目的とするが，本稿では，ソースコードの一致ではなく，ソースコードが示すプログラムの構造が類似するものを発見することを目的としている点が異なる．

2.2 Edit Distance

類似度算出法として Edit Distance (編集距離) という考え方がある．Edit Distance は，二つの比較対象を最短何回の編集操作 (挿入，削除，置換) によって同一の形式に出来るか，という考え方で算出する距離である．元々は文字列を対象とした距離算出法である String Edit Distance [7] が提案され，後に木構造へ適用した Tree Edit Distance [11] が提案された．

しかし，比較する二つの木の大きさが異なっている場合，そのノード数の差の分だけ挿入または削除を行う必要があり，距離が自然と大きくなる．そのため，ノード数の差が大きい場合，二つの木に出現するノードの種類が同一であっても，その出現数が異なることにより類似していないと判断されてしまう．また，ノードのつながり方に着目した値が算出されるが，これだけでは木構造データが有する複数の特徴の一面からしか捉えられていないと言える．そこで，本稿ではこれらの問題点を考慮するために，複数の特徴を考慮した木類似度算出法を用い，ソースコードへ適用することによりソースコードの類似構造抽出を行う．

3. 木類似度算出法とソースコードへの適用

本節では，我々がこれまでに提案してきた木類似度算出法について述べる．まず，類似度算出法に必要な特徴量の選択を行い，その特徴量を用いた類似度算出法について説明し，ソースコードへの適用方法について述べる．

3.1 複数特徴量の観点選択

本手法で用いる複数特徴量を選択するために，一つの DTD に従う XML だけでなく，異なる DTD に従う異なる特徴の XML を用意するために，Java ソースプログラムの XML 版レポジトリである JX-model の DTD に従った Java チュートリアル (注1) のデータ 517 件，SIGMOD Record (注2) にある 1999 年版の論文に関する 5 種類の DTD に従った 920 件，2002 年版の論文に関する 4 種類の DTD に従った 48 件の計 1,555 件の XML データを用い，木構造データから算出可能な数値である，ノード数，深さ，葉ノード数，ノードの種類数，エッジの種類数を求め，主成分分析を行った．その結果，ノード数と葉ノード数，ノードの種類数とエッジの種類数に関連があることが判明した．そのため，これらの組のデータに回帰分析を行ったところ，いずれも決定係数が 0.9 以上で正の線形関係にあるため，これらの組のそれぞれ一方の値を特徴量の観点とする．よって，一つの木構造を特徴付ける値として深さ，葉ノード数，ノードの種類数を用いる．

以上の様に，一つの木から表現される特徴を三つ選択したが，二つの木を比較する際にも取得出来る特徴がある．本手法では五つの観点から類似度の算出を行う．木の比較時の特徴として α と β，一つの木の特徴として δ と γ と ζ を以下で定義する．

3.2 編集操作に関する特徴量 α

α は，編集操作においてノードの個数よりもノードの出現の有無を重視した値であり，頻出するノードへの編集操作のコストを小さくするために α を以下のように定義する．

まず，各編集操作に対する重み付き編集コストである E_i を求める．E_1 が一回目の編集操作のコストを，E_i が i 番目の編集操作のコストを示しており，Tree Edit Distance に必要な編集操作の回数だけ i を増やしていく．編集操作が挿入または削除の場合には，比較を行う二つの木 T_1，T_2 に含まれている編集操作の対象となるノードの出現個数の逆数とすることにより，ノードの出現個数の差による影響ではなく出現有無に着目した値を算出する．編集操作が置換の場合には，対象となるノードが置換するノードとされるノードの二種類となるため，二つのノード出現個数の平均値の逆数とする．n はノード，T は木，$N_t(T, n)$ は T にある n の数，$N(T)$ は T のノード数，$E\{\varepsilon \mapsto n\}$ は n の挿入，$E\{n \mapsto \varepsilon\}$ は n の削除，$E\{n \mapsto m\}$

