ミューテーション法を応用した学習者プログラムの誤り箇所特定方法の考察

月原 花菜[1]　山本 詠一朗[3]　中島 亜美[2]　蜂巣 吉成[1]　吉田 敦[1]　桑原 寛明[1]

概要：本研究では，コンパイルできるが期待した実行結果が得られないプログラムを学習者が自分で誤り箇所に気付いて修正できるような支援を行うことを目的に，ミューテーション法を応用して実行結果から間違い箇所を絞り込む方法を考察する．模範解答プログラムに学習者がよくやる誤りを埋め込んでミュータントを作成し，学習者プログラムの実行結果とミュータントの実行結果が一致したとき，学習者プログラムにミュータントと同じ誤りがあるというように特定を行う．また，実行結果から間違い箇所を特定することで，学習者が間違えていないと思い込んでいる間違いも見つけ出すことができる．

キーワード：ミューテーション法，プログラミング演習，誤り箇所特定

Localize a Fault in a Learner's Program using Mutation Analysis

1. はじめに

大学のプログラミング演習において，学習者のプログラムがコンパイルは可能だが期待する実行結果が得られないことがある．学習者は自身で誤り箇所を見つけようとするが，エラーメッセージが出力されないので誤り箇所が分からないことがある．教員や TA に質問しようとしても学習者の数に比べて教員と TA の数が少ないので，アドバイスがもらえず，やる気が削がれてしまうことがある．

武藤らはコーディングに行き詰まりコンパイルできる状態までソースコードを書くことが困難な学習者に対し，模範解答を使用してチャットボット上で二者択一の質問を行い，どのようなコードを書くべきかを支援する方法を提案している [1]．しかし，学習者が自分自身で間違えているか判断を行うので，学習者が間違えていると認識できていない箇所については支援が十分でないという問題がある．

本研究は，コンパイル可能だが期待する実行結果が得られないプログラムを，学習者が自分で誤り箇所に気付いて修正できるような支援を行うために，実行結果から間違い箇所を絞り込む方法を考察する．ミューテーション法を応用し，模範解答プログラムに学習者のよくある誤りを含んだプログラム (ミュータントと呼ぶ) を作成し，実行結果を得る．学習者のプログラムの実行結果がどのミュータントの実行結果と一致するか比較し，実行結果が一致する場合，ミュータントと同じ誤りをしている可能性があるとして絞り込みを行う．効率よく間違い箇所の絞り込みを行うためにテストケースの入力の値の順番を決める決定木を利用する．同じ実行結果となるミュータントが複数あった場合の誤りの推定が可能か，学習者が模範解答とは異なるプログラムを記述していた場合の誤りの推定が可能か考察する．本研究を実際の教育現場に用いる場合，ルールベース型のチャットボットを使用することで実用化できると考える．教員は，演習の前に模範解答からミュータントを作成し，実行，決定木の作成を行う．決定木からチャットボットのシナリオ (質問木) を作成し，演習中に作成したシナリオに沿って学習者に質問をする．そのため，ルールベース型のチャットボットを利用することを想定し，学習者への質問の順番を決定するためのシナリオを質問木として作成する方法を考察する．

本稿は著者ら (月原，山本，中島) の卒業論文 [5] に基づいており，決定木作成アルゴリズムを見直して提案方法の説明を詳細化，間違い箇所絞り込みに関する評価を追加し，考察を再構成している．

1　南山大学
2　南山大学卒
3　南山大学卒，現シンフォニアテクノロジー株式会社

2. 背景技術・関連研究

ミューテーション解析とは，テストケースの欠陥検出能力を測定する方法のひとつである．テスト対象プログラムに故意に誤りを含ませ，テストケースによってその誤りが検出できたかどうかによって能力の測定を行う [2][3]．本研究では，誤りをミューテーション，誤りを含んだプログラムをミュータント，ミューテーションをどのように作成するかの規則をミューテーションオペレータと呼ぶ．

武藤ら [1] は，教員や TA にすぐ質問できず学習者のやる気が削がれてしまう問題に対し，コーディングに行き詰まりコンパイルまで至っていない学習者を対象とした，チャットボットを使用して模範解答から二者択一の質問を行い，どのような文を書くべきかを支援する方法を提案している．学習者が回答しやすく効率のいい質問を考えることで，問題解決の糸口を掴みやすくし，質問効率を高めることで学習者が欲する解答に早く辿り着くことができ，学習者の学習意欲が向上するのではないかと考えている．武藤ら [1] と本研究の違いは，対象としているソースコード，間違い箇所の特定方法の 2 つである．武藤ら [1] の対象としているソースコードはコンパイルに至っていないコードだが，本研究の対象としているソースコードはコンパイルは可能だが期待される実行結果が得られないコードである．間違い箇所を特定する方法は，武藤ら [1] は模範解答を使用し選択肢を用いた質問を行うことで間違い箇所を特定していたが，この方法では学習者が間違えていないと思い込んでいる間違いを特定できないと考え，本研究では実行結果を用いて間違い箇所を特定する．

