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.

ソースコードのスナップショットに基づいた無作為修正者の検出

大野　　優†　上野　秀剛††　内田　眞司††

† 奈良工業高等専門学校システム創成工学専攻情報システムコース　〒639–1058 奈良県大和郡山市矢田町 22
†† 奈良工業高等専門学校情報工学科　〒639–1058 奈良県大和郡山市矢田町 22
E-mail: †a0899@stdmail.nara-k.ac.jp, ††{uwano,uchida}@info.nara-k.ac.jp

あらまし　教育現場におけるプログラミング演習では，少数の教員が多数の受講者を指導することが多い．そのため，教員が各受講者の学習状況を把握し，理解が不十分な受講者を発見，指導することが難しい．本稿では，内容を理解しないままソースコードの修正を繰り返す受講者を検出することを目的に，受講者が提出したソースコードのスナップショットから無作為な修正を行う受講者を識別するためのメトリクスを提案し，その有用性を検証する．本手法はソースコードの変更履歴から算出されるメトリクスを用いることで，演習の正答にたどり着かない修正を繰り返す受講者の特徴を把握する．

キーワード　プログラミング教育，Online Judge System，自動識別

1. 序　　論

教育現場におけるプログラミング教育では座学と演習を組み合わせた授業形式が一般的である．授業では少数の教員が多数の受講者を対象に講義を行うため，演習中に個々の受講者についてその理解状況や作業の様子を把握することが難しい．また，理解していない学生を識別できた場合でも具体的に何を理解しておらず，どのような指導が適切なのか判断することは時間がかかる．プログラミング講義の受講者の状況識別について調査している研究が複数報告されている [1] [2] [3] [4]．これらの研究から，プログラミング受講者のスナップショットを分析することで，演習中に受講者がどのような箇所で行き詰まっているのかを把握することができる．本研究は，識別対象を無作為修正者に限定し，演習中の受講生の識別を試みる点でこれらの研究と異なる．

プログラミングの学習環境の 1 つとして，Online Judge System(OJS) がある．OJS は課題に対応したソースコードを学生が作成し提出すると，コンパイル・実行し，その実行結果を元にソースコードを採点する．プログラミング講義に OJS を用いることで，受講者は演習課題で作成したソースコードを OJS に提出し即座に採点結果を得ることができる．

本研究では OJS を利用した講義を対象に，教員による指導が必要な受講者を自動的に識別する手法の開発を目的としている．受講者は OJS にソースコードを随時提出するため，提出ごとのすべてのソースコードがスナップショットとして記録される．そのため，スナップショットから各受講者がどのような編集をしたのか履歴をとることができる．OJS から得られるソースコードの編集履歴から各受講者の行動や理解状況を把握できれば，演習中に対象を絞った指導をすることが可能となる．

本稿では，無作為修正者を対象として，編集履歴からの識別を試みる．無作為修正者とは，演習の内容を理解しないままソースコードを闇雲に繰り返す受講者を指す．受講者が提出したソースコードの編集履歴から無作為修正者に特有の性質をメトリクスとして算出し，演習や講義の成績との相関を分析する．無作為修正者を識別できるようになれば，演習時間の早期に指導することで授業時間内により多くの受講者が理解に辿り着くことができると考えられる．

2. 関 連 研 究

プログラミング講義の受講者を対象に，状況の識別を行っている研究が複数報告されている [1] [2] [3] [4]．Fujiwara らはプログラミング演習時に取得した初学者のソースコード編集履歴を分析し，その傾向を明らかにした [1]．分析では (1) プログラミング熟練者の目視による定性的分析と，(2) トークンレベルでの Levenstein 距離の計測に基づく定量的分析を実施した．分析 (1) の結果，初学者が誤りに陥っている際に本来修正すべき箇所ではなく，課題内容とは関係の無い箇所を編集するといった行動パターンが観測された．分析 (2) の結果，演習において理解困難な状態に陥っている場合，距離の変動を確認することで初学者が誤りに陥っているかを検出できる可能性があることがわかった．

また，同著者は受講者のソースコード編集履歴を分析し，どのような箇所で行き詰まっていたのか特定する手法を提案した [2]．提案手法は，ある時点における演習課題のソースコードを，演習課題完了時のソースコードにするために必要な作業を推定残作業量として定量化する．推定残作業量の変化を元に，受講者が作成しているソースコードが正解に近づいていない，行き詰まり区間を検出し，その区間のスナップショットからどのような箇所で行き詰まっていたのかを特定する．

