社団法人　電子情報通信学会
THE INSTITUTE OF ELECTRONICS,
INFORMATION AND COMMUNICATION ENGINEERS

信学技報
TECHNICAL REPORT OF IEICE.

実行経路の近似度を用いたテストケースの重み付けによる SBFL の精度向上

吉岡　　遼†　肥後　芳樹†　楠本　真二†

† 大阪大学大学院情報科学研究科
E-mail: †{h-yosiok,higo,kusumoto}@ist.osaka-u.ac.jp

あらまし　ソフトウェア開発において，デバッグ作業は多大なコストを要する．計算機によるデバッグ作業の支援によって，デバッグに要するコストの削減が期待される．デバッグ作業を支援する技術の 1 つに，自動欠陥限局と呼ばれる技術が存在する．近年盛んに研究されている自動欠陥限局の手法の 1 つに，Spectrum-Based Fault Localization(以降，SBFL) と呼ばれる手法が存在する．テストケースの実行経路から欠陥を含む行を予測する技術である．既存の SBFL 手法ではテストケースに重要度という概念がなく全てのテストケースが等しく扱われているが，テストケースごとの重要度は異なると著者らは考えた．先行研究にて，実行経路の類似度に基づき重み付けを行う研究が行われていた．しかし，単に実行経路の類似度を比較するのみでは，精度があまり向上しないことが報告されている．そのため，提案手法では，ブロック化という手順を踏んだ後に類似度を比較する．ブロック化とは，ある行を実行するテストケースの集合が，その前後の行を実行するテストケースの集合と一致する場合に，それらの行を 1 つのブロックにまとめる処理である．ブロック化を行なった状態での重み付けにより，SBFL と差異が生じた欠陥を含む行の順位が最大 17.05%向上することを確認した．

キーワード　Fault Localization,　SBFL,　自動欠陥限局

1. まえがき

ソフトウェア開発において，デバッグ作業は多大なコストを要する．ソフトウェア開発コストのうち，過半数をデバッグ作業が占めるという報告もなされている [1] [2]．計算機によるデバッグの支援により，コストの削減が期待される．

デバッグ作業を支援する技術の 1 つに，自動欠陥限局と呼ばれる技術がある．自動欠陥限局は，何らかの手段を用いてプログラムに潜む欠陥の位置を予測する技術である．これまでに多くの手法が提案されているが，近年盛んに研究されている手法の 1 つに SBFL と呼ばれる手法がある．SBFL は，テストケースの実行経路から欠陥の位置を予測する技術である．失敗したテストケースで実行した行は，欠陥を含む行である可能性が高く，成功したテストケースで実行した行は低いというアイデアに基づき推定を行う．

既存の SBFL ではテストケースの重要度という概念がなく全てのテストケースが等しく扱われる．しかし，テストケースごとに重要度は異なり，その重要度に基づく重み付けにより精度を向上できると著者らは考えた．

本研究では疑惑値が等しい場合，テストケースで実行された順序を疑惑値の順序付けに用いた．これは，開発者がデバッグ作業を行う場合は，テストケースで実行された経路に従ってプログラムを確認していくからである．

テストケースへの重み付けにより精度が向上する例をあげる．図 1 では，テストケース t_c と実行経路が全く同一のテストケース t'_c を新たに追加することにより精度が向上する．t_a，t_b，t_c はそれぞれ，n の初期値として 1，-1，2 を与えるテストケースである．9 行目が欠陥を含む行である．追加前の 9 行目の順位は 5 位だが，t'_c を追加することで順位が 1 位に向上する．t'_c の追加は，t_c の重みとして 2 を付けたと解釈できる．このことから，失敗したテストケースにより近い実行経路を持つ成功したテストケースに，より大きな重みを付けることで精度を向上できるのではないかと考えた．

提案手法のキーアイデアは，「失敗したテストケースの実行経路により近い実行経路を持つ成功したテストケースは重要度が高いとして，より大きな重みを付ける」である．しかし，既存研究 [3] により，単に実行経路の類似度から重み付けを行うだけでは，精度がそれほど向上しないことが報告されている．

