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
ET2020-16(2020-09)

ロジック構成躓き推定のためのソースコード間の距離推移分析
－類似度算出手法の特徴比較－

立花 隼一[†]　中山 祐貴[††]　大沼 亮[†]　神長 裕明[†]　宮寺 庸造[‡]　中村 勝一[†]

†福島大学 共生システム理工学類／共生システム理工学研究科 〒960-1296 福島県福島市金谷川 1 番地
††早稲田大学 グローバルエデュケーションセンター 〒169-0071 東京都新宿区戸塚町 1 丁目 104
‡東京学芸大学 自然科学系 〒184-8501 東京都小金井市貫井北町 4-1-1
E-mail: †s1970028@ipc.fukushima-u.ac.jp, {kami, nakamura, onuma}@sss.fukushima-u.ac.jp,
††nakayama@aoni.waseda.jp, ‡miyadera@u-gakugei.ac.jp

あらまし　プログラミング演習では，的確な指導を行うために，学習者の躓き等の状況を上手く把握することが重要であるが，人的制約ゆえに困難である．学習状況把握に関する研究が盛んに行われているが，殆どがコンパイルエラーを対象としており，コンパイルエラーに表出しない躓きの把握については，有効な支援方法が実現されていない．本研究では，経験の浅い学習者が「思い描く処理像をプログラムとして形作る段階での躓き（ロジック構成面の躓き）」を自動抽出する手法の開発を目指す．本稿では，主に，ソースコード間の距離推移分析に関するケーススタディについて報告し，その結果に基づいて，類似度算出手法の特徴を比較・考察する．

キーワード　ロジック構成，躓き推定，学習履歴分析，ソースコード間類似度，プログラミング学習支援

Transition Analysis of Distance between Source Codes for Estimating the Stumbling in Construction of Program Logic: Comparison of the Features of Similarity Calculation Methods

Junichi TACHIBANA[†]　Hiroki NAKAYAMA[††]　Ryo ONUMA[†]
Hiroaki KAMINAGA[†]　Youzou MIYADERA[‡]　Shoichi NAKAMURA[†]

†Dept. Computer Science and Mathematics, Fukushima University　1 Kanayagawa, Fukushima 960-1296, Japan
††Global Education Center, Waseda University　1-104 Totsuka-machi, Shinjuku-ku, Tokyo 169-8050, Japan
‡Division of Natural Science, Tokyo Gakugei University　4-1-1, Nukuikita, Koganei, Tokyo 184-8501, Japan
E-mail: †s1970028@ipc.fukushima-u.ac.jp, {onuma, kami, nakamura}@sss.fukushima-u.ac.jp,
††nakayama@aoni.waseda.jp, ‡miyadera@u-gakugei.ac.jp

Abstract　To precisely guide students in programming exercise, it is important for instructors to sufficiently understand the situations of each learner such as stumbling. However, such understanding is usually difficult due to the restrictions of human resources. Although there have been many research projects on grasping learning situations in programming exercise, most methods targeted only compilation errors. Therefore, effective method for understanding the stumbling which does not appear in compilation error has not been developed. In this research, we have developed methods for automatically detecting the stumbling a student faces when he/she expresses an envisions of processing as a program (i.e., stumbling in construction of a program logic that does not appear in compile errors). In this paper, we mainly describe a case study on analysis of the transitions of distance between source codes. Based on the results, we discuss the features of methods for calculating the similarity.

Keywords　Construction of Program Logic, Estimation of Stumbling, Learning History Analysis, Degree of Similarity between Source Codes, Programming Learning Support

1. はじめに

プログラミング演習では，学習者は頻繁に問題に遭遇し，その解決に努める．未熟者にとって重要な経験であると同時に，難しい作業でもある．ここで，学習者が直面する問題は，文法的なエラーと，コンパイルエラーとして表出し難い論理的なエラーに大別できる．文法的エラーは，コンパイルエラーとして表出するため，学習者自身による解決の努力もある程度期待することができる．一方，コンパイルエラーとして表出し難いエラーは，学習者自身による解決のための模索が