(注1)：http://java.sun.com/docs/books/tutorial/

(注2)：http://www.sigmod.org/publications/sigmod-record/xml-edition/

は n から m への置換を意味する．

$$E_i = \begin{cases} \dfrac{1}{N_t(T_1,n)+N_t(T_2,n)} \\ \qquad (\text{if } E\{\varepsilon \mapsto n\} \text{ or } E\{n \mapsto \varepsilon\}) \\ \dfrac{2}{(N_t(T_1,n)+N_t(T_2,n))+(N_t(T_1,m)+N_t(T_2,m))} \\ \qquad (\text{if } E\{n \mapsto m\}) \end{cases} \quad (1)$$

式 (1) で求めた E_i を用い，α を式 (2) のように定義する．E_i の和をさらに比較する二つの木のノード数の平均で割ることにより，木全体のノード数による影響を小さくしている．α の値は，$\alpha \geqq 0$ の値をとり，必要な編集操作の回数が増えるにつれて大きな値をとる．

$$\alpha = \frac{2}{N(T_1)+N(T_2)} \sum_i E_i \quad (2)$$

3.3 兄弟ノードの類似に関する特徴量 β

β は，比較する木の同じ深さにおける兄弟ノードの並びの類似に関する値である．同じ深さにおいて，兄弟ノードの並び方が類似しているかどうかを求めるために，兄弟ノードの並びに対して String Edit Distance で距離を算出し，比較対象の兄弟ノードの数の大きい方で割る．なお，式 (3) の i は $\min\{depth(T_1), depth(T_2)\}$ まで，つまり比較する二つの木が共通して有する深さまで繰り返す．β は，$\beta \geqq 0$ の値をとる．$SED\{Str_1, Str_2\}$ は Str_1 と Str_2 の String Edit Distance の値，$Str_{depth=i}(T)$ は T の深さ i の兄弟ノードの列，$N_{depth=i}(T)$ は T の深さ i の兄弟ノード数を意味する．

$$\beta = \sum_i \frac{SED\{Str_{depth=i}(T_1), Str_{depth=i}(T_2)\}}{\max\{N_{depth=i}(T_1), N_{depth=i}(T_2)\}} \quad (3)$$

3.4 木の特徴の差に関する特徴量 γ，δ，ζ

γ，δ，ζ は，比較を行う木 T_1，T_2 が持つ特徴の違いを表す値であり，それぞれ木の深さ，葉ノード数，ノードの種類数を用いる．用いる値は違うが，算出方法は共通であり，式 (4) で算出する．$f(T)$ は，γ 算出時には木 T の深さ，δ 算出時には T の葉ノード数，ζ 算出時には T のノード種類数を代入する．

$$\gamma \text{ or } \delta \text{ or } \zeta = \begin{cases} 0 & (\text{if } f(T_1)=0 \cap f(T_2)=0) \\ 1 - \dfrac{\min\{f(T_1), f(T_2)\}}{\max\{f(T_1), f(T_2)\}} & (\text{else}) \end{cases} \quad (4)$$

これらの値は，各観点から求めた値が近い場合に，その近さに応じて値を小さくする働きがあり，$0 \leqq \gamma \leqq 1$, $0 \leqq \delta \leqq 1$, $0 \leqq \zeta \leqq 1$ の値をとる．

3.5 非類似度 $disSim(T_1, T_2)$

α，β，γ，δ，ζ の各特徴量に対し，パラメータを付与した非類似度である式 (5) を定義する．この値は，値が小さいほど類似したデータであると判断出来る．また，パラメータ w によって α，β，γ，δ，ζ のどの観点を重視するかが調整可能となっている．

$$disSim(T_1,T_2) = w_1\alpha + w_2\beta + w_3\gamma + w_4\delta + w_5\zeta \quad (5)$$
$$(w_1 + w_2 + w_3 + w_4 + w_5 = 1)$$

3.6 ソースコードへの適用

3.5 で定義した類似度算出法を用い，Java のソースコードへの適用を行う．本稿では，Java のソースコードをそのまま用いるのではなく，Java 言語のソースコードに対して，構文木情報をマークアップするためのモデルである，JX-model を用いて XML へ変換した木構造データを用いる．JX-model では，Java のソースコードを解析し，27 種類の要素から構成される XML でソースコードを表現することが出来る．このモデルで生成された XML を木構造データとして用いることにより，ソースコードの構造が XML の木の構造に反映され，その類似度を求めることでソースコードに存在する類似した構造の抽出を可能とする．そのため，類似度算出法にはプログラミング言語の知識を一切導入する必要がない．また，他のプログラミング言語へ適用する際にはこのモデルをそれぞれの言語へ適用したものを用いれば良い．