実行経路の類似度から重みを求めることの問題点を述べる．問題点とは，プログラムの表現が多岐にわたるという点である．例えば，図 1 において，2-5 行目の書き方は，図に示したように 4 行で書く方法もあれば，$n = (n-5) * 3/6 + 1$; と 1 行で書く方法もある．意味的な側面で見た場合には同じ表現であるが，2 つの表現では行数が異なる．よって，失敗したテストケースの実行経路により近い成功したテストケースにはより大きな重

z

追加前

		t_a	t_b	t_c	疑惑値	順位
1	`if(n > 0){`	●	●	●	0.577	6
2	`  n -= 5;`	●		●	0.707	1
3	`  n *= 3;`	●		●	0.707	2
4	`  n /= 6;`	●		●	0.707	3
5	`  n += 1;`	●		●	0.707	4
6	`if(n < 0){`	●	●	●	0.577	7
7	`  n += 1;`	●	●	●	0.577	8
8	`if(n == 0){`	●	●	●	0.577	9
9	`  n += 2;`	●	●		0.707	5
10	`return n;`	●	●	●	0.577	10
		✖	○	○		

追加後

		t_a	t_b	t_c	t_c'	疑惑値	順位
1	`if(n > 0){`	●	●	●	●	0.500	6
2	`  n -= 5;`	●		●	●	0.577	2
3	`  n *= 3;`	●		●	●	0.577	3
4	`  n /= 6;`	●		●	●	0.577	4
5	`  n += 1;`	●		●	●	0.577	5
6	`if(n < 0){`	●	●	●	●	0.500	7
7	`  n += 1;`	●	●	●	●	0.500	8
8	`if(n == 0){`	●	●	●	●	0.500	9
9	`  n += 2;`	●	●			0.707	1
10	`return n;`	●	●	●	●	0.500	10
		✖	○	○	○		

図 1　同一のテストケースの追加により精度が上がる例

みを付ける方法では，意味的な表現は同じであるにも関わらず異なる重みがついてしまうこととなり問題となる．

そこで，ブロック化という処理を行った後に類似度を比較する手法を提案する．ブロック化とは，ある行を実行するテストケースの集合が，その前もしくは後ろの行を実行するテストケースの集合と一致する場合に，それらの行を 1 つのブロックにまとめる処理である．この処理により，図 1 における 2-5 行目の内容は，その表現の方法によらず 1 つのブロックにまとめられることとなる．ブロックにまとめた後の実行経路の一致度を比較することで，表現の方法によらず同一の重みを付けることができる．

評価実験として，従来の SBFL と提案手法を OSS に適用し，SBFL と提案手法との間で差異が生じた欠陥を含む行の順位がそれぞれ疑惑値が付いた行の上位何パーセントに入るかの比較を行なった．その結果，差異が生じた欠陥を含む行において，欠陥を含む行の順位が上位に入るパーセンテージが最大で 17.05%向上したことを確認した．

2. 準　　備

SBFL では，対象のプログラムをテストスイートに通し，テストケースの成功/失敗情報と実行経路から，疑惑値を算出する．疑惑値は，該当する行がどの程度欠陥を含んでいる可能性が高いかを示す指標であり，0 から 1 までの値をとる．SBFL は，失敗したテストケースで実行した行は，成功したテストケースで実行した行よりも欠陥を含む可能性が高いという考えに基づく．

具体的にどのように疑惑値を算出するか説明する．成功したテストケースの集合を $\mathbb{P}$，失敗したテストケースの集合を $\mathbb{F}$，テストケース全体の集合を $\mathbb{T}$ と定義する．テストケース t_i と，プログラム中の行 s に対して，関数 $fail_i(s)$，$pass_i(s)$ を次のように定義する．

$$fail_i(s) = \begin{cases} 1 & (t_i \in \mathbb{F} \wedge t_i \text{が } s \text{ を実行する}) \\ 0 & (t_i \in \mathbb{P} \vee t_i \text{が } s \text{ を実行しない}) \end{cases}$$

$$pass_i(s) = \begin{cases} 1 & (t_i \in \mathbb{P} \wedge t_i \text{が } s \text{ を実行する}) \\ 0 & (t_i \in \mathbb{F} \vee t_i \text{が } s \text{ を実行しない}) \end{cases}$$

