Vol.2021-SE-207 No.25
2021/3/2

プログラミング入門科目における提出プログラムのセマンティクスを考慮した自動分類手法

西 陽太[1,a)]　石尾 隆[1]　松本 健一[1]

概要：プログラミング演習において，学生から提出されたプログラムを確認し内容に応じたフィードバックを返すことは教育上重要であるが，全ての提出を確認することは講師や TA にとって負担の大きい作業であり，効率化する方法が求められる．本研究では，プログラミング初学者を対象としたプログラミング演習において提出されたプログラムを，その計算手順の同一性によって自動分類する手法を提案する．提出されたプログラムから，類似した記述や間違いを含むものを自動でまとめることにより，グループとなった学生に同じフィードバックメッセージを送ることを可能とし，全てのプログラムを確認するコストの削減を目指す．具体的な方法としては，学生の提出したプログラム群に対して記号実行を適用し，実行パスごとのプログラムの出力値の計算式と出力条件を抽出し，それらの集合の同一性によってプログラムを同値類に分類する．授業で実際に作成されたプログラム群に対して，講師が手動で分類した結果を正解データとして，提案手法の有効性を評価した．

1. はじめに

ソフトウェア開発者の需要増加に伴い，プログラミングを学ぶ学生が増えている [1], [2]．多くの教育機関ではプログラミングを学習する方法として演習形式の授業を行っており，まず講師が学生に対して課題を提示し，学生がプログラムを作成すると，講師はそのプログラム内容に合わせて学生にフィードバックを行うことでコーディングスキルの向上を図る．それぞれの学生に対してメッセージを送ることは教育上重要であるが，受講者数が多い場合は全ての提出の確認に多くの時間と労力が必要であり，講師にとって負担の大きい作業である [3]．

講師による採点の負担を軽減する方法の 1 つが，テストの自動化である．しかし，与えた入力に対して正しい出力が出てくるかどうかを検査するだけでは，特定の入力に対してハードコーディングによりテストケースを通過するようなプログラムを記述する学生を検出することができない [4]．たとえば，C 言語における四則演算の使用方法を学ぶことを目的として，変数に格納された数値の累乗を計算した結果を標準出力に表示するプログラムを作成するという課題を提示したとき，累乗という言葉から Web 上の情報を検索した学生が `pow` 関数を使ったプログラムを提出する場合がある．このようなプログラムの提出に対しては，課題の主旨とは異なることを指摘する必要があるが，テストだけでは検知することができない．プログラム中に特定の言語要素が存在することを構文的に自動確認する方法も提案されている [5] が，使い方の意味的な正しさまでは検査することができない．

学生が作成したプログラムの意味的な内容に応じフィードバックを返す仕組みとして，Marin らは提出されたプログラムに対して，事前に講師が用意した特定の形のコードが含まれているならば対応するフィードバックメッセージを自動的に返す手法を提案している [6]．この手法では，課題ごとに講師が用意したチェックリストをプログラム依存グラフの断片として表現することで，学生ごとの記述の細かい差異を無視して，チェックリストの確認が可能である．しかし，講師は課題ごとに必要となる記述内容や誤りのパターンをあらかじめ列挙する必要があり，多数の小規模な課題を学生に与えるような演習科目で導入することは難しい．

本研究では，学生から提出されたプログラムの集合から計算手順を抽出し，その同一性をプログラム間で比較することで自動的にグループ化する手法を提案する．事前にパターンを用意するかわりに，講師がグループ単位で確認を行い，そのグループに含まれた学生に同一のフィードバックメッセージを送ることで，全てのプログラムの確認に必要なコストを削減する．プログラミング入門科目では，何

[1] 奈良先端科学技術大学院大学
Nara Institute of Science and Technology, Ikoma, Nara 630–0192, Japan
[a)] nishi.yota.nw5@is.naist.jp

らかの入力値や定数に対して簡単な計算を行い，結果を出力するプログラムを作成することが多いと考えられるため，講師が作成したプログラム例から入出力となる変数の個数を取得し，プログラムの実行パスごとに，実行の条件式と，使用される出力値の計算式の組をプログラムの特徴として取得する．条件式と計算式の組の集合が等しいものを同値類としてグループ化することで，各同値類には，同一の入力に対して同一の計算式で得られた値を出力するプログラムだけが含まれるようにする．講師・TA は各同値類から 1 つのプログラムを確認するだけですべてのプログラムの計算手順を確認することができ，プログラムの確認の手間が減少する．

提案手法の有効性を評価するために，C 言語を対象として記号実行エンジン KLEE を使用して提案手法を実装し，著者の所属組織におけるプログラミング演習において学生が作成したプログラムの自動分類を行う．その結果を，講師が各プログラムを目視して分類した，同一フィードバックメッセージを送りたいグループ分けと比較する。

以降，2 章で関連研究について述べ，3 章で提案手法の詳細について述べる．4 章では提案手法の評価実験について述べる．5 章ではまとめと今後の課題について述べる．