槇原らは探索的プログラミングと呼ばれる，プログラムの実現方法が定かではないときに学習者が修正・コンパイル・実行を繰り返すプログラミング行動に着目した手法を提案している [3]．手法は学習者がプログラムのどこで探索しているかを検出し，課題に対する取り組みや行き詰まりをリアルタイムに特

定する．

藤原らはプログラミング学習者の学習状況から取れるソースコードの行数，コンパイル回数，プログラムの実行回数などの時系列データから学習者の特徴を定量化する手法を提案した [4]．複数の時系列データをグラフで可視化し，関連性のあるところに対してデータ値の増減，データ値の変化量，データ値の上限，データ値の下限，データの時間微分値の 5 つの定性的な表現に当てはめ，特定の学習状況に現れるデータ値の推移の特徴を数式として定量化する．分析の結果，一定時間でのソースコードの行数の増減やプログラムの実行回数の増加などから学習者の行き詰まりを定量化できることがわかった．

以上の研究はいずれもプログラミング受講者のスナップショットを分析することで，演習中に受講者がどのような箇所で行き詰まっているのか把握する．本研究はプログラミング学習者の中でも無作為修正者に対象を絞り，スナップショットから得られるメトリクスを用いてその識別を試みる．

3. 準　　備

3.1 OJS を用いたプログラミング講義

本研究が対象とするプログラミング講義は，ある単元に対して座学の後に演習課題を解く形式の講義である．また，受講者は演習課題で作成したソースコードを OJS に提出し即座に採点結果を得る．OJS は提出されたソースコードをコンパイル・実行し，教員があらかじめ作成したテストケースと一致するかどうかを判定する．テストケースとは作成したプログラムが要件を満たしているかテストするための入力と，入力に対して望まれる出力の組みを表す．テストケースとの一致数に応じて以下の式で点数 *score* が計算される．

$$score = \frac{\text{テストケースの正解数}}{\text{テストケース数}} \times 100 \qquad (1)$$

受講者には，各テストケースの正誤判定結果，*score*，コンパイル時のエラー，実行時エラーなどが報告される．受講者は *score* が 100 点になるまで，ソースコードの修正と提出を繰り返す．作成されたソースコードは提出ごとにリビジョンとして，提出日時，点数とともに記録される．

3.2 無作為修正者

本研究では演習の内容を理解しないままソースコードの無作為な修正を繰り返す受講者を「無作為修正者」と定義する．図 1 に無作為に行われる修正の例を示す．図はある課題におけるある受講者の提出履歴のうち，最初の 8 リビジョンのソースコードを示しており，図の灰色の箇所は修正箇所を表す．リビジョン 1 ～ 4 のソースコードを見ると，正しく動くまでに 4 行目の for 文の条件式を繰り返し修正している．同様にリビジョン 5 ～ 8 のソースコードを見ると，5 行目の print 文の表示内容を繰り返し修正している．無作為修正者は複数のリビジョンをかけて，if 文や for 文の条件式，print 文の中身といった，特定の行の修正を闇雲に繰り返す傾向がある．なお，本稿ではソースコードの修正を「追加」,「削除」,「変更」の 3 つの動作から成ることとする．

受講者は演習中に作成したソースコードにエラーが出たり，仕様通りの動作にならないといった行き詰まりに陥った場合，自分で理由を考えたり，調査した上でソースコードを修正するのが理想である [5]．しかし，無作為修正者は行き詰まりの原因について考えず，自分が持っている修正パターンを闇雲に適用することで正解にたどり着こうとすることが考えられる．そのため，闇雲な修正によってプログラムが正しく動いた場合，与えられた課題の仕様やアルゴリズム，新たに学習した文法などを理解せずに演習を終える可能性がある．演習中に無作為修正者をリアルタイムに識別することで，教員が指導して課題が設定された単元の理解を促すことが可能になると考えられる．