次に，$fail_i(s)$，$pass_i(s)$ を用いて，関数 $fail(s)$，$pass(s)$ を次のように定義する．

$$fail(s) = \sum_{t_i \in \mathbb{T}} fail_i(s)$$

$$pass(s) = \sum_{t_i \in \mathbb{T}} pass_i(s)$$

すなわち，$fail(s)$ は行 s を実行して失敗したテストケースの個数を，$pass(s)$ は行 s を実行して成功したテストケースの個数を表す．

これらを用いて，各行の疑惑値を計算する．疑惑値を算出する式はこれまでに数多く提案されている．Abreu らは，7 つの疑惑値の計算手法を比較し，Ochiai [4] が優れた手法であると結論付けている [5]．そのため，本研究では疑惑値の算出に Ochiai を利用する．Ochiai の式は以下の通りである．|S|は集合 S の要素数を表す．ここでは|F|は失敗したテストケースの総数を表す．

$$suspicious(s) = \frac{fail(s)}{\sqrt{|\mathbb{F}| * (fail(s) + pass(s))}} \qquad (1)$$

3. 提 案 手 法

本研究では，SBFL の精度を向上させるため，テストケースに重みを持たせるという手法を提案する．実行経路の情報から，テストケースに重みを計算し，その重みを用いて疑惑値を算出する．重み付けは，失敗したテストケースの実行経路により近

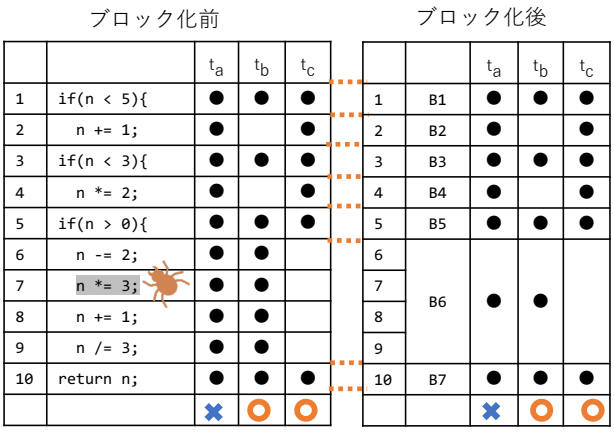


図 2　ブロック化の例

い実行経路を持つ成功したテストケースは重要度が高いというアイデアに基づく．

しかし，単に実行経路が近いテストケースに高い重みを付けるという処理では，精度が十分に上昇しないことが先行研究 [6] により報告されている．著者らは，欠陥を含む行と必ず併せて実行されるその周辺にある行の存在がノイズとなり予測に悪影響を与えているのではないかと考えた．このノイズは，ブロック化を行なった後に実行経路を比較することで解消できるのではないかという仮説を立てた．ブロック化とは，ある行を実行するテストケースの集合が，その前あるいは後ろの行を実行するテストケースの集合と一致する場合，それらの行を 1 つのブロックとして集約する処理である．

本研究ではこの仮説のもと，ブロック化した実行経路情報から実行経路の近似度を求め，テストケースに重み付けを行う．本提案手法を，以降では Blocknized Spectrum-Based weighting Fault Localization(BSBFL) と呼ぶ．

BSBFL は，以下の 3 ステップで構成される．

ステップ 1.
取得した実行経路のブロック化

ステップ 2.
ブロック化した実行経路の近似度から成功したテストケースの重みを計算

ステップ 3.
成功したテストケースへの重み付けを行なった状態での疑惑値の算出

各ステップについて詳しく説明する．

3.1　ステップ 1: 取得した実行経路のブロック化

ブロック化を行う方法を述べる．ある行を実行するテストケースの集合が，その前あるいは後ろの行を実行するテストケースの集合と一致する場合，それらの行を同一のブロックに含める．