3. ミューテーション法を応用した学習者プログラムの誤り箇所特定方法

3.1 概要

本研究では，ミューテーション法を応用した学習者プログラムの誤り箇所特定方法を考察する．ミューテーション法を応用して実行結果から誤り箇所を絞り込む理由は，学習者自身が気がついていない間違いを見つけ出すためである．図 1 は，提案する方法の流れを図にしたものである．青色で色付けしている質問木作成以降の部分は，本研究では対象外である．

本提案方法の工程は次のようになる．1 で使用するミューテーションオペレータはツール上で用意してある．

(1) 模範解答からミュータントの作成
(2) 作成したミュータントのコンパイル・実行と実行結果の保存
(3) 実行結果をテストケースごとに分類し，実行結果グループの作成・学習者に質問するテストケースの順番を決めるための決定木の作成

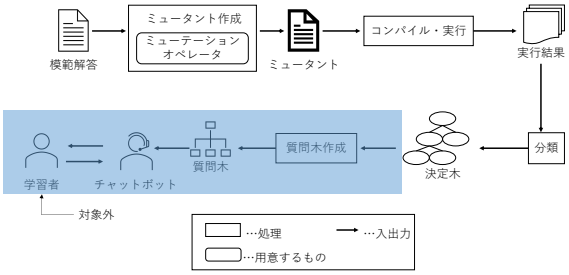

図 1 提案した方法の流れ
Fig. 1 The flow of the proposed method

次に教員が行う作業の説明を行う．初めに，教員は各行に「その行がどのような動きをしているか」というコメントを記入した模範解答とテストケースを用意する．コメントはチャットボットで学習者に質問するときに用いる．ソースコード 1 に例を示す．main 関数で実行結果を”&@”で区切って出力している．教員は 1 のツールを使用して用意した模範解答を書き換え，ミュータントを作成する．その後，教員は 2 のツールを実行して作成したミュータントをコンパイル・実行し，実行結果をテキストファイルに出力する．教員は 3 のツールを用いて，どのテストケースの入力の値から学習者に送れば効率よく目的のミュータントに辿り着けるかを決める決定木を作成する．効率を意識する理由は，教育現場に利用する際，学習者に対しテストケースを 1 つ提示し，その実行結果をもらうことを繰り返すことで誤り箇所を特定するため，チャットボットと学習者のやり取りを少なくし，短時間で誤り箇所を特定するためである．

3.2 前提条件

本研究の対象ソースコードは，コンパイルは可能だが期待される実行結果が得られないソースコードとし，対象言語は C 言語とする．対象とするミュータントは，1 つのミュータントに対し 1 つのミューテーションとし，2 つ以上ミューテーションが存在するプログラムは本研究では対応しないものとする．ミューテーションに書き換える箇所は，main 関数を除く実際に学習者が書く箇所のみとし，main 関数の誤りは対象外とする．ミューテーションオペレータは学習者が行いそうな間違いとする．また模範解答を用意する際に，各行にその行が何をしている部分かをコメントで残すこと，実行結果を出力するプログラムは組み合わせの数を求めるプログラムソースコード 1 の main 関数のように書くものとする．

3.3 ミューテーションオペレータ

ミューテーションオペレータは文献 [2] をもとに設定す

ソースコード 1　組み合わせの数を求めるプログラム 模範解答

```
1  int comb(int a,int b){
2      int result;
3      int upper = 1; // 分子の計算用の変数宣言
4      int lower = 1; // 分母の計算用の変数宣言
5
6      if(a <= 0 || b <= 0){// 全体の個数を表す変数と取り出す個数を表す変数がどちらも 0 以下の時
7          return -1;
8      }
9      if(a == b){// 全体の個数を表す変数と取り出す個数を表す変数が同値の時
10         result = 1;// 全体の個数を表す変数と取り出す個数を表す変数が同値の時の結果の代入
11         return result;
12     }
13     if(a < b){// 全体の個数を表す変数が取り出す個数を表す変数より小さい時
14         return -1;
15     }
16
17     for(int i = 0;i < b;i++){// 分子の計算
18         upper = upper * (a - i);
19     }
20     for(int i = 0;i < b;i++){// 分母の計算
21         lower = lower * (b - i);
22     }
23
24     result = upper / lower;
25     return result;
26 }
27 int main(void){
28     int case1 = comb(5,3);
29     int case2 = comb(8,2);
30     int case3 = comb(4,4);
31     int case4 = comb(2,5);
32
33     printf("%d\n",case1);
34     printf("&@\n");
35     printf("%d\n",case2);
36     printf("&@\n");
37     printf("%d\n",case3);
38     printf("&@\n");
39     printf("%d\n",case4);
40
41     return 0;
42 }
```

