

# ベイズ更新を活用したバグ限局の有効性に関する考察

佐野 一樹 阿萬 裕久 川原 稔

プログラミングコンテスト等で使用されるオンラインジャッジシステムは、有用なプログラミング学習環境の 1 つとなっている。オンラインジャッジシステムは複数のテストデータを使ってプログラムの正誤判定を自動で行えるが、誤り（バグ）があった場合に、その原因箇所を特定して指摘するという点までは難しい。そのため、バグ位置を自動的に絞り込む自動バグ限局技術が有用な支援技術として期待されている。主要なバグ限局手法として、プログラムスペクトル（実行経路情報）に基づいた手法（Spectrum-Based Fault Localization: SBFL）が知られているが、条件分岐の同一ブロック内では実行経路情報がすべて同じになってしまうため、算出される（バグ位置としての）疑惑値も同じになってしまうという問題がある。特に、プログラミング初心者が作成するようなプログラムでは制御構造が単純なことが多く、結果として SBFL ではうまくバグ限局が行えないことが懸念される。そこで本論文では、SBFL とは異なる視点に立った手法として、ソースコード行ごとの内容に基づいてそこでのバグ修正発生率をベイズ更新によって推定する手法を新たに提案し、代表的な SBFL 手法との比較を行っている。

## 1 はじめに

近年、社会のデジタル化や人工知能技術の普及と発展に伴い、プログラミングに対する注目度が高まってきている [10]。それに伴い、多くの初心者がプログラミングを学ぶ機会や環境も増えつつある [9]。代表的な環境として AtCoder [3] といったプログラミングコンテストサイトが挙げられる。そういったプログラミングコンテストでは、参加者に対してプログラムを作って解くような問題が与えられ、参加者は自身の作ったプログラムを提出するとその正誤判定結果が返されるという流れになっている。その際、正誤判定は人手ではなく、オンラインジャッジシステムと呼ばれる自動システムによって行われる。具体的には、提出されたプログラムに対して、それをさまざまなテスト

データでもって実行してプログラムの正しさを確認するという処理が行われる。コンテスト終了後にも、出題された問題とオンラインジャッジシステムは利用できるよう公開されており、それらは有益なプログラミング教材として活用できるようになっている。

このようにオンラインジャッジシステムはプログラミング学習に役立つが、それによって得られる正誤判定結果は単に正解または不正解というだけあり、提出されたプログラムのどこに誤りがあるのかを指摘するものではない。それゆえ、初心者にとってはどの部分を修正すればよいのか分からず、先に進めなくなってしまうことが懸念される。この課題に対し、自動プログラム修正 [5] という技術の活用が期待される。自動プログラム修正とは、バグ（誤り）のある箇所を自動的に絞り込み（これをバグ限局という）、その上でさまざまな書き換えを試すことでバグの自動修正を試みるという手法である。この技術を活用できれば、人手を介さずに誤り箇所を指摘し、なおかつ“ヒント”として修正案を提示することも可能になる。

しかしながら、現状ではバグ限局が必ずしも高精度でないこと、さらには現実的な時間で修正を完了できるのは 1-2 行程度の修正に限られることから、直ち

A Study of Usefulness of Bug Localization Using Bayesian Updating.

Kazuki Sano, 愛媛大学大学院理工学研究科, Graduate School of Science and Engineering, Ehime University.

Hirohisa Aman, Minoru Kawahara, 愛媛大学総合情報メディアセンター, Center for Information Technology, Ehime University.

に上述の課題を解決できるわけではない。それゆえ、まずは修正すべき箇所の特定が重要であり、そのための技術を確立することが必要である。既存のバグ限局手法はテストの際の実行経路情報（プログラムスペクトル）を活用するもの[1][7]が多いが、初心者向けの問題の場合には単純な構造のプログラムであることも珍しくなく、結果としてそのような実行経路情報ではうまく絞り込むことが難しい場合が想定される。

そこで本論文では新たなアプローチとして、あらかじめバグを含む誤りプログラムを多数入手して、ソースコード行に見られる主要な特徴（特定のキーワードの出現や識別子の個数等）ごとのバグ修正発生率を調べる。そして、それらを経験的確率と見なし、そこにベイズ更新[12]の技術を適用することで、各ソースコード行でバグ修正が起こる確率を推定するという手法を新たに提案する。これにより、ソースファイル内のどの行でバグ修正が必要となるかを確率のかたちで表し、バグ限局の助けとすることを目指す。

本論文の構成は以下の通りである。まず、2節で従来手法とその課題について概説し、3節で従来手法での課題に対する新たなアプローチ（バグ限局手法）を提案する。そして、4節では提案手法に対する評価実験を行い、その有用性について考察する。最後に5節で本論文のまとめと今後の課題について述べる。

## 2 スペクトルベースのバグ限局とその課題

スペクトルベースのバグ限局（Spectrum-Based Fault Localization:SBFL）は、軽量で言語非依存かつ使いやすく、テスト実行時間のオーバーヘッド時間が比較的小さいという観点から、最も有力なバグ限局手法の1つとして注目されている[11]。SBFLは、テスト実行時のカバレッジ情報とテスト結果に基づき、各行に対する疑惑値を算出し、各行を疑惑値の降順に並び替え、その上位の行にバグがあると推測する手法である。疑惑値の算出には、疑惑値算出式と呼ばれる数式を用いる。これまでに様々な疑惑値算出式が提案されているが、その中でも広く利用されているの