図 2 がブロック化のイメージ図である．例えば，図 2 において，6-9 行目を実行するテストケースの集合は同じであるため，同一のブロックに集約する．

3.2　ステップ 2: ブロック化した実行経路の近似度から成功したテストケースの重みを計算

まず，次のように $Exec(i)$，$NotExec(i)$ を定義する．

$Exec(i)$
テストケース t_i で実行するプログラム要素の集合

$NotExec(i)$
テストケース t_i で実行されないプログラム要素の集合

定義中で使用しているプログラム要素とは，BSBFL においてはステップ 1 で述べたブロックである．

失敗したテストケース t_j と成功したテストケース t_i に対し，新たに $numNotExec(j,i)$，$numExec(j,i)$ を定義する．

$$numExec(j,i) = |Exec(j) \wedge Exec(i)|$$
$$numNotExec(j,i) = |Exec(j) \wedge NotExec(i)|$$

例として図 2 において numNotExec(a,b) を求める．$Exec(a) = \{B1, B2, B3, B4, B5, B6, B7\}$，$NotExec(b) = \{B2, B4\}$ であるから，$Exec(a) \wedge NotExec(b) = \{B2, B4\}$ となる．従って，$numNotExec(a,b) = 2$ となる．

これらの数値を用いて，成功したテストケース t_i の重み $w(i)$ を求める．失敗したテストケース t_j に対する，成功したテストケース t_i の重みを $w(j,i)$ とする．t_j，t_i の実行経路の一致度を $x = numExec(j,i)/(numExec(j,i) + numNotExec(j,i))$ とする．$w(j,i)$ は，次のように定義され，x の値が $thsld$ 未満の場合には 1 を，それ以上の場合には 1 よりも大きい値をとる．

$$w(j,i)) = \begin{cases} 1 & (0 \leqq x < thsld) \\ \dfrac{numExec(j,i)}{\sqrt{numExec(j,i) + numNotExec(j,i)}} & (thsld \leqq x \leqq 1) \end{cases} \tag{2}$$

$w(j,i)$ の平均をとった値がテストケース t_i の重み $w(i)$ である．すなわち，

$$w(i) = \frac{\sum_{t_j \in \mathbb{F}} w(j,i)}{|\mathbb{F}|}$$

例として，図 2 の場合における $w(c)$ を求める．まずは $w(a,c)$ を求める．閾値 $thsld$ は 0.8 であるとする．$numExec(a,c) = 6$，$numNotExec(a,c) = 1$ であるため，t_a，t_b の一致度は $x = \dfrac{6}{1+6} \fallingdotseq 0.86$ となる．閾値以上の一致度であるため，$w(a,c)$ は $\dfrac{6}{\sqrt{6+1}} \fallingdotseq 2.27$ となる．$\mathbb{F} = \{t_a\}$ であるため，$w(c) = \dfrac{w(a,c)}{|\mathbb{F}|} = 2.27$ となる．同様に計算をして，$w(b) = 1$ となる．

3.3　ステップ 3: 成功したテストケースに重み付けを行なった状態での疑惑値の算出

まず，$pass'(s)$ を次のように定義する．

$$pass'(s) = \sum_{t_i \in \mathbb{T}} pass_i(s) * w(i)$$

Ochiai の式 (1) において $pass(s)$ を，$pass'(s)$ で置き換えた式が提案手法における疑惑値を算出する式である．

図 2 において，3.2 項で求めた $w(b)$，$w(c)$ を用いて疑惑値を算出する．例として，1 行目の疑惑値を求める．

$$fail(1) = 1$$
$$pass'(1) = w(b) + w(c) = 2.27 + 1 = 3.27$$

であるため，1 行目の疑惑値次のようになる．

$$\frac{fail(1)}{\sqrt{|\mathbb{F}| * (fail(1) + pass'(1))}} = \frac{1}{\sqrt{1 * (1 + 3.27)}} = 0.483$$