る．文献 [2] では，関係演算子が違う，変数に代入する値が違う，初期値の宣言忘れなどが提案されている．我々自身の経験から関係演算子間違いと初期値の値間違いに着目した．この間違いに着目した理由は，プログラミング演習の授業でよくあった間違いであり，初学者が起こしやすい誤りと考えたからである．

本研究のミューテーションオペレータを次に示す．箇条書きの 7 つ目は，ソースコード 1 の 3 行目の変数に初期値など値を代入する際の誤りを想定している．

- $==$ が $<=, >=, <, >, !=$
- $<=$ が $==, >=, <, >, !=$
- $>=$ が $==, <=, <, >, !=$
- $<$ が $==, >, <=, >=, !=$
- $>$ が $==, <, <=, >=, !=$
- $!=$ が $==, <, >, <=, >=$
- $=0$ が$=1$,$=1$ が$=0$

3.4　ミュータント作成

教員が用意した模範解答プログラムから 3.3 節に基づいてミュータントを作成する．ソースコード 1 を用いて作成されるミュータントの例を次に示す．NO はミュータントのファイル名である．以下の例では 35 種類のミュータントが作成される．

3.5　プログラムの実行

ミュータント作成ツールで作成されたミュータントを自動でコンパイル・実行し，実行結果をテキストファイルに出力する．ミュータントが無限ループであった場合には，タイムアウトを使って判断する．ミュータントの実行結果と模範解答の実行結果が一致した場合，正しいプログラムと判断する．

3.6　決定木作成

学習者プログラムの実行結果がどのミュータントの実行結果と一致するか調べるために決定木を用いる．作成する決定木は，どのテストケースの入力の値から学習者に送れば効率よく目的のミュータントに辿り着けるかを決める決定木である．決定木は，作成したミュータントの実行結果を各テストケースごとに同じ実行結果のものを同じグループになるように分類して作成する．決定木のノードはミュータントの集合，枝はテストケースと実行結果の値である．

貪欲法によるアルゴリズム 1 で決定木を作成する．最初は makeNode(root) となる．root は全ミュータントの集合である．アルゴリズム内の divide(M, t) は，ミュータントの集合 M をテストケース t の結果で同値分割する関数で，実行結果とミュータントの組の集合を返す．この方法でテストケースの順番を決める理由は，枝のテストケースの順

表 1 ミュータントの例
Table 1 Example of Mutants

NO	説明
1-1.c	9 行目：== が !=
2-1.c	9 行目：== が >
3-1.c	9 行目：== が >=
4-1.c	9 行目：== が <
5-1.c	9 行目：== が <=
21-1.c	13 行目：< が ==
21-2.c	17 行目：< が ==
21-3.c	20 行目：< が ==
22-1.c	13 行目：< が !=
22-2.c	17 行目：< が !=
22-3.c	20 行目：< が !=
23-1.c	13 行目：< が >
23-2.c	17 行目：< が >
23-3.c	20 行目：< が >
24-1.c	13 行目：< が >=
24-2.c	17 行目：< が >=
24-3.c	20 行目：< が >=
25-1.c	13 行目：< が <=
25-2.c	17 行目：< が <=
25-3.c	20 行目：< が <=
26-1.c	6 行目：<= が ==
26-2.c	6 行目：<= が ==
27-1.c	6 行目：<= が !=
27-2.c	6 行目：<= が !=
28-1.c	6 行目：<= が >
28-2.c	6 行目：<= が >
29-1.c	6 行目：<= が >=
29-2.c	6 行目：<= が >=
30-1.c	6 行目：<= が <
30-2.c	6 行目：<= が <
31-1.c	3 行目：=1 が=0
31-2.c	4 行目：=1 が=0
31-3.c	10 行目：=1 が=0
32-1.c	17 行目：=0 が=1
32-2.c	20 行目：=0 が=1

番で木の高さが変わるが，木の高さを低くするためである．

ソースコード 1 のプログラムを例に説明をする．まず，ソースコード 1 から節 3.4 に基づいてミュータントを作成し，コンパイル・実行すると実行結果は**表 2** ようになる．T1 から T4 は testcase1 から testcase4 のことである．testcase1 の入力の値は comb(5,3)，testcase2 の入力の値は comb(8,2)，testcase3 の入力の値は comb(4,4)，testcase4 の入力の値は comb(2,5) である．

