

ランダムな変異を用いたバグ入りプログラムの生成

寺田 実^{1,a)}

概要：デバッグ学習などのためには誤りを含むプログラムが必要である。ここで、プログラムにランダムな変異を加えることによりエラーのあるプログラムを生成して、ソフトウェアテストシステムのエラー検出能力の評価に用いる mutation testing という手法が知られている。本発表では、この手法を用いて「良い」バグ入りプログラムを生成し、デバッグ技術の習得やデバッグ手法のベンチマークに利用することを目指す。「良さ」の一つの基準として、網羅的なテストデータに対する正解率を見ることで「ほとんど正しいがまれなケースにのみ誤る」タイプのバグを生成できると考え、実際に行った生成結果について考察する。

キーワード：バグ生成, Mutation Testing

1. はじめに

たとえばデバッグ練習用の問題集を考えてみる。どのような練習問題があればよいだろうか。誰もがやりがちな定型的なバグも必要であろうが、そうではない意外性のあるバグも不可欠だと思う。後者のタイプのバグをプログラムに変異をくわえることで自動的に生成できないだろうか、というのが本研究の目指すところである。大量のバグ入りプログラムを自動生成し、そのなかから目指す特性を持つものを選別することにより、練習問題として通用するような「よいバグ入りプログラム」の作成を試みた。

2. 目的

2.1 デバッグの必要性

ソフトウェア作成において、デバッグの必要性はまだ解消されていない。しかし、たとえばプログラミング入門教育においてはデバッグは主たる

テーマとしては扱われていないことが多い。「注意深くコードを見直す」とか「print 文を挿入して値を確認する」といった手段を紹介するにとどまっているのが大半であろう。デバッグツールの紹介にも至っていない。また、自分自身が作成したコードをただちにデバッグするのが主で、時間が経った自分のコードとか、他人が作成したコードのデバッグなどは経験することもない。したがって、プログラミング教育の中にもうすこしデバッグに注目したテーマがあってもよいと考える。その際に必要となるのはデバッグの練習問題である。

一方、デバッグのためのさまざまなソフトウェアツールが作成されている。古くからあるデバッガに加えて、コードを静的に解析して知識ベースと照合をおこない、バグである可能性の高い箇所を指摘するツールもある（一例をあげれば PMD, Findbugs）。しかし、それらのツール自身を評価するためには、標準的なバグのセットをベンチマークとして用意する必要があるだろう。

こうした要求のために、バグ入りプログラムのセットを用意する必要がある。本研究は、そのよ

¹ 電気通信大学

^{a)} terada.minoru@uec.ac.jp

うなセットの自動的な生成を試みるものである。

2.2 バグの集め方

バグ入りプログラムを揃えるにはいろいろな方法がある。

2.2.1 実際のコードから収集

現実性を持つコードを確保できる点でのぞましいが、いろいろ問題点がある

- 収集が容易ではない

個人の集められる件数はそう多くはなく、多くの人が集めたバグを共有する仕組みが必要となる。バグトラッキングシステムの利用が一つの解決策であるが、次項で述べる問題点もある。

- バグが単離できていない

現実のソフトウェアは大規模であるので、個々のバグには大きな関連情報が付随しているのが普通である。そうしたバグを学習やベンチマークで利用するには自己完結したバグとして単離（いわゆる “Short, Self Contained, Correct Example”）しておく必要がある。

2.2.2 経験をもとに作成

よくあるバグを再現するコードを手で作成してセットとする。たとえば、繰り返しの構造において回数を一回だけ多く/少なく行ってしまうバグ（いわゆる “Off-by-one error”）などを念頭に置いている。学習用に使うのであれば、こうしたセットを標準の問題集として利用するのはよいと思う。ただ、実際に直面するバグのうちにはそうしたイディオムでは解決の難しいものが存在し、そうしたものは人手で作るのは難しい。意外性が必要になるのだ。