はOchiai[2]という算出式（次式）である：

$$\text{Susp}(s) = \frac{\text{fail}(s)}{\sqrt{\text{total\_fail}(s) \times \{\text{fail}(s) + \text{pass}(s)\}}},$$

ただし、

- $\text{Susp}(s)$  :  $s$  行目の疑惑値
- $\text{fail}(s)$  : 行  $s$  が失敗テストで実行された回数
- $\text{pass}(s)$  : 行  $s$  が成功テストで実行された回数
- $\text{total\_fail}(s)$  : 失敗テストの総数

である。

しかしながら、SBFLでは複数の連続する行に対して同じ疑惑値を割り当ててしまうという“同順位問題”が課題点として知られている。疑惑値の計算はテストの実行経路情報に基づくため、条件分岐により分割される同一ブロック内の全てのソースコード行の疑惑値が同じになってしまうことがその要因である。特に、初心者が作成するような比較的単純なプログラムでは条件分岐が少なく、結果的にほとんどのテストケースの実行パスが同じになってしまうためにこの問題が起こりやすい。そのため、SBFLでは、その種の単純な構造のプログラムに対してバグ限局がうまくいかない指摘されている[8]。

## 3 ベイズ更新を活用したバグ限局

前節で説明したように、SBFLでは単純な構造をした誤りプログラムではバグの原因箇所（修正すべき箇所）をうまく絞り込めない恐れがある。そのため、SBFLでの精度向上に関する取り組みはいくつかあるが（例えば、Liら[8]は正解プログラムと誤りプログラムの実行時に取得される変数値のシーケンスに着目し、その差異からバグ潜在が疑われるソースコード行を推定する手法を提案している）、本論文ではより軽量の別のアプローチを考え、その適用可能性について検討する。具体的には、バグを含んだプログラムにおける主要な特徴に着目し、それらに対するバグ修正の実績を調べた上でベイズ統計学を活用した新たな手法を提案する。これは、初心者がよくやる誤りは特定の構文やパターンに集中しやすいのではないかという仮説の下、各ソースコード行に現れる特徴とバグ修正発生率を関連付け、複数の特徴の組合せに対し

てバイズ更新によってバグ修正発生率の統合を図ると  
いうアイデアである。

提案手法は次の 3 つの段階で構成される：

- (1) バグ修正実績データの収集
- (2) バイズ更新による確率推定
- (3) バグ修正箇所の推定

以下、これらについて順を追って説明する。

### 3.1 (1) バグ修正実績データの収集

まずは、プログラミングコンテストサイトから誤りプログラムとそれを修正した正解プログラムのペアを取得する。ただし、ユーザが複数のプログラムを提出する場合を考慮し、正解プログラムはあるユーザが初めて提出した正解、誤りプログラムはその直前に提出されたものとする。そして、集めたプログラムに対して次の 4 つの方針で正規化を行う：

- 各行のインデントは削除する。
- 1 行は 1 つの実行可能文のみで構成されるようにする。つまり、セミコロン等の区切り文字で複数の実行可能文が併記されていた場合は、それらを別々の行へ分割する。
- 空白及びタブ文字はそれが文字リテラルまたは文字列リテラルの一部でない限り削除する。
- コメント文は削除する。

続いて、誤りプログラムと正解プログラムの差分を調べ、バグ修正が行われたソースコード行の位置を特定する。ただし、プログラムの修正が新たなコード行の追加のみとなっていた場合、修正前には対応するコード行が存在しないため、バグが含まれていたソースコード行を特定できない。そのため、本論文ではそのような修正は対象外とする。

さらに、誤りプログラムの各ソースコード行に対し、主要な特徴の出現状況を調べる。本提案では、以下の特徴に注目する：

- (i) プログラムの実行の流れ（制御フロー）に何らかの影響を及ぼす、関係すると考えられる命令・演算子の出現の有無（表 1）
- (ii) メソッド呼出しや変数宣言の有無、並びに識別子やリテラルの出現頻度（表 2）

なお、表 2 における“識別子の出現頻度 (ID1~ID4)”

表 1 注目するプログラム要素

if	synchronized
else	throw
for	代入演算子 $(=, +=, -=, *=, /=, %=, \&=,  =, ^=, \<<=, \>>=, \>>>=)$
do	
while	
continue	インクリメント・デクリメント $(++, --)$
break	
return	算術演算子 $(+, -, *, /, \%)$
assert	
try	比較演算子 $(==, !=, <, <=, >, >=)$
catch	$>=, ?$
switch	論理演算子 $(\&\&,   , !, \^)$
default	ビット演算子 $(\&,  , \^, \<<, \>>)$
case	