本稿では，無作為修正者の性質について以下の 3 つの仮説を元に識別する手法を提案する．

H1： 同じ箇所（同じ行）を頻繁に修正する

H2： 一定時間の修正・コンパイルの回数が多い

H3： 1 回あたりの修正量が小さい

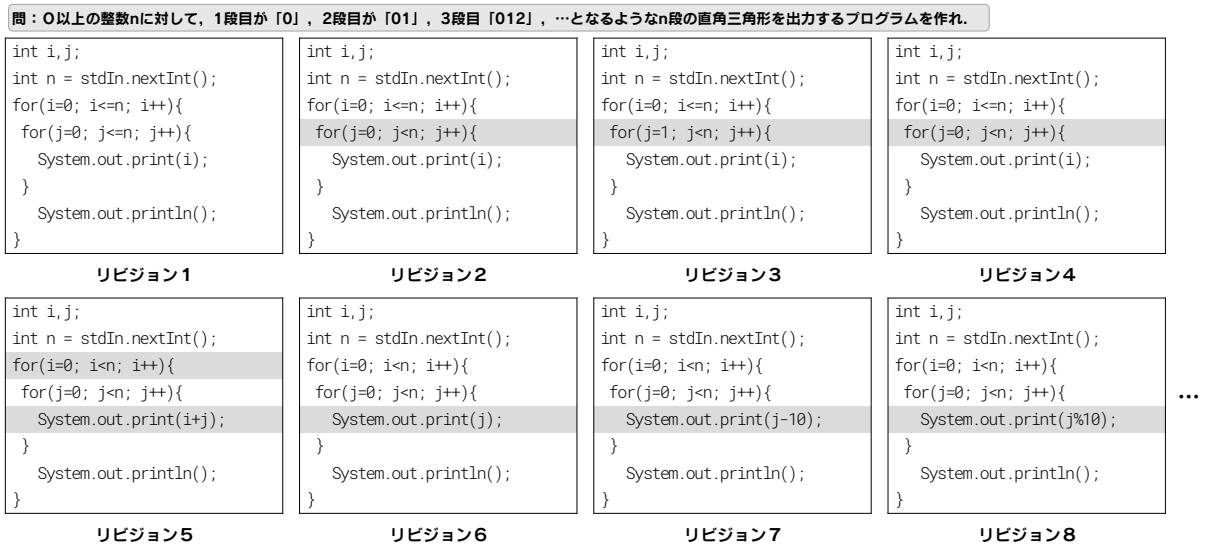

図 1　無作為な修正の例

4. 提 案 手 法

4.1 メトリクス

提案手法は以下の 4 つのメトリクスを用いて無作為修正者の特徴を把握する．提案手法は，演習の時間中に無作為修正者を識別し，指導を行うことを目的としているため，リビジョン 1 の提出から t 分間に提出されたソースコードの変更履歴からメトリクスを計算して識別に用いる．

$Freq(t)$： 修正行の偏り

$Revs(t)$： t 分間のリビジョン数

$DiffLine(s,t)$： s 行以下の修正が続いた回数

$AveDiff(t)$： 1 回あたりの平均修正行数

$Freq(t)$ は t 分間に修正された行の偏りを表す．無作為修正者はエラーの原因と思われる箇所に対して，自分が持っている修正パターンを闇雲に適用する傾向があると考えられる．そのため，たとえば if 文の条件式のような 1 行に対して繰り返し修正を行うと考えられ，一定時間内の修正がわずかな行に集中している可能性がある（仮説 H1）．提案手法ではソースコードの各行に対する修正回数から分散を求めることで，t 分間に行われた修正がどれだけ特定の行に集中しているかを分析する．無作為修正者は $Freq(t)$ が大きくなる傾向があると考えられる．

$Revs(t)$ は，t 分間に提出されたリビジョン数である．無作為修正者はエラー箇所と思われる部分に自分の修正パターンを闇雲に適用する傾向があると考えられる．そのため一定時間内の修正・コンパイル回数が多く（仮説 H2），$Revs(t)$ が大きくなる傾向があると考えられる．

$DiffLine(s,t)$ は，s 行以下の修正が続いた回数から求められる．無作為修正者はエラーが見つかると原因と思われる箇所をわずかに修正し，その正しさを考えることなくコンパイル・実行を繰り返す傾向があると考えられる．そのため，たとえば if 文の条件式だけのような 1 行の修正を繰り返す可能性がある（仮説 H3）．提案手法では，t 分間に提出されたリビジョン群において s 行以下の修正が連続した回数の最大値と，$Revs(t)$ から $DiffLine(s,t)$ を算出する．

$$DiffLine(s,t) = \frac{s \text{ 行以下の修正が続いた回数の最大値}}{Revs(t) - 1} \quad (2)$$

無作為修正者は $DiffLine(s,t)$ が大きくなる傾向があると考えられる．なお，本稿では $s = 1$ とし，1 行以下の修正が続いた回数を用いて計算する．

$AveDiff(t)$ は，1 回あたりの平均修正行数である．無作為修正者はエラーが見つかると複数個所が原因になっている可能性を考えず，ごく少量の修正を行い，コンパイル・実行を繰り返す傾向があると考えられる．そのため，1 回あたりの修正量は小さくなる可能性がある（仮説 H3）．無作為修正者は $AveDiff(t)$ が小さくなる傾向があると考えられる．

