遺伝的アルゴリズムを用いた自動プログラム修正手法を応用したプログラミングコンテストの回答の自動生成に向けて

富田 裕也[1,a)] 松本 淳之介[1,b)] 柗本 真佑[1,c)] 肥後 芳樹[1,d)] 楠本 真二[1,e)]
倉林 利行[2,f)] 切貫 弘之[2,g)] 丹野 治門[2,h)]

概要：本稿では，遺伝的アルゴリズムを用いた自動プログラム修正手法を応用したプログラミングコンテストの回答の自動生成に向けた取り組みについて報告する．本研究では，プログラミングコンテストの問題文，テストケースおよび過去のプログラミングコンテストの回答を格納したデータベースを用いて，プログラミングコンテストの回答を生成する．まず，問題文から必要とする制御構文を推定する．次に，必要とする制御構文のみを含む回答の雛形を作成し，作成した雛形に対して過去に開催されたプログラミングコンテストの回答に含まれるプログラム文を再利用して回答を生成する．過去のプログラミングコンテストの回答に含まれるプログラム文の再利用によって回答の生成が行えるかを検証するために，必要とする制御構文の推定に成功したと仮定した上で実験を行った．AtCoder というプログラムコンテストサイトで開催されたプログラミングコンテストの問題のうち条件分岐や繰り返し処理を使わずに解ける平易な問題を実験の対象にした．実験の結果，全ての問題に対して与えられたテストケースに成功する回答の生成に成功した．

1. はじめに

自動プログラム生成とは，プログラムを自動で生成する技術である．つまり，開発者はプログラムを書くのではなく，作成したいプログラムの仕様を与えると与えられた仕様を満たすプログラムが生成される．

自動プログラム生成の手法には，Microsoft 社が開発した DeepCoder[3] がある．DeepCoder は与えられた入出力仕様からその入出力仕様を満たす 34 種類ある命令の組み合わせを深さ優先探索を行い探索する．DeepCoder は深層学習を用いて制御構文，演算子や定数などの出現確率を予測したモデルを利用して，効率的に深さ優先探索を行う．

Becker らが提案した AI Programmer[4] は，遺伝的アルゴリズムを用いてプログラムを生成する手法である．AI Programmer は，与えられた入出力仕様からポインタの操作やメモリの内容の書き換えなどのプリミティブな操作を繰り返し行いその入出力仕様を満たすプログラムを生成する．

入出力仕様ではなく自然言語からプログラムを生成する手法もある．それらの手法の 1 つに Gvero らが提案した AnyCode[7] がある．AnyCode は関数の機能を自然言語で与えるとその機能を持つ関数の候補を出力する．

本稿では，自動プログラム生成の実現に向けた第一歩として，プログラミングコンテストの問題の回答を過去に開催されたプログラミングコンテストの問題に対する回答を再利用することで自動生成する取り組みについて紹介する．生成を試みる前に，過去に開催されたプログラミングコンテストの問題に対する回答中のプログラム文の再利用によって理論上回答を生成できるかを確認するために調査を行った．調査対象は，AtCoder というプログラミングコンテストサイトで開催されているプログラミングコンテストの問題に対する回答である．調査の結果，80%の回答は過去の問題の回答に含まれるプログラム文の再利用によって生成できることがわかった．プログラミングコンテストの

1 大阪大学大学院情報科学研究科
Graduate School of Information Science and Technology, Osaka University
2 日本電信電話株式会社
Nippon and Telephone Corporation
a) y-tomida@osaka-u.ac.jp
b) j-matumt@ist.osaka-u.ac.jp
c) shinsuke@ist.osaka-u.ac.jp
d) higo@ist.osaka-u.ac.jp
e) kusumoto@ist.osaka-u.ac.jp
f) kurabayashi.toshiyuki@lab.ntt.co.jp
g) kirinuki.hiroyuki@lab.ntt.co.jp
h) tanno.haruto@lab.ntt.co.jp

ⓒ 2020 Information Processing Society of Japan

問題にはさまざまな種類があるため，プログラム文の再利用だけでは現実的な時間内での回答の生成は難しい．プログラミングコンテストの問題を分類するために回答中に存在する制御構文の構造に着目し，140 問の問題を対象に調査を行った．調査の結果，90%の回答が上位 5 つの構造であった．この調査結果を踏まえ，過去に開催されたプログラミングコンテストの問題に対する回答の再利用による回答の生成を試みた．本手法の入力は，プログラミングコンテストの問題文・テストケース，過去の回答を集めたデータベース（以降，回答データベースという）である．出力として与えられたプログラミングコンテストの問題文に対する回答が得られる．本手法の流れは，最初に入力として与えられたプログラミングコンテストの問題文から回答に必要である制御構文を推定し，回答の雛形を作成する．次に，作成した雛形に対して回答データベースに保存されているプログラム文を再利用し与えられた全てのテストケースに成功する回答を生成する．