2.2.3 自動生成

上記の「意外性のあるバグ」を作る方法として、計算機による自動生成を考える。本研究が着目したのはこの方法である。

3. バグ生成の手法

3.1 概要

本研究では、正解のプログラムを一つ用意し、それに変異を加えることでバグを作り出す。変異は

ランダムに行い、多数の試行を繰り返すことで望ましいバグをバリエーションを持って生成する。

この手法は遺伝的プログラミング (Genetic Programming) と類似している。そこでは、単純で正しくないコードを出発点とし、変異を繰り返すことで仕様を満たすコードを生成しようとする。

ソフトウェアテストの文脈では、コードに意図的にバグを埋め込みそれが発見できるかどうかでテストの進捗度を測定するという手法がある。アイデアそのものは [1] で Bebugging という名で提示されている。「プログラマがプログラムに対して信頼感を持ちすぎるのを防止するための方法として、テスト中のシステムにランダムな誤りを導入する、というものが有り得よう。」(訳書初版 P.315) 現在では、正しいプログラムに変異を加えてバグを導入し、その変異のうちのどれだけをテストが検出できるかによってテストセットの完成度を評価する手法として、Mutation Testing と呼ばれている [2]。これは本研究とアプローチに共通点が多いが、本研究ではバグ入りプログラムの生成に目標を置いている点で異なっている。

ソフトウェアテストにおけるもう一つの手法として Fuzzing がある。これは誤りを含むデータを生成してプログラムに与えて動作を観察するもので、近年ソフトウェアの脆弱性検出などに利用されてきている。

3.2 生成目標とするバグ

本研究で生成しようとするバグは、学習用の観点から「なるべく発見の難しいもの」とする。発見の難しさはなかなか定量化することができないが、たとえば以下のような観点が可能であろう。

(1) ソースコードレベルでの正解との類似

文字列の編集距離などの観点から、正解に似ているものをよいバグとする。これは人間が目視でバグを発見しようとする際には困難さの有効な尺度であり、つづりの類似した二つの変数を併用するなど悪いプログラミングスタイルである。ただ、支援環境が整っていれば異なる変数を弁別しやすく表示するなどが可能である。

(2) 誤りの発生が稀

デバッグにおける最初のステップが「確実にエラーを発生させるケース」を求めることであることからわかるように、発生率の低いバグは難しい。並行性が導入された場合、同一の入力によっても再現できないバグもありえて、このようなバグはもっとも対処の難しいものである。本研究は再現性のある逐次プログラムを対象とするので、こうした確率的なバグは扱わないが、それでも大多数のテストケースには正しい出力を行い、ごく少数のテストにだけ誤った出力を行うものがこうした希少性のあるバグに相当する。

(3) プログラムの振る舞いが正解と類似

データを与えたときのコードの振る舞いを実行トレースを用いて正解と比較した場合に、それらの違いがほとんどないものはデバッグが難しい。条件分岐が正解と同じ枝をたどるとか、繰り返しの回数が正解と同じなどである。これは前項のテストケースの一致率をコードの振る舞いに拡張したものといえる。

(4) 意外性のあるバグ

変数の初期化を忘れた、`>=`と`>`の間違い、繰り返しの回数が一回足りないなどはよくあるパターンであるが、そうではないバグは難しい。上記の観点のうち、本研究では(2)の誤りの発生が稀なものを生成の目標とする。具体的にはテストデータセットを入力としてプログラムを実行し、出力を正解と比較して正解数の多いものをよいバグと見ることにする。もちろん全データで正解と一致するものは正しいプログラムのはずなので、それらはバグからは除外する。

3.2.1 対象言語

本研究ではJavaを対象言語とした。これはプログラムに対する操作が比較的容易であることからである。Javaの統合開発環境であるEclipseには、Javaプログラムから抽象構文木(AST)を作成するクラスライブラリが含まれており、構文木レベルでプログラムに変異を与えるのが容易である。ASTではなくソースコードレベルでの文字単位の変異はコンパイルエラーを引き起こす可能性