4.2 算 出 手 順

各メトリクスは以下 (1) ～ (5) の手順で算出する（図 2）．ここで，リビジョン k を R_k と表す．

（1） t 分間のスナップショット群の抽出

OJS に記録された各ソースコードの提出日時を元に，R_1 から t 分間に提出されたソースコードと，受講者 ID や評価結果 $score$ を抽出する．

（2） $Revs(t)$ の算出

課題 ID と受講者 ID から同じ受講者が同じ課題に対して提出したソースコード数を求め，$Revs(t)$ を算出する．

（3） リビジョン間の修正箇所の検出

手順 (1) で抽出されたソースコードに対して，R_i と R_{i+1} 間の各行に対する修正箇所を diff コマンドと-W オプションを用いて検出し，差分テキストファイルとして生成する．差分テキストファイルは R_i と R_{i+1} 間の各行で，「追加」，「削除」，「変更」があれば，その行に各々「<」，「>」，「|」の記号が記され，修正がなければ何も記されない．この生成を各受講者に対して $i = 1 \sim (Revs(t) - 1)$ まで繰り返す．

（4） $DiffLine(1,t)$, $AveDiff(t)$ の算出

手順 (3) で生成した差分テキストファイルに対し，UNIX の wc コマンドを用いて各リビジョン間の修正行数を算出する．この結果を用いて，1 行以下の修正が続いた回数と $Revs(t)$ から $DiffLine(1,t)$ を算出する．同様に，各リビジョン間の修正行数と $Revs(t)$ から，各受講者に対する $AveDiff(t)$ を算出する．

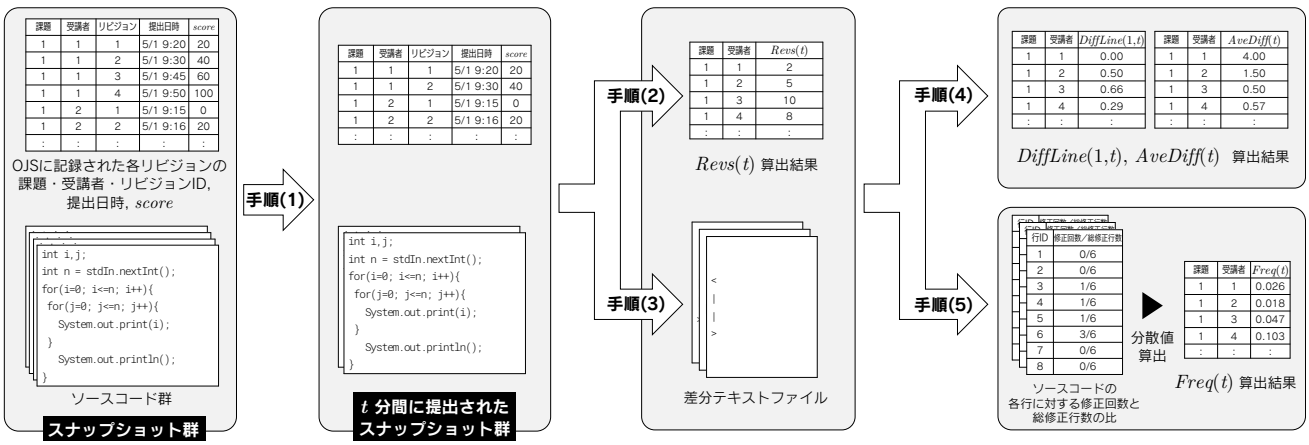

図 2 各メトリクスの算出手順

表 1　$score(t)$ との相関

t	分析対象人数	$Freq(t)$ 相関係数	$Freq(t)$ p 値	$Revs(t)$ 相関係数	$Revs(t)$ p 値	$DiffLine(1,t)$ 相関係数	$DiffLine(1,t)$ p 値	$AveDiff(t)$ 相関係数	$AveDiff(t)$ p 値
5	223	0.185	0.005**	-0.289	0.000**	-0.161	0.016*	-0.021	0.759
10	227	0.194	0.003**	-0.400	0.000**	-0.221	0.001**	-0.044	0.514
15	232	0.185	0.005**	-0.407	0.000**	-0.246	0.000**	-0.013	0.839
20	233	0.159	0.015*	-0.393	0.000**	-0.200	0.002**	0.003	0.967
25	233	0.171	0.009**	-0.331	0.000**	-0.211	0.001**	-0.007	0.915
30	233	0.186	0.004**	-0.380	0.000**	-0.183	0.005**	-0.035	0.596
35	234	0.194	0.003**	-0.405	0.000**	-0.169	0.010*	-0.126	0.055
40	234	0.199	0.002**	-0.355	0.000**	-0.168	0.010*	-0.111	0.090
45	234	0.194	0.003**	-0.366	0.000**	-0.135	0.039*	-0.118	0.071