表 2 その他の特徴

メソッド呼び出しの有無
変数宣言の有無
識別子の出現頻度 (ID1: 低い)
識別子の出現頻度 (ID2: やや低い)
識別子の出現頻度 (ID3: やや高い)
識別子の出現頻度 (ID4: 高い)
リテラルの出現頻度 (LT1: 低い)
リテラルの出現頻度 (LT2: 高い)

とは、各ソースコード行における識別子の出現回数に基づいて全体を四分位数で分けた 4 つのグループを意味し、ID1 は最も少ない 25% の範囲に該当することを意味する。ID2, ID3, 及び ID4 についても同様である。“リテラルの出現頻度 (LT1, LT2)” は、各行に出現するリテラル数に基づいて分類したものであり、LT1 は少ない場合（出現数が 1）、LT2 は多い場合（出現数が 2 以上）を示す。本研究では、JavaParser [6] を使用してこれらの特徴を抽出した。

そして、各特徴について、それが出現（該当）しているソースコード行でバグ修正が行われた割合を当該特徴の“バグ修正率”（実績値）として算出する。例えば、if 文に注目したとき、実験対象となる全ての誤りプログラムにおいて、全部で  $n$  行に if 文が 1 回以上出現していたとする。そして、そのうちの  $b$  行ではバグ修正が行われていたとする。この場合、if が登場するソースコード行  $s$  のバグ修正率  $p(s|if)$  を

$$p(s|if) = \frac{b}{n}$$

として算出する。

### 3.2 (2) ベイズ更新による確率推定

各特徴に対応するバグ修正率を経験的確率と見なすことで、各ソースコード行におけるバグ修正発生確率を推定することはできるが、一般に1つのソースコード行には前出した複数の特徴が同時に現れることも考えられる。そこで、各特徴に対応する確率をベイズの定理とベイズ更新を使って統合し、各ソースコード行におけるバグ修正発生率を推定する。この手順を以下に示す。

いま、あるソースコード行に着目し、その行がバグ修正されるという事象を  $A_1$ 、バグ修正されないという事象を  $A_2$  で表す。次に、特定のプログラムの特徴（例えば if 文）に注目するものとして、あるソースコード行にそのプログラムの特徴が出現しているという事象を  $B_1$ 、出現していないという事象を  $B_2$  と表す。ここで、その注目しているプログラムの特徴が出現しているという条件の下でバグ修正が起こる条件付き確率  $p(A_1|B_1)$  は、ベイズの定理より

$$p(A_1|B_1) = \frac{p(B_1|A_1)p(A_1)}{p(B_1|A_1)p(A_1) + p(B_1|A_2)p(A_2)} \quad (1)$$

と書ける。ここで上式における各要素はそれぞれ次の確率を表している：

- $p(A_1|B_1)$ ：注目している特徴が見られるソースコード行でバグ修正が行われるという事後確率である。上述した各特徴を前提としたバグ修正率がこれに該当する。
- $p(A_1)$ ：当該ソースコード行でバグ修正が行われるという事前確率である。
- $p(A_2)$ ：当該ソースコード行でバグ修正が行われないという事前確率である。これは  $1 - p(A_1)$  で求まる。
- $p(B_1|A_1)$ ：バグ修正が行われたソースコード行で、注目している特徴が見られる確率である。
- $p(B_1|A_2)$ ：バグ修正が行われなかったソースコード行で、注目している特徴が見られる確率である。

(1) 式の左辺値だけであれば実績データ収集の時点ですべて得られていることになるが、実際には1つのソースコード行に複数のプログラムの特徴が見ら

れることもあるため、それらを組み合わせたバグ修正確率を推定する必要がある。そのため、ベイズ更新と呼ばれる手法を活用する。

便宜上、別の特徴がソースコード行に見られるという事象を  $C_1$ 、見られないという事象を  $C_2$  で表す。このとき、2種類の特徴が同時に見られるという条件付きでのソースコード行でのバグ修正確率  $p(A_1|B_1, C_1)$  を次式で算出するというのがベイズ更新である：

$$p(A_1|B_1, C_1) = \frac{p(C_1|A_1)p(A_1|B_1)}{p(C_1|A_1)p(A_1|B_1) + p(C_1|A_2)\{1 - p(A_1|B_1)\}}$$

(1) 式に示したベイズの定理と比べて、事前確率  $p(A_1)$  が  $p(A_1|B_1)$  に置き換わっている。つまり、事象  $B_1$  が生起しているという条件が既に成立している状況を改めて前提として、その上でベイズの定理を利用するというものである。ただし、ベイズ更新は要素間の独立性を前提としているため、プログラムの特徴間の共起関係を調べる必要がある。例えば、“for” と “比較演算子” は同じソースコード行に存在する可能性が極めて高く、独立した事象とは言い難い。そのため、実績データにおいて共起確率が上位5%以内の場合は、強い共起関係があるペアとみなし、ベイズ更新を適用しないようにする。同様の処理を繰り返すことで、ソースコード行に任意個の特徴が現れてもそれに合わせたバグ修正確率を求めることができる。

例として、あるソースコード行に3種類の特徴が見られた場合の計算例を示す。便宜上、それらの出現を事象  $B_1, C_1, D_1$  で表す。まず、 $p(A_1|B_1)$  は実績データ収集によって既に求まっていることになるが、ここでは例として  $p(A_1|B_1) = 0.2$  としておく。