2. 関連研究

プログラミング演習科目における効率的なフィードバック手法に関する研究は，Web 上で大学の講義が受講できるサービス，すなわち Massive Open Online Courses (MOOCs) のプログラム入門科目を対象として数多く行われている．Marin らは Java の課題に対してパーソナライズされたフィードバックを提供するために，講師が定義したチェックリストに基づいてプログラムの構成要素の存在を検査し，プログラムがどこまで目標を達成しているかを通知する手法を提案している [6]．この手法では，講師側は課題を解くにあたり必要と思われるプログラムの雛形をプログラム依存グラフ [7] の形で用意する．たとえば，ループ処理の中で特定の条件を満たした場合に変数に値を加算するという処理が必要な課題の場合，`for` 文のブロック内に `if` 文があり，`if` 文のブロック内に加算代入の式がある，というグラフの形を knowledge base と呼ばれるパターンとして作成する．学生の提出プログラムを制御フローおよびデータフローを結合した拡張プログラム依存グラフ (Extended Program Dependence Graph) に変換したとき，knowledge base に対応する部分グラフが存在する場合はループ処理は実装されている，存在しない場合はループ処理の実装をせよ，というようにプログラムの状態を学生に提示する．

Gulwani らは C 言語または Python の入門コースを対象とした提出プログラムの自動分類および修正フレームワーク CLARA を提案している [8]．CLARA は新たに提出された誤りのあるプログラムに対して，過去にコースへ提出された正解プログラムの情報をもとに修正案を自動で作成しユーザにフィードバックする．この手法では，過去に提出されテストによって正解と判定されたプログラム群を，(1) 制御フローが一致しており，(2) 出現する変数に全単射関係があるプログラム同士を同一とみなし，事前にクラスタに分類しておく．そして，新たに提出されたプログラムに誤りがあった場合，そのプログラムを各クラスタに所属するプログラムへと変換する（すなわち誤りを修正する）修正案を作成し，作成された修正案の中から修正コストが最も低いものを選択しユーザにフィードバックする．実験では MOOC のデータセットを用いて修正案を生成できたかを確認し，良質な修正案を 81%生成可能だったことが報告されているが，この手法が利用できるのは多数の正解プログラムが存在するような授業に限られている．また，修正の理由は提示しないため，何が問題だったのかを学生は自分で考える必要がある．

藤原らは，学生がプログラムを作成している最中に行き詰まった箇所（コーディング中に一時的に課題を解くことが困難な状態に陥った部分）を特定し，講師に提示することで，講師からのフィードバックを支援する手法を提案している [3]．具体的な方法としては，まず，学生が作成しているプログラムのスナップショットを 1 分間隔あるいはコンパイルや実行が行われるたびに保存する．次に，スナップショットが保存された時点の推定残作業量を計測する．そして，推定残作業量の遷移から学生がコーディングを進めているにも関わらず，正解の状態に近づいていない箇所を行き詰まり区間とし，行き詰まり区間内および周辺のスナップショットを講師に提示する．講師は学生が行き詰まりやすい箇所を効率的に把握できるが，フィードバックそのものは，学生に対して個別に実行する必要がある．

Cardell-Oliver は，コードの大きさや実行時間などの計測可能な指標と，PMD や Checkstyle などの静的解析ツールを用いて得られる可読性や未初期化変数の参照，使用していない変数の検出などのコード品質の情報を自動的に学生へのフィードバックメッセージへと変換する環境を提案している [9]．本研究は，学生が実装に用いた言語要素などの確認を目的としており，着目する観点が異なっている．

3. 提案手法

提案手法は，プログラム集合 P を入力として，集合内の各要素に対して計算手順の同一性を判定することで，同値類に分割する．各プログラム $p \in P$ は，標準入力から読み込んだ値とソースコード中に定義された定数を用いて値を計算し，標準出力に書き出すものとする．本研究では，出力値の計算手順のみを考慮するものとし，たとえば最終的な出力に影響を与えないような無駄な行や変数の存在については考慮しない．以降，C 言語と KLEE [10] を用いた

実装に基づいて提案手法の説明を行うが，実行パスと計算手順の取得方法については，特に言語に対する依存性はなく，プログラミング言語に対応する記号実行系が存在すれば，その機能を使って実装可能であると考えている．

本手法におけるプログラム p の計算手順とは，p における実行パスを k 個列挙したときの，各実行パス $i(1 \leq i \leq k)$ を通過するための入力値の条件 $cond_i$ と，パスを通過する際に実行される出力命令 (C 言語では `printf` 関数) の出力値の計算式の列 $expr_i$ の組 $(cond_i, expr_i)$ の集合である．ここで，条件および計算式は，プログラムに対する入力値 (C 言語では `scanf` 関数で読み込まれる値) の列 $in_1, in_2, \cdots, in_n$ と定数値を使って表現される式である．例として，以下の課題 (以降，課題 K と呼ぶ) を考える：

- 3 つの数値を入力として受け取り，底面の横幅 `w` と縦幅 `h`，高さ `d` とみなして直方体の底面積・体積を計算し，横幅，縦幅，高さ，底面積，体積を順に標準出力に出力するプログラムを作成せよ．