同様に，全ての行の疑惑値を求めると表 1 のようになる．SBFL では欠陥を含む行である 7 行目の順位が 4 位であるが，成功したテストケースへの重み付けを行なった BSBFL では 2 位であり，精度が向上していることがわかる．

4. 実　　験

4.1 実験概要

以下の内容を確認することを目的に実験を行なった．

- SBFL と BSBFL 間に生じた精度の差異
- ブロック化を行う場合と行わない場合での重み付けによる結果の差異

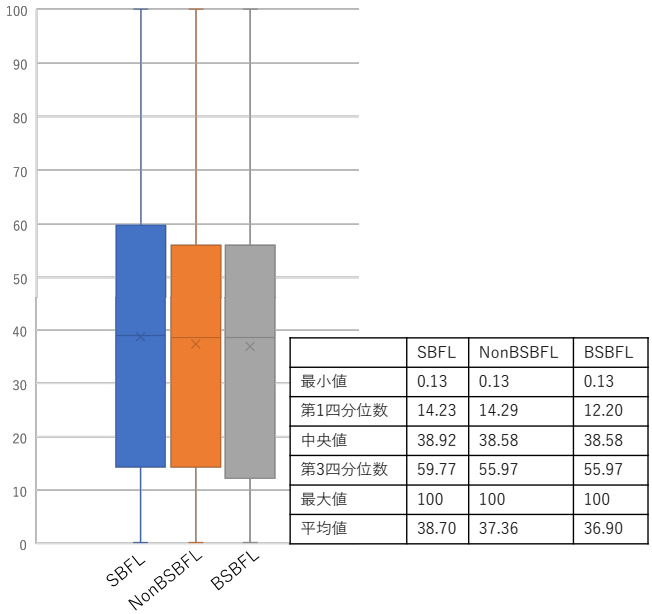
SBFL と BSBFL を OSS に適用し，それぞれの手法において欠陥を含む行が，疑惑値が付いた行全体の上位何パーセントに含まれるかという観点で比較を行う．

また，ブロック化を行う場合と行わない場合での重み付けにより精度にどの程度の差異が生じるかを確認するために，ブロック化を行わない状態で成功したテストケースへの重み付けを行う手法と BSBFL の比較を行う．以降，ブロック化を行わない状態で成功したテストケースへの重み付けを行なった BSBFL を，Non-Blocknized Spectrum-Based weighting Fault Localization(NonBSBFL) と呼ぶ．

BSBFL では，$Exec(i)$，$NotExec(i)$ の定義におけるプログラム要素をブロックとした．NonBSBFL では，このプログラム要素をプログラムの行であるとする．疑惑値の算出方法は

表 1　成功したテストケースへの重み付けによる疑惑値の変化

行	SBFL の疑惑値	順位	BSBFL の疑惑値	順位
1	0.577	7	0.483	7
2	0.707	1	0.553	5
3	0.577	8	0.483	8
4	0.707	2	0.553	6
5	0.577	9	0.483	9
6	0.707	3	0.707	1
7	0.707	4	0.707	2
8	0.707	5	0.707	3
9	0.707	6	0.707	4
10	0.577	10	0.483	10



	SBFL	NonBSBFL	BSBFL
最小値	0.13	0.13	0.13
第1四分位数	14.23	14.29	12.20
中央値	38.92	38.58	38.58
第3四分位数	59.77	55.97	55.97
最大値	100	100	100
平均値	38.70	37.36	36.90

図 3　各手法における精度を表した箱ひげ図

BSBFL と同様である．

4.2 実験対象

本実験では，Defects4j [7] の Math に含まれる 100 個の欠陥を実験対象とする．Math を利用する理由は，本プロジェクトが多くの自動欠陥修正や自動欠陥限局のベンチマークとして利用されているためである．100 個の欠陥には，309 行の欠陥を含む行が存在する．このうち，SBFL で検出できる欠陥を含む行の総数は 263 行である．失敗したテストケースで通過しない場所に欠陥を含む行がある場合，疑惑値が付かず SBFL では検出できない．