難しい．特に，思い描く処理等をプログラムとして形作る（ロジック構成）段階での躓きは，後のより高度な学習段階にも大きな影響を及ぼし，その把握と指導の重要性が指摘されている[12]．指導・助言のためには，個々の学習状況の把握がとりわけ重要だが，実際の演習授業の制約下では限界がある．これに対して，演習進捗状況把握支援[1-4]や，プログラム誤り検出システム[5-8]など，プログラミング演習支援に関する多くの研究が報告されているが，その殆どが，いわゆる文法エラーを扱ったもので，ロジック構成面の躓きの把握は，十分な支援が実現されていない．

本研究では，コンパイル履歴とソースコード間類似度の分析に基づいて，プログラムロジック構成面の躓きを推定する手法の開発を目指す．

2. 問題点と支援方針

2.1. 支援対象

プログラミング演習授業では，的確な指導のために，個々の学習者の状況把握が重要だが，躓きは多種多様である．ここで，プログラミング演習における学習段階として，(1)全くの初心者で，文法などプログラミングの基礎を学ぶ段階，(2)プログラミングの基礎は習得しており，個別の命令文だけは記述できる段階，(3)自ら簡単ならプログラムを作成できる段階，の 3 つが考えられる．

本研究では，プログラムを形作ることに苦慮する(2)の段階の学習者を対象として，把握が特に困難なロジック構成面の躓き把握の支援を目指す．

2.2. 関連研究

学習者の躓き推定，及び学習状況の把握に関する研究は，数多く行われている．主なものとして，プログラミング演習における進捗状況把握支援に関する研究[1-4]，プログラムの誤り検出に関する研究[5-8]，プログラミング学習者の思考分析に関する研究[9-11]がある．

プログラミング演習における進捗状況把握支援に関する研究では，長ら[1]が学習者によるエラーなどの詳細な学習履歴から，学習者の特徴的な行動を抽出して，学習者の出来を判断したり，学習者に自動的にアドバイスを与えたりできるシステム「proFrep」を開発している．しかし，このシステムは予めパターンに合うアドバイスを登録するため，支援には限界がある．

プログラムの誤り検出に関する研究では，藤原ら[5]がプログラミング演習時に収集したソースコードのスナップショットを分析して，学習者がいつ，どのような箇所で行き詰まっていたのかを特定する手法を提案している．しかし，この手法は各学生の回答からの編集距離を算出しているため，学生が回答を提出するまでは行き詰まりの特定を行えないという制限がある．

プログラミング学習者の思考分析に関する研究では，大場ら[9]が，プログラミング力と論理的文章作成力との類似戦を分析している．

2.3. 問題点

プログラミング初学者が直面する問題は，コンパイルエラーとして表出する文法的エラーと，コンパイルエラーとして表出し難いエラーに大別できる．「文法的エラー」は，コンパイルエラーとして表出するため，学習者自身による解決の努力もある程度期待することができる．既存研究は，殆どが，この文法的エラーを扱うものであった．一方，「論理的エラー」は，コンパイルエラーとして表出しないため，学習者自身による解決のための模索が難しい．同時に，教授者による躓き状況の把握が容易ではない．

そこで本研究では，主に以下の 2 つに焦点を当てる．

（問題点 1）コンパイルエラーとして表出し難いロジック構成面の躓きを把握することは困難である．

（問題点 2）少人数の教授者で，学習者それぞれの状況を把握するのは困難である．

2.4. 方針

支援の概要を図 1 に示す．本研究では，まず，プログラミング演習におけるコンパイル履歴の分析に基づいて，躓きの一次候補を抽出する手法を開発する．次に，抽出した一次候補について，ソースコード間の類似度推移の分析に基づいて，ロジック構成面の躓きを推定する手法を開発する．これらの手法により，問題点 1 の解決を目指す．その上で，開発した手法に基づいて，ロジック構成面での躓き状況を視覚化するシステムを開発することで問題点 2 の解決を目指す．

さらに，分析の過程で，対象ソースとその分析戦略自体を動的に切り替える仕組みを開発することでより精度の高いシステムの開発を目指す．実際の履歴データを用いた検証を行い，課題を含めた提案手法の特徴を明らかにする．