現段階でプログラムを生成できるか検証するために，AtCoder で開催された問題のうち分岐や繰り返しを用いずに解ける 4 つの問題を対象として実験を行った．実験の結果，対象とした全ての問題に対して与えられたテストケースに成功する回答の生成に成功した．さらに，3 つの問題に対してオーバーフィットの無い回答が得られた．

2. 準備

2.1 自動プログラム修正

ソフトウェア開発においてデバッグは多大な労力を必要とする作業であり，開発工数の半数以上を占めるといわれている [5]．そのため，デバッグの支援はソフトウェア開発の効率化やソフトウェアの信頼性および安全性の向上に有益である．デバッグの労力を削減するために，デバッグの支援に関する研究が数多く行われている [6], [9], [12], [13]．デバッグの支援に関する研究の 1 つに自動プログラム修正がある．

自動プログラム修正は，欠陥を含むプログラムおよびそのプログラムに対するテストケースを入力として，全てのテストケースに成功するプログラムを出力する手法である．自動プログラム修正の手法の 1 つに GenProg[10] がある．GenProg は遺伝的アルゴリズム [11] を用いた自動プログラム修正手法である．遺伝的アルゴリズムとは，解の候補を生物の個体に見立て，遺伝的操作を繰り返し行い解を探索するアルゴリズムである．

GenProg の流れを図 1 に示す．まず，GenProg は欠陥限局 [2] を行う．欠陥限局とはプログラム中の欠陥の位置を推定する手法である．欠陥限局後に，欠陥を含むプログラムに対して推定した欠陥箇所に変更を加え，プログラムを複数生成する．次に，生成したプログラムの評価を行う．

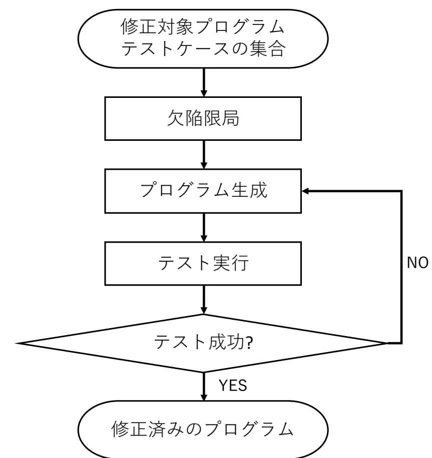


図 1 GenProg の流れ

プログラムの評価には適応度を用いる．適応度はプログラムの良さを表しており，適応度にはテストの通過率が用いられる．GenProg は全てのテストケースに成功するプログラムが得られるまでプログラムの生成および評価を繰り返す．プログラム生成処理で行われる操作を以下に示す．

選択 生成されたプログラムに対してテストを実行する．それらから適応度が高いプログラムを一定数取り出す．取り出されたプログラムは変異や交叉の対象となる．

変異 選択によって取り出されたプログラムの欠陥箇所に変更を加え，新しいプログラムを生成する．加える変更はプログラム文の挿入・削除・置換のいずれかである．挿入や置換で用いるプログラム文は入力プログラムに含まれるプログラム文からランダムに選ばれる．変異の際に，変更を加えた箇所と変更内容を記録する．

交叉 親となる 2 つのプログラムに記録されている変更箇所と変更内容をいくつか選び，新たなプログラムを生成する．生成されたプログラムは親に記録されている変更箇所と変更内容の一部を持っている．

2.2 AtCoder

AtCoder[*1]は，AtCoder 社が運営しているプログラミングコンテストサイトである．AtCoder では，定期的にいくつかのコンテストが開催されている．それらで最も難易度が低いコンテストは AtCoder Beginner Contest（以降，ABC という）である．ABC はプログラミングコンテスト初心者向けのコンテストであり，最も難易度が低い A 問題から最も難易度が高い F 問題の 6 つの問題で構成されている．各問題は問題を説明する問題文，入力の制約，入力・出力の形式およびテストケースに当たる入出力例から構成されている．図 2(a)，(b) に A 問題の例および回答例を示す．

---
*1 https://atcoder.jp
*2 https://atcoder.jp/contests/abc130/tasks/abc130_a

問題文
X，A は0以上9以下の整数である．
X が A 未満の時 0，A 以上の時10を出力せよ．

制約
- 入力は全て整数
- $0 \leq A, X \leq 9$

入力
入力は以下の形式で標準入力から与えられる．

```
X    A
```

出力
X が A 未満の時0，A 以上の時10を標準出力に出力せよ．

入出力例

入力	出力
$X=3$，$A=10$	0
$X=7$，$A=5$	10
$X=6$，$A=6$	10

(a) 問題

```java
import java.util.Scanner;

public class Main {

  public static void main(String[] args) {
    // 標準入力から読み込む
    Scanner scanner = new Scanner(System.in);
    int X = scanner.nextInt();
    int A = scanner.nextInt();

    if (X < A) {
      System.out.println(0);
    } else {
      System.out.println(10);
    }
  }
}
```