次に、 $p(A_1|B_1, C_1)$  をベイズ更新によって求める。仮に、 $p(C_1|A_1) = 0.6$ 、 $p(C_1|A_2) = 0.4$  とすると、

$$\begin{aligned} p(A_1|B_1, C_1) &= \frac{p(C_1|A_1)p(A_1|B_1)}{p(C_1|A_1)p(A_1|B_1) + p(C_1|A_2)\{1 - p(A_1|B_1)\}} \\ &= \frac{0.6 \times 0.2}{0.6 \times 0.2 + 0.4 \times (1 - 0.2)} \\ &\approx 0.273 \end{aligned}$$

となり、2種類の特徴が同時に現れていることで、条件付き確率は0.2から0.273へ更新される。

さらに第 3 の特徴も同時に現れていることを考慮した  $p(A_1|B_1, C_1, D_1)$  をベイズ更新によって求める。仮に,  $p(D_1|A_1) = 0.04$ ,  $p(D_1|A_2) = 0.02$  とすると,

$$\begin{aligned} & p(A_1|B_1, C_1, D_1) \\ &= \frac{p(D_1|A_1)p(A_1|B_1, C_1)}{p(D_1|A_1)p(A_1|B_1, C_1)+p(D_1|A_2)\{1-p(A_1|B_1, C_1)\}} \\ &= \frac{0.04 \times 0.273}{0.04 \times 0.273 + 0.02 \times (1 - 0.273)} \\ &\simeq 0.429 \end{aligned}$$

が得られる。2 種類目に比べて確率の上昇幅が大きいが, これは  $p(D_1|A_1)$  と  $p(D_1|A_2)$  の比率が影響している。3 番目の例では前者の確率が後者の 2 倍になっているため, その要素の出現が重要であることを意味している。

### 3.3 (3) バグ修正箇所の推定

ベイズ更新を用いて各ソースコード行に対するバグ修正確率を算出した後, それらをプログラム内で降順に並べ, 上位に位置する行でバグ修正が行われると推定する。この推定の評価尺度として EXAM スコアを用いる。EXAM スコアは, ソースコード全体のうち, バグのある行に到達するまでにどれだけのコードを調査する必要があるかを表す指標であり, バグ箇所特定手法の性能評価によく用いられる。その値が小さいほど, バグ限局の精度が高いことを意味する。誤りプログラムに対して, バグのある行の順位を  $r$  とし, 実行可能なソースコード行の総数を  $N_{\text{exec}}$  としたとき, EXAM スコアは次式で定義される:

$$\text{EXAM} = \frac{r}{N_{\text{exec}}} .$$

ただし, 疑惑値が同一の行が複数存在し, 同順位となる場合には文献 [8] に従い, 最下位の順位を  $r$  として採用する。つまり, バグのある行の疑惑値の順位が  $x$  であった場合, 他に同順位の行が存在しなければ  $r = x$  としてよいが, そうでない場合は順位  $x$  の行の個数を  $k$  とし,  $r = x + k - 1$  とする。例えば, バグ限局の候補となる行が全部で 6 行あり (つまり,  $N_{\text{exec}} = 6$ ), 各ソースコード行の疑惑値順位がプログラムの上から順に (1, 1, 1, 4, 4, 6) であったとする。そして, 実際の (正解となる)

バグは上から 3 番目の行 (順位  $x = 1$ ) にあったとする。この時,  $1(=x)$  位の行が  $3(=k)$  つあるため,  $r = x + k - 1 = 1 + 3 - 1 = 3$  となる。したがって, この場合の EXAM スコアは  $3/6 = 0.5$  となる。

## 4 実験

前節では, 誤りプログラムの修正事例に基づいて, プログラムの特徴ごとにバグ修正の起こりやすさを定量化し, ベイズ更新の概念を適用することで変更すべき箇所を推定する方法を提案した。本節では, 提案手法の有用性について検討するために行った評価実験について報告する。以下, 4.1 節, 4.2 節及び 4.3 節で本実験の目的, 対象及び手順をそれぞれ説明する。そして, 4.4 節で実験結果を示し, 4.5 節で実験結果に対する考察を述べる。

### 4.1 目的

前述したように, プログラミング初心者が作成するようなプログラムは比較的小規模かつ単純な構造のものが多く, それゆえ, 一般によく使われるテストの実行経路情報 (プログラムスペクトル) を使ったバグ限局手法では修正すべき箇所の絞り込みが難しい可能性がある。そこで本論文ではプログラムの修正データ, 具体的には誤りプログラムとそれを修正した正解プログラム間の差分情報に着目し, どういった特徴 (命令文や演算子の出現の有無, 識別子の出現頻度等) のソースコード行ではバグ修正が起こりやすいかを定量的に分析して推定に活用する手法を提案している。これを踏まえて, 本実験では以下の研究課題 (Research Question: RQ) についてデータの収集と分析を行い, 提案手法の有用性について検討していく。

- **RQ1:** プログラミング初心者が作成するような小規模プログラムを対象とした場合, 代表的な SBFL 手法によるバグ限局では, 同順位問題がどの程度発生するか?
- **RQ2:** ベイズ更新を用いた提案手法によるバグ限局は, プログラミング初心者が作成するような小規模プログラムに対して有効か?