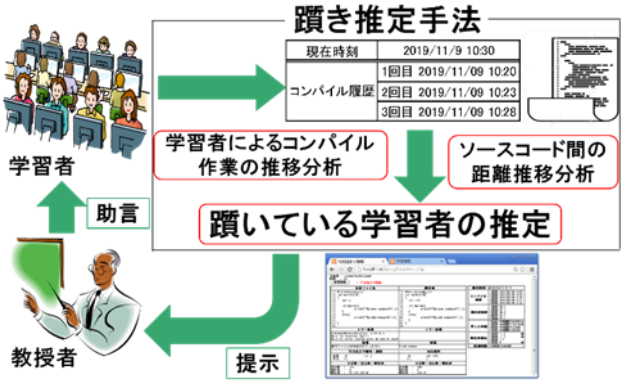

図 1　支援の概要

3. ロジック構成面の躓き推定手法

3.1. 躓き推定手順

ロジック構成面の躓き推定の概要を図 2 に示す．推定は以下の手順で行う．

① **一次候補抽出**：学習者のコンパイル履歴を分析し，エラーを伴わないコンパイルが一定回数以上続いているケースを抽出し，各コンパイル時点でのソースコードを取得する．

② **ソースコード間の類似度算出**：①で抽出した各コンパイル時点のソースコードについて，正解ソースコードに対する類似度を算出する．

③ **躓き推定**：②の算出結果から「類似度の推移」を分析し，ロジック構成面の躓きを推定する．

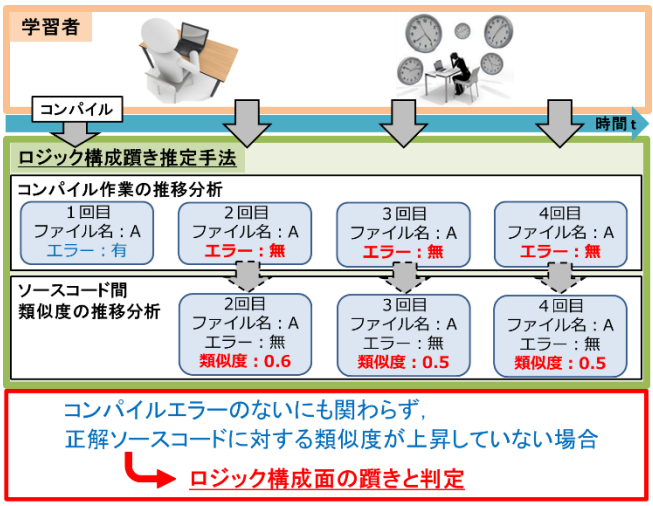

図 2 ロジック構成躓き推定手法

3.2. コンパイル作業の推移分析による一次候補抽出

コンパイル履歴を分析し，エラーの無いコンパイルが繰り返されている（しかし，完成には至っていない）ケースを抽出し，該当する学習者，コンパイル日時，ソースコードを一次候補として抽出する．例えば，図 2 では，2 回目から 4 回目のコンパイル実施時が一次候補となり得る．

3.3. ソースコード間の類似度推移分析

抽出した一次候補について，当該ソースコードの正解ソースコードに対する類似度を算出し，その推移を分析する．エラーの無いコンパイルが繰り返されており，なおかつ，正解ソースコードに対する類似度が上昇していない場合（図 2 の例では，0.6→0.5→0.5）に，ロジック構成面の躓きと判定する．

3.4. ソースコード間の類似度算出

ソースコード間の類似度の算出には，ソースを一種の文書と捉えた編集距離に基づく方法と，構文木の類似性に基づく方法を用いる．

1) 編集距離に基づく類似度算出
- Order Levenshtein Distance（OLD）
- Jaro-Winkler Distance（Jaro）
- Overlap Coefficient（OC）

2) 構文木による類似度算出
- Tree Edit Distance（TED）
- Tree Overlapping（TO）

3.4.1. Order Levenshtein Distance

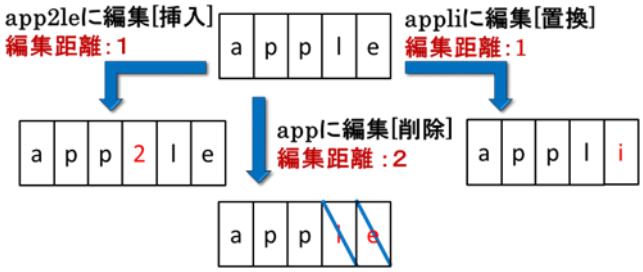

図 3 Order Levenshtein Distance

OLD では，2 つの命令順列間のレーベンシュタイン距離を求める．つまり，教授者の命令順列に一致させるために，学習者の命令順列を何回挿入・削除・置換したかを求める（図 3）．2 つの命令順列A, Bを式 3.1，式 3.2 に定義する．