4.3 実験設定

実行経路情報は自動欠陥修正ツールである kGenProg [8] から取得する．kGenProg はプラグインとして JaCoCo [9] を呼び出し，実行経路情報を取得している．

SBFL では，Ochiai の式 (1) を利用する．Ochiai の式が最も精度が高いと評されている [5] ためである．

4.4 評価指標

本実験では，欠陥を含む行の疑惑値の順位が，疑惑値が付いた行全体の上位何パーセントかという観点で評価を行う．欠陥を含む行がより上位に入っているほど精度が高い手法である．

疑惑値が等しい場合，テストケースで実行された順序を疑惑値の順序付けに用いた．これは，開発者がデバッグ作業を行う場合は，テストケースで実行された経路に従ってプログラムを確認していくからである．

SBFL，BSBFL，NonBSBFL において差異が生じた欠陥を含む行が，平均で上位何パーセントに含まれるか比較する．

4.5 結果と考察

各手法における精度の変化

SBFL，NonBSBFL，BSBFL において欠陥を含む行が上位何パーセントに含まれるかを図 3 の箱ひげ図に示す．既存手法と比べ，提案手法が第 1 四分位数，中央値，第 3 四分位数，

平均値で優れていることがわかる．しかし，数値の変化は非常に小さく，視認性が低い．変化が非常に小さい理由は，SBFL，NonBSBFL，BSBFL において差異が生じた欠陥を含む行数が 54 行と少ないためである．そこで，以降においては順位の変動がわかりやすいように各手法で順位に差異が生じた欠陥を含む行のみを取り上げ比較を行う．

SBFL と BSBFL を比較し，どの程度精度が向上したか

本実験において，重み付け (2) の閾値は，0.6-0.95 の区間を 0.05 区切りで取る．各閾値における SBFL と BSBFL 間で BSBFL の方が順位が高い欠陥を含む行の行数，SBFL の方が順位が高い欠陥を含む行の行数を表 2 に記す．また，それぞれの手法において，欠陥を含む行が上位何パーセントに入るかを表 3 に記す．

BSBFL では，表 2 に示した通り，0.85 を除く閾値において，順位が向上した欠陥を含む行の行数よりも，順位が低下した行数の方が多い．しかし，表 3 からわかる通り，閾値 0.75，0.8，0.85 では BSBFL の方が精度が高い．このことから，BSBFL は精度が低下する欠陥を含む行こそ多いが，欠陥を含む比較的少数の行で劇的に順位を向上させる傾向にあると考えられる．

このような傾向が見られることに関しての考察を行う．本実験では，類似度が高いテストケースに対してのみ，大きな重みを付ける．すると，大きな重みの付いた成功したテストケースで実行されず，失敗したテストケースで実行される行の疑惑値の順位が飛躍的に上昇する．この部分に欠陥を含む行が存在する場合，欠陥を含む行の順位の上がり幅は大きくなる．対して，大きな重みの付いた成功したテストケースで実行され，なおかつ失敗したテストケースでも実行される部分に欠陥を含む行が存在する場合，順位は低下．失敗したテストケースで実行され，成功したテストケースで実行されない行数分のみの順位が低下する．成功したテストケースと失敗したテストケースのブロックした後の一致度が閾値を超える成功したテストケースにのみ重みをつけているため，失敗したテストケースで実行され成功したテストケースで実行されない行の行数は少ないと考えられる．従って，順位の下り幅は小さくなる．

BSBFL の精度が最も高い閾値 0.8 での SBFL と BSBFL の精度を比較したグラフを図 4 に記す．前述の通り，順位が低下する部分の下り幅は比較的小さく，順位が上昇する部分での上がり幅は大きい傾向にあるとわかる．

表 2　SBFL-BSBFL 間で結果に差異が生じた欠陥を含む行の行数

閾値	BSBFL が優れていた個数	SBFL が優れていた個数
0.6	22	77
0.65	21	68
0.7	25	62
0.75	21	52
0.8	12	14
0.85	12	10
0.9	1	2
0.95	1	1