図 1 が，このプログラムの実装例である．変数 `area` に底面積，変数 `volume` に体積の計算結果を格納し，課題 K の指定順に `printf` でそれぞれの値を表示する．このプログラムにおける計算手順 $C(p)$ は，入力変数の名称を w, h, d とすると，以下のように定義される．

$$C(p) = \{(cond_1, expr_1)\}$$
$$cond_1 = true$$
$$expr_1 = [w, h, d, (w * h), ((w * h) * d)]$$

たとえば 8 行目と 9 行目を入れ替えるなど，計算の順序が違ったとしても同一の出力を行うプログラムであれば計算手順は等しくなる．一方でたとえば `printf` の命令の順序が変わると $expr_i$ の列の要素の順序が入れ替わり，異なる計算手順として認識される．

プログラムが if 文などで複数の実行パスを持つ場合は，それらに対応する $cond_i$ と，その実行経路を通過したときの $expr_i$ を求める．プログラムは入力値に応じて無限に近い実行パスが存在しうるため，探索の上限としてのパラメータ k が必要である．ループや if 文の中を通過しない (条件が false になる) パスを優先して k 個の列挙を行うことで，プログラムの実行完了につながるパスを収集する．

提案手法は，複雑な実行パスでのみ違いが発現するようなプログラムの分類はできないが，プログラムの入力にシンボル値を用いる動的記号実行であるため，テストデータの準備が不要であるという利点がある．また，計算手順がすべて同一であるため，同値類として得られたグループの中から 1 つのプログラムを確認すれば，他のプログラムについても同一の計算を行っていることから，条件漏れなどの誤りの指摘などを行いやすいという性質を持つ．

本研究では，KLEE を用いて，プログラム p から $C(p)$ を求める方法を以下のような手順として実装した．

```
1  #include <stdio.h>
2  int main(void){
3    int w;
4    int h;
5    int d;
6    scanf("%d␣%d␣%d", &w, &h, &d);
7
8    int area = w * h;
9    int volume = w * h * d;
10
11   printf("Width=%d,␣", w);
12   printf("Height=%d,␣", h);
13   printf("Depth=%d\n", d);
14   printf("Area=%d\n", area);
15   printf("Volume=%d\n", volume);
16 }
```

図 1 課題 K の実装例

(1) ソースコードを KLEE で処理可能な形式に変換する
(2) 記号実行により実行パスを列挙する
(3) 計算式を正規化する
以降，各手順について詳細を述べる．

3.1 記号実行のためのソースコード変換

提出プログラムを KLEE で実行し実行パスを解析するために，記号実行が可能となる表現にソースコードを書き換える．具体的には，以下の 3 つの操作を行う．

入力値のシンボル化． 提出されたプログラム内にあるユーザからの入力を任意の変数に格納する関数 (`scanf`) をコメントアウトし，その変数をシンボルとして扱う表現に書き換える．プログラミングの演習課題では，通常，入力値の順序が指定されているため，読み込みに使われる変数の順序を，そのままシンボル値としてプログラム間で対応づける．

シンボル式の出力の追加． プログラム内で使用される標準出力関数 (`printf`) を全てコメントアウトし，`printf` が引数を持つ場合，その値に対応するシンボル式を出力する命令に書き換える．

ヘッダファイルの追加． 書き換えた表現部分の命令を解釈できるように KLEE 用の命令が宣言されているヘッダファイルを include する処理を追加する．

出力結果をあとで人間が解釈する都合上，実装したツールでは，入力値と出力値に，人間が名前を与えられるようにしている．たとえば課題 K の場合，入力変数 w, h, d，出力変数 $w, h, d, area, volume$ という名称を与えておくと，プログラム内での `scanf`, `printf` での出現順に，これらの変数の名称を割り当てて，計算式を出力する．

本研究ではこの変換をパーサジェネレータの ANTLR[*1] と C 言語用の文法ファイル[*2] を用いて，機械的に行うよ

[*1] ANTLR: https://www.antlr.org/
[*2] c/C.g4 file in https://github.com/antlr/grammars-v4/

```c
#include <klee/klee.h>
#include <stdio.h>
int main(void){
  int w;
  int h;
  int d;
  /* scanf("%d %d %d",&w,&h,&d); */
{ klee_make_symbolic(&w, sizeof(w), "w");
  klee_make_symbolic(&h, sizeof(h), "h");
  klee_make_symbolic(&d, sizeof(d), "d");
}

  int area = w * h;
  int volume = w * h * d;

  /* printf("Width=%d, ",w); */
{ klee_print_expr("$=", w);
}
  /* printf("Height=%d, ",h); */
{ klee_print_expr("$=", h);
}
  /* printf("Depth=%d\n",d); */
{ klee_print_expr("$=", d);
}
  /* printf("Area=%d\n",area); */
{ klee_print_expr("$=", area);
}
  /* printf("Volume=%d\n",volume); */
{ klee_print_expr("$=", volume);
}
}
```

図 2 課題 K に対する提出プログラムを手順 2〜4 により変換したソースコード

う実装した．図 1 のプログラムに対し変換を行った結果を図 2 に示す．scanf 関数の引数をシンボルとして定義する命令 (klee_make_symbolic 関数) が挿入され，printf 関数の出力のかわりに，引数のシンボル式を取得する命令 (klee_print_expr 関数) を追記している．