（5）　$Freq(t)$ の算出

手順 (3) で生成したテキストファイルに対し，R_i と R_{i+1} 間の各行に対する修正箇所と R_{i+1} と R_{i+2} 間の各行に対する修正箇所をそれぞれ対応づける．これを $i = 1 \sim (Revs(t) - 1)$ まで繰り返し，全リビジョンにおいて修正した行とその修正回数を求める．各行で求めた修正回数と全リビジョンの総修正行数から $Freq(t)$ を算出する．

5. 実　　験

5.1 対象データ

奈良工業高等専門学校の 2 年生が受講する授業であるプログラミング I で収集されたデータを分析対象とする．本講義は Java のプログラミング演習形式による，1 回あたり 90 分間の講義である．Learning Management System(LMS) の 1 つである Moodle と連携して OJS の機能を提供する CodeRunner を用いて演習の出題と回収，評価を行っている．

CodeRunner は学生が提出したソースコードを課題 ID，受講者 ID，リビジョン番号，提出日時，*score* と共に記録する．なお，$Revs(t) = 1$ の場合，$Freq(t)$, $DiffLine(1,t)$, $AveDiff(t)$ の 3 つのメトリクスを算出できないため分析から除外する．本稿では，2018 年 6 月 21 日 ∼ 9 月 1 日に収集された計 2,529 回 (11 演習，述べ 364 人）の提出ソースコードのデータを対象に実験を行う．

5.2 評　　価

提案したメトリクスの有用性を評価するために，各メトリクスと受講者の能力の関係を分析する．無作為修正者は，演習内容を理解しないままソースコードの修正を繰り返すため，各リビジョンごとに OJS が算出する *score* が低いと考えられる．一方で，演習の内容やエラーの原因を理解して修正する受講者は各リビジョンの *score* が高いと考えられる．そこで，本稿では t 分間のスナップショット群における最新リビジョンの点数 $score(t)$ を求め，各メトリクスとの相関を求める．

本研究では演習中に無作為修正者を識別することを目標としている．そのため，リビジョン 1 からの経過時間を表す t について複数の条件で分析を行うことが望ましい．そこで本稿では対象講義 1 コマ分の半分の時間である 45 分を t の最大とし，5 分刻みで各メトリクスを算出し，比較する．

5.3 結　　果

各メトリクスと $score(t)$ の相関係数および無相関検定の結果を表 1 に示す．以降では有意な相関が見られなかった $AveDiff(t)$ 以外の各メトリクスについて結果を説明する．

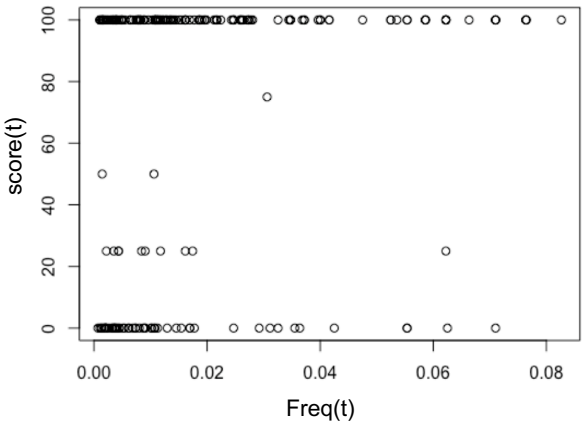

図 3　$Freq(t)$ と $score(t)$ の散布図 ($t = 10$)

5.3.1 $Freq(t)$

$Freq(t)$ と $score(t)$ の相関係数は，+0.159 ∼ +0.199 と弱い正の相関で，すべての t で有意な相関がみられた ($p < 0.01$)．この結果は，一定時間内の修正がわずかな行に集中している受講者の $score(t)$ が高いことを示しており，4.1 節で述べた仮説 H1（同じ行を頻繁に修正する）とは反対の特徴を有している可能性を示唆する．また，t の値が異なっても相関係数はほとんど変化しなかった．

図 3 に $t = 10$ における $Freq(t)$ と $score(t)$ の散布図を示す．図から $Freq(t)$ と $score(t)$ の両方が高い受講者が多く見られた．また，このような受講者は $Revs(t)$ が小さい場合が多かった．これは，少数のリビジョンで特定の行を数回修正することで正答に到達していることを示している．