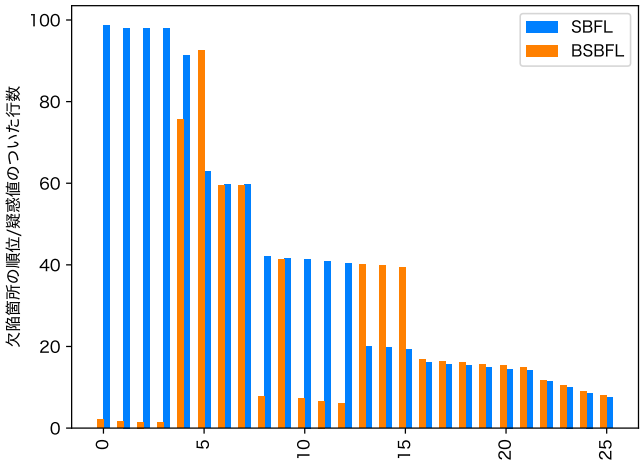


図 4　SBFL と BSBFL の順位の比較 thsld = 0.8

ブロック化を行う場合と行わない場合での重み付けにより結果にどの程度の差異が生じるか

BSBFL と NonBSBFL の精度を比較することで，ブロック化の有無によりどの程度結果に差異が生じるか確かめる．各手法で最も精度が高い閾値が異なるため，2 つの手法を用いるにあたり，それぞれの手法の精度が最も高くなる閾値を用いて比較する．

表 4 は，SBFL，BSBFL，NonBSBFL の各手法において欠陥を含む行が上位何パーセントに入るかをまとめた表である．この表から，BSBFL では閾値が 0.8 の時に精度が最も高く，NonBSBFL では閾値が 0.95 の時に精度が最も高いことがわかる．

これらの閾値において 2 つの手法での精度を比較したグラフ

表 3　SBFL と BSBFL 間で順位に差異が生じた欠陥を含む行の疑惑値が上位何パーセントに入るかの比較

閾値	SBFL の平均 (%)	BSBFL の平均 (%)
0.6	41.47	43.21
0.65	44.45	45.72
0.7	45.09	45.33
0.75	43.93	40.09
0.8	40.74	23.69
0.85	44.95	29.35
0.9	53.87	58.71
0.95	77.05	77.34

表 4　SBFL，BSBFL，NonBSBFL 間で順位に差異が生じた欠陥を含む行の疑惑値が上位何パーセントに入るかの比較

閾値	SBFL(%)	NonBSBFL(%)	BSBFL(%)
0.6	38.88	48.73	40.32
0.65	39.05	48.62	40.02
0.7	39.17	39.39	39.35
0.75	41.57	39.24	38.72
0.8	41.6	34.97	34.68
0.85	37.47	40.76	35.15
0.9	38.11	41.58	38.22
0.95	38.47	32.55	38.48

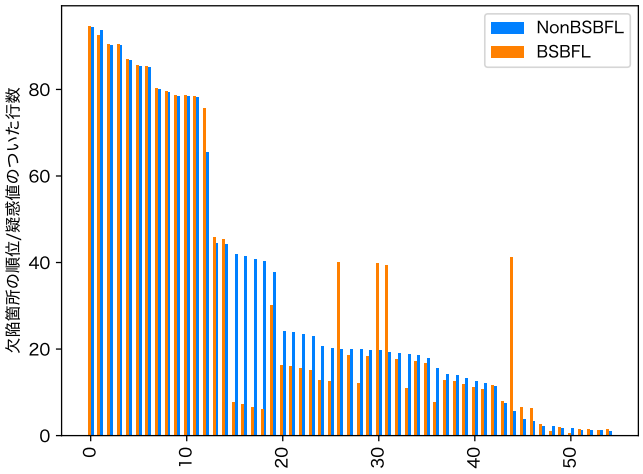


図 5　BSBFL と NonBSBFL の精度比較

が，図 5 である．NonBSBFL に対して BSBFL の方が精度が高く出る箇所が多いことがわかる．BSBFL と NonBSBFL 間で差異が生じた欠陥を含む行の順位が，BSBFL では平均で上位 31.28%に入っており，NonBSBFL では平均で上位 33.30%であった．ブロック化を行うことで，ブロック化を行わなかった場合と比較して精度は 2.02%向上することが確認される．