分類する前のミュータントの集合を M_{all} として，表 2 の実行結果を，各テストケースごとに分類していく．アルゴリズム 1 の devide の結果である．

T1 の実行結果グループは，
$divide(M_{all}, T_1)$ ={(1,{1-1, 2-1, 3-1}), (10,{4-1, 5-1, 21-1, 31-3}), (0,{21-2, 23-2, 24-2, 25-3, 31-1, 31-2}), (60,{21-3,

アルゴリズム 1 決定木作成アルゴリズム

```
function makeNode(n)
    M = n のミュータントの集合
    if M の要素が 1 つ then
        return
    end if
    root から n に至る枝に出現しないテストケースの集合を T とする
    if T = 空集合 then
        return
    end if
    X = 空集合
    for t in T do
        X に divide(M, t) を追加
    end for
    D = X の中で最も多く組ができた集合
    if D の組の集合が 1 つでない then
        for d in D do
            d のミュータントの集合を持つノード c を作る
            n から c に d の出力をラベルにした枝を作る
            makeNode(c)
        end for
    end if
end function
```

表 2 ミュータントの実行結果
Table 2 Result table of mutants

NO.	T1	T2	T3	T4
1-1.c	1	1	1	1
2-1.c	1	1	1	-1
3-1.c	1	1	1	-1
4-1.c	10	28	1	1
5-1.c	10	28	1	1
21-1.c	10	28	1	0
21-2.c	0	0	1	-1
21-3.c	60	56	1	-1
22-1.c	-1	-1	1	-1
23-1.c	-1	-1	1	0
23-2.c	0	0	1	-1
23-3.c	60	56	1	-1
24-1.c	-1	-1	1	0
24-2.c	0	0	1	-1
24-3.c	60	56	1	-1
25-2.c	20	168	1	-1
25-3.c	0	0	1	-1
27-1.c	-1	-1	-1	-1
27-2.c	-1	-1	-1	-1
28-1.c	-1	-1	-1	-1
28-2.c	-1	-1	-1	-1
29-1.c	-1	-1	-1	-1
29-2.c	-1	-1	-1	-1
31-1.c	0	0	1	-1
31-2.c	0	0	1	-1
31-3.c	10	28	0	-1
32-1.c	2	3	1	-1
32-2.c	30	56	1	-1

23-3, 24-3}), (-1,{22-1, 23-1, 24-1, 27-1, 27-2, 28-1, 28-2, 29-1, 29-2}), (20,{25-2}), (2,{32-1}), (30,{32-2})} の 8 グループである．

T2 の実行結果グループは，
$divide(M_{all}, T_2)$ ={(1,{1-1, 2-1, 3-1}), (28,{4-1, 5-1, 21-1, 31-3}), (0,{21-2, 23-2, 24-2, 25-3, 31-1, 31-2}), (56,{21-3, 23-3, 24-3, 32-2}), (-1,{22-1, 23-1, 24-1, 27-1, 27-2, 28-1, 28-2, 29-1, 29-2}), (168,{25-2}), (3,{32-1})} の 7 グループである．

T3 の実行結果グループは，
$divide(M_{all}, T_3)$ ={(1,{1-1, 2-1, 3-1, 4-1, 5-1, 21-1, 21-2, 21-3, 22-1, 23-1, 23-2, 23-3, 24-1, 24-2, 24-3, 25-2, 25-3, 31-1, 31-2, 32-1, 32-2}), (0,{31-3}), (-1,{22-1, 23-1, 24-1, 27-1, 27-2, 28-1, 28-2, 29-1, 29-2})} の 3 グループである．

T4 の実行結果グループは，
$divide(M_{all}, T_4)$ ={{1,(1-1, 4-1, 5-1)}, {-1,(2-1, 3-1, 21-2, 21-3, 22-1, 23-2, 23-3, 24-2, 24-3, 25-2, 25-3, 27-1, 27-2, 28-1, 28-2, 29-1, 29-2, 31-1, 31-2, 32-1, 32-2, 31-3, 32-1, 32-2)}, {0,(21-1, 23-1, 24-1)}} の 3 グループである．

よって，実行結果グループが一番多いのは testcase1 であるため，testcase1 の入力の値から決定木のノードと枝を作成する．testcase1 で分けた集合を利用して，できた組の数だけ葉ノードを作成していく．ノードのラベルにミュータントプログラム，root から葉ノードに出力 r をラベルにした枝を作る．できた葉ノードに対してまた同じ操作を繰り返していき，決定木を作成していく．ソースコード 1 についてアルゴリズム 1 で作られる決定木の例を図 2 に示す．