このような修正は OJS の採点方法が厳密であることが原因の 1 つとして起こっていると考えられる．OJS による採点はテストケースの出力部分との文字列のマッチングによって行われる．そのため，課題に対する回答のアルゴリズムが正しくても，空白の数やアルファベットの大文字・小文字，全角半角の違い，文字化けなどのフォーマットが異なると不正解と判定される．$Freq(t)$ と $score(t)$ の両方が高い受講者のリビジョン間のソースコード変化を見ると，正しい出力`"OutPut:0"`に対して誤って`"Output:0"`, `"OutPut:␣0"`と出力している場合が複数見られた．また，これらの誤りは数回のリビジョンで修正され，正答に到達していた．$Freq(t)$ を仮説 H1(同じ行を頻繁に修正する) のような振る舞いを検出するメトリクスとして使用するためには，同時にリビジョン数 ($Revs(t)$) について考慮する必要があると考えられる．

表 2　$Revs(t)$ の値ごとにおける $score(t)$ の受講者数 ($t = 10$)

$score(t)$	$Revs(t)$							
	2–3	4–5	6–7	8–9	10–11	12–13	14–15	16–
100	97	35	13	10	4	2	0	0
75	1	0	0	0	0	0	0	0
50	0	1	0	1	0	0	0	0
25	0	2	1	4	2	0	0	1
0	18	8	7	10	3	7	0	0

5.3.2　$Revs(t)$

$Revs(t)$ と $score(t)$ の相関係数は，-0.289 ～ -0.407 と $Freq(t)$ と比べてやや強く，すべての t で有意な相関がみられた ($p < 0.01$)．この結果は，一定時間内の修正・コンパイル回数が多い受講者の $score(t)$ が低いことを示しており，4.1 節で述べた仮説 H2（一定時間の修正・コンパイルの回数が多い）の特徴を有している可能性を示唆する．

また，t について 5 の時に最も相関が小さく，10 以上に大きな違いが見られなかった．本研究で定義した $Revs(t)$ は t 分間に提出された累積リビジョン数のため，早い時間に 100 点に到達した受講者は t が大きくなったときにも $Revs(t)$ が変化せず，$Revs(t)$ と $score(t)$ の相関も一定のまま変化しない．そのため，t が増加したときに $score(t)$ が低いまま $Revs(t)$ のみが増大するような受講者がいても全体としての相関が変化しないことが考えられる．$Revs(t)$ を用いて無作為修正者を検出するときには，すでに $score(t)$ が 100 点に到達している受講者を除外することが望ましいと考えられる．

表 2 に $Revs(t)$ と $score(t)$ のクロス集計表を示す．表は $score(t)$ が低く $Revs(t)$ が小さい受講者や，$score(t)$ が高く $Revs(t)$ が高い受講者が複数いることを表している．リビジョン間のソースコード変化を見ると，$score(t)$ が高く $Revs(t)$ が高い受講者は特定の行を繰り返し修正し，正答に到達していた (図 4)．これは，無作為な修正を繰り返しているうちに，たまたま正答に到達した可能性がある．

5.3.3　$DiffLine(1, t)$

$DiffLine(1, t)$ は，相関係数が -0.135 ～ -0.246 と弱かったが，すべての t で有意な相関がみられた ($p < 0.05$)．この結果は，1 行以下の修正が多い受講者の $score$ が低いことを示しており，4.1 節で述べた仮説 H3（1 回あたりの修正量が小さい）の特徴を有している可能性を示唆する．$DiffLine(1, t)$ と $score(t)$ の相関係数が低い原因として，$Freq(t)$ や $Revs(t)$ と同様に，点数が高い受講者も細部の修正を施すことが考えられる．また，(2) 式より，$DiffLine(1, t)$ は修正回数 $Revs(t)$ に依存するメトリクスとして定義されている．そのため，1 行の修正を 1 回行ったときの影響が $Revs(t)$ が小さい受講者で大きくなる．$DiffLine(1, t)$ を仮説 H3(1 回あたりの修正量が小さい) のような振る舞いを検出するメトリクスとして使用するためには，同時にリビジョン数 ($Revs(t)$) について考慮する必要があると考えられる．

5.3.4　$Freq(t)$ と $Revs(t)$ による群の差

$Freq(t)$ や $DiffLine(1, t)$ は $Revs(t)$ の大小によって異なる受講者の様子を表している可能性がある．そのため，$Revs(t)$ の大小でグループ分けしたうえで，$Revs(t)$ の大きいグループ