$A = (a_1, a_2, \cdots, a_m) \cdots$ A の接頭辞は，$A_{1,2,\cdots m}$ (3.1)

$B = (b_1, b_2, \cdots, b_n) \cdots$ B の接頭辞は，$B_{1,2,\cdots n}$ (3.2)

$OLD(A_i, B_j)$は，式 3.3 により定義される．

$$OLD(A_i, B_j) = \begin{cases} \min(A_i, B_j) & if\ \min(A_i, B_j) = 0 \\ min = \begin{cases} OLD(A_{i-1}, B_j) + 1 \\ OLD(A_i, B_{j-1}) + 1 \\ OLD\left(A_{i-1}, B_{j-1} + 1_{A_i \neq B_j}\right) \end{cases} & otherwize. \end{cases} \quad (3.3)$$

この式に標準化をすることで類似度Q_1とし，式 3.4，式 3.5 に示す．

$$Q_1 = 1 - \frac{OLD(A_i, B_j)}{D_{OLD}} \quad (3.4)$$

$$D_{OLD} = \max(|A|, |B|) \quad (3.5)$$

3.4.2. Jaro-Winkler Distance

Jaro-Winkler Distance では，文字列間の距離を，共通する文字数と置換の要不要から求める．また，ジャロ・ウィンクラー距離は Jaro Distance を使って定義される．Jaro Distance を式 3.6 に定義する．

$$\Phi = W_1 \cdot c/d + W_2 \cdot c/r + W_\tau \cdot (c - \tau)/c \quad (3.6)$$

W_1：最初の文字列中の文字に掛かる重み
W_2：2 番目の文字列中の文字に掛かる重み
W_t：置き換えに掛かる重み
d：最初の文字列の長さ
r：2 番目の文字列の長さ
τ：文字の置換の数
c：区間内で一致する文字の数

ただし，c=0 で Φ=0 とする．具体的には，c は式 3.7 のように定義される．

$$c = \frac{\max(d, r)}{2} - 1 \qquad (3.7)$$

Jaro-Winkler Distance を式 3.8 に示す．

$$\Phi_n = \Phi + i \cdot 0.1 \cdot (1 - \Phi) \qquad (3.8)$$

このとき i=1，2，3，4 である．

3.4.3. Dice Coefficient

Dice Coefficient では 2 つの集合の平均要素数と共通要素数の割合を表している．ある集合 A とある集合 B についての OC（A,B）は，式 3.9 で定義される．

$$\mathrm{Dice(A,B)} = \frac{2|A \cap B|}{|A| + |B|} \qquad (3.9)$$

3.4.4. Tree Edit Distance

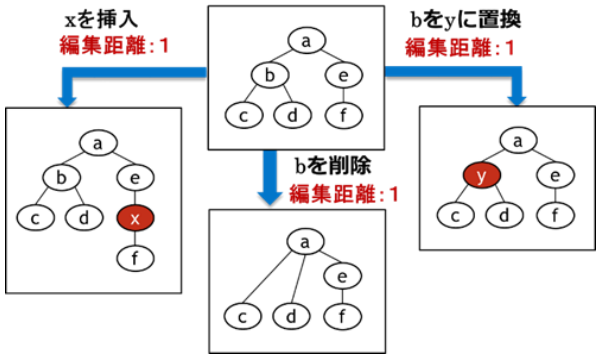

図 4　Tree Edit Distance

Tree Edit Distance では 2 つのツリーの間のレーベンシュタイン距離を求める（図 4）．元の木について，最左木のルートノードを抜き出すことで得られる残りの複数の木の集合が森となる．森について，最左木のルートノードを L(F)，L(F)の子ノードをルートノードとする木の集合を C(F)，残りの木の集合を R(F)とする．このとき，森F_1，F_2間の編集距離 d(F_1, F_2)は式 3.10 のようになる．

$$d(F_i, F_2) = \min \begin{cases} \gamma(R(F_1), R(F_2)) + d(C(F_1), C(F_2)) + (d(L(F_1), L(F_2)), \\ \gamma(R(F_1), 0) + d(L(F_1) + C(F_1), F_2), \\ \gamma(0, R(F_2)) + d(F_1, L(F_2) + C(F_2)) \end{cases} \qquad (3.10)$$

