初心者のプログラムを正誤と質で評価するシステム

鄭　佳健[1,a)]　寺田　実[1,b)]

概要：プログラミングを習得するためには練習問題を解くのが効果的である. しかし, 初心者が正解にたどり着くのはそれほど簡単なことではないので, 正解に近い,「惜しい」プログラムを評価する方法が必要である. 一般のプログラミングコンテストなどでは, テスト入力に対する出力が正解と一致するかどうかで正誤を判定する. 本研究ではこれに加えて、本研究では正解プログラムとの近さ、言い換えればプログラムの質の評価を目指す。これによって, 結果は正しくなくても良いプログラムをそれなりに評価でき, また, 逆に結果が正しくとも構造や振る舞いの点で問題のあるプログラムも発見できる. この評価方法から, 正解を得た初心者が過剰に自信を持つことがなく, 不正解を得た初心者も自信をなくすことがないことが期待される. 具体的な評価の方法としては, いくつかのパターンの正解プログラムを用意し, 静的な解析としてプログラムの抽象構文木を用いた類似度と, 動的な解析として変数値の系列の比較による類似度を用いる.

キーワード：初心者プログラム評価

1. はじめに

情報技術をさらに発展させるには情報教育の重要性が指摘されている. 情報技術の中ではプログラミング教育が非常に重要な部分である.2020 年にプログラミング教育は小学校で義務化されている点からプログラミング教育が近年重視されていることが分かる.

プログラミングの勉強には練習問題を解くのが効果的である. 練習問題を正解まで解いたらある程度のプログラミングの基本知識が分かっていると判定できる.

しかし, 初心者に対して, 練習問題を正解まで辿り着くにはそれほど簡単な事ではない. 練習問題を解いてトラブルに遭って, 勉強する自信がなくなるケースがよく見かける. 故に, 初心者に対してプログラミング技術の習得を支援するシステムが必要となる.

2. 目的

本研究は完全な正解以外にも「部分点」を与えることでユーザの意欲を引き出すシステムを目指す.

現在, インターネット上で練習問題を解くサイト (LeetCode [*1], AIZU ONLINE JUDGE [*2]) が多数存在している. しかし, これらのサイトは図 1 のように出力の正誤だけでプログラムを評価している. これはある程度のプログラミング知識が持っている人に対しては問題ないが, 初心者に対しては厳しいと考えられる. 原因としては, 初心者に対して

[1] 電気通信大学大学院, 情報・ネットワーク工学専攻
[a)] t1931091@edu.cc.uec.ac.jp
[b)] terada.minoru@uec.ac.jp

[*1] https://leetcode.com/problemset/all/
[*2] http://judge.u-aizu.ac.jp/onlinejudge/

「間違っている」と知らせても, どのように修正するかが分からない状態がよく発生している. 一方, 正しいという結果が得られても,「どうなっているのかは分からないが, とりあえず動きました」という考えで正解を得てしまう学生も少なくない. 本研究は python を対象として, 正誤だけではなく, ソースコードの質などほかの尺度でプログラミングコードを評価し, プログラムの理解を確かめ, 初心者のプログラミング技術の習得支援を目標としている.

本研究では新たなプログラム評価方法が得られ, 初心者のプログラミング技術の習得に役立つことが期待される. 例えば, 以下のような結果が得られる.

- 結果は正しくないが良いプログラムをそれなりに評価できる.
- 結果が正しくとも構造やスタイルがよくないプログラムを発見できる.

Time Submitted	Status
11/25/2020 17:21	Accepted
11/25/2020 17:20	Runtime Error

図 1 現存の評価方法の例

図 1 は leetcode で問題を解いた後返された結果である. 一行目は正しいなので, 正解という結果が返された. 二行目は構文が間違ったので, 誤りがあったことしか返されない. これだけを見ても, 初心者はどう修正するのかが変わらないので, 自信が無くなる可能性が高い. 本研究で提案した質の点数を加えることで, 方向性やコードスタイル面で評価し, 正解にどれくらい近づいているのを知らせることで, 自信を保つことに繋がる.

3. 関連研究

3.1 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 [1]

槇原らは学生達がどこで探索的プログラミングを行っているのかを検出し, 手間を掛けている所の自動検出手法を提案した.