また，表 4 において，BSBFL では閾値 0.8 前後で SBFL よりも精度が上昇することが確認される．しかし，NonBSBFL では特定の閾値の近傍で SBFL よりも精度が向上する特徴は確認されない．NonBSBFL において，最も精度が高い閾値は 0.95 であるが，その前の閾値 0.85 と 0.9 では SBFL よりも精度が低下している．NonBSBFL では特定の閾値の近傍において精度が向上する特徴が見られないと考えられる．BSBFL にのみ特定の閾値の近傍において精度が向上する傾向があると仮定すると，他のプロジェクトを対象に欠陥限局をした場合にも閾値 0.8 の近傍で精度が上がることが期待される．

これらの結果から，精度を高める上でブロック化は有効であると著者らは考える．

5. 妥当性の脅威

5.1 疑惑値の計算

本実験における SBFL，BSBFL，NonBSBFL の疑惑値を算出する式として，精度が高いと報告されている [5]Ochiai の式 (1) を利用した．その他の式を利用した場合には，異なる結果が得られる可能性がある．

5.2 実験対象

本実験では，Defects4j [7] に含まれる Math を実験対象とした．他のプロジェクトを対象に実験を行なった場合，異なる結果が得られる可能性がある．

5.3 重み付け

成功したテストケースへの重み付けにおいて，本実験で使用した以外の重み付けを使用した場合，異なる結果が得られる可能性がある．

6. あとがき

本研究では，SBFL の精度を高めるためにブロック化を行い成功したテストケースに重みを持たせる手法を提案した．既存手法の SBFL と提案手法の精度を比較し，既存手法よりも精度が高いことを示した．また，ブロック化の有無により重み付けを行った時の精度の変化を確認し，ブロック化を行うことで精度が上昇することを確認した．以上の結果から，提案手法が有効である可能性を示した．

謝辞　本研究は JSPS 科研費 JP20H04166 および JP21K18302 の助成を得て行われた．

文　献

[1] B. Hailpern and P. Santhanam, “Software debugging, testing, and verification,” IBM Systems Journal, vol.41, no.1, pp.4–12, 2002.

[2] T. Britton, L. Jeng, G. Carver, and P. Cheak, “Quantify the time and cost saved using reversible debuggers,” Technical report, Cambridge Judge Business School, 2012.

[3] A. Bandyopadhyay and S. Ghosh, “Proximity based weighting of test cases to improve spectrum based fault localization,” Proc. International Conference on Automated Software Engineering, pp.420–423, 2011.

[4] R. Abreu, P. Zoeteweij, and A.J. vanGemund, “On the accuracy of spectrum-based fault localization,” Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION, pp.89–98, 2007.

[5] R. Abreu, P. Zoeteweij, R. Golsteijn, and A.J.C. vanGemund, “A practical evaluation of spectrum-based fault localization,” Journal of Systems and Software, vol.82, no.11, pp.1780–1792, 2009.

[6] A. Bandyopadhyay and S. Ghosh, “Proximity based weighting of test cases to improve spectrum based fault localization,” Proc. International Conference on Automated Software Engineering, pp.420–423, 2011.

[7] R. Just, D. Jalali, and M.D. Ernst, “Defects4j: A database of existing faults to enable controlled testing studies for java programs,” Proceedings of the 2014 International Symposium on Software Testing and Analysis, pp.437–440, 2014.

[8] Y. Higo, S. Matsumoto, R. Arima, A. Tanikado, K. Naitou, J. Matsumoto, Y. Tomida, and S. Kusumoto, “kGenProg: A high-performance, high-extensibility and high-portability apr system,” Proc. Asia-Pacific Software Engineering Conference, pp.697–698, 2018.

[9] Hoffmann M.R., “Jacoco java code coverage library.,” `https://www.eclemma.org/jacoco/`. [Online; accessed 12-December-2021].