この式に標準化をすることで類似度Q_2とし，式 3.11，式 3.12 に示す．

$$\mathrm{Q_2} = 1 - \frac{d(F_1, F_2)}{D} \qquad (3.11)$$

$$\mathrm{D_{TED}} = \max(|F_1|, |F_2|) \qquad (3.12)$$

3.4.5. Tree Overlapping

Tree Overlapping では 2 つのツリーの間について，親と子の導出規則の等しいノード組数を求める（図 5）．木T_1，T_2間の重なるノード数$S_{To}(T_1, T_2)$は式 3.13，式 3.14 で定義される．

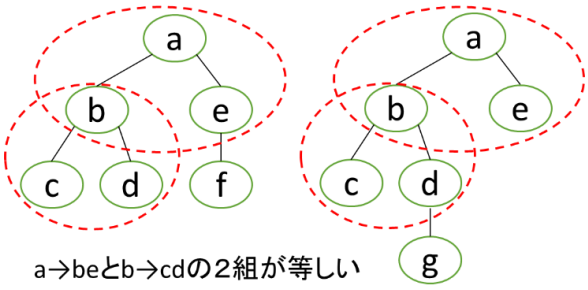

図 5　Tree Overlapping

$$S_{To}(T_1, T_2) = \max_{n_1 \in N_1} \max_{n_2 \in N_2} C_{TO}(n_1, n_2) \qquad (3.13)$$

$$C_{TO}(n_1, n_2) = \left| (m_1, m_2) \;\middle|\; \begin{array}{c} m_1 \in nonterm(T_1) \\ \wedge\, m_2 \in nonterm(T_2) \\ \wedge\, (m_1, m_2) \in L(n_1, n_2) \\ \wedge\, prod(m_1) \in prod(m_2) \end{array} \right| \qquad (3.14)$$

この式に標準化をすることで類似度Q_3とし，式 3.15,式 3.16 に示す．

$$\mathrm{Q_3} = \frac{S_{TO}(T_1, T_2)}{D_{TO}} \qquad (3.15)$$

$$\mathrm{D_{TO}} = \max(|nonterm(T_1)|, |nonterm(T_2|) \qquad (3.16)$$

3.4.6. ソースコード間類似度の動的切替

本研究では，「一次候補」と「正解」を比較し，類似度を算出する際に，先述した算出法の中からその時の状況に応じて，一番適したものを自動的に採用する．そのために，いくつかの手法で類似度を算出する体制を準備し，実際のソースコードを用いた検証を行い，どの算出方法を用いれば最も効率よく正確なデータを得られるか比較・検証する．この作業を通して，各類似度算出法の特徴の把握に努める．その結果に基づいて，手法切替の「指標」を整備する．指標に基づいて動的切替を担うモジュールを開発する．それを用いて，反復的な検証と手法改善を試みる．

4. 実験と考察

4.1. 実験概要

ソースコード間の類似度算出の基本的な有効性検証と課題抽出を目的として実験を行った．具体的には，まず，実際のプログラミング演習における課題を参考に，完成したプログラム（正解ソースコード）と，未完成段階のソースコードを準備した．その上で，各ソースコードの正解ソースコードに対する類似度を算出した．ここでは，上述した 5 つ（OLD，Jaro，Dice，TED，TO）の算出法を用いた．正解ソースコードを図 6，図 7 に示す．また，未完成段階のソースコードは，以下の要領で準備した．

<正解ソースコード 1>

・ソース 1：①の部分を除いたもの

・ソース 2：②の部分を除いたもの

・ソース 3：③の部分を除いたもの
・ソース 4：④の部分を除いたもの
・ソース 5：①の部分を誤った位置に移動（②のループの外側に移動）したもの
・ソース 6：①の部分を誤った位置に移動（③のループの内側に移動）したもの
・ソース 7：①の部分を誤った位置に移動（④のループの外側に移動）したもの