(b) 回答例

図 2　A 問題の例（Rounding）[*2]

```java
public int solveA (int r) {
  [単文]
  return [式];
}
```

単文
式

(a) flat テンプレート

```java
public String solveA (int N, int M) {
  if ([式]) {
    [単文]
    return [式];
  } else {
    [単文]
    return [式];
  }
}
```

単文
式

(b) if{}テンプレート

図 3　テンプレート

2.3　テンプレート

テンプレートとは，回答に出現する制御構文のうち if 文，for 文，switch 文，while 文および do-while 文の構造である．テンプレートの分類は排他的である．2 つのテンプレートの例を図 3 に示す．図 3(a) は分岐や繰り返しがない flat テンプレートである．図 3(b) は分岐が 1 つだけある if{}テンプレートである．水色の部分に単文，緑色の部分に式が入る．

3.　調査

3.1　再利用率の調査

3.1.1　調査目的

本調査の目的は，過去に ABC で出題された A 問題に対する回答中のプログラム文の再利用によって ABC の A 問題の回答を生成できるかの確認である．

3.1.2　調査方法

ABC の A 問題の回答に含まれる過去の ABC の A 問題に対する回答のプログラム文の割合を調査した．調査対象は，ABC1～140 の A 問題に対する Java で記述された 35,275 個の回答である．変数の正規化をした回答としていない 2 種類の回答に対して本調査を行った．変数の正規化の流れを図 4 に示す．まず，Java のソースコード解析ツールである Java Development Tools[*3]（以降，JDT という）を用いて回答からプログラム文を抽出する．次に，抽出したプログラム文に対して変数を出現順に正規化する．

3.1.3　調査結果

再利用率を箱ひげ図にプロットした結果を図 5 に示す．図 5(a) は，対象にした全ての回答の再利用率をプロットした箱ひげ図である．図 5(b) は，対象にした各 A 問題に対して再利用率の最大値をプロットした箱ひげ図である．いずれの図も橙色が変数の正規化をしていない回答，青色が変数の正規化をした回答である．図 5(a) より調査対象の回答の半分に含まれるプログラム文の 80%は過去に出題された A 問題の回答に含まれるプログラム文であった．また，図 5(b) より調査対象にした 140 問の A 問題の半分に当たる 70 問の回答中のプログラム文は，変数を正規化すれば過去に開催された ABC の A 問題に対する回答中のプログラム文と一致した．

[*3] https://www.eclipse.org/jdt

ⓒ 2020 Information Processing Society of Japan

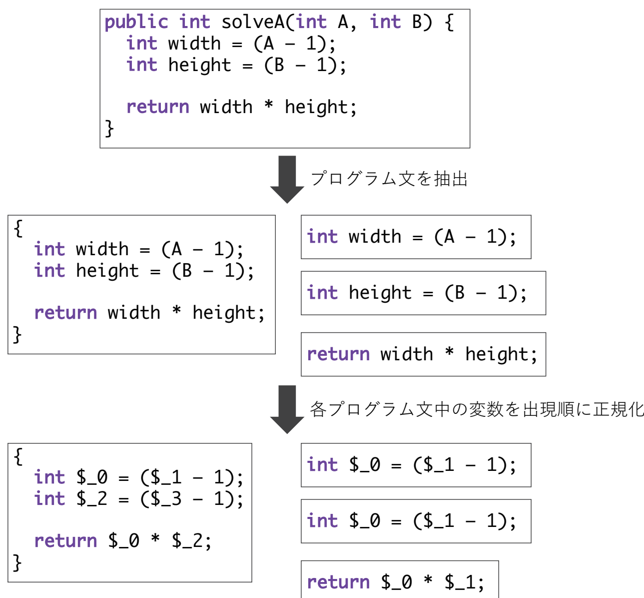
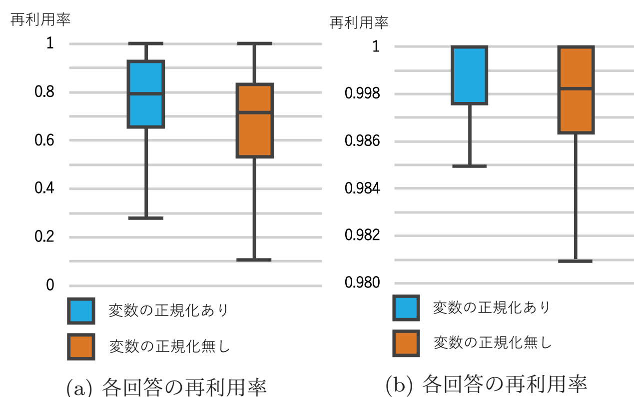