3.7 ツールの設計と実現

本研究では，模範解答から決定木の自動生成を行うツールを試作した．決定木の作成までには以下の 4 つの段階がある．また，図 3 は，ツールの内部の図である．ミュータントの作成には属性付き字句系列に基づくプログラム書換え処理系である TEBA[4] を用いて実現し，ミュータントのコンパイル・実行，決定木作成のプログラムは Perl を用いて実現した．

- 模範解答からミュータントの作成・コメントの抽出・模範解答実行結果保存
- ミュータントの実行・無限ループの削除
- 等価ミュータントの削除
- 実行結果から決定木を作成

4. 考察

4.1 決定木の考察

本研究ではツールで作成したミュータントを実行し，各テストケースごとに実行結果のグループ分けを行い，一番グループが分かれたテストケースから順に決定木を作成する．これにより，高さの低い決定木を作成することができ，葉の要素数を抑えることができる．

本研究の作成方法で作成した決定木と，testcase1 から順に作成する決定木の各枝の末端に辿り着くまでに必要な実行結果の数を比較し，どちらが少ないか比較する．計算方法は，各枝の一番下の葉に到着するまでに使用したテストケースの総数/全ての葉の数としている．図 2 では，各枝の葉に到達するまでに平均で約 1.917 個の実行結果が必要である．次に testcase1-testcase2-testcase3…という順番で作成される組み合わせの決定木を図 4 として以下に示す．testcase1-testcase2-testcase3…という順番で作成される決定木の場合は，平均で約 2.692 個必要になる．次に文字列が回文か判定する問題で，ミュータントを 16 個作成し，テストケース 5 個で決定木を作成した．アルゴリズム 1 で作成した決定木では，各枝の葉に到達するまでに平均で約 2.4 個の実行結果が必要であった．次に testcase1-testcase2-testcase3…という順番で作成される決定木では，各枝の葉に到達するまでに平均で約 4.8 個の実行結果が必要であった．

以上のことから testcase1 から順番に決定木を作るよりも，実行結果グループが一番分かれるテストケースの入力の値から作る方が高さが低くなる．

4.2 間違い箇所絞り込みの評価

模範解答と違う書き方をされたソースコードの誤り箇所を特定できるか評価を行った．実験方法は次の通りである．

(1) 組み合わせのプログラム，'*' で三角形を出力するプログラム，文字列中の大文字を小文字に，小文字を大文字に変換するプログラム，文字列が回文か判定するプログラムの計 4 種類の模範解答プログラムを用意
(2) 学習者が書いたとする (学習者プログラムとする) 組み合わせのプログラム，'*' で三角形を出力するプログラム，文字列中の大文字を小文字に，小文字を大文字に変換するプログラム，文字列が回文か判定するプログラムの計 4 種類のプログラムを用意 (プログラムには誤りは含まれていないものとし，模範解答プログラムと書き方が異なるものとする)
(3) 用意した模範解答から決定木を作成
(4) 用意した学習者プログラムに，本研究のミューテーションオペレータを前提とした誤りを無作為に埋め込み，各プログラムに 5 種類ずつ，計 20 種類の誤りを含んだ学習者プログラムを作成する
(5) 末端の葉に着くまで決定木の順番に沿ってテストケースの入力の値を実行
(6) 末端の葉に書かれたファイルの誤り箇所・ミューテーションの種類と埋め込んだ誤りが一致するか確認する

学習者が書いたとする誤りを含んだ組み合わせプログラムの例を 3 つあげる (ソースコード 2)．組み合わせの数を求める学習者プログラムと模範解答プログラムの違いは，