が高く試行の効率が悪い。

本研究では入門プログラミングのレベルを対象とするので、整数および整数の配列の範囲で記述できるプログラムに限った。つまりオブジェクト指向は関係しない。Javaでは配列の添字チェックが行われるのでコードに含まれる誤りでプログラム全体が異常終了することはないことも利点である。

また、今回は利用していないが、Javaプログラムの実行トレースを採取することも容易なので対象言語に選んだ理由のひとつとなっている。

3.3 バグ生成手順

本研究では以下の手順でバグ入りプログラムを生成する。まず、正解プログラムを用意する。このとき、なるべくJava特有の言語機能を使用しないで作成するようにした。つぎに、テストデータセットを用意する。そのあと、変異と実行の繰り返しを行う。

3.3.1 AST生成と変異

正解プログラムを読み込み、ASTをメモリ上に生成する。そのASTを深さ優先でトラバースし、変異の候補となる節点に出会うたびに乱数を用いて一定確率で変異を加える。(したがって、プログラム全体での変異の発生数は一定しない。)変異候補節点は以下である。

● 変数参照

ASTのクラス名は`SimpleName`。その時点での名前表を検索し、ローカル変数であったなら一定確率で型の一致する別のローカル変数と置き換える。

● 二項演算

ASTのクラス名は`InfixExpression`。演算子が比較演算子(`<`, `<=`, `>`, `>=`, `==`, `!=`), 条件演算子(`&`, `|`, `&&`, `||`), 算術演算子(`+`, `-`, `*`, `/`, `%`)のいずれかのグループに属していたら、一定確率で同一グループの別のものと置き換える。

● ブロック

ASTのクラス名は`Block`。0個以上の文の並びである。文が二つ以上あった場合、以下のいずれかを一定確率で行う。

- 文の削除
並びのうちからランダムに一つの文を選び削除する
 - 文の複製
並びのうちからランダムに一つの文を選び、ランダムな位置に複製を挿入する
 - 文の交換
並びのうちからランダムにふたつの文を選び交換する
- 変異によって複製や移動のあった文についても再帰的に変異処理を加えていく。なお、Javaではブロック中のどこにでもローカル変数の宣言を置くことができるが、その削除や複製を許すとコンパイル時にエラーとなるので対象外としている。

変異は確率的であるため、試行の繰り返しの結果として同一のコードが生成されてしまう可能性がある。これを防止するために生成したコードのハッシュを保存しておき、重複を発見した場合にはそのあとの実行段階は行わない。

3.3.2 テストセット実行

変異を加えたプログラムをファイルに書き出し、コンパイルしてテストデータを入力として実行する。出力を正解と比較して正解率を算出し記録する。あとからコードを吟味する必要があるがコードそのものを保存するのはデータ量に問題があるため、変異を与えるもととなった乱数のシードもあわせて記録することでコードの再現を可能としている。

コンパイルの際にエラーになったもの、実行時に例外発生で異常終了したもの、規定時間内に実行が終了しなかったものは記録を残しておく。

ここで、規定時間で実行を打ち切る方法について説明しておく。変異を加えたプログラムは監視スレッドのもとで別スレッドとして動作している。Javaではスレッドを強制終了させる手段として `Thread.stop()` メソッドが用意されていたが現在では非推奨となっており、協力的でないスレッドを終了させることができない。本実験ではプログラムに変異を加える目的でASTを作成し加工しているので、その最終段階でループ本体の先頭に

```
if(killed) throw new Error("killed");
```

という文を挿入する。実行時に規定時間が経過したら監視スレッドが `volatile` 変数 `killed` を真にすることでスレッドを自発的に終了させる。

3.4 対象プログラム

本研究で対象とするプログラムは、プログラミング入門教育レベルを想定して、整数配列操作を中心とする小規模のものとした。整数配列に着目した理由は、添字と要素の型が同一であるためにそれらの混用によるバグを誘発するためである。具体的には以下である。