3.2 記号実行による実行パスの列挙

変換が完了したソースコードを KLEE 上で実行するために，Clang[*3]コンパイラを用いて LLVM bitcode にコンパイルする．LLVM bitcode 形式 (.bc ファイル) にコンパイルするには，コンパイル時に Clang のオプションで "-c -emit-llvm" を指定する．ここで，提出プログラムの実行パスと演算過程をプログラムの記述通りに取得するために，"-O0"オプションによりコンパイラによる最適化を行わないようにする．また，klee.h を include 可能にするために，"-I"オプションにより klee.h が用意されているディレクトリのパスを指定する．

次に, コンパイルしたプログラムに対して KLEE を実行する．本研究では, 計算手順の取得のために KLEE の実行オプションで "-search=dfs"，"-write-kqueries"，および，"-write-print-expr" を指定する．"-search=dfs"オプションは，記号実行を行う際に実行パスの探索方法として深さ優先探索を指定するものであり，KLEE では条件分岐に到達すると条件が偽のブランチから先に探索されること

[*3] Clang: https://clang.llvm.org/

```
array x[4] : w32 -> w8 = symbolic
(query [(Eq false (Sle 0 (ReadLSB
    w32 0 x)))] false)
```

図 3 .kquery ファイルの一例

```
$=:(ReadLSB w32 0 w)$
$=:(ReadLSB w32 0 h)$
$=:(ReadLSB w32 0 d)$
$=:(Mul w32 (ReadLSB w32 0 w) (
    ReadLSB w32 0 h))$
$=:(Mul w32 (Mul w32 (ReadLSB w32
    0 w) (ReadLSB w32 0 h)) (
    ReadLSB w32 0 d))$
```

図 4 図 2 のプログラムから得られるシンボル式

から，ループの実行がなるべく起こらないような実行パスの探索が進行する．"-write-kqueries"オプションは，各実行パスに至るために必要な制約条件をそれぞれの実行パスごとにファイルで出力する．"-write-print-expr"オプションは，本研究において KLEE を改造して作成したオプションで，klee_print_expr 関数で得られるシンボル式を各実行パスごとにファイルで出力する．元々の klee_print_expr 関数は標準エラー出力にシンボル式を出力していたが，それを探索中の制約条件と容易に対応づけられるようにしたものである．

KLEE を実行すると，実行パスが列挙され，実行パスの個数だけ，それぞれの制約条件（提案手法における $cond_i$）を保存した.kquery ファイルが生成される．ファイルの中身は KQuery [*4] という制約式とクエリをテキスト表現した言語で記述されている．ファイルの中身の例を図 3 に示す．ファイル 1 行目は x をシンボル変数として定義し，2 行目は KLEE が内部で実行する SMT ソルバに渡される制約条件を KQuery のクエリで表しており，"[ ]" で囲まれた式 "(Eq false (Sle 0 (ReadLSB w32 0 x)))" が制約式である．図 3 の例では制約式が一つのみであるためこれが制約条件となるが，複数の制約式を持つ場合はそれらを ”∧” で繋げたものが制約条件となる．

各 .kquery ファイルに対応して，その実行パスでプログラムが行う出力を klee_print_expr 関数の出力結果として得る．これが提案手法における出力値の計算式，すなわち $expr_i$ となる．なお，klee_print_expr 関数が出力するシンボル式の文法は KQuery に従い，木構造となっている．図 2 のプログラムを実行すると，実行パスに対応する .expr ファイルが 1 つ生成され，図 4 のような内容が得られる．シンボル式は演算順序が存在しており，たとえば，5 行目のシンボル式は，中置記法に変換すると，$((w * h) * d)$ を意味する．

[*4] KQuery language: https://klee.github.io/docs/kquery/

3.3 計算式の正規化

提出されたすべてのプログラムを記号実行し，得られた $cond_i$ と $expr_i$ の組の集合が同一であるようなプログラムを同値類として分類すれば，プログラムの分類は完了する．しかし，実際の数値計算では，学生ごとの命令の書き方の違いによって，同一の計算であっても式の差異が発生しうるため，提案手法では計算式の正規化を行う．たとえば，課題 K における体積の計算方法では，`"volume = w * h * d"` という命令で計算した計算結果のシンボル式は `"((w * h) * d)"` となるが，変数名の順序を変更して `"volume = w * d * h"` という命令で計算した場合に得られるシンボル式は `"((w * d) * h)"` となり，同一とはみなされない．本研究では，交換則や分配則などの単純な四則演算の書き方の違いを吸収するため，得られた $expr_i$ の表現を，記号計算用ライブラリ SymPy[*5] の簡略化関数 `simplify` を適用して，正規化を行う．ただし，実際のシンボル式は KQuery 形式であり SymPy で扱える Python の演算子を使用した中置記法の式に変換する必要があるため，パーサジェネレータ ANTLR を用いて作成した KQuery 用パーサを用いて変換を行った．