図 4　変数の正規化の流れ



(a) 各回答の再利用率　(b) 各回答の再利用率

図 5　再利用率の調査結果

3.2　テンプレートの調査

3.2.1　調査目的

本調査の目的は ABC の A 問題の分類である．

3.2.2　調査方法

3.1 節の調査で対象にした各 A 問題（計 140 問）に対して再利用率が最大であった回答を対象に調査を行った．本調査時に以下の処理は著者が手動で削除あるいは変形した．

独自実装の入出力処理　問題を解くための処理とは関係ないため，全て削除した．

三項演算子　全て if 文に変形した．

3.2.3　調査結果

テンプレートの調査結果を表 1 に示す．A 問題の回答は 14 種類のテンプレートに分類でき，90%の回答は 5 種類のテンプレートのいずれかであった．さらに，43%の回答は繰り返しや分岐がない flat テンプレートに分類できた．

4.　現在の取り組み

4.1　概要

本節では，現在取り組んでいるプログラミングコンテストの回答を GenProg を応用し生成する手法について説明する．図 6 は手法の流れである．入力は以下の 3 つである．

- 回答を生成したいプログラミングコンテストの問題文
- テストケース
- 回答データベース

出力として与えられたテストケースに成功する回答が得られる．

本手法は以下のステップで構成される．

Step 0　前準備

Step 1　テンプレートの推定

Step 2　拡張した GenProg への適用

Step 0 では，前準備として JDT を用いて過去に開催されたプログラミングコンテストの問題に対する Java で記述された回答からプログラム文を抽出し，回答データベースに保存する．この Step は一度だけ実行すればよい．

Step 1 では，プログラミングコンテストの問題文からテンプレートを推定し，回答の雛形を作成する．

Step 2 では，Step 1 で作成した雛形と入出力例をテストケースとして拡張した GenProg に与え，プログラムの生成を行う．GenProg は雛形ではなく回答データベースから再利用するプログラム文を検索する．

4.2　回答データベース

回答データベースのスキーマを以下に示す．

- コンテスト名
- プログラム文
- プログラム文の種類
- プログラム文に含まれる変数の情報（型，final 修飾子の有無）

回答データベースのレコード数を削減するために，プログラム文に含まれる変数を正規化し，回答データベースに保存する．変数の正規化は，3.1 節と同じ方法を用いて行

表 1　テンプレートの調査結果

テンプレート	数	割合	割合の累積和
flat	60	0.43	0.43
if{}	47	0.34	0.76
if{if{}}	8	0.06	0.82
for{}	8	0.06	0.88
if{}if{}	3	0.02	0.9
for{if{}}	3	0.02	0.92
if{}if{}if{}	3	0.02	0.94
if{}if{}if{}if{}	2	0.01	0.96
for{if{}if{}}if{}	1	0.01	0.96
while{}	1	0.01	0.97
for{if{}}if{}	1	0.01	0.98
for{}if{}	1	0.01	0.99
if{if{if{}}}	1	0.01	0.99
if{}if{}if{if{}}	1	0.01	1.00

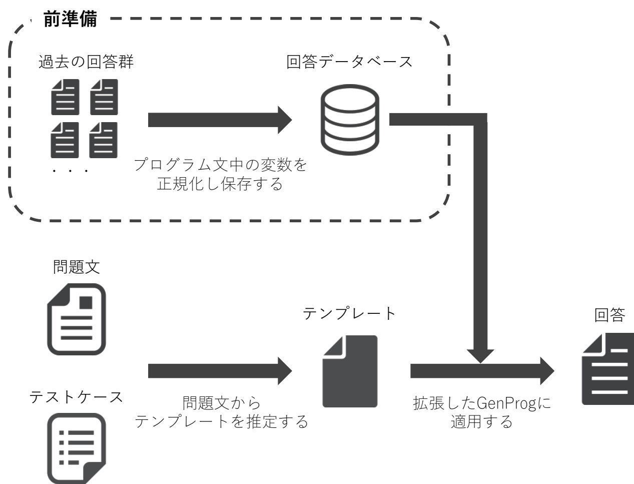


図 6　手法の流れ

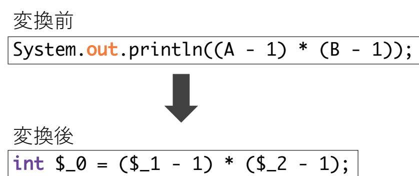


図 7　出力処理の変換の例

う．回答データベースには全く同じプログラム文は存在しない．さらに，再利用しても回答の生成に影響しないと考えられるプログラム文は回答データベースに保存しない．例えば，int j; のような初期化をしていない変数宣言文は，その変数を利用した式から回答を得られるとは考えにくい．また，標準出力への出力処理は図 7 のように出力を行うメソッドへの引数を抽出し変数を正規化した上で変数宣言文に変換する．

