特別研究報告

題目

SBFL における実行経路の類似度に基づく テストケースへの重み付けの提案

指導教員

楠本 真二 教授

報告者

吉岡遼

令和 4 年 2 月 8 日

大阪大学 基礎工学部 情報科学科

令和 3 年度 特別研究報告

SBFL における実行経路の類似度に基づく テストケースへの重み付けの提案

吉岡遼

内容梗概

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，欠陥限局と呼ばれる技術が存在する．計算機によりプログラム中に含まれる欠陥の箇所を推定する技術である．これまでに多くの手法が提案されてきた．近年盛んに研究されている欠陥限局の手法の 1 つに，Spectrum-Based Fault Localization(以降，SBFL) と呼ばれる手法が存在する．SBFL は，テストケースの実行経路から欠陥を含む行を推定する技術である．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．しかし著者はテストケースごとの重要度は異なると考えた．本研究では，テストケースの実行経路が失敗テストケースにより近い成功テストケースに大きな重みを付与する手法を提案する．テストケースの成否を分けた小さな違いにこそ大きな価値があると考えたためである．しかし，先行研究により実行経路の類似度を単純に比較し重み付けを行う手法ではあまり精度が向上しないことが報告されている．著者はこの原因を，連続して実行される行の存在が重み付けに悪影響を与えているためだと捉えた．提案手法では．ブロック化という処理を行いこの問題を解決した上で実行経路を比較し，重み付けを行う．実験として，既存の SBFL と提案手法を比較し，提案手法が既存の SBFL よりも高い精度で欠陥を含む行を推定できることを示した．このことから，ブロック化を行った後の実行経路の類似度に基づく重み付けが有効である可能性を示した．

主な用語

欠陥限局，重み付け，ブロック化

目次

図目次

表目次

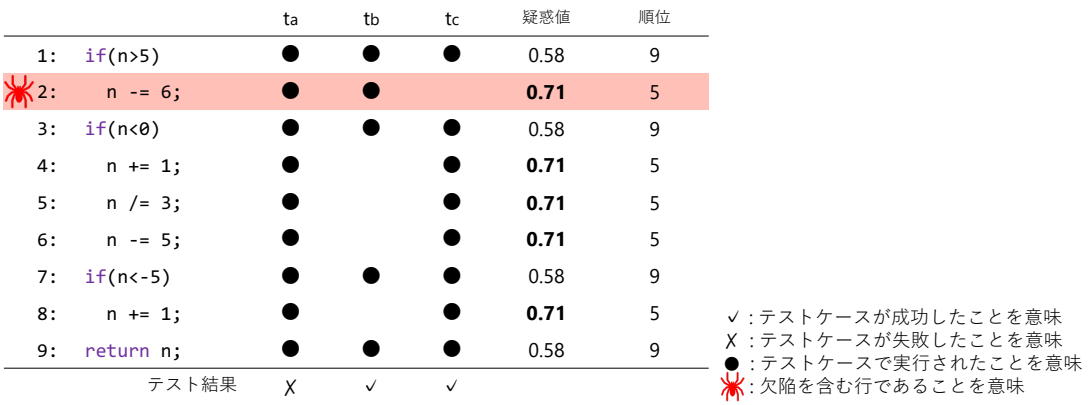

図 1: 既存の SBFL 手法

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，半分以上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，欠陥限局と呼ばれる技術がある．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推定する技術である．これまでに多くの手法が提案されている [3, 4, 5]．その中で近年最も盛んに研究されている手法が，テストケースによる実行経路の情報を用いて欠陥限局を行う Spectrum-Based Fault Localization(SBFL) である [6]．SBFL では，失敗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．SBFL は，入力として欠陥を含むプログラムとテストスイートを与えられると，各行ごとの疑惑値を出力する．疑惑値とは，その行が欠陥を含む可能性を表す指標であり，通常 0 から 1 の範囲で与えられる．値が大きいほど欠陥を含んでいる可能性が高いことを表す．

既存の SBFL の例を図 1 に記す．疑惑値が高い順に，順位付けする．同じ疑惑値の行が複数存在する場合，欠陥を含む行の順位はそれらの中で最も低い順位とする．例えば，欠陥を含む行の疑惑値が 1 番目に大きく，欠陥を含む行と同じ疑惑値の行が他に 2 つ存在する場合，欠陥を含む行の順位は 3 位とする．