4. 評価実験

本節では，Java のソースコードへ類似度算出法を適用し，算出された値でクラスタリングを行う．パラメータの調整を行うことにより，ソースコードの構造を考慮したクラスタ分けが出来るか実験を行う．

4.1 実験手順

本実験では，Java のソースコードであるテストデータを用意し，それらを JX-model によって XML へ変換したデータを用いる．その XML を用いて類似度算出法を適用し，各データ間の類似度でクラスタリングを行う．類似度算出時にはパラメータを調整可能なため，パラメータを変更していき，有意なクラスタリング結果になるかを確認する．有意なクラスタリング結果とは，テストデータとしてソースコードの構造に二つの異なる規則性を与えているため，それらの規則によってクラスタリングされることを指す．規則に関しては 4.2 で述べる．また，JX-model によって XML 形式にしたデータに対して Tree Edit Distance を用いて距離を算出した結果を比較対象として用いる．なお，クラスタリングに用いたアルゴリズムはウォード法，群平均法，最遠隣法，最近隣法の四種類である．

4.2 対象データ

対象とするデータは，類似度算出法によりソースコードの構造を考慮できるのかを確認出来るよう二つの規則性を用いた図 1 〜 6 に示すソースコードを利用する．一つ目の規則性として，図 7 の模式図のようにパターン A とその処理回数を倍にしたパターン B のデータを用意したものであり，二つ目の規則性として，各パターンの Step2 の処理に当たる部分を単純な代入文，for 文，入れ子の for 文の三種類を用意した，計 6 種類のデータを用いる．具体的には，図 1 に示す `Test1.java` が入れ子構造も無く，これらの中で一番シンプルなものであり，図 4 に示す `Test1_2.java` がその際の処理回数を増やしたものである．図 2 に示す `Test2.java` は，図 1 に示

```
class Test1 {
  public static void main (String[] args) {
    int a = 0;
    int b = 0;
    int c = 0;

    a = a + 10;
    b = b + 10;
    c = c + 10;

    System.out.println(a);
    System.out.println(b);
    System.out.println(c);
  }
}
```

図 1　Test1.java

```
class Test2 {
  public static void main (String[] args) {
    int a = 0;
    int b = 0;
    int c = 0;

    for (int i = 0; i < 10; i++) {
      a = a + i;
    }
    for (int j = 0; j < 20; j++) {
      b = b + j;
    }
    for (int k = 0; k < 30; k++) {
      c = c + k;
    }

    System.out.println(a);
    System.out.println(b);
    System.out.println(c);
  }
}
```

図 2　Test2.java

```
class Test3 {
  public static void main (String[] args) {
    int a = 0;
    int b = 0;
    int c = 0;

    for (int i = 0; i < 10; i++) {
      a = a + i;
      for (int j = 0; j < 20; j++) {
        b = b + j;
        for (int k = 0; k < 30; k++) {
          c = c + k;
        }
      }
    }

    System.out.println(a);
    System.out.println(b);
    System.out.println(c);
  }
}
```

図 3　Test3.java

```
class Test1_2 {
  public static void main (String[] args) {
    int a = 0;
    int b = 0;
    int c = 0;
    int d = 0;
    int e = 0;
    int f = 0;

    a = a + 10;
    b = b + 10;
    c = c + 10;
    d = d + 10;
    e = e + 10;
    f = f + 10;

    System.out.println(a);
    System.out.println(b);
    System.out.println(c);
    System.out.println(d);
    System.out.println(e);
    System.out.println(f);
  }
}
```

図 4　Test1_2.java

```
class Test2_2 {
  public static void main (String[] args) {
    int a = 0;
    int b = 0;
    int c = 0;
    int d = 0;
    int e = 0;
    int f = 0;

    for (int i = 0; i < 10; i++) {
      a = a + i;
    }
    for (int j = 0; j < 20; j++) {
      b = b + j;
    }
    for (int k = 0; k < 30; k++) {
      c = c + k;
    }
    for (int l = 0; l < 40; l++) {
      d = d + l;
    }
    for (int m = 0; m < 50; m++) {
      e = e + m;
    }
    for (int n = 0; n < 60; n++) {
      f = f + n;
    }

    System.out.println(a);
    System.out.println(b);
    System.out.println(c);
    System.out.println(d);
    System.out.println(e);
    System.out.println(f);
  }
}
```