4.3　GenProg の拡張

GenProg の Java 実装である kGenProg[8] をプログラム文の挿入時に変数名の書き換えを行うように拡張した．挿入するプログラム文内で変数が宣言済みか否かによって変数名の書き換え方法が異なる．挿入するプログラム文内

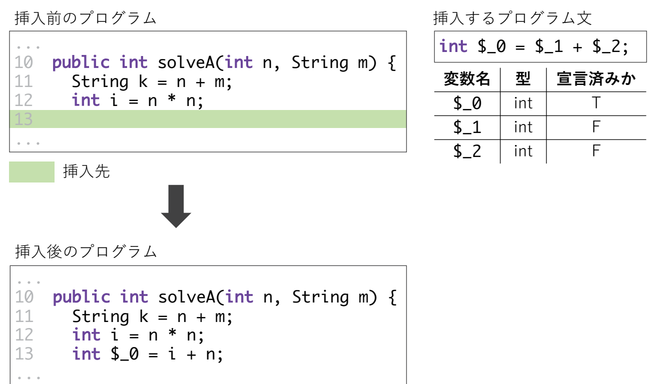


図 8　変数の書き換え例

で宣言されている変数は，挿入先よりも前の行で宣言されている変数と変数名が衝突しないように変数名を書き換える．挿入するプログラム文内で宣言されていない変数は挿入先よりも前の行で宣言されているかつ型が一致する変数のうちいずれか 1 つと同じ変数名に書き換える．条件を満たす変数が存在しない場合はプログラム文の挿入をしない．

変数の書き換えの例を図 8 に示す．13 行目（緑色の行）がプログラム文の挿入先である．挿入するプログラム文に含まれる宣言されている変数は$_0 である．13 行目以前で宣言されているかつ int 型である変数は n および i であり，そのままの変数名でも変数名は衝突しないため，$_0 の変数名は書き換えない．挿入するプログラム文に含まれる宣言されていない変数は$_1, $_2 である．これらの変数名を n または i のいずれかに書き換える．

5.　実験

5.1　実験対象

以下の条件を満たす ABC の A 問題 23 問から 2019 年 9 月 9 日時点で最新の 4 問を実験対象とした．

- 入力および出力が整数のみである．
- 分岐や繰り返しをせずに解ける，すなわち flat テンプレートを用いれば解ける問題である．
- 3.1 節の調査結果において再利用割合の最大値が 1 である．

対象とした問題を表 2 に示す．

[4] https://atcoder.jp/contests/abc121/tasks/abc121_a
[5] https://atcoder.jp/contests/abc125/tasks/abc125_a
[6] https://atcoder.jp/contests/abc128/tasks/abc128_a
[7] https://atcoder.jp/contests/abc134/tasks/abc134_a

表 2　実験対象

名前	概要	回答	テストケース数
White Cells[4]	H 行 W 列の白色のマス目から h 個の行と w 列の行を選び，黒色に塗った時に残った白色のマスの数を求める．	$(H-h)*(W-w)$	3
Biscuit Generator[5]	A 秒毎に B 枚のビスケットを作る機械が $T+0.5$ 秒後までに作るビスケットの数を求める．	$(T/A)*B$	3
Apple Pie[6]	A 個のリンゴと P 個のリンゴの欠片でできるアップルパイの数を求める．1 個のリンゴからは 3 個の欠片ができ，2 個の欠片から 1 個のアップルパイができる．	$(A*3+P)/2$	3
Dodecagon[7]	半径 r の円に内接している正十二角形の面積を求める．	$3*r*r$	3

ⓒ 2020 Information Processing Society of Japan

5.2 実験方法

過去のプログラミングコンテストの回答に含まれるプログラム文の再利用によって回答の生成が行えるかの検証が本実験の目的である．そのため，テンプレートの推定に成功したと仮定して実験を行った．つまり，著者が手動で作成した回答の雛形を拡張した kGenProg に入力として与えた．実験に用いた計算機の性能を表 3，kGenProg のパラメータを表 4 に示す．各パラメータに対して制限時間を 2 時間に設定し，Seed 値を変えて実験を 10 回行った．与えられた全てのテストケースに成功する回答が得られた場合は，残り時間に関わらず kGenProg を停止させた．