探索的プログラミングとはプログラミングを書く時にソースコードの修正, コンパイル, 実行, デバッグのサイクルを繰り返すことであり, 簡単に言うと, 試行錯誤である. 探索的プログラミングの繰り返しにより, プログラミングの理解が深まる.

図 2 は探索的プログラミングを行った例である. 学生達の手間をかけている場所が分かれば, より良い指導やより良い教材の作成に繋がる.

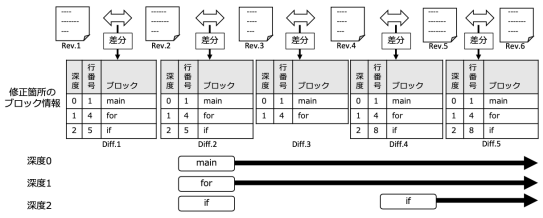

図 3 探索的プログラミング行動の検出

図 2 探索的プログラミングの例 ([1] より)

3.2 Properties of “Good” Java Examples[2]

Abbas らは Java サンプルプログラムの質が可読性 (Readability) と複雑性 (Complexity) から評価できると発表した. 可読性とはコードを理解するために必要な労力のこと, 複雑性とはコードを修正, デバッグ, テストなどに掛かる労力のこと. そして, 古典理論 [4][5] とプログラム理解モデルを通してサンプルプログラムのどこが可読性と複雑性に影響しているのかを分析し, 新しいプログラムの質の評価方法を提案した.

3.3 Generating Data-driven Hints for Open-ended Programming[3]

Thomas らは学生達の Scratch 言語の各段階のコーディング履歴を取得し, 抽象構文木を作成する. そして, 学生達の違う答えにより抽象構文木のノードが複数の状態が保存されている. 例えば, 図 3 の抽象構文木の破線の部分は図 4 のように複数の状態を持っている. 現状は “==” の両辺が変数と

なっているが, 図 4 に示すようにいろいろな構文要素がありうる. これらのうち, もっとも正解へ到達しやすいケースをヒントとして推薦し, プログラムの作成をサポートする.

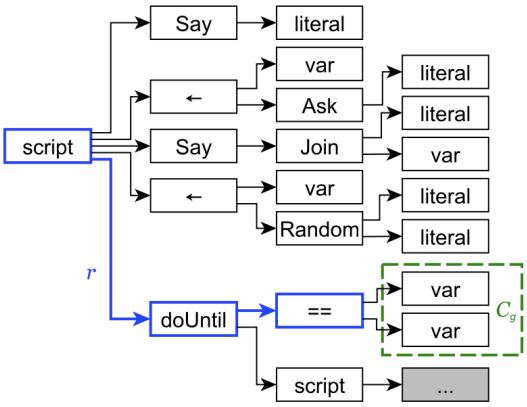

図 3　抽象構文木の例

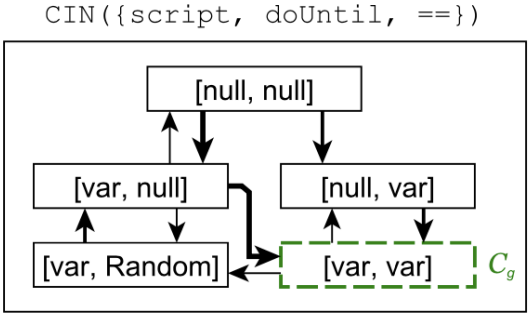

図 4　ノードの状態

4. 提案システム

4.1 概要

本研究は学習者が作成したプログラムの出力に対して正解と一致するかに基づく正誤だけで判定する一般的な判定方法と違い, プログラムの質という視点でプログラムの評価を行う.

プログラムの評価方法について, 以下の 2 つの方法を考えた. 一つ目の方法は静的にソースコードを解析し, プログラムの構造を比較する. 二つ目の方法はプログラムを実行し, 動的にプログラムの動きを解析する.

- 解析方法 1:抽象構文木を用いた正解プログラムとの構造比較
- 解析方法 2:プログラム実行の追跡と解析

4.2 解析方法 1:抽象構文木を用いた正解プログラムとの構造比較

ソースコードを静的に解析するには全体を要素に分解し, 各要素が予約語か, 関数か, 変数かを知る必要がある. 本研究は抽象構文木を利用し, この要件を解決する. 現存の評価方法では実行時エラーが発生すると不正解が返される. この点に対して, 本システムのこの部分は静的に解析するので, 実行時エラーがあっても質の点数をつけることができ, より優しい評価基準になる.