図 5　Test2_2.java

```
class Test3_2 {
  public static void main (String[] args) {
    int a = 0;
    int b = 0;
    int c = 0;
    int d = 0;
    int e = 0;
    int f = 0;

    for (int i = 0; i < 10; i++) {
      a = a + i;
      for (int j = 0; j < 20; j++) {
        b = b + j;
        for (int k = 0; k < 30; k++) {
          c = c + k;
          for (int l = 0; l < 40; l++) {
            d = d + l;
            for (int m = 0; m < 50; m++) {
              e = e + m;
              for (int n = 0; n < 60; n++) {
                f = f + n;
              }
            }
          }
        }
      }
    }

    System.out.println(a);
    System.out.println(b);
    System.out.println(c);
    System.out.println(d);
    System.out.println(e);
    System.out.println(f);
  }
}
```

図 6　Test3_2.java

す Test1.java の処理に繰り返し文を組み込んだものであり，図 5 に示す Test2_2.java がその際のステップを増やしたものである．図 3 に示す Test3.java は，図 2 に示す Test2.java の繰り返し処理を更に入れ子にしたものであり，図 6 に示す Test3_2.java がその際のステップを入れ子によって増やしたものである．これらのソースコードを JX-model を用いて XML へ変換した木構造を用いる．その変換された木が有する特徴量と変換後の XML の行数は表 1 の通りである．

表 1　テストデータの特徴量

データ	XML 行数	ノード数	深さ	葉ノード数	ノード種類数
Test1	17	263	10	75	16
Test1_2	26	470	10	135	16
Test2	23	431	12	123	16
Test2_2	38	806	12	231	16
Test3	23	431	16	123	16
Test3_2	38	806	22	231	16

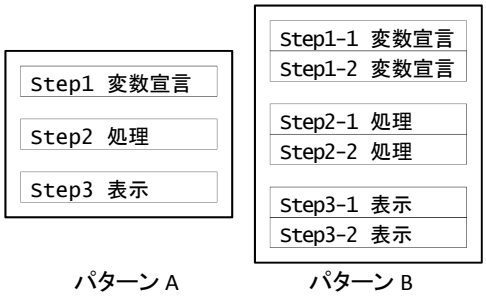

図 7　テストデータの構造の模式図

4.3　結果・考察

実験の結果を図 8 ～ 11 に示す．図 8 は Tree Edit Distance による結果である．これを見ると，特にソースコードの行数が多い，つまりノードの数が多い Test2_2 と Test3_2 でクラスタが構成されており，クラスタリングに用いた Tree Edit Distance の値がソースコードの行数・ノード数が影響していると読み取ることが出来る．なお，図 8 はウォード法によるクラスタリング結果であるが，群平均法，最遠隣法，最近隣法によるクラスタリングでも同様のクラスタ構造になる．

次に，JX-model へ類似度算出法を適用した結果の中で有意なクラスタリング結果であると判断したものを図 9，10，11 に示す．図 9 は $w_1 = w_3 = w_4 = w_5 = 0, w_2 = 1$ のウォード法に

よるクラスタリング結果である．また，$w_1 = w_3 = w_4 = w_5 = 0.15, w_2 = 0.40$ の場合も同様の結果となる．なお，これらはウォード法だけでなく，群平均法，最近隣法，最遠隣法でも同様のクラスタ分けになる．また，$w_1 = w_2 = w_3 = w_4 = w_5 = 0.20$ の場合，$w_1 = w_2 = w_3 = w_4 = 0.15, w_5 = 0.40$ の場合，$w_2 = w_3 = w_4 = w_5 = 0.15, w_1 = 0.40$ の場合では最近隣法以外のウォード法，群平均法，最遠隣法の場合に同じクラスタ構造になる．これらのパラメータは，β の重みである w_2 を大きくした場合，γ の重みである w_3 を小さくした場合，δ の重みである w_4 を小さくした場合である．w_2 を大きくすることは各深さでの処理の類似を重視することになり，パターン A と B では対応する処理の数が異なるため，パターンごとのクラスタリングを可能としている．また，w_3 は深さに着目した観点である．処理の内容は入れ子の深さが異なっているため，γ はその処理ごとの類似を重視することになる．そのため，w_3 を小さくすることにより，パターンごとのクラスタリングを可能としている．さらに，w_4 は葉ノード数に着目した観点である．3.1 で述べたように，ノード数と葉ノード数は線形関係にあり，本実験で用いたデータでも，表 1 に示す通り，葉ノード数とノード数・行数は正の線形関係にある．そのため，δ は Tree Edit Distance による結果の様にノード数・行数の影響を受ける．そのため，w_4 を小さくすることによりパターンごとのクラスタリングを可能としている．