- LinkedList 上での挿入ソート
- QuickSort における partition
- HeapSort
- 8 Queen

このうち、LinkedList はリンクの実現にポインタは使用せずリンク先のデータの配列上の添字を使用している。また、8 Queen は再帰的に一段ずつ女王を置き、縦と斜め二方向の整数配列をもちいて利きの関係をチェックする。

テスト実行に用いた正解データは、ソートについては長さ 0 から 5 までの可能な整数列すべてに対するソート結果を用いた。可能な整数列とは要素の大小関係だけに着目したもので、たとえば {3, 1, 2} は {4, 1, 2} は等価であるが、{2, 1, 2} とは等価でない。こうした列の総数は Ordered Bell Number と呼ばれ、長さ 1, 2, 3, 4, 5 に対して 1, 3, 13, 75, 541 となる。

8 Queen については解の総数である 92 だけを用い、具体的な解の内容は考慮しなかった。

4. 結果

4.1 定量的結果

まず変異によって生成したプログラムに関する定量的な結果を示す。

4.1.1 エラー

変異を加えたプログラムを実行する段階で、コンパイルエラーと実行時エラーが発生しうる。

変異が原因となるコンパイルエラーを避けるために一定の配慮はしているが、たとえばブロック

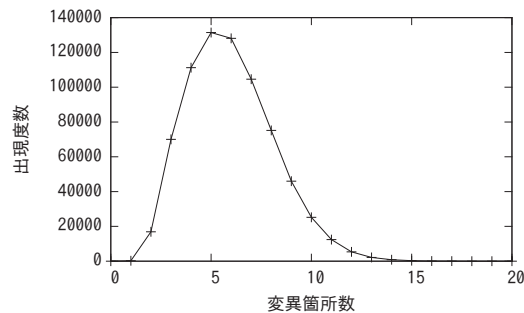


図 1 heapsort 変異箇所数の分布

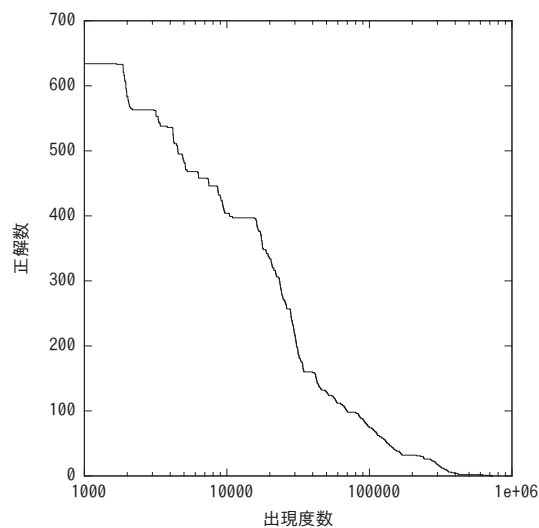


図 2 heapsort 正解数の分布

表 1 heapsort 正解数ごとの度数 (一部)

正解数	度数
634 (全テスト正解)	1681
633	175
632	15
628	2
...	...
4	21131
3	4810
2	202455
1	87403
0	18793
全試行数	730054

中での文の交換が原因でローカル変数が宣言より前に参照されてしまうなどがありうる。コンパイルエラーが発生した場合、テスト実行は行わない。コンパイルエラーの発生率は heapsort の場合で 3.8% であった。

いっぽう実行時エラーには、配列の範囲外参照、ゼロ除算、規定実行時間超過による強制終了などがある。これらは区別せずまとめて実行時エラーとして扱っている。テスト実行の際にある特定のデータに対して実行時エラーが発生した場合、誤った結果を出力したものとみなした。