そして, 直観的に抽象構文木をみることが出来れば, 解析に役立つので, 抽象構文木を可視化し, グラフを作成した. 例えば以下のソースコードから抽象構文木を生成すると, 図 5 のようになる.

抽象構文木を作成できたので, 次は得られた抽象構文木を解析し, 質の点数を出す部分について説明する. ここで使う方法は初心者が書いたプログラムと正解のプログラムの編集距離を求め, その編集距離から質の点数を算出するという考えである.

```
1 class test_class:
2     def __init__(self,a,b):
3         self.b = b
4         self.c = "b"
5         self.a = 123
```

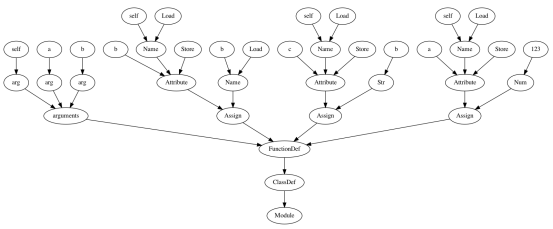

図 5　抽象構文木の例

4.2.1 変数名の問題

構文木を比較する際, 変数名やプログラム中の定数をどう扱うかが問題になる. 初心者がソースコー

ドを書く時に”a”,”b”,”c”などの無意味な変数名を作成する場合が多い, それに比べ, 正解ソースコードは読み手が理解しやすいために適切に意味のある変数名を付けている. 従って, そのまま初心者のソースコードを正解ソースコードとの編集距離を求めると, 例え正解かつ構造の良いプログラムにしても, 想像以上の編集距離が計算されると考えられる. この問題を避けるために, 以下の 2 つ方法を考えた.

- 初心者に問題を与える時, 予め使う変数を目的ごとに決める
- 全部の変数を同じものとして扱い, プログラム中の定数値は元の値に保持する

まず, 一つ目の方法のメリットとしては簡単に実装できる. 使う変数名を目的ごとに指定することにより, 前の問題を解消できる. 例えば,「結果を入れる変数は “res”, ループのカウンタには “i” を使うこと」と問題文に入れることである. しかし, そのように変数を指定するというのはプログラムの構造を指定しているわけで, 初心者の自由な考えを制限してしまうことが問題になる. その上,“ループ” や “カウンタ”などの単語には初心者に理解できない可能性があり, より一層混乱させることになる.

二番目の方法は変数を同じものとして扱う方法である. この方法の実現は最初に抽象構文木を作成するときにすべての変数は同一のノード var とする. 変数名と違って定数の値は重要だからそれぞれの値を別のものとして扱う. 以上の処理をする根拠は変数はただ値を入れる箱で, 変数名が違ってもソースコード全体の結果に影響を与えない, すなわち, 違う変数により構造的な影響を与えないと言うことである. しかし, 値が変化すると最後の結果に影響を及ぼすので, その値が大事な評価基準になる.

本研究の方針としては「なるべく甘く部分点を与える」ことであり, 変数をすべて同一視することにより, ソースコードの比較を大きく甘くすることができ, 初心者の自信を保つことに繋がる. 従って,「全部の変数を同じものとして扱い, 定数値は元の値に保持する」を採用した.

4.2.2 編集距離

編集距離 [7] は元々二つの文字列がどの程度異なっているかを示す距離の一種である. 具体的には,1 文字の挿入, 削除, 置換によって, 文字列 A を文字列 B に変形するのに必要な手順の最小回数として定義される。例えば, 以下の文字列 A:abcde を文字列 B:aaced に変形するには以下の 3 ステップが必要である. 故に, 文字列 A と文字列 B の編集距離は 3 である.

- A[1] を “a” に置換する
- A[3] を “e” に置換する
- A[4] を “d” に置換する

この編集距離を文字列から木構造へ拡張したのが zss アルゴリズム [6] である. 木の編集距離は「1 ノードの挿入, 削除, 置換によって, 木 A を木 B に変形するのに必要な手順の最小回数」と定義した. 例えば, 図 6 は抽象構文木の一部を抽出している. この部分だけを見るとの編集距離は 4 である. すなわち, 以下の 4 ステップで左の木を右の木に変更できる.