一方，図 10，11 は $w_1 = w_4 = w_5 = 0.125, w_3 = 0.625, w_2 = 0$ の場合であり，図 10 が群平均法及び最近隣法によるクラスタリング結果であり，図 11 がウォード法及び最遠隣法によるクラスタリング結果である．これらのパラメータは γ である木の深さの影響を大きくし，β である各深さでの処理の違いの影響を小さくした場合である．この場合，処理回数の違いであるパターンよりも，処理の内容によってクラスタリングが行われている．図 10 では，`Test3` と `Test3_2` に関してのみクラスタの距離が大きいが，これは `Test1` と `Test2` が単純に処理の回数を倍増させているのに対し，`Test3` ではそれを入れ子として増やしたため，他の組よりも距離が増加したと考えた場合には妥当な結果だと解釈出来る．また，図 11 では，入れ子が存在するクラスタと存在しないクラスタに分割された後に，各処理ごとにクラスタリングされているため，このクラスタ構造も有意であると判断出来る．パラメータは w_2 を小さくした場合で，β の影響を小さくした場合である．これは図 9 とは逆に各深さでの処理の類似の影響を抑えることによりパターンではなく処理ごとのクラスタリングを可能としている．また，w_3 を大きくし，深さの影響を大きくしている．同じ処理を行うソースコードは同じ深さを有しているため，処理ごとのクラスタリングを可能としている．

以上の結果より，木類似度算出法を用いることにより Tree Edit Distance では抽出出来なかったソースコードの類似構造を捉えることが出来たと言える．また，パラメータの調整によりその観点を切り替えることが可能なため，ソースコードの長さや処理の内容など，必要に応じた観点による類似構造を保持したソースコードの検索への応用が期待出来る．本実験より得られる JX-model による木構造の特徴とソースコードの構造の特徴の関連としては，木構造での葉ノード数はソースコードの長さと関連し，木構造での各深さでの処理の類似は本稿でソースコードのパターンと呼んでいた処理の長さと関連し，木構造での深さはソースコードでの入れ子の数と関連していることが読み取れる．よって，これらのパラメータを適切に用いることにより，ソースコードからの構造の抽出が可能となると言える．