3.4 実装上の制限

本提案手法は KLEE の実装に基づいているため，浮動小数点数を扱った課題に対して適用ができない．また，記号実行の特性上，ソートなどの実行パスが指数関数的に増えるような課題に対してはパス爆発が発生してしまう．この問題に関しては，KLEE の実行時間やパス探索数，SMT ソルバの実行時間に上限を設定するか，KLEE 上で実行したいプログラムの内部に，シンボル値の取ることができる値の制約を埋め込むことで回避できると考えられる．

4. 評価実験

提案手法の有用性を確認するために評価実験を行う．プログラミング演習における提出プログラム集合に対して，提案手法は振る舞いに基づくグループ化を行う．この結果が，講師が個別にプログラムを確認して作成した，プログラム改善のための指摘事項に基づくグループ化と等しいかどうかを評価する．

グループ化の質を評価するために，MoJoFM [11] という指標を用いる．これは，あるグループの要素を別のグループへ移動する Move 操作と，2 つのグループを 1 つに結合する Join 操作のみを用いて，グループ化の結果 A を．別の結果 B に変換するための操作の量 $mno(A,B)$ によって，以下のように定義される．

$$MoJoFM(A,B) = 1 - \frac{mno(A,B)}{max(mno(\forall X,B))}$$

ここで，分母は Move と Join の回数が最も多い最悪の状態から B を作成するときの手間を意味しており，MoJoFM の値が 1 に近づくほどグループ化の質が高いことになる．これは，講師がグループ化されたプログラムの中から無関係なプログラムを取り除くなどの，分類結果の確認コストが小さいことを対応する．

提案手法の結果を評価するためのベースラインとして，(1) 提案手法を用いず，講師が個別に確認する（各プログラムをそれぞれ 1 グループとみなす）場合と，(2) ソースコードの字句に基づく類似度によるグループ化を用いる．ソースコードの字句に基づく類似度としては，Ishio ら [12] のトークン単位での N-gram の Jaccard 係数によって定義されたファイル単位の類似度を使用する．プログラム p_1, p_2 の間の類似度 $sim(p_1,p_2)$ が閾値 th 以上であるようなプログラムの組を同じグループに含める，最短距離法に基づくクラスタリングである．

4.1 実験方法

本評価実験で用いるデータセットは，奈良先端科学技術大学院大学のプログラミング入門科目に相当する「プログラミング演習 (2020 年度) [*6]」の受講生の提出プログラム (2020 年度分) を使用した．この演習では講師が事前に用意した様々な課題を学生に提示し，学生は課題内容を読み取り解答となる C 言語のプログラムを作成し提出する．提出されたプログラムは講師か TA が，目視，または，学生側に提示していない隠されたテストケースを手元で実行し確認を行い，解答としての正否を判断する．実験に使用した提出プログラムは，前述した確認方法によって TA に受理されたプログラムである．

プログラミング演習 (2020 年度) では，C 言語を用いる課題として 59 個の課題が用意された．そのうち，課題内容，および，事前に用意された各課題の模範解答を確認し，

(1) 課題を解くにあたり浮動小数点数を必要としない
(2) 課題を解くにあたり `scanf` の返り値を利用する必要がない
(3) 文字列のみを表示するタイプの課題でない (出力のうち最低一つでも数値を表示すれば対象とする)

以上の 3 つの条件を満たす 29 個の課題を実験対象として抽出した．このような条件を設定した理由として，KLEE の技術的な制限がある．1 つ目の条件は，浮動小数点数をサポートしておらず，動作を保証できないからである [10]．2 つ目の条件は，`scanf` 関数の置換となる `klee_make_symbolic` 関数は返り値を持たないため，本来のプログラムの意味を保ったままの変換ができないからである．3 つ目の条件は，出力値の計算式取得のために使用する `klee_print_expr` 関数の引数に文字列を指定してもアドレスしか取得できず，計算式として比較できないためである．

[*5] SymPy: `https://www.sympy.org/en/index.html`

[*6] `https://syllabus.naist.jp/subjects/preview_detail/427`

表 1 データセットの課題が使用するプログラムの構成要素

課題名	入力の読み取り	加減算	乗算	除算	剰余	if/switch	for/while	配列	関数	ポインタ
1-4	x	x	o	x	x	x	x	x	x	x
1-5	x	x	o	x	x	x	x	x	x	x
1-6	x	x	o	x	x	x	x	x	x	x
1-7	o	x	o	x	x	x	x	x	x	x
1-8	o	o	o	x	x	x	x	x	x	x
1-9	x	x	o	x	x	x	x	x	x	x
1-10	x	x	o	x	x	x	x	x	x	x
2-1	o	o	o	o	o	x	x	x	x	x
2-3	o	x	x	o	o	x	x	x	x	x
2-4	o	o	o	x	x	x	x	x	x	x
2-5	o	o	o	o	o	x	x	x	x	x
2-6	o	o	o	o	o	x	x	x	x	x
2-7	o	o	x	o	o	x	x	x	x	x
2-8	o	o	o	o	x	x	x	x	x	x
2-9	o	x	x	o	o	x	x	x	x	x
2-10	o	o	o	o	x	x	x	x	x	x
3-3	o	x	x	o	o	o	x	x	x	x
3-4	o	o	x	o	x	o	x	x	x	x
3-5	o	o	x	x	x	x	o	x	x	x
3-7	o	o	x	x	x	o	o	x	x	x
3-10	o	o	x	x	x	o	o	x	x	x
3-12	o	o	o	o	o	o	o	x	x	x
5-1	o	o	x	x	x	x	o	o	x	x
6-1	x	x	x	x	x	x	x	x	x	o
6-2	o	x	x	x	x	x	x	x	o	o
6-3	o	x	x	x	x	o	x	x	o	o