RQ1 は, 従来手法の課題である同順位問題が, 本論文で想定している “初心者が作成するプログラム”

表 3 実験対象の問題とプログラムペア数

問題 ID (プログラムペア数)							
101.a (19)	101.b (59)	101.c (129)	101.d (17)	111.a (29)	111.b (63)	111.c (135)	111.d (0)
102.a (18)	102.b (33)	102.c (128)	102.d (11)	112.a (26)	112.b (86)	112.c (159)	112.d (64)
103.a (44)	103.b (84)	103.c (59)	103.d (37)	113.a (10)	113.b (149)	113.c (36)	113.d (8)
104.a (24)	104.b (164)	104.c (47)	104.d (28)	114.a (58)	114.b (59)	114.c (35)	114.d (13)
105.a (116)	105.b (113)	105.c (31)	105.d (38)	115.a (36)	115.b (14)	115.c (68)	115.d (56)
106.a (13)	106.b (140)	106.c (131)	106.d (15)	116.a (28)	116.b (60)	116.c (46)	116.d (33)
107.a (15)	107.b (30)	107.c (113)	107.d (12)	117.a (0)	117.b (68)	117.c (26)	117.d (77)
108.a (27)	108.b (43)	108.c (84)	108.d (0)	118.a (43)	118.b (32)	118.c (38)	118.d (30)
109.a (56)	109.b (44)	109.c (86)	109.d (48)	119.a (70)	119.b (0)	119.c (28)	119.d (27)
110.a (44)	110.b (0)	110.c (66)	110.d (23)	120.a (57)	120.b (143)	120.c (36)	120.d (33)

で実際にどの程度発生するかを明らかにすることが狙いである。これにより、一般的な SBFL 手法のオンラインジャッジシステムへの適用可能性を確認するとともに、別の視点に立ったアプローチの必要性についても論じることができる。そして、RQ2 は提案手法の有用性に関する問いであり、本論文で想定している小規模で単純なプログラム群に対して、提案手法がバグ限局の精度向上に貢献できるか検討していく。

#### 4.2 対象

本実験では、AtCoder Beginner Contest の第 101 回から第 120 回までに出题された A 問題、B 問題、C 問題、D 問題の全 80 問に対し、同一ユーザによる正解プログラムと誤りプログラムのペア 4168 組を実験対象とする。各問題に対しての取得できたプログラムペア数は表 3 に示す通りである。なお、ペア数が 0 となっている問題については、テストケースに不備があったため、SBFL でのバグ限局が不可能であることから実験対象から除外した。

#### 4.3 手順

本実験の手順を以下に示す。

##### (1) SBFL によるバグ限局の課題点 (RQ1)

Java プログラムを対象とした SBFL ツールである GZoltar [4] を用いて、各誤りプログラムの各ソースコード行に対し、Ochiai の式に基づい

て疑惑値を算出する。その後、疑惑値が最も高い行が全体に占める割合（疑惑値 1 位の割合）を算出し、同順位問題がどの程度起きているのかを調査する。

##### (2) バイズ更新によるバグ限局 (RQ2)

実験対象である 4168 個の誤りプログラムの集合をランダムに 10 分割する。それらを便宜上 “ $D_1$ ”, “ $D_2$ ”, ..., “ $D_{10}$ ” と呼ぶ。そして、そのうちの 1 つ（便宜上  $D_k$  とする）を除いた 9 つ ( $\bigcup_{i \neq k} D_i$ ) をひとかたまりのデータセットする。まず、 $\bigcup_{i \neq k} D_i$  から、各特徴に対応したバグ修正率を算出する。

続いて、3.2 節で説明したバイズ更新を適用して、 $D_k$  に含まれる各ソースコード行で “バグ修正が行われる確率（バグ修正確率）” を求める。そして、バグ修正確率の上位に位置するソースコード行で “実際にバグ修正が起こったかどうか” を調べる。この操作を  $k = 1, 2, \dots, 10$  について繰り返す。各繰り返しにおける評価基準には 3.3 節で紹介した EXAM スコアを使用し、10 回の平均値をもって総合評価とする。なお、EXAM スコアの計算においては、SBFL とバイズ更新の両手法で算出される疑惑値の対象行の和集合を分母として用いることで、評価対象を統一している。これにより、両手法の比較における公平性を保っている。

## 4.4 結果

### 4.4.1 RQ1 (同順位問題がどの程度発生するか?)

Ochiai を用いた SBFL 手法によって疑惑値 1 位となったソースコード行が当該ソースファイルに占める割合をまとめた結果を図 1 に示す。

図 1 から、全体としては 0 に近い値も一定数見られるものの、どちらかといえば 1 に近い値を示すプログラムが多く存在していたことを確認できた。なお、平均値は 0.449、中央値は 0.250 であり、疑惑値 1 位の割合は過半数のプログラムで 20% を超えており、実際に同順位問題が起こりやすい傾向にあったといえる。