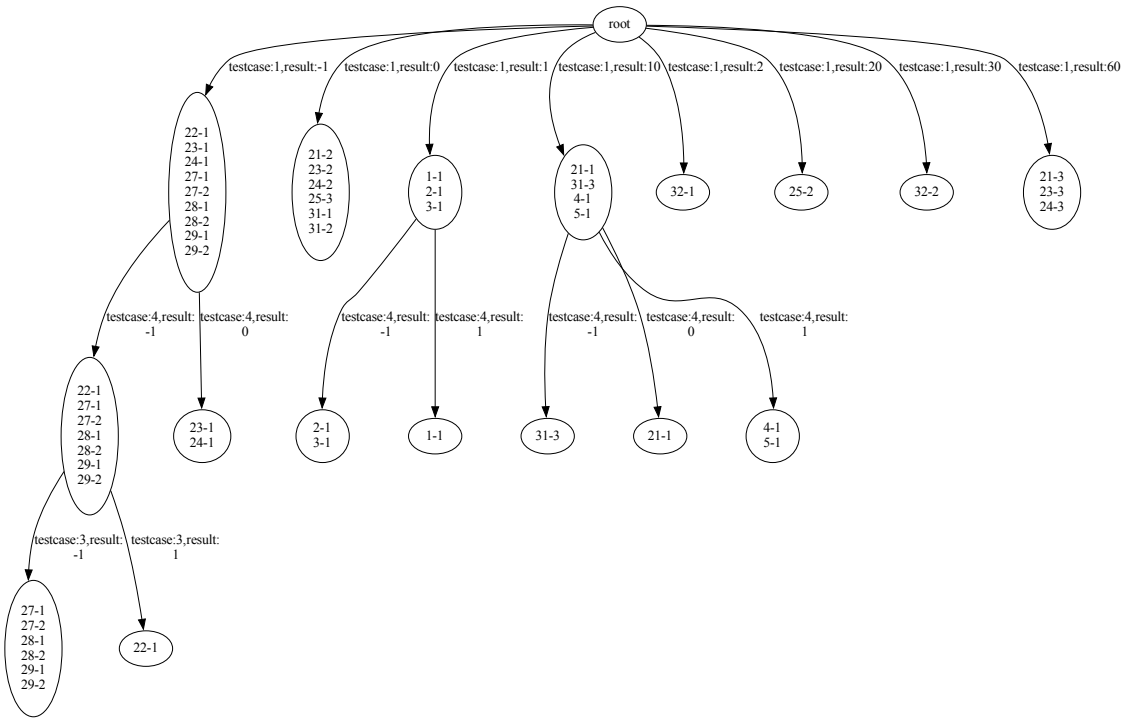

図 2　決定木の例 (アルゴリズム 1)
Fig. 2　Example of a decision tree(algorithm1)

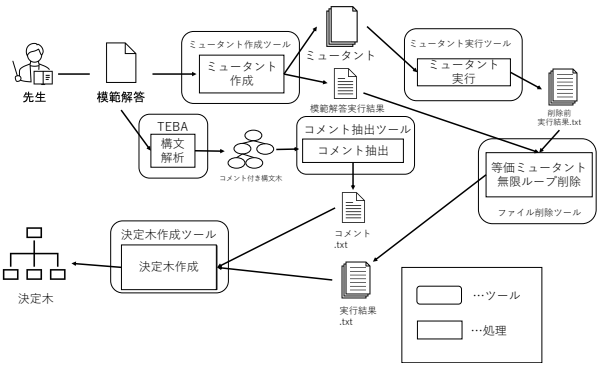

図 3　ツール内部の流れ
Fig. 3　Flow of our tool

分母の計算を分けているところである．

学習者が書いたとする誤りを含んだ組み合わせプログラム 1 つ目 (以下 ver1 とする) はソースコード 2 の 3 行目の $= 1$ が $= 0$ になっているプログラムである．ver1 の実行結果は testcase1 が 0，testcase2 が 0，testcase3 が 1，testcase4 が-1 である．学習者が書いたとする誤りを含んだ組み合わせプログラム 2 つ目 (以下 ver2 とする) はソースコード 2 の 19 行目の $i < a$ が $i <= a$ になっているプログラムである．ver2 の実行結果は testcase1 が 0，testcase2 が 0，testcase3 が 1，testcase4 が-1 である．学習者が書いたとする誤りを含んだ組み合わせプログラム 3 つ目 (以下 ver3 とする) はソースコード 2 の 25 行目の $i < c$ が $i > c$ になっているプログラムである．ver3 の実行結果は testcase1 が 20，testcase2 が 20160，testcase3 が 1，testcase4 が-1 である．図 2 の決定木を用いて末端に書いてあるファイル ($[1-32]-[1-9].c$) と ver1 ver3 を比較すると，ver1 は $31-1.c$ と同じ初期値の定義をしている箇所が誤っており，また誤り方も一致しているため，学習者プログラムの誤りの特定が可能であると考えられる．ここで言う「特定」とは，学習者プログラムの誤りを確実に絞り込めることとする．ver2 は $24-2.c$ と同じ分子の計算する箇所が誤っているが，誤り方は一致しなかった．ver2 については，学習者に対するヒントの出し方によっては学習者プログラムの誤りの推定が可能だと考えられる．ここで言う「推定」