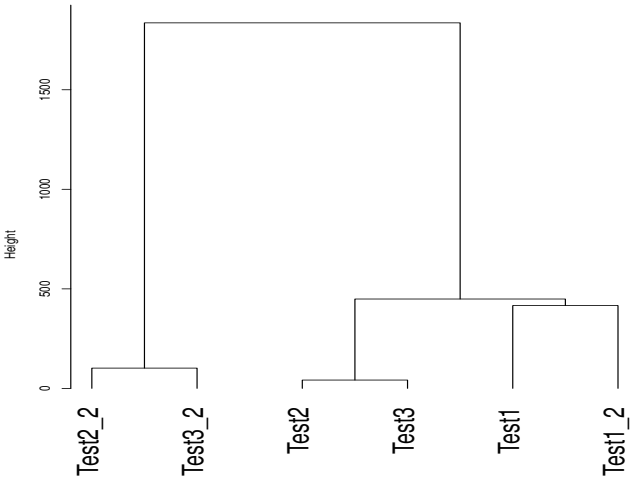

図 8　Tree Edit Distance によるクラスタリング

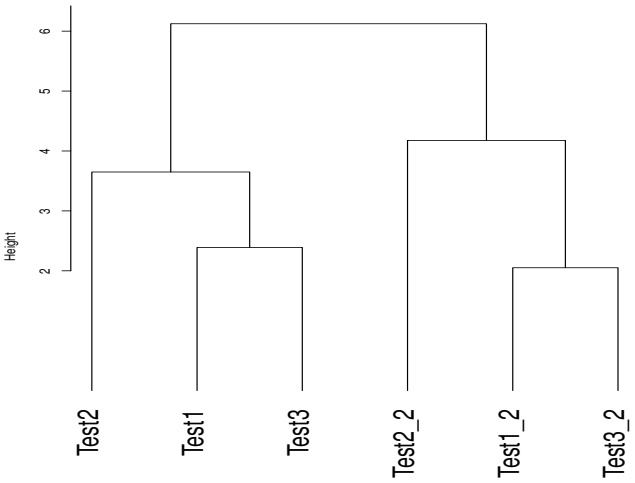

図 9　提案手法によるクラスタリング 1

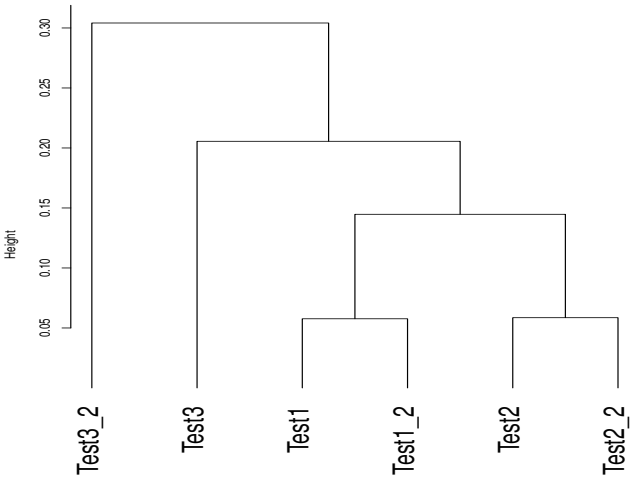

図 10　提案手法によるクラスタリング 2_1 (群平均法，最近隣法)

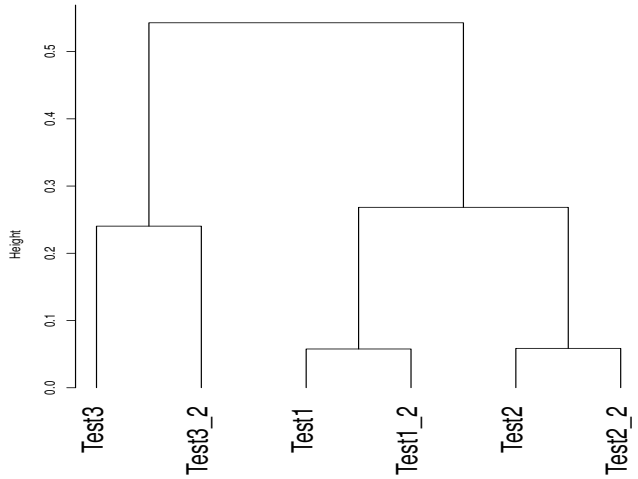

図 11　提案手法によるクラスタリング 2_2 (ウォード法，最遠隣法)

5. おわりに

本稿では，木構造データを対象とした複数の観点を用いた類似度算出法のソースコードへの適用を行った．パラメータの調整を行うことにより，ソースコードが持つ構造の類似をそれぞれ抽出出来ることを確認した．その際に JX-model を用いることにより，類似度算出法にはプログラミング言語の知識を一切必要としない．そのため，本手法を他のプログラミング言語に適用したい場合は，他の言語を対象としたソースコードを XML に変換する技術も考案されているため，それらを用いることにより適用が可能であり，汎用性が高いと言える．

しかし，本手法において以下のように今後の課題が三点存在する．

（1） パラメータ調整の自動化

（2） 複数観点間の独立性の議論

（3） 複雑なソースコードへの適用

まず，課題 (1) にパラメータ調整の自動化が挙げられる．本手法ではパラメータの調整による観点の重要度の調整を可能にしたが，パラメータの調整によって類似度が変化するため，本手法を有効に利用するためには適切なパラメータの設定が必要となってくる．ユーザが重要視したい観点が決まっていれば，そのパラメータを大きくすれば良いが，対象のデータ群の特徴によっても適切なパラメータが異なってくる．そのため，パラメータの自動設定もしくは自動推定の仕組みが必要になってくると考えられるため，今後はその考案を行う必要がある．