図 1 では，欠陥を含む 2 行目と，欠陥を含まない 4，5，6，9 行目に同じ疑惑値がついている．実際に欠陥が含まれていないにも関わらず高い疑惑値が付与されることは問題である．この問題の一因は，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を等しく扱うことにあると著者は考える．提案手法では，テストケースに重要度という概念を設け，重要度に基づく重み付けをすることで，この

		ta	tb	tc × 2		疑惑値	順位
1:	`if(n>5)`	●	●	●	●	0.50	9
2:	`n -= 6;`	●	●			**0.71**	1
3:	`if(n<0){`	●	●	●	●	0.50	9
4:	`n += 1;`	●		●	●	**0.58**	5
5:	`n /= 3;`	●		●	●	**0.58**	5
6:	`n -= 5;`	●		●	●	**0.58**	5
7:	`}`					0.00	10
8:	`if(n<-5)`	●	●	●	●	0.50	9
9:	`n += 1;`	●		●	●	**0.58**	5
10:	`return n;`	●	●	●	●	0.50	9
	テスト結果	✗	✓	✓			

図 2: テストケースへの重み付けにより欠陥を含む行の順位が向上する例

問題の解消に取り組む．

提案手法は，「実行経路が失敗テストケースに近い成功テストケースは重要度が高い」というアイデアに基づき重み付けを行う．テストケースの成否を分けた小さな違いにこそ大きな価値があると著者は考えたためである．

図 2 は，テストケース t_c の重みを 2 倍にした例である．成功テストケース t_b，t_c のうち，実行経路が失敗テストケース t_a により近いのは t_c であるため，t_c の方が重要度が高いとして重みを増やした．テストケース t_c の重みを増やすことでで，欠陥を含む行の順位が 5 位から 1 位に向上している．このことから，テストケースに対し適切に重要度を定め重み付けすることにより，SBFL の精度を高められることがわかる．

評価実験として，重み付けを行わない既存の SBFL と提案手法を OSS に適用し，それぞれの手法が算出した欠陥を含む行の疑惑値がどのように変化するか確かめた．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に入るかの比較を行なった．その結果，欠陥を含む行の順位が疑惑値がついた行全体の上位に入るパーセンテージが最大で 3.86% 良くなることを確認した．

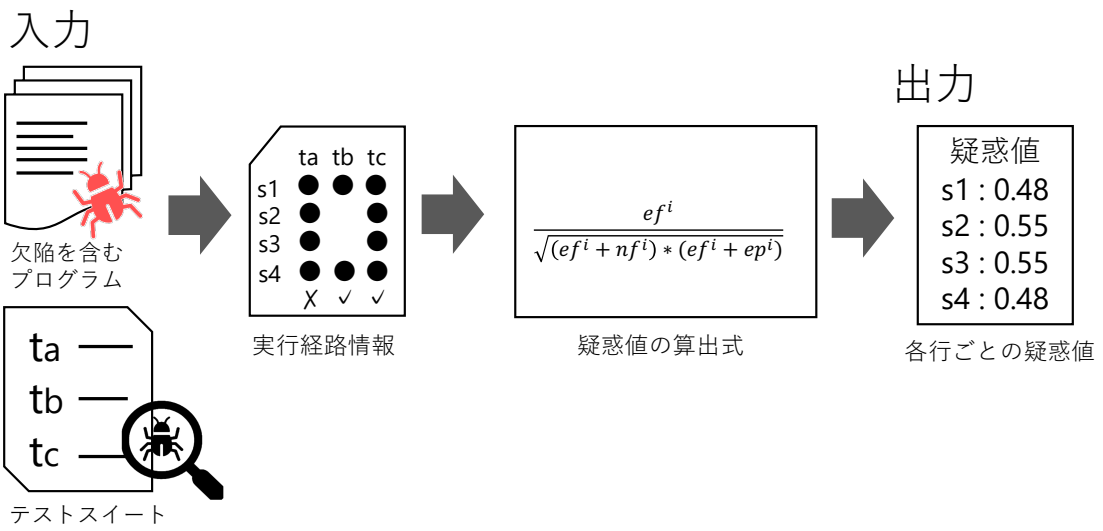

図 3: SBFL の入力から出力までの大まかな流れ

2 準備

SBFL では，入力として欠陥を含むプログラムとテストスイートを与える．対象のプログラムをテストスイートに通し，テストケースの成否情報と実行経路から，疑惑値を算出する．失敗テストケースで実行した行は，成功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き疑惑値を算出する．求めた疑惑値が，SBFL の出力である．

疑惑値の算出方法について説明する．プログラムの行 s_i に対し，ef^i，nf^i，ep^i，np^i を次のように定義する．

ef^i　行 s_i を実行する失敗テストケースの個数
nf^i　行 s_i を実行しない失敗テストケースの個数
ep^i　行 s_i を実行する成功テストケースの個数
np^i　行 s_i を実行しない成功テストケースの個数