4.1.2 変異箇所数

前述のとおり、AST における候補箇所ごとに一定確率で変異を発生させるため、変異の箇所数はさだまらない。heapsort プログラムについて変異箇所数の分布を図 1 に示す。このときのコンパイルエラーを除外した試行数は 730054、AST の変異候補ノード数は 80、変異確率は 0.1 である。

4.1.3 テスト実行での正解数の分布

heapsort でコンパイルできたプログラムについて、テストケースに対する正解数の分布を図 2 に示す。正解数の高い順にソートしてその累積度数を横軸 (対数) にとってある。全テストデータ 634 個すべてに正解したプログラムは 1681 通りあったので、図の横軸は 1 からでなく 1000 から始めてある。また、いくつかの正解数についてその度数を表 1 にも示した。正解率の高いプログラムは非常に稀であること、しかし全テストデータに近いスコアを得ているコード (惜しいコード) もわずかではあるが存在することがわかる。

8 queen で各プログラムが答えた解総数の分布を図 3 に示す。試行総数は 734111 であるが出力の解総数は広く分布し、最大が 334072、最小が 0 であった。試行を解総数順に並べ、真の値 92 に近い範囲だけをプロットしてある。正解である 92 のところに平坦な領域があることがわかる。また、正解の近くに「惜しいコード」があることもわかる。

4.2 生成したバグの類型

以下では変異によって生成したプログラムのうち、目視によってひろいあげた特徴的な例をいく

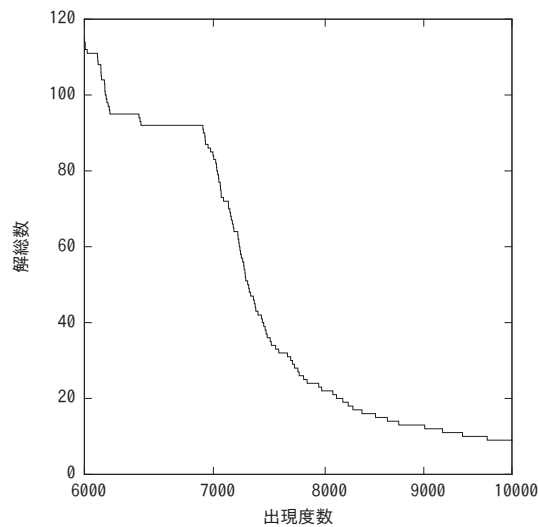


図 3 8 queen 解総数の分布

つか紹介する.

4.2.1 惜しいコード

本研究が目指した、テスト正解数が高い「惜しいコード」の例を示す. 図 4 に示すコードが heapsort での正解コードであるが, その 11 行目のあとに図 5 のコードを挿入したものがその「惜しいコード」となる. このコードは入力データの長さが 0 と 1 のときにだけ配列範囲外参照を起こす. 挿入部分は図 4 の 3 行目から 11 行目の for ループの複製にあたるが, 条件式の不等号を $<$ から $>=$ に変更してある. 入力データの長さが 2 以上の場合には挿入部分が実行されないで, このような振る舞いをすることになる.

また, 同じく図 4 のうちの 23, 24 行目を

```
if(a[j] >= a[l]){
    if(a[j] >= a[r]) break;
```

から

```
if(a[j] >= a[i]){
    if(a[j] <= a[r]) break;
```

と変更したものは, 空入力を含む 634 通りの入力のうち 616 通りに対して正しい出力を行う. この変更がなぜそのような「惜しいコード」になるかは自明ではない.