### 4.4.2 RQ2 (バイズ更新を用いた提案手法は有効か?)

表 4 に本実験で注目した特徴ごとのバグ修正率を示す。ただし、バグ修正の件数が少ないプログラムの特徴のバグ修正率については数値としての信頼性が十分でない可能性があるため、バグ修正率の 95% 信頼区間に基づいて統計的に有意と判断された 25 個の特徴のみを対象とすることにした。

そして、表 4 に示す各特徴に着目し、各ソースコード行でのバグ修正発生率をバイズ更新によって推定してバグ限局を行うという実験をデータセットを変えながら 10 回行った。なお、表 4 に示すバグ修正率は収集した全データを使って算出されたものであり、EXAM スコアを使った評価ではデータセットごとに取得し直した点に注意されたい。結果を表 5 に示す。同表では同じデータセットに対して、Ochiai を用いた

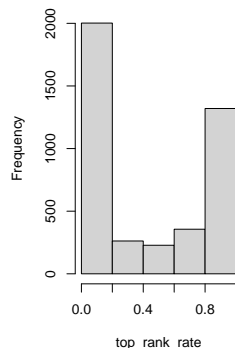


図 1 Ochiai での疑惑値 1 位の割合の分布

表 4 各特徴に対応したバグ修正率

プログラムの特徴	バグ修正率
if	0.276
for	0.170
while	0.133
continue	0.222
break	0.217
return	0.075
try	0.024
catch	0.023
do	0.050
else	0.167
throw	0.022
代入演算子	0.207
インクリメント・デクリメント	0.172
算術演算子	0.305
論理演算子	0.256
比較演算子	0.234
ビット演算子	0.165
メソッド呼び出し	0.185
変数宣言	0.153
識別子の出現数 Q1	0.132
識別子の出現数 Q2	0.146
識別子の出現数 Q3	0.160
識別子の出現数 Q4	0.237
リテラルの出現数 1	0.209
リテラルの出現数 2 以上	0.288

SBFL によるバグ限局結果についても併記している。

表 5 から、バイズ更新の EXAM スコアの平均は 0.197 となり、全実行可能行のうち上位 20% を調査すればバグのある行に到達できることが分かった。一方で、SBFL の EXAM スコアの平均は 0.490 となり、全実行可能行のうち半分近くを調査しないとバグのある行に到達できなかったことが分かった。

続いて、EXAM スコアの平均だけでなく分布についても比較を行ったところ、それぞれ図 2 及び図 3 に示すかたちとなっていた。図 2 から、バイズ更新

表 5 EXAM スコア

データセット	バイズ更新	SBFL
dataset-1	0.202	0.492
dataset-2	0.191	0.494
dataset-3	0.197	0.472
dataset-4	0.201	0.510
dataset-5	0.203	0.497
dataset-6	0.204	0.513
dataset-7	0.215	0.496
dataset-8	0.182	0.475
dataset-9	0.202	0.495
dataset-10	0.173	0.462
平均	0.197	0.490

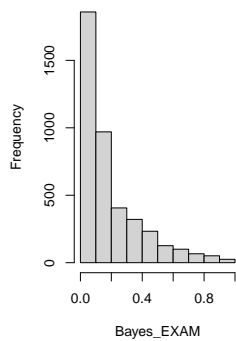


図2 提案手法（ベイズ更新）での EXAM スコアの分布

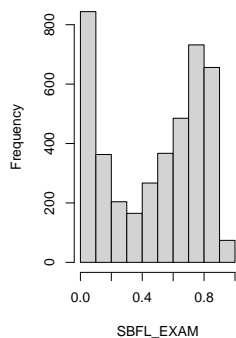


図3 従来手法（SBFL）での EXAM スコアの分布

を用いた提案手法によるバグ局限では、EXAM スコアは 0.0~0.2 の範囲に集中しており、バグのある行を早期に特定できたケースが多かったことが分かる。一方、図3から、SBFL での EXAM スコアは 0.8 付近に多く集まっており、バグのある行を上位に特定できなかったケースが多かったことが分かる。

#### 4.5 考察

##### 4.5.1 RQ1 (同順位問題がどの程度発生するか?)

図1に示したように、Ochiai の疑惑値 1 位の平均値は 0.449 であり、1.0 に近いものも多く確認できた。これは実行可能行の半分もしくは全てを見ないとバグを見つけられないものが多いことを示している。したがって、プログラミング初心者が作成するような小規模プログラムの場合、テスト結果だけではバグ位置の絞り込みが難しい場合が多いと言える。

この傾向についてより詳しく考察するため、データセットに含まれるプログラムの中で、比較的小規模なプログラムと大規模なグループに分け、更なる分析を行った。ここでは、“行数 (LOC)” と “サイクロマ