また，課題 (2) に複数観点間の独立性の議論が挙げられる．複数観点は対象とするデータに依存し，さらにパラメータにも依存するため，一概に独立性の議論を行うのは難しい．相関係数を用いて観点間の関係を見ると，データにも依存するが α と β には相関があると判定されるが，相関係数で相関があると判定されても，類似度に用いる観点として適している，適していないと判断出来るとも言い切れない．実際のクラスタ構造を見ると相関があると判定されても異なるものとなる．二観点間で回帰分析を行うといずれも決定係数は小さく，線形関係には無い．パラメータの推定を行う際に対象データの特徴をあらかじめ取得することによる独立性の高いパラメータの推定も可能であると考えられる．そのため，今後はさらに独立性と類似度の関連性に関して詳しく見ていく必要がある．

最後に，課題 (3) として本手法の複雑なソースコードへの適用が挙げられる．本稿では解釈しやすいように単純なデータに対しての実験であるため，実際のさらに複雑なソースコードでも実験う必要がある．その際，本稿で取り上げた二つの規則の両方を考慮した実験を行う必要がある．そのため，複雑かつ評価を妥当に行えるデータを用意し，本手法のさらなる評価を行っていく予定である．

謝　辞

本研究の一部は，同志社大学大学院文化情報学研究科研究推進補助金及び財団法人 日揮・実吉奨学会 研究助成金 (研助第 2144 号) によるものである．ここに記して謝意を表す．

文　献

[1] Diffutils. http://www.gnu.org/software/diffutils/.

[2] Ira D. Baxter, Andrew Yahin, Leonardo Moura, Marcelo Sant'Anna, and Lorraine Bier. Clone detection using abstract syntax trees. In *Proceedings of the International Conference on Software Maintenance*, ICSM '98, pp. 368–377. IEEE Computer Society, 1998.

[3] Stéphane Ducasse, Matthias Rieger, and Serge Demeyer. A language independent approach for detecting duplicated code. In *Proceedings of the International Conference on Software Maintenance*, ICSM '99, pp. 109–118. IEEE Computer Society, 1999.

[4] Lingxiao Jiang, Ghassan Misherghi, Zhendong Su, and Stephane Glondu. DECKARD: Scalable and accurate tree-based detection of code clones. In *Proceedings of the 29th international conference on Software Engineering*, ICSE '07, pp. 96–105. IEEE Computer Society, 2007.

[5] J. Howard Johnson. Substring matching for clone detection and change tracking. In *Proceedings of the International Conference on Software Maintenance*, ICSM '94, pp. 120–126. IEEE Computer Society, 1994.

[6] 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.

[7] Vladimir Iosifovich Levenshtein. Binary codes capable of correcting deletions, insertions and reversals. *Soviet Physics Doklady*, Vol. 10, No. 8, pp. 707–710, 1966.

[8] Zhenmin Li, Shan Lu, Suvda Myagmar, and Yuanyuan Zhou. CP-Miner: Finding copy-paste and related bugs in large-scale software code. *IEEE Transactions on Software Engineering*, Vol. 32, pp. 176–192, 2006.

[9] Katsuhisa Maruyama and Shinichiro Yamamoto. A CASE tool platform using an XML representation of java source code. *4th IEEE International Workshop on Source Code Analysis and Manipulation (SCAM'04)*, pp. 158–167, 2004.

[10] Webb Miller and Eugene W. Myers. A file comparison program. *Software: Practice and Experience*, Vol. 15, pp. 1025–1040, 1985.

[11] Kuo-Chung Tai. The tree-to-tree correction problem. *J. ACM*, Vol. 26, pp. 422–433, 1979.

[12] 小堀一雄, 山本哲男, 松下誠, 井上克郎. 類似度メトリクスを用いた Java ソースコード間類似度測定ツールの試作. 電子情報通信学会技術研究報告, SS2003-2, Vol. 103, No. 102, pp. 7–12, 2003.