こうした「惜しい」バグは網羅的なテストが可能であれば検出可能だが, そうでない場合には発

```
1 static void heap0(int a[]){
2     int n = a.length;
3     for(int i=1; i<n; i++){
4         int j = i;
5         while(j > 0){
6             int p = (j-1) / 2;
7             if(a[j] <= a[p]) break;
8             swap(a, j, p);
9             j = p;
10        }
11    }
12    for(int i=n-1; i>0; i--){
13        swap(a, 0, i);
14        int j = 0;
15        while(true){
16            int l = j*2+1;
17            int r = l+1;
18            if(l >= i) break;
19            if(r == i){
20                if(a[j] < a[l]) swap(a, j, l);
21                break;
22            }
23            if(a[j] >= a[l]){
24                if(a[j] >= a[r]) break;
25                swap(a, j, r);
26                j = r;
27            } else if(a[l] < a[r]){
28                swap(a, j, r);
29                j = r;
30            } else {
31                swap(a, j, l);
32                j = l;
33            }
34        }
35    }
36 }
```

図 4 heapsort の正解コード

```
1 for(int i=1; i<=n ; i++){
2     int j = i;
3     while(j > 0){
4         int p = (j-1) / 2;
5         if(a[j] <= a[p]) break;
6         swap(a, j, p);
7         j = p;
8     }
9 }
```

図 5 heapsort の「惜しい」バグコード
図 4 の 11 行目のあとにこれを挿入する.

見がむずかしい。

4.2.2 変異を含んでも全正解

変異を含んでも全テストデータについて正解を与えるプログラムがあった。独立な代入文の入れ替えやその時点で同じ値をもつ変数への参照などの意味的に等価なものが大部分であったが、なかには本質的な変更となっている例があった。

図4のheapsortの正解コードにおいて、5行目を

```
while(j > 0){
```

から

```
while(true){
```

と変更してもすべてのテストを通過できる。(実際に変異で生成されたのは「while(n > 0){」であったが、意味的にはtrueと等価である。)jが0となってもループは終了せずもう一周余計に回ることになるが、そこでもう一つの脱出条件(7行目)で終了する。これはループの終了判定を一つ省略できることを示しており、データ件数によっては実行効率向上につながる可能性もある。

4.2.3 人間でも間違えやすいもの

人間が行いやすい誤りももちろん変異の結果として生成される。たとえば不等号の向きの誤り、不等号に等号を含むかどうかの誤りなどであり、意外性はない。

しかしこうした誤りはテスト合格数の観点からは正解数が大きく劣る場合が多く、かならずしも「惜しいバグ」とはならない。逆に言えば、比較的少数のテストケースでも発見が容易であるといえる。

4.2.4 不自然な変異

$x \geq x$, $x \neq x$ などの不自然な変異もしばしば見られた。これらが制御構文の条件式として出現すると、コードの一部を無効化してしまい結果としてコードをコメントアウトしていることになる。これは変異の効果をキャンセルするので、「惜しいバグ」を作る観点からは生成効率の向上のためにはできれば避けたい。両辺が変数参照のみ、といった単純な場合は変異の実装の工夫である程度は回避可能であろうが、一般の場合を対象とするには静的解析が必要となろう。

```
1 static int nq0(int r, int n,
2               int c[], int u[], int d[]){
3     if(r == n) return 1;
4     int s = 0;
5     for(int x = 0; x < n; x++){
6         int ui = r + x;
7         int di = r - x + n - 1;
8         if((c[x]==0)&&(u[ui]==0)&&(d[di]==0)){
9             c[x] = 1;
10            u[ui] = 1;
11            d[di] = 1;
12            s += nq0(r+1, n, c, u, d);
13            c[x] = 0;
14            u[ui] = 0;
15            d[di] = 0;
16        }
17    }
18    return s;
19 }
```

図6 8 queen の正解コード

4.2.5 データセットによる差

ソートのテストデータとして、1,2,3,4,5を要素とするデータセットと、100,200,300,400,500を要素とするデータセットを比較してみた。その結果、前者の方が正解数が高い(つまり「惜しいコード」が多い)という結果となった。

これは前者では配列の要素の値と配列の添字の範囲が重なっているため、それらの混用によって動作してしまう場合があるのに対して、後者では混用が直ちに実行時エラーとなるためである。逆にいうと、テストケースとしては後者の方がバグ検出力が高いということになる。