ABC の各問題に対する入出力例の数は 2〜4 個であるため，与えれられたテストケースのみ成功する回答が得られる可能性がある．得られた回答が以下の条件を満たす場合のみ，オーバーフィットがない回答と判断した．

- 無作為に作成した全てのテストケースに成功する．
- 著者が目視で正しい答えが得られると判断する．

5.3 実験結果

実験の結果を表 5 に示す．表 5 より，対象とした全ての A 問題に対して与えられたテストケースに成功する回答が得られ，Biscuit Generator 以外に対してオーバーフィットがない回答が得られたことがわかる．以降では実験で得られた回答の例を紹介する．

図 9 は White Cells に対するオーバーフィットがない回答の例である．図中の灰色の部分は回答に影響しないプログラム文である．この回答は $(W-w)*(H-h)$ を返している．White Cells は $(H-h)*(W-w)$ を返せば解ける問題であるから，図 9 の回答にはオーバーフィットがない．

図 10 は Dodecagon に対するオーバーフィットがある回答の例である．図中の灰色の部分は回答に影響しないプログラム文である．この回答は 10 と $r+r$ と $r*r$ を比較し，大きい数の 3 倍を返している．Dodecagon は $3*r*r$ を返せば解ける問題であるから，この回答にはオーバーフィットがある．例えば，入力が $r=1$ の時正しい出力は 3 であるが，図 10 の回答の出力は 6 となる．

図 11 はオーバーフィットがない回答が得られるまでに要した時間である．図の縦軸は実行時間（秒）である．横軸は対象とした A 問題の名前であり，左から White Cells, Apple Pie, Dodecagon である．いずれの A 問題も実行時間に大きなばらつきがあった．

表 3　実験に用いた計算機の性能

CPU	AMD Ryzen 9 3900X（3.8GHz，12 コア）
メモリ	64GB

表 4　kGenProg のパラメータ

変異で生成するプログラムの数	交叉で生成するプログラムの数	1 世代あたりに選択するプログラムの数
40	0	4
100	0	10
400	0	40

表 5　実験結果

問題名	テストケースに成功した回答の数	オーバフィットがない回答の数
White Cells	19	1
Biscuit Generator	2	0
Apple Pie	10	9
Dodecagon	8	7

```
// (H - h) * (W - w) が答え
public int solveA(int H, int W, int h, int w) {
  W = (W - w) * (H - h);
  w = w + ((H - W) * h);
  double $_0 = ((double) (H + w) / 2);
  return W;
}
```

回答に影響しないプログラム文

図 9　オーバーフィットがない回答の例（White Cells）

```
// 3 * r * r が答え
public int solveA(int r) {
  int $_0 = r - r / 2 * 2;
  int $$_6 = Math.max(r * r, r + r);
  $_0 = 24 + (24 - $_0);
  int $$_7 = ($_0 + $_0 + 2 - 1) / 2;
  $_0 = $_0 * (-1);
  $_0 -= 999;
  $_0 = r + r;
  $$_6 = $$_6 + (2 * $$_6);
  return $$_6;
}
```

回答に影響しないプログラム文

図 10　オーバーフィットがある回答の例（Dodecagon）

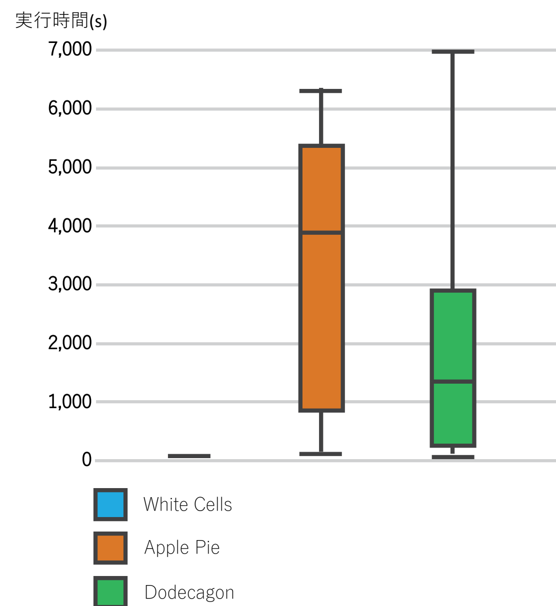


図 11　オーバーフィットがない回答が得られるまでに要した時間（秒）

ⓒ 2020 Information Processing Society of Japan

```
// (A * 3 + P) / 2 が答え
public int solveA(int A, int P) {
  int $_0 = A + ((P + A) / 2);
  A = P * P + P;
  return $_0;
}
```