表 2 使用した環境

OS / ツール	バージョン
macOS Catalina	10.15.7
Docker Desktop	3.1.0
KLEE	2.2-pre
LLVM (+ Clang)	9.0.1
Java	13.0.1
Antlr v4	4.8
Python	3.7.5
SymPy	1.7

提案手法の実行では，記号実行によって探索するパスの上限数を $k = 20$ とした．これは，データセットの課題に使用されたテストケースの内容から，高々 20 で十分な実行パス数が揃うためである．$k = 20$ のパスを探索する前に実行時間が 30 秒に到達したプログラムが存在した 3 つの課題については，分類対象から除外した．これらの課題はいずれも `for/while` 文の条件式にプログラムの入力値を使用しており，KLEE によるループの終了条件の探索が完了しなかった．

本実験の対象となる課題 26 個について，模範解答内で使用されるプログラムの構成要素を表 1 に示す．表内の `"o"` が使用，`"x"` が不使用を表す．課題は 1-x から 6-x まで番号が振られており，課題 1-x は変数への代入の練習，2-x は入力の読み取りと四則演算，3-x は条件分岐，5-x は配列，6-x はポインタの演習に対応する．

本研究で使用した環境を表 2 に示す．KLEE および LLVM は，ソースコードホスティングサービスの Github 上で公開されているリポジトリ[*7] の Commit id 53ace1a119c5ede520d2e597f9bcdfeece31229a から clone したソースコードに対して新たにオプション `"-write-print-expr"` の機能を追加したものとなっている．また，*mno* と MoJoFM の算出にはソフトウェアのアーキテクチャ分類ライブラリ SADE [13] が公開しているソースコードの mojo パッケージ内の実装[*8]を用いる．

[*7] `https://github.com/klee/klee`

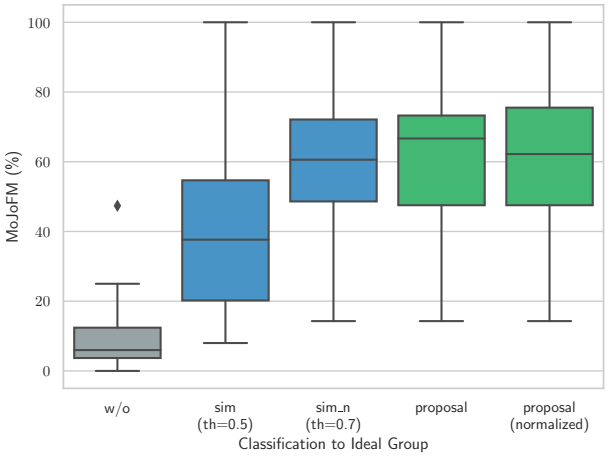

図 5 各分類の MoJoFM の分布

4.2 結果

図 5 に，提案手法およびベースラインの MoJoFM の分布を示す．一番左の “`w/o`” は提案手法を使用せず，各ファイルを個別にグループとみなした場合の値である．左から 2 番目の “`sim (th=0.5)`” は，ソースファイルの類似度を用いた手法で，識別子をそのままトークンとして扱ったものの中で最も結果が良かった場合の結果（$th = 0.5$）である．続く “`sim_n (th=0.7)`” は，識別子を匿名のトークンに変換した（識別子の違いを無視した）中で最も結果が良かった場合の結果（$th = 0.7$）である．そして右側の “`proposal`” は提案手法を式の正規化なしで用いた場合，“`proposal (normalized)`” は式の正規化ありで用いた場合の結果である．MoJoFM の値は，提案手法を式の正規化なしで用いた場合が中央値，平均値とも最も高かった．Wilcoxon の符号順位検定で，“`sim (th=0.5)`” と比べると有意に差があることを確認した ($p < 0.001$)．一方で，“`sim_n (th=0.7)`” と比べると，有意差は確認できなかった ($p = 0.191$)．

課題ごとの提案手法の性能の詳細を表 3 に示す．分析数/提出数の列は，提出数のうちソースコードの変換とコンパイルが可能だったファイルの数を示している．グループ数，MoJoFM における Sim 1 は “`sim (th=0.5)`” の結果に，Sim 2 は “`sim_n (th=0.7)`” の結果に，それぞれ対応する．また，MoJoFM における「個別」が，各ファイルを個別にグループとみなした場合の値である．