これらを疑惑値の算出式に入れることで疑惑値が求まる．疑惑値の算出式はこれまでに多数提案されている．例として Ochiai[7] の算出式を以下に示す．

$$suspicious(s_i) = \frac{ef^i}{\sqrt{(ef^i + nf^i) * (ep^i + ef^i)}} \tag{1}$$

疑惑値を大きい順に並べた際に，欠陥を含む行の疑惑値が何番目に出てきたかを順位と呼ぶ．疑惑値の順位に関して，同じ疑惑値を持つ行が複数ある場合，欠陥を含む行の順位はそれらの中で最も低い順

位となるように評価する．例えば，欠陥を含む行の疑惑値が 1 番目に大きい値で，欠陥を含む行と同じ疑惑値の行が他に 2 つ存在する場合，欠陥を含む行の順位は 3 位として評価する．

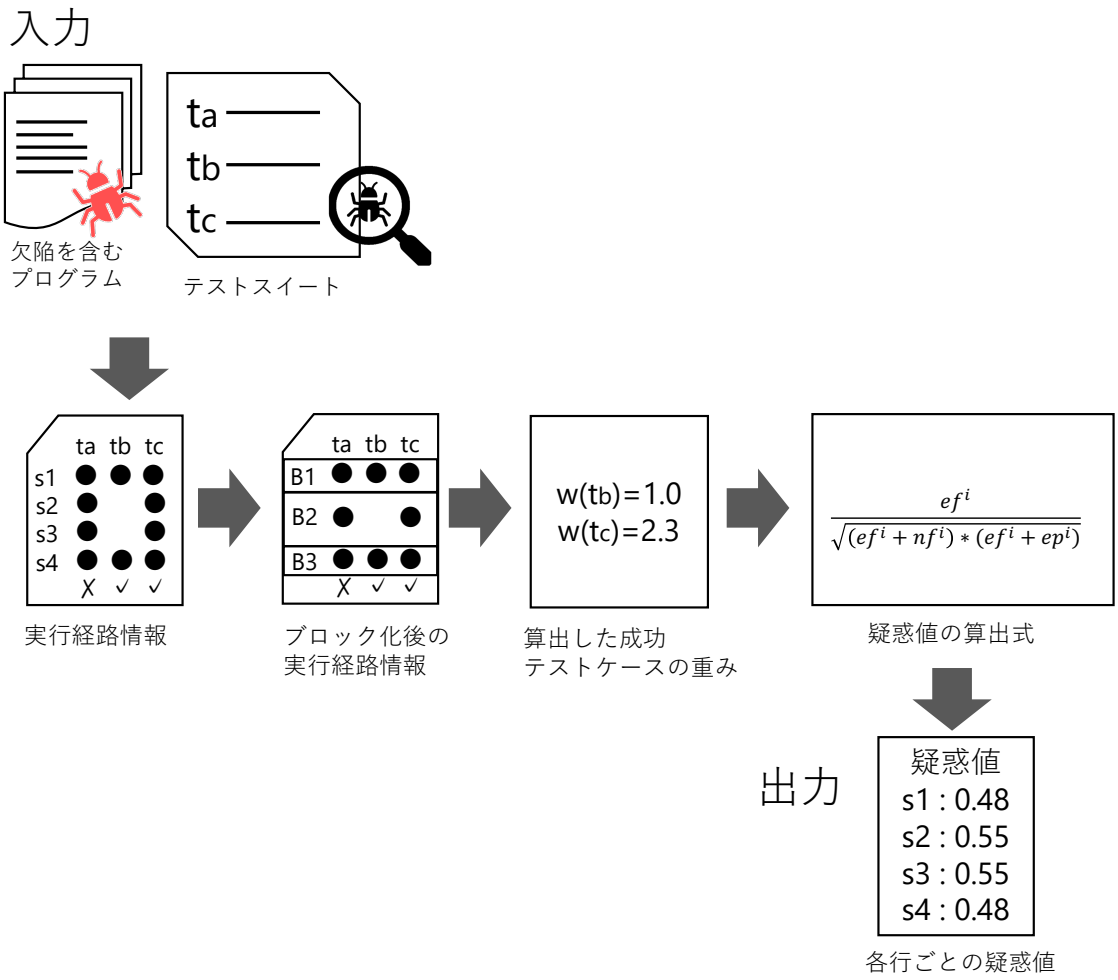

図 4: 提案手法の入力から出力までの流れ

3 提案手法

3.1 提案手法の概要

本研究では，SBFL の精度を向上させるため，成功テストケースに重みを持たせる手法を提案する．提案手法では，「実行経路が失敗テストケース近い成功テストケースは重要度が高い」というアイデアに基づき重み付けを行う．テストケースの成否を分けた小さな違いにこそ大きな価値があると考えたからである．

提案手法の大まかな流れを図 4 に記す．入力として欠陥を含むプログラムとテストスイートを与えられると，各行ごとの疑惑値を出力する．提案手法は，次の 4 ステップからなる．

ステップ 1.

欠陥を含むプログラムをテストスイートに通し，実行経路情報を取得

ステップ 2.