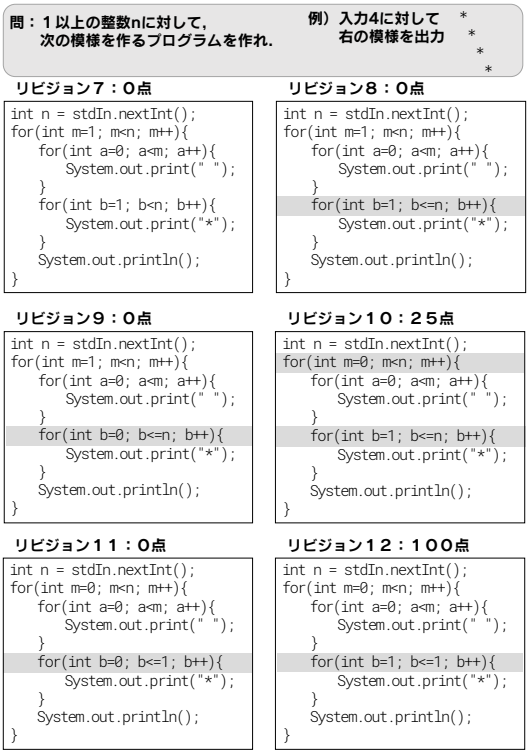

図 4　無作為な修正の繰り返しで正答に到達した例

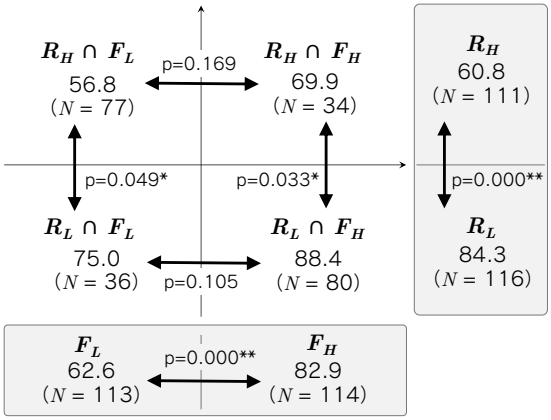

図 5　各群の平均 $score(t)$

に対して $Freq(t), DiffLine(1, t)$ で識別する必要があった．そこで，受講者を $Freq(t)$ や $Revs(t)$, $DiffLine(1, t)$ の大小で 2 群に分け，各群の $score(t)$ に差があるか検証する．本稿では紙面の都合上，$Freq(t)$ と $Revs(t)$ の検証結果のみ述べる．

$Revs(t)$ が中央値以上の受講者群を R_H，中央値より小さい受講者群を R_L とする．同様に，$Freq(t)$ が中央値以上の受講者群を F_H，中央値より小さい受講者群を F_L とする．

$t = 10$ における各群の $score(t)$ の平均値と検定 (Welch の t 検定) の結果を図 5 に示す．結果は $score(t)$ の平均は F_L より F_H が，R_L より R_H が有意に高い．また，各群の平均値について $R_H \cap F_L$ より $R_L \cap F_L$ が有意に高く，$R_H \cap F_H$ より $R_L \cap F_H$ が有意に高い．$R_H \cap F_L$ と $R_H \cap F_H$，および $R_L \cap F_L$ と $R_L \cap F_H$ の間には平均値の差が見られたものの，有意差は見られなかった．

問：キーボードで入力されたnに対し，0〜nまでのうち
3の倍数だけ加算するプログラムをfor文を用いて作れ

リビジョン9
0点
Syntax Error

```
for(int i=0;i<=n;i=i+1){
    sum = sum + 3%n == 0;
}
```

リビジョン10
0点
Syntax Error

```
for(int i=0;i<=n;i=i+1){
    sum = sum + (3%n == 0);
}
```

リビジョン11
0点
Syntax Error

```
for(int i=0;i<=n;i=i+1){
    sum = sum + (i%3 == 0);
}
```

リビジョン12
0点
Syntax Error

```
for(int i=0;i<=n;i=i+1){
    sum = sum && (i%3 == 0);
}
```

リビジョン13
50点

```
for(int i=0;i<=n;i=i+1){
    sum = sum + i/9 +i/6 + i/3;
}
```

リビジョン14
0点
Syntax Error

```
for(int i=0;i<=n;i=i+1){
    sum = sum +(i%9 == 0)+
      (i%6 == 0)+(i%3 == 0);
}
```

図 6　戦略が見られない修正やコンパイルエラーを含むような修正

F_H に属する受講者は修正行が偏っているため，演習中にエラーが出た時に，受講者がソースコードを修正すべき箇所は掴んでいたことが考えられる．しかし，F_H 群の中でも $Revs(t)$ の多い受講者 $(R_H \cap F_H)$ は特定の行に対して戦略が見られない修正やコンパイルエラーを含むような修正を連続して提出している様子が修正履歴から確認できた（図 6）．一方で，$Revs(t)$ の少ない受講者 $(R_L \cap F_H)$ は特定の行に対して 1 回もしくは 2 回の（主にフォーマットに関する）修正で正答に到達していた．これの受講者はいずれも $Freq(t)$ が高くなるが，$Revs(t)$ と組み合わせる事で，適切に弁別できると思われる．