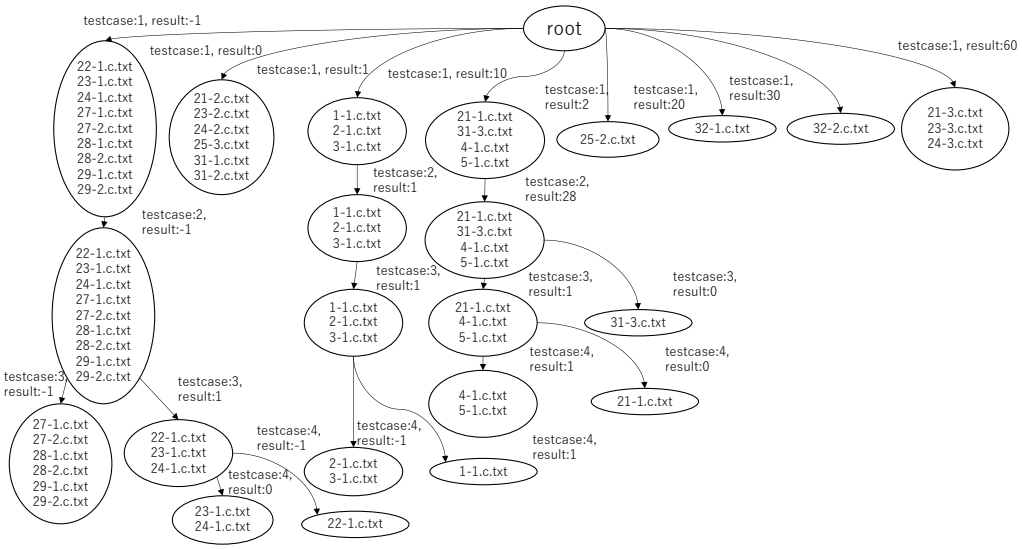

図 4　組み合わせ決定木 (testcase1 から順)

Fig. 4　Example of a decision tree(From testcase1)

ソースコード 2　組み合わせの数を求める学習者プログラム

```
int comb(int a,int b){
    int result;
    int upper = 1; //ver1 int upper = 0;
    int lower = 1;
    int lower2 = 1;
    int c = a - b;

    if(a <= 0 || b <= 0){
        return -1;
    }
    if(a == b){
        result = 1;
        return result;
    }
    if(a < b){
        return -1;
    }

    for(int i = 0;i < a;i++){ //ver2 i <= a
        upper = upper * (a - i);
    }
    for(int i = 0;i < b;i++){
        lower = lower * (b - i);
    }
    for(int i = 0;i < c;i++){ //ver3 i > c
        lower2 = lower2 * (c - i);
    }

    result = upper / (lower * lower2);
    return result;
}
```

とは，学習者プログラムの誤りを絞り込める可能性はあるが，確実に絞り込める保障がない状態のことである．そのため，誤ったヒントを学習者に提示してしまう可能性がある．ver3 は分母の計算をする部分が誤っているが，決定木の末端に書いてあったファイル 25 − 2.c と異なる動きをしている箇所が誤っており，また誤り方も一致しなかった．ver3 のようなパターンは，学習者に対するヒントも見当違いなヒントになると想定できるため，学習者プログラムの誤りの特定・推定が不可能であると考えられる．

他プログラムも同様に実験を行うと，合計で 20 種類中 9 種類は学習者プログラムの誤りの特定が可能，5 種類は学習者プログラムの誤りの推定が可能，6 種類は学習者プログラムの誤りの特定・推定が不可能であることがわかった．

以上の結果からアルゴリズムが異なる学習者プログラムの誤りであっても，用意された模範解答と同じ動きをしている部分の誤りであれば，誤りを特定をすることは可能だが，模範解答にない動きをしている箇所の誤りの特定は難しいことがわかった．そのため，模範解答にない動きをしている箇所の誤りの特定を行うためには，模範解答の別解をいくつか用意しておく必要があると考えられる。

5.　教育現場での活用案

本研究を実際の教育現場に活用するためには，ルールベース型のチャットボットを利用し，学習者と対話をすることでサポートを行うことが考えられる。チャットボットから学習者に対し質問するには以下のようなことが考えられる。