取得した実行経路のブロック化

ステップ 3.

ブロック化後の実行経路の類似度から成功テストケースの重みを計算

ステップ 4.

成功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

提案手法と先行研究の違いを 3.2 節で述べる．各ステップについては 3.3 節-3.6 節にかけて詳しく説明する．ブロック化の必要性を 3.7 節で言及する．3.8 節では，ブロック化の処理における考えを記す．提案手法では，ステップ 3 において類似度の算出に広く利用される Jaccard 係数という指標を用いない．その理由を 3.9 節で述べる．ブロック化という処理を行い重み付けを行う手法であることから，以降において提案手法を Blocknized Spectrum-Based weighting Fault Localization(BSBFL) と呼ぶ．

3.2 先行研究との違い

成功テストケースに重み付けを行う手法は，先行研究 [8] でも提案されている．先行研究では，失敗テストケースと成功テストケースの実行経路の類似度を Jaccard 係数 [8] から求め，類似度による重み付けを行う．しかし，この手法では精度があまり向上しないことが報告された．その原因は以下の点であると著者は考える．

- 単純に実行経路を比較するだけでは，連続して実行される分岐を含まない行の存在が特定のテストケースの類似度を高めることがある．重み付けでは，分岐を重要視すべきだと筆者は考察する．先行研究の重み付けでは，分岐よりも分岐を含まず連続して実行される行の長さに大きく左右されるため，不適切な重みを付与する可能性がある．
- 失敗テストケースの実行経路との類似度が低い成功テストケースにも重み付けを行っている．類似度が低い成功テストケースには重み付けを行わないべきである．

重み付けでは分岐を重要視するべきだとする根拠は，3.7 節にて述べる．

失敗テストケースの実行経路との類似度が低い成功テストケースに重み付けを行うべきでない理由を述べる．成功テストケースへの重み付けは，失敗テストケースのみで実行され，成功テストケースでは実行されない行が少ないほど，それらの行に欠陥が含まれる可能性が高いというアイデアに基づく．行が少ないほど，欠陥を含む行が絞り込めていると捉えられるためである．失敗テストケースと成功テストケースの類似度が低いことは，実行経路の差分が大きいことを意味する．差分が大きいということ

は，欠陥を含む可能性の高い行を絞り込めていないということである．欠陥の箇所が絞れていないにも関わらず，大きな重みを付与することはかえってノイズになってしまう．したがって，失敗テストケースとの類似度が低い成功テストケースには重み付けを行うべきでない．

本研究の提案手法は，次の点で先行研究 [8] の手法と大きく異なる．

- 分岐を含まず連続して実行される行の影響を取り除くためにブロック化という処理を行う．
- 失敗テストケースとの実行経路の類似度が高い成功テストケースにのみ大きな重み付けを行い，類似度が低い成功テストケースには重み付けを行わない．

ブロック化の目的は，連続して実行される行により特定のテストケースの類似度が非常に高くならないようにすることである．ブロック化の必要性は，3.7 節にて述べる．

失敗テストケースと成功テストケースの実行経路の類似度が高いことは，失敗テストケースで実行され，成功テストケースで実行されなかった行が少ないことを意味する．欠陥を含む行の絞り込めていると捉えられる．したがって，一定以上の類似度を持つ成功テストケースへの重み付けは有効だと結論づける．

3.3 ステップ 1: 欠陥を含むプログラムをテストスイートに通し，実行経路情報を取得する

実行経路情報は自動欠陥修正ツールである kGenProg[9] から取得する．kGenProg は GenProg[10] という自動欠陥修正ツールの Java 実装版である．kGenProg はライブラリとして JaCoCo[11] を用いる．JaCoCo を使うことで欠陥を含むプログラムをテストスイートに通した時の実行経路をテストケースごとに得ることができる．しかし，JaCoCo から実行経路情報を取得するには，JaCoCo から出力されるカバレッジレポートに適切な処理を行わなければならない．kGenProg にて，JaCoCo から実行経路情報を取り出すための適切な処理が実装されているため，これを利用する．

3.4 ステップ 2: 取得した実行経路のブロック化

ブロック化とは，特定のテストケースのみによって連続して実行される行を 1 つのブロックにまとめる処理である．図 5 を用いて説明する．図 5(a) において，4-6 行目は連続して実行されており，なおかつ 4-6 行目を実行するテストケースの集合は $\{t_a, t_c\}$ で一致している．そのため，ブロック化処理を行うと，4-6 行目は同一のブロック $B4$ にまとめられている．図 5(a) の 1，3 行目を実行するテストケースは t_a，t_b，t_c で一致している．しかし，間で 2 行目が実行されており，連続して実行されていないためそれぞれ異なるブロック $B1$，$B3$ にまとめられる．