<正解ソースコード 2>
・ソース 1：②の部分を除いたもの
・ソース 2：③の部分を除いたもの
・ソース 3：④の部分を除いたもの
・ソース 4：①の部分を誤った位置に移動（②のループの外側に移動）したもの
・ソース 5：①の部分を誤った位置に移動（③のループの外側に移動）したもの
・ソース 6：①の部分を誤った位置に移動（④のループの外側に移動）したもの

```
1   #include<stdio.h>
2   int main(void)
3   {
4       int retry;
5       int i,no;
6
7       do{
8
9       do{
10        printf("正の整数を入力してください:");
11        scanf("%d",&no);
12        if(no<=0)
13          puts("¥a 正でない数を入力し①      ②
14          ないでください．");
15      }while(no<=0);
16
17      for(i=1;i<=no;i++`      ③
18        putchar('*');
19      putchar('¥n');                          ④
20
21      printf("もう一度? [Yes…0/No…9]:");
22      scanf("%d",&retry);
23    }while(retry==0);
24
25      return 0;
26  }
```

図 6 正解ソースコード 1

```
1   #include <stdio.h>
2   int main(void)
3   {
4       int x, y, z;
5
6       for (x=1; x<=10; x++) {                    ④
7       for (y=1; y<=10; y++) {                    ③
8         for (z=1; z<=10; z++) {                  ②
9           if(2*x<y && x*x+3*y==2*z) {            ①
1             printf("(x,y,z)=(%d,%d,%d)¥n",x,y,z);
0           }
1         }
1       }
1       }
2       return 0;
1   }
```

図 7 正解ソースコード 2

4.2. 結果と考察

正解ソースコード 1 に対する類似度の算出結果を表 1 に示す．

表 1 ソースコード 1 の類似度算出結果

ソース	OLD	Jaro	Dice	TED	TO
1	0.864	0.919	0.947	0.453	0.907
2	0.682	0.861	0.857	0.302	0.209
3	0.545	0.591	0.750	0.314	0.209
4	0.318	0.647	0.519	0.291	0.093
5	0.909	0.970	1.000	0.302	0.894
6	0.727	0.955	1.000	0.326	0.913
7	0.727	0.917	1.000	0.302	0.816

まず，図 6 における①は，②の内側に内包される局所的な部分であるため，正解ソースコード 1 に対する類似度は，ソース 1>ソース 2 となるはずである．他の部分も同様に，ソース 1>ソース 2，ソース 3>ソース 4 となるものと考えられる．また，ソース 5～7 は徐々に正解ソースコードに対する距離を遠ざけているため，正解ソースコード 1 に対する類似度は，ソース 5，ソース 6>ソース 7 になるものと考えられる．

表 1 より，ソース 1～4 では 5 つの算出法に関して，概ね狙い通りの値を得ることができた．ソース 5～7 では，Jaro，TO は狙い通りの値が得られた．一方で，OLD はソース 6>ソース 7 となるところがソース 6=ソース 7，Dice は値が変わらない，TED はソース 5>ソース 7 となるところがソース 5=ソース 7 という結果が出た．

正解ソースコード 2 に対する類似度の算出結果を表 2 に示す.

表 2 ソースコード 2 の類似度算出結果

ソース	OLD	Jaro	Dice	TED	TO
1	0.867	0.956	0.957	0.407	0.469
2	0.733	0.911	0.909	0.272	0.306
3	0.600	0.748	0.857	0.111	0.143
4	0.867	0.956	0.917	0.272	0.469
5	0.733	0.908	0.917	0.099	0.286
6	0.600	0.896	0.917	0.111	0.122

まず，図 7 における②は，③の内側に内包される局所的な部分であるため，正解ソースコード 2 に対する類似度は，ソース 1>ソース 2 となるはずである．他の部分も同様に，ソース 1>ソース 2>ソース 3 となるものと考えられる．また，ソース 4～6 は徐々に正解ソースコードに対する距離を遠ざけているため，正解ソースコード 1 に対する類似度は，ソース 4>ソース 5>ソース 6 になるものと考えられる．

表 2 より，ソース 1～3 では 5 つの算出法に関して，概ね狙い通りの値を得ることができた．ソース 4～6 では，OLD，Jaro，TO は狙い通りの値が得られた．一方で，Dice は値が変わらない，TED はソース 5>ソース 6 となるところがソース 5<ソース 6 という結果が出た．