まず，質問は「以下の値を入力として実行してください」とテストケースの入力の値を送ることとし，選択肢は実行結果とする．質問木のノードには質問，エッジには選択肢

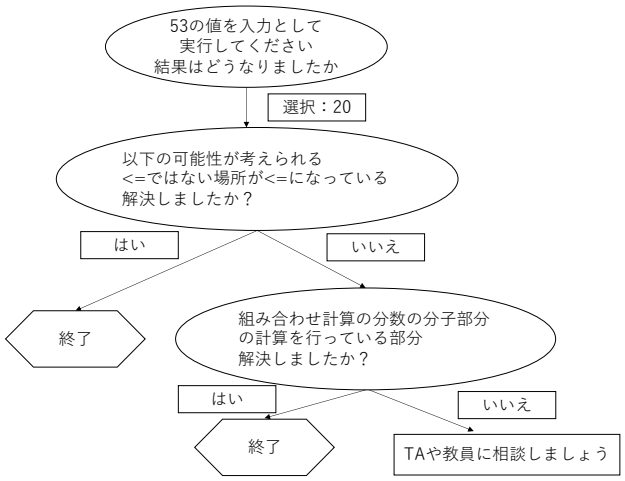

図 5 質問木の例 (一部)
Fig. 5 Example of a part of a question tree

をラベルとしてつける．決定木の葉ノードに該当する部分までは決定木と同じように作成していき，葉ノードに達したときにヒントの設定を行う．ヒントは原則として，学習者に考えさせる内容とし直接的な内容は避けるよう設定する．用意するヒントは 2 つで，書き換えられた後の演算子について，書き換えられた箇所のコード内での役割についてとする．ヒントを二段階に分けた理由は，まず学習者にどの演算子に着目すれば良いかのヒントを与えることで，該当演算子が含まれている文がどのように動いているのかを考えさせるようにし，それでもわからない場合さらに確認する箇所を限定させることで，放置しないようにするためである．2 段階目のヒントは書き換え箇所が同じミュータントが多い順にヒントを出力することとする．2 段階目のヒントの順序の決め方は，葉ノードにあるミュータントを書き換え箇所ごとにグループ分けを行い，分類されたグループのうち最も多くのミュータントが含まれているグループのヒントから出力する．

例としてソースコード 1 の 9 行目が == が ! = に書き換えられていたとする．最初に「! = でない箇所が ! = になっています」というヒントを出す．その後学習者に解決したか尋ねる質問し，解決していない場合コメントから「全体の個数を表す変数と，取り出す個数を表す変数が同値の時の対応をする箇所に間違いがあります」とヒントを出す．このヒントを出しても学習者が理解できない，または誤りが訂正できなかった場合は，教員に相談するよう促す文を提示する．図 5 に質問木の例を示す．円の中の文は学習者に送る質問であり，四角は学習者に送る選択肢である．

しかし，この方法で出力するヒントが学習者が必要としているヒントとは限らない．対処法として，ツールを実行し作成したミュータントの中で学習者が行いやすい誤りを順位付けし，その順番でヒントを提示するように改善する方法がある．順位付けを行うために，教員等の経験則以外に実際にどの誤りが多いかデータを集める必要がある．

6. おわりに

本研究では，コンパイル出来るが期待する実行結果が得られないプログラムの誤りを見つけ出し，自身で解決できるようなヒントを与えることを目的に，ミューテーション法を応用して実行結果から間違い箇所を特定する方法を考察した．テスト実行の結果でミュータントをグループ化して，決定木を作る方法を提案し，その有効性を確認した．

今後の課題として，想定されるミューテーションの再度検討，2 つ以上のミューテーションが存在する際の対応，複数のミュータントが同一の実行結果グループに存在する場合の提示するヒントの出す順番を決めるために，学習者が行いやすい誤りの調査などが挙げられる．

謝辞

本研究の一部は JSPS 科研費 23K11359，2023 年度南山大学パッヘ奨励金 I-A-2 の助成を受けた．

参考文献

[1] 武藤健太，西山雄也：チャットボットを利用したプログラミング学習者に対するコーディング支援方法の提案，南山大学理工学部 2021 年度卒業論文 (2022) http://www.st.nanzan-u.ac.jp/info/gr-thesis/2021/hachisu/pdf/18se052.pdf

[2] Agrawal, H., DeMilo, R.A. et al.: Design of Mutant Operators for the C Programming Language, Technical Report SERC-TR41-P (1989)

[3] S. Hamimoune and B. Falah: Mutation testing techniques: A comparative study, 2016 International Conference on Engineering & MIS (ICEMIS), pp. 1-9, (2016)

[4] 吉田敦，蜂巣吉成，沢田篤史，張漢明，野呂昌満: 属性付き字句系列に基づくソースコード書き換え支援 環境，情報処理学会論文誌，Vol.53，No.7，pp.1832-1849，(2012).

[5] 中島亜美，月原花菜，山本詠一朗：ミューテーション法を応用した学習者プログラムの誤り箇所特定方法の提案，南山大学理工学部 2022 年度卒業論文 (2023) https://www.st.nanzan-u.ac.jp/info/gr-thesis/2022/hachisu/pdf/19se042.pdf