ブロック化を行う上で，どのテストケースでも実行されない行は無視する．図 5(a) の 7 行目はどのテストケースでも実行されないので，どのブロックにも入れない．4-6 行目で，5 行目が仮に空行であっ

		ta	tb	tc
1:	if(n>5)	●	●	●
2:	n -= 6;	●	●	
3:	if(n<0){	●	●	●
4:	n += 1;	●		●
5:	n /= 3;	●		●
6:	n -= 5;	●		●
7:	}			
8:	if(n<-5)	●	●	●
9:	n += 1;	●		●
10:	return n;	●	●	●
	テスト結果	✗	✓	✓

(a) ブロック化前の実行経路

ブロック	ta	tb	tc
B1	●	●	●
B2	●	●	
B3	●	●	●
B4	●		●
B5	●	●	●
B6	●		●
B7	●	●	●
テスト結果	✗	✓	✓

(b) ブロック化後の実行経路

図 5: ブロック化の例

た場合，5 行目はどのテストケースでも実行されない行なので，無視する．この時，4，6 行目は，間に 5 行目が挟まっているが，連続して実行されると判断する．よって，4，6 行目は同一のブロックに含める．ソースコードを静的に解析して if 文や for 文の分岐に基づきブロック化を行わない理由は 3.8 節で述べる．

3.5 ステップ 3: ブロック化した実行経路の類似度から成功テストケースの重みを計算

ステップ 2 では，失敗テストケースと成功テストケースの類似度を求め，求めた類似度から成功テストケースの重みを算出する．類似度に閾値 $thsld$ を設け，類似度が閾値以下の時には等しく 1 の重みを，類似度が閾値よりも大きい場合には 1 よりも大きく，類似度の増加に伴い単調に増加する重みを与える．

以降において，失敗テストケースの集合を $\mathbb{F}$ とする．また，集合 $\mathbb{F}$ の要素の数を $|\mathbb{F}|$ と表す．次のように $only(t_i, t_j)$，$both(t_i, t_j)$ を定義する．

$only(t_i, t_j)$

テストケース t_i で実行され，t_j では実行されないプログラム要素の個数

$both(t_i, t_j)$

テストケース t_i，t_j の両方で実行されるプログラム要素の個数

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．

例として図 5 において $only(t_a, t_b)$，$both(t_a, t_b)$ を求める．テストケース t_a で実行されるブ

ロックの集合は $\{B1, B2, B3, B4, B5, B6, B7\}$，テストケース t_b で実行されるブロックの集合は $\{B1, B2, B3, B5, B7\}$ である．したがって，t_a で実行され，t_b で実行されないブロックは $\{B4, B6\}$ となる．よって，$only(t_a, t_b) = 2$ となる．また，t_a，t_b の両方で実行されるブロックは $\{B1, B2, B3, B5, B7\}$ である．よって，$both(t_a, t_b) = 5$ となる．

t_i，t_j の実行経路の類似度 x を次のように定義する．2 つの集合の類似度を計算するにあたり，広く利用されることが多い Jaccard 係数を利用しなかった理由は 3.9 節に記す．

$$x = \frac{both(t_i, t_j)}{only(t_i, t_j) + both(t_i, t_j)} \tag{2}$$

これらの数値を用いて，成功テストケース t_j の重み $w(t_j)$ を求める．失敗テストケース t_i に対する，成功テストケース t_j の重みを $w(t_i, t_j)$ とする．$w(t_i, t_j)$ は式 (3) のように定義される．$w(t_i, t_j)$ では，類似度 x の値がある閾値 $thsld$ 未満の場合には等しく 1 を，それ以上の場合には 1 よりも大きい値をとる．閾値 $thsld$ は，0-1 の範囲で与えられる．

$$w(t_i, t_j) = \begin{cases} 1 & (0 \leq x < thsld) \\ \dfrac{both(t_i, t_j)}{\sqrt{only(t_i, t_j) + both(t_i, t_j)}} & (thsld \leq x \leq 1) \end{cases} \tag{3}$$

閾値よりも類似度が低い場合には小さな重みを，高い場合には大きな重みを付与することから，重み付けにおいて，機械学習でよく用いられる活性化関数が参考にできると考えた．$thsld \leq x \leq 1$ の範囲でこの関数は類似度が大きくなるにつれ増加率が小さくなる．類似度の分母にルートをつけることで，活性化関数に近い値の取り方をするため，この関数を利用することにした．

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

$$w(t_j) = \frac{\sum_{t_i \in \mathbb{F}} w(t_i, t_j)}{|\mathbb{F}|} \tag{4}$$

例として，図 5 で，閾値 $thsld$ として 0.8 を利用した時の $w(t_c)$ を求める．まずは $w(t_a, t_c)$ を求める．$only(t_a, t_c) = 1$，$both(t_a, t_c) = 6$ であるため，t_a，t_c の類似度は次のようになる．