プログラミング教育の文脈でいえば、前者はバグの発見を難しくするという意味で教育的であり、後者はプログラミング作成で早期に誤りに気づくことができるという意味で教育的である。後者の視点からは、配列の要素型と添字型に互換性がない方が例題としてのぞましく、たとえば整数のソートよりは文字列のソートの方が適切だといえるであろう。

4.2.6 理解を越えるもの

図6に8 queenの正解コードを示す。女王の利き検査用の縦と斜めの配列c, u, dを用意し、

```
nq0(0, 8, c, u, d)
```

と呼び出すことで解の総数が返る仕様となっている。この12行目の再帰呼び出しの部分を

```
s += nq0(r+1, n, c, u, d);
```

から

```
s += nq0(x+1, n, c, d, d);
```

と変更すると、解の総数の正解92に対して93を返す。再帰での変数の渡し方が明らかにおかしいのは一目でわかるが、逆になぜこれが93という惜しい結果を出しているのかは不明である。コードの動作を解析してみると正解とはまったく異なる動きをしており、とうぜん解もまったく異なる。単なる偶然以上の説明ができない。

5. 考察

5.1 デバッグと新規作成

プログラムをデバッグするとき、現状のバグ入りコードを理解して修正する方法と出発点となる仕様から新規にコードを作成する方法がある。

前者の方法はバグの理解が前提だが、それは常に可能とは限らない。他人の作成したものや自分が作成しても時間の経ってしまったものの理解は簡単ではないし、たとえば前述した8 queenでの93を返すプログラムのように、まったく説明のつけられないバグもある。

一方、後者の新規作成を行うのもいろいろ問題がある。まず新規作成のコストに加えて、仕様が完全でなかった場合にそれまでのコードだからこそ動作していたケースに新たなバグを持ち込む危険性がある。また、自分自身が作成したコードについては、新規に作り直したとしても同様のバグに陥る可能性が高いであろう。

重要なのはバグの理解可能性の見極めだと考えられるが、本研究で生成したコードを使って練習することでそれを習得できるならば面白いと思う。

5.2 デバッグ教育

筆者は教員としてプログラミング入門教育における学生のコードを見てきた。そのなかにはバグを含むものも多数あるが、適切な採点のためにはその原因をできるだけ探求する必要がある。バグがあるからといって新規作成するわけにはいかな

いのだ。

教員という立場からやむなくバグ理解につとめてきたわけだが、実はこうした訓練は学生自身のプログラミング能力の向上にも意義のあるものだと考える。学生が自分のコードのバグについて深く考えることがのぞましいし、さらには自分では書かないようなコードについても考えることが重要である。そのために本研究で生成した「惜しいコード」を利用できるとよいと考える。

6. おわりに

正しいコードに確率的な変異を加えテストケース通過数という尺度でフィルタリングすることによって、たとえばデバッグの練習問題として使えるような「よいバグ入りプログラム」の作成を試みた。コードの自然さ、動的な振る舞いの類似など、異なる尺度によるフィルタリングも併用することでより有用な結果が作れるのではないかと考えている。

謝辞 本稿は2017年の夏のプログラミングシンポジウムで発表した内容を増補したものです。同発表においていただいたコメントに感謝いたします。本研究はJSPS科研費15K00091の助成を受けたものです。

参考文献

- [1] Gerald M. Weinberg: **The Psychology of Computer Programming**. 1971. (木村泉, 角田博保, 久野靖, 白濱律雄 訳: プログラミングの心理学. 技術評論社, 1994.)
- [2] Andreas Zeller, David Shuler, **Seeding Bugs to Find Bugs: Beautiful Mutation Testing**, in *Beautiful Testing Leading Professionals Reveal How They Improve Software*, O'Reilly, 2009. (大西 建児 監訳, 児島 修 訳: ビューティフルテストイングーソフトウェアテストの美しい実践, O'Reilly Japan, 2010.)