回答に影響しないプログラム文

図 12　Apple Pie の回答の例

```
public static void main(String[] args) {
  // 標準入力から読み込む
  Scanner sc = new Scanner(System.in);
  int a = sc.nextInt();
  int b = sc.nextInt();
  int c = 0;

  if ((a + b) % 2 != 0) {
    c = 1;
  }

  System.out.println(c + ((a + b) / 2));
}
```

再利用したと考えられるプログラム文

図 13　再利用元と考えられる回答[*9]

6. 考察

本節では，各問題について回答の生成に成功あるいは失敗した理由について述べる．

回答の生成に成功した理由として，以下の 2 点が挙げられる．

- 似ていないが同じ値を返す式を用いれば解ける問題が存在した．
- 正しい答えを得られるようにプログラム文を組み合わせた．

White Cells と Apple Pie に対して生成した回答には似ていないが同じ値を返す式を用いれば解ける問題が存在した．例えば，Apple Pie に対して生成したオーバーフィットがない 9 個の回答のうち 8 個の回答には Round Up the Mean[*8] の回答に含まれるプログラム文が存在した．Apple Pie の回答の例を図 12 に示す．この回答は $A+((P+A)/2)$ を返している．この式を変形すると，$(A*3+P)/2$ となり，図 12 の回答にオーバーフィットはない．Round Up the Mean は，整数 a，b の平均値の小数点以下を切り上げた値を求める問題である．Apple Pie は A 個のリンゴと P 個のリンゴの欠片からできるアップルパイの数を求める問題であるから Apple Pie と Round Up the Mean は似ていない問題といえる．図 13 は再利用元と考えられる回答である．水色の部分が再利用したと考えられるプログラム文である．このプログラム文中の変数を書き換えた結果，図 12 の回答が得られたと考えられる．

Dodecagon の回答には Apple Pie に対する回答と異なり多くの回答に共通するプログラム文が存在しなかった．

*8 https://atcoder.jp/contests/abc082/tasks/abc082_a
*9 https://atcoder.jp/contests/abc082/submissions/5875920

```
// 3 * r * r が答え
public int solveA(int r) {
  r = (r * r);
  r += r + r;
  int $_0 = 24 + 24 - r;
  int $$_7 = Math.min($_0, ($_0 + r));
  int $$_8 = Math.min(r, r * $$_7);

  return r;
}
```

回答に影響しないプログラム文

図 14　Dodecagon の回答の例

回答

```
// (T / A) * B が答え
public int solveA(int A, int B, int T) {
  T = 10;
  int $_0 = A % 1000;
  $_0 = $_0 % 10;
  // 10 と B * (A % 1000 % 10) を比較し
  // 小さい数を返す
  int $$_9 = Math.min(T, B * $_0);
  String $$_7 = String.valueOf(A + "¥n");
  int $$_11 = (T + B) * T / 2;
  return $$_9;
}
```

回答に影響しないプログラム文

テストケース

入力	期待値	出力	成否
$A = 3,\ B = 5,\ T = 7$	10	10	○
$A = 3,\ B = 2,\ T = 9$	6	6	○
$A = 20,\ B = 20,\ T = 19$	0	0	○

図 15　Biscuit Generator の回答の例および各テストケースに対する出力

ゆえに，これらの回答は正しい値を出力するようにプログラム文を組み合わせた結果，偶然正しい回答と出力が得られる回答が生成できたと考えられる．Dodecagon の回答の例を図 14 に示す．この回答は 3 つの r の和を返している．r には $r*r$ が代入されているため，図 14 の回答は $3*r*r$ を返していることになり，オーバーフィットがない回答と判断できる．

オーバーフィットがある回答が生成できた原因として，テストケースに成功するようにプログラム文を組み合わせたことが考えられる．図 15 は Biscuit Generator の回答の例および各テストケースに対する出力である．図中の灰色の部分は回答に影響しないプログラム文である．この回答は 10 と A を 1,000 で割った余りを 10 で割った余りを比較して小さい数を返している．Biscuit Generator は T を A で割った商に B を掛けた値を返せば解ける問題であるから，図 15 の回答にはオーバーフィットがある．例えば，入力が $A=10$，$B=1$，$T=10$ の時正しい出力は 1 であるが，図 10 の回答の出力は 0 となる．

7. 妥当性の脅威

異なる 10 個の Seed 値を用いて実験を行った．そのた

め，本稿と異なる Seed 値を用いるあるいは Seed 値の数を増やすと実験の結果大きく変わる可能性がある．また，回答のオーバーフィットの有無を自動的に生成したテストケースに全て成功した回答に対して目視で確認したため，オーバーフィットがあるにも関わらずオーバーフィットがない回答と判断されたり，オーバーフィットがないにも関わらずオーバーフィットがある回答と判断されている可能性がある．