$x = \dfrac{6}{1+6} \fallingdotseq 0.86$

閾値以上の類似度であるため，$w(t_a, t_c)$ は $\dfrac{6}{\sqrt{1+6}} \fallingdotseq 2.27$ となる．$\mathbb{F} = \{t_a\}$ であるため，$w(c) = \dfrac{w(t_a, t_c)}{|\mathbb{F}|} = 2.27$ となる．

3.6 ステップ 4: 成功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

ep'^i，np'^i を次のように定義する．

ep'^i　行 s_i を実行する成功テストケースの重みの合計

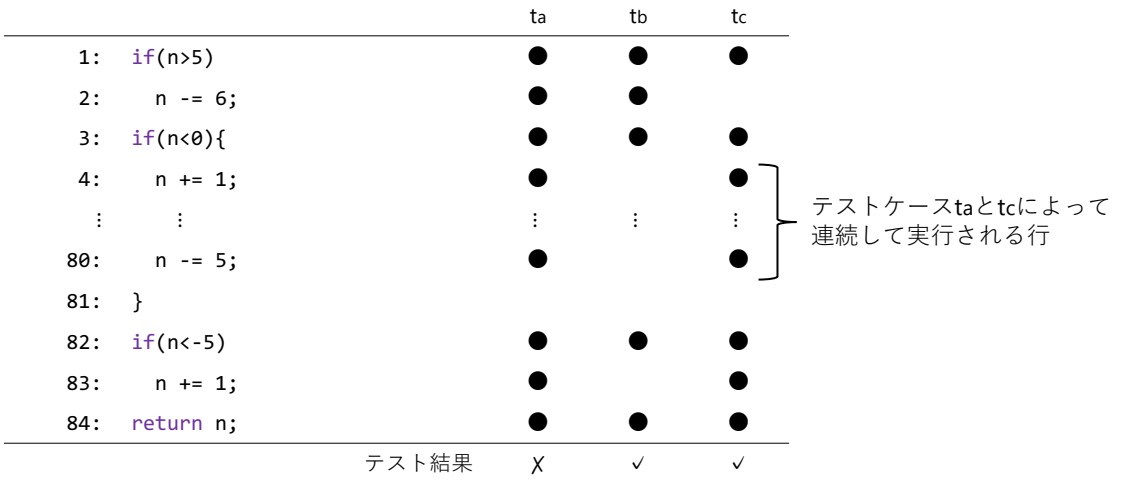

図 6: 連続して実行される行の存在により類似度が非常に高くなる例

np'^i　行 s_i を実行しない成功テストケースの重みの合計

既存の SBFL における ep^i，np^i を ep'^i，np'^i で置き換えた式が，提案手法における疑惑値の算出式である．例えば，Ochiai の式 (1) の ep^i を ep'^i で置き換えた提案手法における疑惑値の算出式は，以下のようになる．

$$suspicious(s_i) = \frac{ef^i}{\sqrt{(ef^i + nf^i) * (ep'^i + ef^i)}} \tag{5}$$

3.7　ブロック化の必要性

単純に実行経路を比較すると，テストケースの類似度は分岐を含まず連続して実行される行の存在に大きく影響される．後述の理由により，SBFL では分岐を含まない連続する行の長さは重要ではなく，分岐の存在が重要である．そのため，テストケースへの重み付けにおいても分岐を重要視するべきだと著者は捉える．ブロック化の目的は，連続して実行される行の影響を抑え，分岐を重視した重み付けを可能にすることである．

SBFL では分岐の存在が重要であると捉える理由を述べる．SBFL は分岐によってのみ疑惑値が変動する．例えば，図 6 では，4-80 行目は t_a，t_c によって連続して実行される分岐を含まないプログラムである．4-80 行目は全て同一のテストケース t_a，t_c によってのみ実行されるため，全ての行の疑惑値は同じである．1 行目や 3 行目の if 文によって分岐が生まれることで疑惑値に変化が生じる．分岐を含まず連続して実行される行の長さは，疑惑値に全く影響を与えない．4-80 行目が，仮に 4 行目で 1 行にまとめられて書かれても，その 1 行に付与される疑惑値の値は，元の 4-80 行目に付与される疑惑値と同じ値となる．したがって，SBFL では，分岐を含まず連続する行の長さは疑惑値に影響を与えないため重要ではなく，疑惑値に影響を与える分岐こそが疑惑値を決める重要なファクタであると推測する．