ティック複雑度” を指標とし、それぞれを四分位範囲で分割して得られる 16 種類にプログラムのグループ分けを行った。そして、各問題の誤り（修正前）プログラムについて、16 個のグループの中で該当するプログラムが最も多かったグループを“その問題が属する”グループと見なすことにした。その結果、データセットに含まれる全問題は全体で 7 つのグループに分類されたが、その中でも該当する問題数が多い（10 以上の）グループが 4 つ見られた。これら 4 グループの特徴を図4~図7に示す。Group1 は行数とサイクロマティック複雑度がそれぞれ最も小規模なグループであり、Group2、Group3 となるにつれて徐々に規模が大きくなる。Group4 は両指標において最も大規模なグループである。

そして、これらのグループごとに Ochiai による疑惑値 1 位の割合を算出した結果（箱ひげ図）を図8に示す。図8より、最小値、第1四分位数、中央値、第3四分位数、最大値のいずれにおいても Group1 が最も高く、プログラムの規模・複雑さが大きくなるにつれてそれらの値が低くなっていることが確認できた。特に中央値に着目すると、比較的小規模である Group1 及び Group2 が約 0.6 であるのに対して、

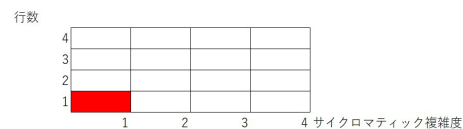


図4 Group1

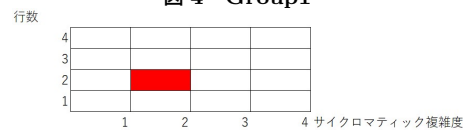


図5 Group2

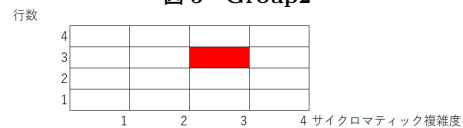


図6 Group3

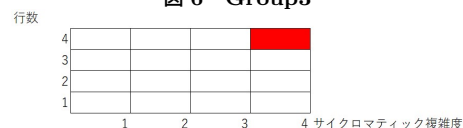


図7 Group4

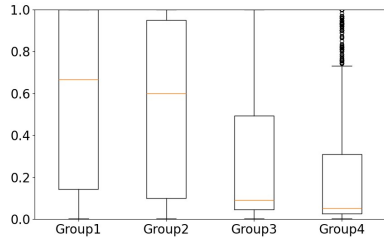


図 8 グループ別の Ochiai の疑惑値 1 位の割合

比較的大規模な Group3 及び Group4 では約 0.1 にとどまっており、明確な差が見られる。以上の結果から、プログラムの規模が小さく単純な構造であるほど、同順位問題が発生しやすい傾向があると考えられる。

以上より、RQ1 に対しては次の通り回答する。

**RQ1: プログラミング初心者が作成するような小規模プログラムを対象とした場合、代表的な SBFL 手法によるバグ限局では、同順位問題がどの程度発生するか？**

初心者が作成することの多い小規模で比較的単純なプログラムを対象とした場合、Ochiai を使った SBFL では疑惑値 1 位の割合が平均で 0.449、中央値でも 0.25 であり、1.0 に近いケースも多く確認された。さらに、プログラムの規模と複雑さに応じたグループ分けを行ったところ、その中でも特に小規模なプログラムほど疑惑値 1 位の割合が高くなる傾向が明らかとなった。したがって、小規模なプログラムほど同順位問題が発生しやすく、代表的な SBFL 手法だけではバグ限局はより難しいと思われる。

#### 4.5.2 RQ2 (ベイズ更新を用いた提案手法は有効か?)

表 5 に示したように、ベイズ更新を用いた提案手法での EXAM スコアの平均は 0.197 であるのに対して、SBFL の EXAM スコアは 0.490 と 2 倍以上の値になっていた。このことから、ベイズ更新を活用したバグ限局は SBFL によるバグ限局に比べて半分以下の労力でバグを発見できる可能性があることを示している。また、図 2 と図 3 に示した分布の違いを見比べても、ベイズ更新を用いた提案手法の方が安定してバグ位置を上位で推定できていたことが確認できた。

ベイズ更新を用いた提案手法が、SBFL の課題である同順位問題にどの程度対処できているのかについても調べるため、各誤り (修正前) プログラムにおける疑惑値 1 位の割合を算出したところ図 9 に示す分布が得られた。結果として、疑惑値 1 位の割合の平均値は 0.071、中央値は 0.059 であった。最大でも 0.449 にとどまっており、8 割以上が 0.1 以下という結果になった。Ochiai を用いた SBFL と比較して平均は 0.379、中央値は 0.191 程度低くなっているため、ベイズ更新を用いた提案手法は同順位問題の発生を大幅に抑制できていると考えられる。

さらに、RQ1 と同様に、プログラムの規模・複雑さによってベイズ更新と SBFL での EXAM スコアに差異が見られるかも検討した。図 10 にベイズ更新の場合の、図 11 に SBFL の場合での EXAM スコアのグループごとの箱ひげ図をそれぞれ示す。

図 10 及び図 11 より、いずれのグループでもベイズ更新の EXAM スコアの方が SBFL のスコアよりも良い数値であると分かる。このことから、ベイズ更新を活用した提案手法は、本実験のデータセットにおいて有効性が高いことを確認できた。