全体をみると，正解ソースコードの一部を削除したものに関しては，5 つの算出法で狙い通りの値を得ることができた．一方，徐々に正解ソースコードに対する距離を遠ざけているケースでは，算出法により差が出た．Jaro，TO を用いた算出法は，概ね狙い通りの値を得ることができた．しかし，Dice は値が変わらない，TED も狙いとは違う結果が出た．また，OLD はソースコード 1 の実験では，狙いとは違う結果であったが，ソースコード 2 の実験では狙い通りの値を得ることができた．

今回の実験は限定的なものであるが，提案手法のロジック構成面の躓き推定に対する有効性について，良好な感触を得ることができた．また，類似度算出法について，各々特徴的な傾向を窺うことができた．

5. おわりに

本稿では，コンパイルエラーに表出し難いロジック構成面の躓き状態の推定手法について提案した．具体的には，「学習者によるコンパイル作業の推移」と「ソースコード間の距離推移」の分析に基づいた推定手法について述べた．テストケースデータを用いて，5 つの手法による類似度算出の実験を行った．その結果から，Jaro，TO を用いた算出法は，概ね狙い通りの値を得ることができた．一方，一部上手く行かないケースの観察から，分析対象によって手法の適用が制限され得ることが分かった．

今後は実際のデータを用いた提案手法の検証と改善を重ね，課題を含めた知見集約を試みる．また，支援システムのプロトタイプを用いてロジック構成躓きの推定実験を行い，実践面での検証を進める．

文　献

[1] 長慎也，筧捷彦，“proGrep-プログラミング学習履歴検索システム”，情報処理学会研究報告（CE），Vol.78，No.15，pp.29-36，2014.

[2] 井垣宏，齊藤俊，井上亮文，中村亮太，楠本真二，“プログラミング演習における進捗状況把握のためのコーディング過程可視化システム C3PV の提案”，情報処理学会論文誌，Vol.54，No.1，pp.330-339，2013.

[3] 加藤利康，石川孝，“プログラミング演習のための授業支援システムにおける学習状況把握機能の実現”，情報処理学会論文誌，Vol.55，No.8，pp.1918-1930，2014.

[4] Suin Kim, Jae Won Kim, Jungkook Park, Alice Oh, “Elice: An online CS Education Platform to Understand How Students Learn Programming”, *Proc. of the Third ACM Conference on Learning @ Scale*, pp. 225-228, 2016.

[5] 藤原賢二，上村恭平，井垣宏，吉田則裕，伏田享平，玉田春昭，楠本真二，飯田元，“スナップショットを用いたプログラミング演習における行き詰まり箇所の特定”，コンピュータ ソフトウェア，Vol.35，No.1，pp.1-13，2018.

[6] 藤原理也，田口浩，島田幸廣，高田秀志，島川博光“ストリームデータによる学習者のプログラミング状況把握”，電子情報通信学会第 18 回データ工学ワークショップ，D9-5，2007.

[7] 宮地恵佑，高橋直久，“構造誤り検出機能を有するアセンブリプログラミング演習支援システムの実現と評価，”電子情報通信学会論文誌，Vol.J91-D，No.2，pp.280-292，2008.

[8] Brett A. Becker, Graham Glanville, Ricardo Iwashima, Claire McDonnell, Kyle Goslin, Catherine Mooney, “Effective compiler error message enhancement for novice programming students”, *Computer Science Education*, Vol. 26, Issue 2-3, pp. 148-175, 2016.

[9] 大場みち子，伊藤恵，下郡啓夫，“プログラミング力と論理的思考力との相関に関する分析”，情報処理学会研究報告，Vol.2015-DD-97，No.2，pp，2015.

[10] Lisa Wang, Angela Sy, Larry Liu, Chris Piech, “Deep Knowledge Tracing On Programming Exercises”, *Proc. of the Fourth ACM Conference on Learning @ Scale*, pp. 201-204, 2017.

[11] Thomas W. Price, Tiffany Barnes, “Comparing Textual and Block Interfaces in a Novice Programming Environment”, *Proc. of the eleventh annual International Conference on International Computing Education Research*, pp. 91-99, 2015.

[12] P.Kinnunena，B.Simonb，“My program is ok ? am I? Computing freshmen’s experiences of doing programming assignments,” *Computer Science Education*, Taylor & Francis, Vol.22, No.1, pp.1-28, 2012.