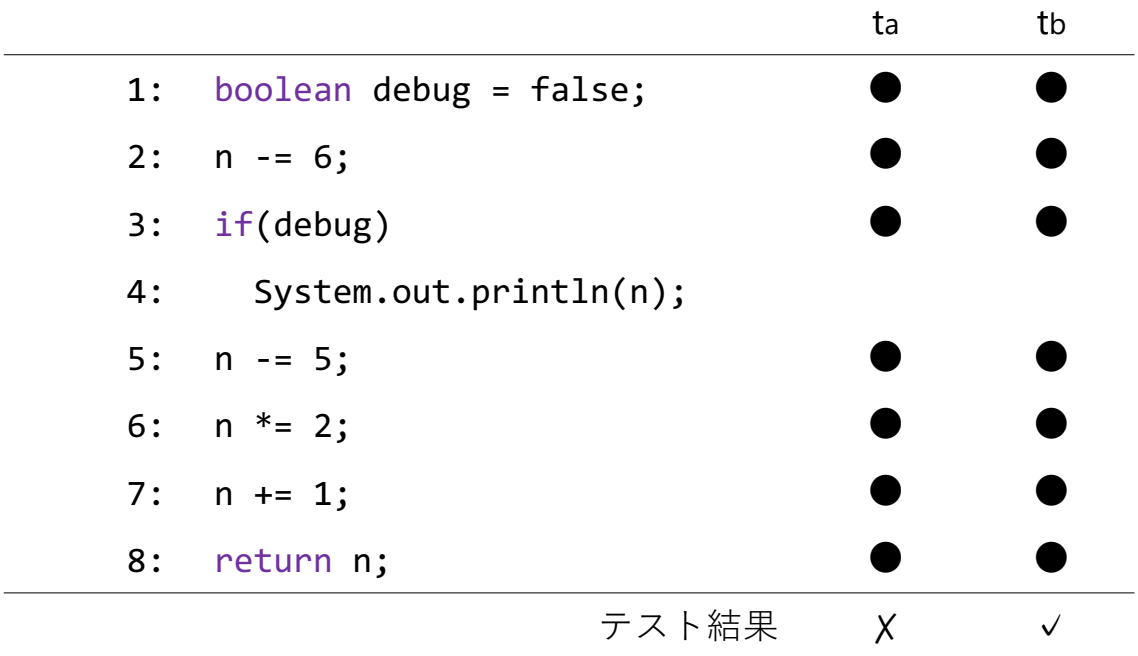

図 7: if 文や for 文に基づきブロック化を行わない理由

ブロック化を行うことで，連続して実行される行の長さによる影響を減らすことができる．連続して実行される行を，1 つのブロックにまとめることで，等価に扱うことができるようになるからである．例えば，図 6 の 2 行目と 4-80 行目は，ブロック化によりそれぞれ 1 つのブロックにまとめられる．提案手法ではブロック化後の実行経路の類似度を比較するため，2 行目と 4-80 行目はその長さによらず，等価に扱うことができる．分岐を含まず連続する行の影響を抑えることで，相対的に分岐に重きを置いた類似度の算出を行える．

3.8 if 文や for 文によりブロック化を行わない理由

実行経路情報に基づきブロック化を行う理由を述べる．ブロック化の目的は，連続して実行される行が重み付けに与える影響を抑えることである．この目的は，ソースコードを静的に解析し，if 文や for 文等の分岐を行うプログラム文において，その分岐先をまとめることでも達成できるように思われる．例えば，図 5(a) では，if 文の分岐先の行をまとめることで，図 5(b) と同じブロック化後の実行経路を得ることができる．しかし，ソースコードを静的に解析する方法では，テストケースで検出できない分岐まで検出してしまう．SBFL は，テストケースの実行経路情報にのみ基づき，元のソースコードの記述内容は一切考慮しない欠陥限局手法である．そのため，SBFL への重み付けにおいても実行経路情報にのみ基づき，元のソースコードの記述内容は考慮しないべきだと推測する．よって，提案手法のブロック化では，テストケースで検出できる内容にのみ基づくこととした．

テストケースの実行経路からは検出できないが，ソースコードを静的に解析することで検出してしまう分岐の例を，図 7 に挙げる．ソースコードを静的に解析する方法では，3 行目に if 文があることから，1-3 行目，4 行目，5-6 行目がそれぞれ異なるブロックにまとめられる．しかし，3 行目の if 文の条件式は，デバッグ時にのみ真となる条件式で，テストケースに通す時には常に偽をとる．そのため，全てのテストケースで 4 行目は実行されず，1，2，3，5，6，7，8 行目は必ず連続して実行される．つまり，3 行目の if 文による分岐は，実行経路情報からは検知できない分岐となる．