以上より、RQ2 に対しては次の通り回答する。

**RQ2: ベイズ更新を用いた提案手法によるバグ限局は、プログラミング初心者が作成するような小規模プログラムに対して有効か？**

ベイズ更新を用いた本手法は、SBFL に比べて高精度かつ安定したバグ限局を実現しており、プログラミング初心者が作成するような小規模プログラムに対

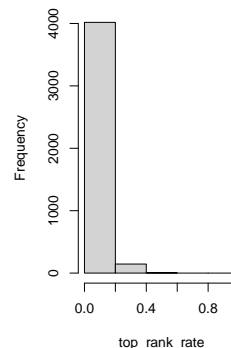


図 9 提案手法での疑惑値 1 位の割合の分布

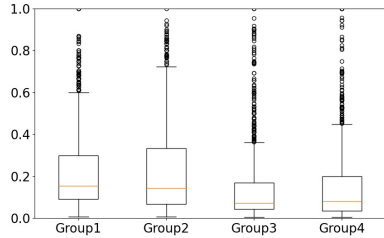


図 10 グループごとの EXAM スコア (バイズ更新)

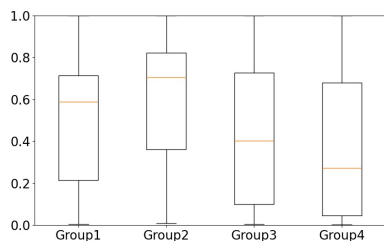


図 11 グループごとの EXAM スコア (SBFL)

しても有効である。

## 5 まとめと今後の課題

本論文では、初心者が作成するような小規模で比較的単純なプログラムに着目し、それに対するバグ限局手法を新たに提案した。従来から広く知られている SBFL 手法は、本論文で注目している小規模プログラムに対しては同順位問題が起りやすく、結果としてバグ限局精度が低くなってしまうことが分かった。これに対し、本論文では SBFL とは異なるアプローチとして、バイズ更新を用いたバグ限局手法を提案し、実験によりその有効性を示すことができた。ただし、この結果はあくまでも初心者向けのプログラミングコンテストで作成されたプログラムを対象とした場合の話であり、提案手法はそのような限定的な条件下では既存の SBFL よりも優れていたということであって、どのようなプログラムに対しても優れているという主張ではない点に注意されたい。

提案手法はテストの結果を活用できていないため、

より汎用性を持たせるためには SBFL との統合が現実的であると考えられる。その統合方法について検討していくのが我々の今後の重要な課題である。

**謝辞** 本研究の一部は JSPS 科研費 23K11382, 25K15059, 25K15062 の助成を受けたものです。

## 参考文献

- [1] Abreu, R., Zoetevej, P., Golsteijn, R., and van Gemund, A. J. C.: A practical evaluation of spectrum-based fault localization, *J. Syst. Softw.*, Vol. 82, No. 11(2009), pp. 1780–1792.
- [2] Abreu, R., Zoetevej, P., and van Gemund, A. J.: On the Accuracy of Spectrum-based Fault Localization, *Testing: Academic & Industrial Conf. Practice & Research Techniques - MUTATION*, September 2007, pp. 89–98.
- [3] AtCoder Inc.: AtCoder: 競技プログラミングコンテストを開催する国内最大のサイト, <https://atcoder.jp/?lang=ja>.
- [4] Campos, J., Riboira, A., Perez, A., and Abreu, R.: GZoltar: an eclipse plug-in for testing and debugging, *Proc. 27th IEEE/ACM Int. Conf. Autom. Softw. Eng.*, September 2012, pp. 378–381.
- [5] Goues, C. L., Pradel, M., and Roychoudhury, A.: Automated program repair, *Commun. ACM*, Vol. 62, No. 12(2019), pp. 56–65.
- [6] JavaParser.org: JavaParser, <https://javaparser.org/>.
- [7] Jones, J. A. and Harrold, M. J.: Empirical Evaluation of the Tarantula Automatic Fault Localization Technique, *Proc. 20th IEEE/ACM Int'l Conf. Auto. Softw. Eng.*, November 2005, pp. 273–282.
- [8] Li, Z., Wu, S., Liu, Y., Shen, J., Wu, Y., Zhang, Z., and Chen, X.: VsusFL: Variable-suspiciousness-based Fault Localization for novice programs, *J. Syst. Softw.*, Vol. 205(2023), pp. 111822.
- [9] 文部科学省: プログラミング教育, [https://www.mext.go.jp/a\\_menu/shotou/zyouhou/detail/1375607.htm](https://www.mext.go.jp/a_menu/shotou/zyouhou/detail/1375607.htm), 2020.
- [10] PR TIMES Corporation: 日本最大のプログラミングコンテストサイト AtCoder 全世界での登録者数が 50 万人を突破!, <https://prtimes.jp/main/html/rd/p/000000038.000028415.html>, May 2023.
- [11] Sarhan, Q. I. and Beszédés, A.: A Survey of Challenges in Spectrum-Based Software Fault Localization, *IEEE Access*, Vol. 10(2022), pp. 10618–10639.
- [12] 豊田秀樹: 基礎からのバイズ統計学, 朝倉書店, 東京, 2015.