Step 1 による KLEE で記号実行を行うためのソースコード変換の結果として，平均提出プログラム数が約 28 個に

[*8] `https://github.com/papachristoumarios/sade/tree/master/sade/mojo`

表 3 分類ごとのグループ数および MoJoFM

課題名	分析数/提出数	LOC	グループ数					MoJoFM				
		中央値	正解	Sim 1	Sim 2	提案手法		個別	Sim 1	Sim 2	提案手法	
						正規化なし	あり				正規化なし	あり
1-4	30/31	9	4	2	1	2	1	3.7	77.78	81.48	85.19	81.48
1-5	30/31	12	3	3	2	1	1	0	44.44	48.15	44.44	44.44
1-6	31/31	12	3	3	2	1	1	0	39.29	50	46.43	46.43
1-7	31/31	15	5	3	1	1	1	3.7	44.44	44.44	44.44	44.44
1-8	31/31	14	5	3	2	10	2	3.7	55.56	59.26	74.07	70.37
1-9	31/31	10	5	26	9	2	1	0	15.38	73.08	65.38	61.54
1-10	31/31	10	6	26	9	3	1	3.85	23.08	69.23	61.54	53.85
2-1	29/29	15	6	10	4	5	5	8	44	88	88	88
2-3	29/29	11	5	18	7	9	6	4	36	48	60	56
2-4	29/29	10	5	11	3	4	2	7.69	61.54	76.92	84.62	80.77
2-5	29/29	19	4	27	11	11	7	7.41	14.81	66.67	66.67	74.07
2-6	29/29	15	4	26	11	11	4	3.85	15.38	57.69	69.23	65.38
2-7	29/29	18	6	29	14	8	4	8	8	68	48	48
2-8	29/29	9	6	26	9	6	4	4.17	12.5	45.83	70.83	62.5
2-9	29/29	9	5	23	5	6	6	0	20.83	79.17	83.33	83.33
2-10	24/24	12	8	24	21	22	19	20	20	20	20	30
3-3	23/29	41	13	12	16	23	23	47.37	31.58	57.89	47.37	47.37
3-4	28/29	20	10	27	24	24	24	25	29.17	37.5	33.33	33.33
3-5	28/29	13	5	13	10	11	9	8	52	56	68	76
3-7	27/29	12	9	11	11	7	6	18.18	40.91	50	59.09	50
3-10	26/28	15	8	11	10	10	9	14.29	28.57	61.9	66.67	61.9
3-12	18/18	31	6	17	17	17	17	14.29	14.29	14.29	14.29	14.29
5-1	26/26	9	5	3	3	3	3	8.7	86.96	86.96	86.96	86.96
6-1	26/26	8	1	1	1	1	1	0	100	100	100	100
6-2	26/26	14	5	3	2	2	2	4.55	68.18	68.18	68.18	68.18
6-3	26/26	16	7	5	3	4	4	13.64	59.09	63.64	68.18	68.18
平均値	27.9/28.4	14.58	5.73	13.96	8	7.85	6.27	8.93	40.15	60.47	62.47	61.42
中央値		12.5	5	11.5	8	6	4	5.98	37.645	60.58	66.67	62.2

対し，変換が可能だったプログラムは約 27 個であった．コンパイルに失敗した変換後の提出プログラムを含む課題が 26 個中 7 個あり，中でも課題 3-3 は提出プログラムのうち 6 個が変換に失敗していた．変換に失敗した提出プログラムを実際に確認したところ，`if-else` 文のブロック（“`{}`”）を省略した書き方の then 節で引数を持たない `printf` を呼び出すなどしていたため，構文変換の結果がコンパイルエラーを起こしていた．また，課題 1-4, 1-5 では `printf("Width=%d\, ", w)` の文字列リテラル内で ‘`\,`’ のように誤った文字をエスケープしているなど，演習環境のコンパイラは許容するが，提案手法の実装（ANTLR で作成した C 言語のパーサ）では処理できないプログラムがあった．これらは，提案手法の実装の改善や，構文エラーの検査ツールによって対応可能であると考えている．

グループ数について見ると，講師による理想的なグループ数の平均は 5.73 であり，提案手法による分類を行った場合のグループ数の平均は 7.85 であった．式の正規化を行った場合の提案手法による分類のグループ数の平均は 6.27 に減少しており，結果 1 と比べたとき平均約 1.4 グループをまとめることが可能となったが，提出プログラムのまとめすぎによって MoJoFM の値を悪化させる結果となっていた．また，Sim 1 と Sim 2 を比べると，Sim 2 のほうがグループ数が大きく減少しており，学生ごとの変数名などの付け方の違いが分類に大きく影響したことが分かる．

Sim 2 と提案手法を比べた場合，課題 1-8 では提案手法が大幅に優れており，課題 1-9 では逆転するといったように，課題ごとに性能の違いが見られた．提案手法は授業で実施されたテストケースでは検知できなかったような条件節の小さな誤りなどを別グループへと分類できたほか，実行パスが 1 つしかあり得ないはずのプログラムに if 文を使っているなど，特殊な書き方をしたプログラムを見つけ，分類することができた．一方，提案手法は，出力に影響しない余分な計算をしているなどといったプログラムについては検知できないことから，字句的な類似度に劣っている部分もあった．2 つの分類手法をうまく組み合わせることができれば，性能をさらに改善できる可能性がある．