- 左のノード “a” を “b” に置換する
- 右のノード “Num” を “Name” に置換する
- 右のノード “1” を “b” に置換する
- 右のノード “Load” を挿入する

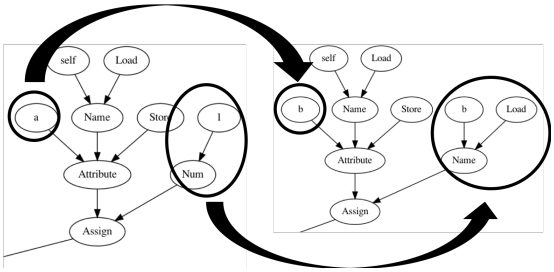

図 6 木の編集距離の例

しかし, そのまま木の編集距離を求めると, 以下の様な見た目がよく似ている, すなわち, 構造がよく似ている 2 つのソースコードに対して, 想像より大きな編集距離が計算される. こうなる原因は「部分木の置換」という操作を基本操作としていないから. 結果, ほぼ同じことをしているソースコード

の編集距離が大きく求められた.

```
class test_class:
    def __init__(self,a,b):
        self.a = 1
        self.b = b
```

```
class test_class:
    def __init__(self,a,b):
        self.b = b
        self.a = 1
```

4.2.3 hash 値によるソート

上で述べた文の順序の入れ替えなどによる編集距離の増大を軽減するために, 編集距離を求める前に同じレベルの子ノードに対してソートを行うことを提案した. 具体的な方法は全てのノードに対して, hash 値を決め, 抽象構文木を hash 値によりソートを行う.

hash 値の決め方としては, 再帰呼び出しを使って, 一番深いノードからルートまで求める. 抽象構文木の一番深いノードは終端記号なので, 終端記号であるノードの hash 値は予め決める. 終端記号は以下の terminal_symbols に定義した.

```
expr_context = ["Load", "Store", "Del
    ", "AugLoad", "AugStore", "Param
    "]
boolop = ["And", "Or"]
operator = ["Add", "Sub", "Mult", "
    MatMult", "Div", "Mod", "Pow", "
    LShift", "RShift", "BitOr", "
    BitXor", "BitAnd", "FloorDiv"]
unaryop = ["Invert", "Not", "UAdd",
    "USub"]
cmpop = ["Eq", "NotEq", "Lt", "LtE",
     "Gt", "GtE", "Is", "IsNot", "In
    ", "NotIn"]
other = ["arg","Num","Str"]
terminal_symbols = [expr_context,
    boolop, operator,unaryop,cmpop,
    other]
```

hash 値の決め方の例を図 7 に示した. この例では終端記号の “Load” と “Store” の hash 値が予め決まっていって, それぞれ 12 と 13 である. そして, 逆戻りに計算し,”Attribute” の hash 値が 25 であることを計算できる. この過程を繰り返し, 全ノードの hash 値を決めることができる.hash 値を計算できれば,hash 値の大きい順にソートを行い, 前節の問題点[*3]を解決できる.

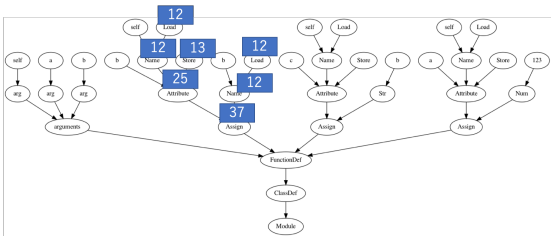

図 7 hash 値の決め方

4.3 解析方法 2:プログラム実行の追跡と解析

プログラムの最終の目的としては正しい結果を得ることなので, ソースコードを静的に解析するだけでの評価は不十分である. この節でソースコードを動的に解析する方法を紹介する. 動的に解析するとはプログラムの実行を追跡し, 正解プログラムと似たような動きであれば, 良いプログラムと判定する.