F_L に属する受講者は修正行が偏っておらず，広範囲にわたっている．修正履歴を確認すると，$Revs(t)$ の少ない受講者 $(R_L \cap F_L)$ は最初に提出したリビジョンで複数の行が誤っているが，次の 1 回もしくは 2 回の修正で正答に到達している．一方で，$Revs(t)$ の多い受講者 $(R_H \cap F_L)$ は複数の行に対して戦略が見られない修正やコンパイルエラーを含むような修正を連続しており，修正方法が把握できていない様子であった．

6. 結　　論

本稿では，演習の内容を理解しないままソースコードの修正を繰り返す「無作為修正者」を検出することを目的に，受講者が提出したソースコードのスナップショットから無作為修正者を識別するメトリクスを提案し，その有用性を検証した．実験の結果，修正行の偏りを表す $Freq(t)$ と t 分間のリビジョン数を表す $Revs(t)$ の両方が高い受講者の点数が低く，また，その編集履歴から無作為修正者である事が示唆された．本稿の結果から，$Freq(t)$ と $Revs(t)$ を組み合わせることで，無作為な修正を繰り返し正答にたどり着かない受講者を把握し，指導することで教育効果を高められると考えられる．OJS にメトリクスを動的に算出・表示する機能を追加することで，支援が不要な受講者の作業を妨げることなく無作為修正者に対して優先的に支援が可能になる．

今後の課題として，メトリクスと分析方法の見直しが挙げられる．本稿で提案した 4 メトリクスのうち $AveDiff(t)$ には $score(t)$ との有意な相関が見られず，他のメトリクスについても単体では弱い相関しか見られなかった．$Freq(t)$ と $Revs(t)$ の高低による平均点の差を分析した結果から，$Freq(t)$ が大きい受講者には無作為修正者だけでなく，少ない回数の細部修正を行っている受講者がおり，そのため相関が弱くなったと考えられる．同様に，$Revs(t)$ が小さい受講者の中には演習を理解している受講者だけではなく，時間あたりの提出回数が少ない（作業速度が遅い，または，正答がわからず提出できない）受講者も含まれており，相関が弱くなったと考えられる．今後の研究では，本稿で定義したメトリクスを組み合わせる事で受講者をグループ分けし，修正履歴などからその特徴を分析する必要がある．

また，リビジョン 1 からの経過時間を表す t を用いた分析の中で，$Freq(t)$ の値が急激に大きくなる区間と変化しない区間が存在する場合と，時間の経過と共に安定して値が大きくなる場合が見られた．これは無作為な修正を行う受講者であっても，常に無作為な修正を行っているわけではなく，修正箇所の特定とその修正を交互に行っていると考えられる．今後，メトリクス値の時系列変化を分析する事で，受講者が修正箇所の特定をできたか否かを識別することができると期待される．

本研究の発展として，受講者の特性を表すメトリクスを特定することで機械学習を用いた無作為修正者の識別が可能になると思われる．受講者の特性を自動で識別することができれば，教員による指導が行われない e-learning コンテンツを利用した学習への適用も可能になる．また，本稿の分析では最初のリビジョンを提出してからの経過時間 t の時点におけるメトリクスとその時点の点数について分析を行っている．t の時点におけるメトリクスと，t から一定時間経過した（たとえば $t+30$）時点における点数やメトリクスの関係を見ることで，長時間正答にたどり着かない受講者をより早い時点で検出することは興味深い発展の 1 つと考えられる．

文　　献

[1] K. Fujiwara, K. Fushida, H. Tamada, H. Igaki, and N. Yoshida, “Why novice programmers fall into a pitfall?: Coding pattern analysis in programming exercise,” Fourth International Workshop on Empirical Software Engineering in Practice (IWESEP 2012)IEEE, pp.46–51 2012.

[2] 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.3–13，2018.

[3] 槇原絵里奈, 井垣　宏, 吉田則裕, 藤原賢二, 飯田　元, “プログラミング演習における探索的プログラミング行動の自動検出手法の提案，” コンピュータ ソフトウェア，vol.35，no.1，pp.110–116，2018.

[4] 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，” 第 18 回データ工学ワークショップ，pp.1–6，2007.

[5] 松永賢次，“導入プログラミング教育におけるオンラインジャッジシステムの活用の試み，” 情報科学研究，vol.31，pp.25–41，2010.