課題 1-5, 1-6 は演習科目序盤の単純なプログラム課題にも関わらず，どの手法でも MoJoFM の値が低くなっている．これらの課題では，ユーザの入力を用いず，定数値を使って出力を計算する課題であったため，KLEE がシンボル値を使わず，定数値を直接計算してしまっていた [10]．結果として，提案手法では，すべてのプログラムが正しい定

数を出力するという 1 つのグループにまとめられてしまっていた．ソースコードの類似度を用いても，書き方に大きな差異がなく，分類に失敗したと考えられる．また，課題 1-7 はユーザから入力される縦，横，高さの 3 つの値を用いて直方体の底面積・体積を出力する課題であったが，やはり MoJoFM の値が低い．この理由として，講師は，分類の基準に適切な変数の利用の有無や，途中の計算結果（底面積）の再利用で計算の効率化が図れているかに注目していたが，本提案手法で得られる計算式は，出力値を入力値のみで表現しようとするため，プログラム中で同じ計算式を繰り返し記述しても，1 つにまとめて記述しても，区別することができなかった．このような課題には，たとえば gdb などのデバッガを利用して，実行された命令の個数を求めるなど，異なるアプローチが必要となる可能性がある．

MoJoFM の値が最も低かった課題 3-12 は，演算過程の最適化も効果がなく，分類をほとんど行えなかった．課題内容は，入力として与えられた数値を，同様に与えられた桁位置で四捨五入した値を出力するというものであったが，答えを求める方法が複数あり，それにより条件の書き方が多様になってしまうため，同じ戦略で実装されたプログラムであっても，実行パスが同一とは判定されなかった．この問題を解決するためには，制御構造の正規化を行い，SMT ソルバを用いて条件式の等価性を判定する方法などが必要であると考えられる．

5. おわりに

本論文では，プログラミング入門演習におけるフィードバック支援のためのプログラム自動分類手法を提案した．実験の結果，プログラムの振る舞いに基づく分類が，ソースコードの字句的な類似度に基づく分類よりも中央値，平均値で優れた結果を出力することを確認した．課題によって性能にばらつきがあったため，有意な差は見られなかったが，プログラムの計算方法に基づく分類であるため，グループから代表だけを検査すればその動作を把握することができ，講師の学生へのフィードバックの効率化につながると考えられる．

今後の課題としては，制御構造の正規化を行い，複数考えられる実行パスの表現をまとめることで同一グループの判断基準を広げる，実行トレースなどを用いて計算に実行した命令数などの特徴を集めることが挙げられる．また，提案手法と字句的なソースコードの類似度を組み合わせることで，これらの手法個別ではできない分類を可能とすることが考えられる．

謝辞

本研究は JSPS KAKENHI Nos. JP18H03221, JP18KT0013, JP20H05706 の助成を受けたものです．

参考文献

[1] 独立行政法人情報処理推進機構. IT 人材白書 2016, 2016.
[2] Tracy Camp, Stuart Zweben, Duncan Buell, and Jane Stout. Booming enrollments: Survey data. In *Proceedings of the 47th ACM Technical Symposium on Computing Science Education*, pp. 398–399, February 2016.
[3] 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_3–1_13, 2018.
[4] Daniel Bruzual, Maria L. Montoya Freire, and Mario Di Francesco. Automated assessment of android exercises with cloud-native technologies. In *Proceedings of the 2020 ACM Conference on Innovation and Technology in Computer Science Education*, ITiCSE '20, pp. 40–46, June 2020.
[5] Colin A. Higgins, Geoffrey Gray, Pavlos Symeonidis, and Athanasios Tsintsifas. Automated assessment and experiences of teaching programming. *Journal on Educational Resources in Computing*, Vol. 5, No. 3, pp. 5–es, 2005.
[6] Victor J. Marin, Tobin Pereira, Srinivas Sridharan, and Carlos R. Rivero. Automated personalized feedback in introductory java programming moocs. In *2017 IEEE 33rd International Conference on Data Engineering (ICDE)*, pp. 1259–1270, April 2017.
[7] Susan Horwitz and Thomas Reps. The use of program dependence graphs in software engineering. In *Proceedings of the 14th International Conference on Software Engineering*, pp. 392–411, 1992.
[8] Sumit Gulwani, Ivan Radiček, and Florian Zuleger. Automated clustering and program repair for introductory programming assignments. *ACM SIGPLAN Notices*, Vol. 53, No. 4, pp. 465–480, June 2018.
[9] Rachel Cardell-Oliver. How can software metrics help novice programmers? In *Proceedings of the Thirteenth Australasian Computing Education Conference - Volume 114*, pp. 55–62, AUS, 2011.
[10] Cristian Cadar, Daniel Dunbar, and Dawson Engler. Klee: unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, pp. 209–224, December 2008.
[11] Zhihua Wen and V. Tzerpos. An effectiveness measure for software clustering algorithms. In *Proceedings. 12th IEEE International Workshop on Program Comprehension, 2004.*, pp. 194–203, 2004.
[12] Takashi Ishio, Yusuke Sakaguchi, Kaoru Ito, and Katsuro Inoue. Source File Set Search for Clone-and-Own Reuse Analysis. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, pp. 257–268, May 2017.
[13] Marios Papachristou. Software clusterings with vector semantics and the call graph. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 1184–1186, 2019.