ソースコードを実行し, テストケースの出力で比較するのは一番普通の動的解析である. 本研究は実行する過程を注目した. 具体的に言うと変数の変化過程を追跡し, その変化過程を値の列として保存する. そして, 初心者ソースコードと正解ソースコードの保存された変数の変化過程の編集距離を求め, その編集距離により動的にソースコードの良さを評価する. 例えば, 以下のプログラムの変数の変化過程は {’res’: ’0 1 3 6 10 15 21 28 36 45’, ’i’: ’0 1 2 3 4 5 6 7 8 9’} となる. この変数の変化過程は文字列として保存したので, 4.3.1 で紹介した二つの文字列の編集距離を求める方法を使い, 編集距離を求める. ただし, 変数の対応付けが難しいので, ここでは総当たりをし, 一番小さな編集距離を採用する. その理由は変数の名前が違っても, 変化過程が大体同じだったら, 機能的に同じ変数だと考えられ

[*3] ほぼ同じことをしているソースコードの編集距離が大きく求められる問題

るからである.

```
1 class test_class:
2     def test_func(self):
3         res = 0
4         for i in range(10):
5             res += i
6         print(res)
```

4.4 本システムと現存システムの比較

本システムと現存のシステムの流れを以下の表 1 にまとめた. 現存のシステムはソースコードを実行し, テストケースの全入力に対する出力結果が一致すれば正解を返す, 一致ではない場合は不正解を返す. これに対し, 本システムの場合, 結果は正しくないが良いプログラムを質の視点からそれなりに評価できる. 結果が正しいけど構造やスタイルがよくないプログラムを発見できる. これにより, 正解の人は過剰に自慢することがなく, 不正解の人は自信を無くすことも無くなることを期待できる。

表 1 本システムと現存のシステムの比較

現存システム	本システム
練習問題を解く	練習問題を解く
実行する	実行する
テストケースの値と一致？	静的+動的で評価
正解/不正解	正解/不正解+xx%

5. 実装

本システムは python3.7 で実装し, 評価対象とするプログラミング言語も python3.7 である. 各部分に使ったモジュールは以下の表 2 にまとめた. システムの全体図を図 8 に示した.

ユーザはシステムを起動し, 表示された画面に対して, コーディングを行う. コーディングが終わり, ソースコードを提出すると「正解 or 不正解+質的点数」が返される. 正解の場合は質的点数を確認し, 高い点数であれば問題がないが, 低い場合は構造的に問題があるので, もう一回ソースコードを見直しする必要がある. 不正解の場合について, もし質的点数が高い場合は構造的に良いので, あと少し頑張れば正解になるかもしれない. この結果により, 学習者の勉強意欲を高めることを目指している.

表 2 各部分で使ったモジュール

使った部分	モジュール名	役割
システム全体	tkinter	UI を作成する
静的解析	ast	抽象構文木を作成する
静的解析	graphviz	グラフを作成する
静的解析	zss	木の編集距離を求める
動的解析	pysnooper	プログラムを動的に観察する
動的解析	Levenshtein	文字列間の編集距離を求める

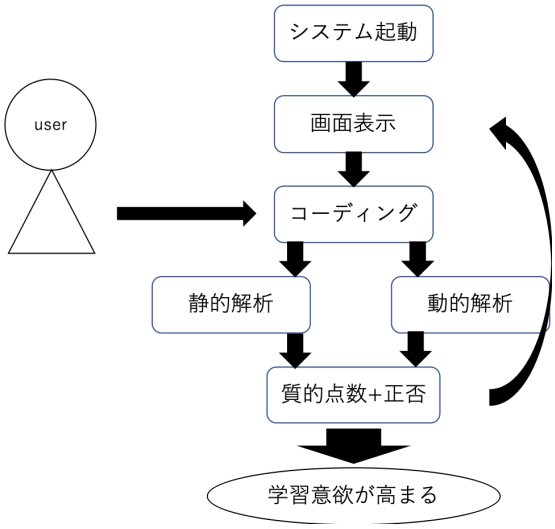

図 8 システム全体

5.1 静的解析の実装

静的解析は ast モジュールを使ってソースコードから抽象構文木を生成する.4.3.2 の説明により, 抽象構文木をソートする必要がある. そのため,hash 値を入れたノードクラスを新しく定義した. ノードの種類を示す self.node と自分の子供はなにかを示す self.children 以外に self.node_hash と self.value を定義した.self.node_hash はソートするために hash 値を保存する変数で,self.value はこのノードが変数か値かを区別するためのものである.

```
1 class New_node:
2     def __init__(self,node,children,
          node_hash,value):
```

```
3         self.node = node
4         self.children = children
5         self.node_hash = node_hash
6         self.value = value
```

ソートした抽象構文木を作成したのち zss モジュールを使い木の編集距離を求めることができる. ただし, 木の構造を zss が処理対象とする標準構造に書き換える必要がある. 最後, 二つのソースコードの名前を与えるだけで静的編集距離を出るように作成した.