8. おわりに

本稿では，過去に開催されたプログラミングコンテストの問題に対する回答に含まれるプログラム文の再利用によってプログラミングコンテストの回答を生成する取り組みについて紹介した．また，分岐や繰り返しがなく解けるプログラミングコンテストの問題に対し実験を行った．実験の結果，対象にした 4 つ問題に対してテストケースに成功する回答が得られ，3 つの問題に対してオーバーフィットがない回答が得られた．

今後の課題として以下の 3 点が挙げられる．

- kGenProg の拡張
- テンプレートの推定の実現
- 適応度の計算方法の改良

現時点では回答を生成できない分岐や繰り返しがある問題に対しても回答の生成ができるように kGenProg を拡張する予定である．また，プログラム文の挿入時に挿入対象を平坦化 [1] することも考えている．平坦化とは複数の演算子を含むプログラム文を演算子が高々 1 つだけ含まれるプログラム文に分解する手法である．平坦化により細かい粒度でプログラム文を再利用できると著者は考えている．例えば，int a = b + c + 1; を挿入する時に int $_0 = b + c; と int a = $_0 + 1; に分解する．この分解によって，分解後のプログラム文の片方だけの再利用が可能になる．

現時点では，与えられた問題文からテンプレートの推定を行っておらず，今後実現する予定である．

現在，適応度はテストの成功率を用いている．適応度の計算方法を改良すればオーバーフィットがある回答が得られ難くなると考えている．例えば，入力として与えられた変数を全て使っていない回答は正しい回答とは考えにくい．そのため，テストの通過率に加えて入力として与えられた変数の使用率も適応度の計算に用いればオーバーフィットがある回答が得られ難くなると考えられる．

参考文献

[1] Higo , Y. and Kusumoto , S.: Flattening Code for Metrics Measurement and Analysis, *2017 IEEE International Conference on Software Maintenance and Evolution*, pp. 494–498 (2017).

[2] Abreu, R., Zoeteweij, P., Golsteijn, R. and van Gemund, A. J. C.: A Practical Evaluation of Spectrum-based Fault Localization , *J. Syst. Softw.*, Vol. 82, No. 11, pp. 1780–1792 (2009).

[3] Balog, M., Gaunt, A. L., Brockschmidt, M., Nowozin, S. and Tarlow, D.: DeepCoder: Learning to Write Programs, *Proceedings International Conference on Learning Representations 2017* (2017).

[4] Becker, K. and Gottschlich, J.: AI Programmer: Autonomously Creating Software Programs Using Genetic Algorithms, *CoRR*, Vol. abs/1709.05703 (2017).

[5] Britton, T., Jeng, L., Carver, G., Cheak, P. and Katzenellenbogen, T.: Reversible Debugging Software: Quantify the time and cost saved using reversible debuggers (2013).

[6] Clarke, L. A.: A System to Generate Test Data and Symbolically Execute Programs , *IEEE Transactions on Software Engineering*, Vol. SE-2, No. 3, pp. 215–222 (1976).

[7] Gvero, T. and Kuncak, V.: Synthesizing Java expressions from free-form queries, *Acm Sigplan Notices*, Vol. 50, No. 10, pp. 416–432 (2015).

[8] Higo, Y., Matsumoto, S., Arima, R., Tanikado, A., Naitou, K., Matsumoto, J., Tomida, Y. and Kusumoto, S.: kGenProg: A High-performance, High-extensibility and High-portability APR System , *the 25th Asia-Pacific Software Engineering Conference*, pp. 697–698 (2018).

[9] Jones, J. A. and Harrold, M. J.: Empirical Evaluation of the Tarantula Automatic Fault-localization Technique , *Proceedings of the 20th ACM/IEEE International Conference on Automated Software Engineering*, pp. 273–282 (2005).

[10] Le Goues, C., Dewey-Vogt, M., Forrest, S. and Weimer, W.: A Systematic Study of Automated Program Repair: Fixing 55 out of 105 Bugs for $8 Each , *Proceedings of the 34th International Conference on Software Engineering*, pp. 3–13 (2012).

[11] Man, K. F., Tang, K. S. and Kwong, S.: Genetic algorithms: concepts and applications [in engineering design] , *IEEE Transactions on Industrial Electronics*, Vol. 43, No. 5, pp. 519–534 (1996).

[12] Pacheco, C., Lahiri, S. K., Ernst, M. D. and Ball, T.: Feedback-Directed Random Test Generation , *Proceedings of the 29th International Conference on Software Engineering*, pp. 75–84 (2007).

[13] Zuddas, D., Jin, W., Pastore, F., Mariani, L. and Orso, A.: MIMIC: Locating and understanding bugs by analyzing mimicked executions , *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, pp. 815–825 (2014).

ⓒ 2020 Information Processing Society of Japan