5.2 動的解析の実装

動的解析は pysnooper を使い、実行中のソースコードの全ての変数を追跡した. その中の必要な部分を正規表現で取り出し, 最後ディクショナリ型としてまとめた. そして, 総当たりを行い, 二つのソースコードの動的編集距離が求められる.

6. 今後の課題

本研究は完全な正解以外にも「部分点」を与えることでユーザの意欲を引き出すシステムを目指す. 現段階では静的と動的の編集距離を求めた, その編集距離を質的評価にどう繋ぐのかがまだ具体的に定めていない状態である. 現在考えているふたつの方法を以下に示す.

一つ目の方法について, 編集距離はソースコード A からソースコード B に書き換えるために必要な労力なので, その労力を正解ノード数で割れば, かかる労力の率になる. そして, その値を 1 から減算することで質的評価の点数とする. この方法で正しく評価できるのかはたくさんのサンプルコードで実験する必要がある.

二つ目の方法は人手で部分点の判定を行い, 判定したデータと編集距離のデータを一つのまとまったデータと見なす. 例は表 3 に示した. 大量なまとまったデータを機械学習により学習し, 最終的には学習したモデルにより自動的に点数をつける.

いずれの方法についても, 大量なサンプルコードのデータが必要なので, データの取得方法について考える必要がある. そして, 他の質的評価の繋ぎ方も検討していく. さらに, 動的と静的の解析が両方できない場合 (構文的なエラーが発生した場合) について, 他の評価方法が必要とする. 今のシステムではこの状況に対して編集距離を求めることが出来ないので, 質的評価を与えることが困難である。. この状況でも初心者にある程度の点数を与えたいので, ソースコードのフラット化を提案した. フラット化とは, ソースコードのインデントを全部揃えることである. フラット化の例を以下に示す. フラット化した後, 正解プログラムと行ごとの文字列比較を行い, 初心者ソースコードのある行が正解ソースコード内に存在すればある程度の点数を与えることである. この方法により, 実行できないかつ構文的エラーがあるソースコードに対しても評価できるようになる.

表 3 学習対象とするデータの想定例

編集距離	人手でつけた点数	ソースコード長	他のパラメータ
20	70	60	..
32	80	116	..
13	70	50	..

```
1 class test_class:
2     def __init__(self,a,b):
3         self.a = 1
4         self.b = b
```

```
1 class test_class:
2 def __init__(self,a,b):
3 self.b = b
4 self.a = 1
```

参考文献

[1] 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**. 実践的 IT 教育シンポジウム, 2018.

[2] Abbas, N: **Properties of "good" java examples**. Umea's 13th Student Conference in Computer Science, 2009.

[3] Thomas W., Yihuan Dong, Tiffany Barnes: **Generating Data-driven Hints for Open-ended Programming**. Proceedings of the 9th International Conference on Educational Data

Mining, 2016.

[4] R. Flesch: **A New Readability Yardstick**. The Journal of Applied Psychology, 1948.

[5] J. Borstler, M.E. Caspersen, and M. Nordstrom: **Beauty and the Beast—Toward a Measurement Framework for Example Program Quality**. UMINF-07.23(UMEA UNIVERSITY Department of Computing Science), 2007.

[6] KAIZHONG ZHANG, DENNIS SHASHA: **SIMPLE FAST ALGORITHMS FOR THE EDITING DISTANCE BETWEEN TREES AND RELATED PROBLEMS**. Society for Industrial and Applied Mathematics, 1989.

[7] Levenshtein, V. I.: **Binary codes capable of correcting deletions, insertions, and reversals**. Doklady Akademii Nauk SSSR 163 (4): 845-848, 1965.