3.9 類似度の算出において Jaccard 係数を用いない理由

類似度として一般的に用いられることが多い Jaccard 係数を用いない理由を述べる．以下の説明において，t_{fail} を失敗テストケース，t_{pass} を成功テストケースとする．3.5 節で定義した $only(t_i, t_j)$，$both(t_i, t_j)$ を用いて，t_{fail}，t_{pass} の Jaccard 係数を表すと次のようになる．

$$\frac{both(t_{fail}, t_{pass})}{only(t_{fail}, t_{pass}) + only(t_{pass}, t_{fail}) + both(t_{fail}, t_{pass})} \tag{6}$$

重み付けは，「失敗テストケースで実行され，成功テストケースで実行されない部分に欠陥が含まれる可能性が高い」という仮説に基づく．そのため，失敗テストケースでのみ実行され成功テストケースでは実行されないブロックは重要だが，成功テストケースでのみ実行され，失敗テストケースでは実行されないブロックは重要ではない．よって，成功したテストケースのみで実行される行を考慮に入れるとかえってノイズになると考えた．したがって，提案手法では類似度を求める上で，成功テストケースでのみ実行されるブロック，すなわち，$only(t_{pass}, t_{fail})$ の存在を Jaccard 係数の式から省いた式を類似度の算出式として用いる．

4 実験

4.1 実験の目的

本実験の目的は，BSBFL における成功テストケースへの重み付けが有効であるかを明らかにすることである．したがって，以下の調査項目を設定する．以降，既存手法とは重み付けを行わない状態での SBFL のことを表す．

- 重み付け，ブロック化の有無，疑惑値の算出式による欠陥限局の精度の変化
- 既存手法と BSBFL での実行時間の変化

欠陥限局の技術は，自動欠陥修正でも用いられる技術である．自動欠陥修正の手法の中には，欠陥の修正が終わるまでに何度も欠陥限局を行う手法が存在する [12]．そのような手法では欠陥限局の実行時間の増加が自動欠陥修正の実行時間の増加に大きく関わる．したがって，重み付けによる精度の変化のみでなく，実行時間の増加も調査項目に加えた．

4.2 実験概要

ブロック化の有無による欠陥限局の精度を確かめるために，ブロック化を行わない状態で成功テストケースへの重み付けをする手法を用意する．以降，ブロック化を行わない状態で成功テストケースへの重み付けを行う手法を，Non-Blocknized Spectrum-Based weighting Fault Localization(NonBSBFL) と呼ぶ．BSBFL では，3.5 節における $only(t_i, t_j)$, $both(t_i, t_j)$ 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とする．$only(t_i, t_j)$，$both(t_i, t_j)$ の定義におけるプログラム要素をプログラムの行とする以外の，入力から出力までの流れは BSBFL と同様である．実験で用いる手法は表 1 にまとめて記す．

重み付け，ブロック化の有無，疑惑値の算出式の違いによる欠陥限局の精度の変化を調査するために，以下の比較を行う．

既存手法と BSBFL の比較

BSBFL の重み付けにより，既存手法よりも良い精度で欠陥限局できるか確かめるために行う．

表 1: 実験で用いる手法一覧

既存手法	重み付けを行わない既存の SBFL 手法 [6]
BSBFL	ブロック化を用いて成功テストケースへの重み付けを行う手法
NonBSBFL	ブロック化を用いずに成功テストケースへの重み付けを行う手法

BSBFL と NonBSBFL の比較

ブロック化の有効性を確かめるために行う．

さまざまな疑惑値の算出式における既存手法，BSBFL，NonBSBFL の比較

算出式によってどのような変化が生じるかを調査するために行う．

4.3 実験対象

本実験では，Defects4J[13] の Math プロジェクトに含まれる 100 個の欠陥を実験対象とする．Math を利用する理由は，本プロジェクトが欠陥限局の論文のベンチマークとして広く利用されているためである [14, 15, 16, 17]．100 個の欠陥には，339 の欠陥を含む行が存在する．このうち，Ochiai の算出式を用いた時に SBFL で検出できる欠陥を含む行の総数は 263 行である．検出できなかった欠陥を含む行の総数は 76 行である．検出できなかった理由の内訳は表 2 に記した通りである．

4.4 実験設定

疑惑値の算出式として，Ochiai[7]，Ample[18]，Jaccard[18]，Zoltar[18]，Tarantula[19] の式を用いる．各算出式は以下の通りである．

Ochiai

$$suspicious(s_i) = \frac{ef^i}{\sqrt{(ef^i + nf^i) * (a'^i_{ep} + ef^i)}} \tag{7}$$

Jaccard

$$suspicious(s_i) = \frac{ef^i}{ef^i + nf^i + ep^i} \tag{8}$$

Zoltar

$$suspicious(s_i) = \frac{ef^i}{ef^i + nf^i + ep^i + \frac{10000 * nf^i * ep^i}{ef^i}} \tag{9}$$

Ample

$$suspicious(s_i) = \left| \frac{ef^i}{ef^i + nf^i} - \frac{ep^i}{ep^i + np^i} \right| \tag{10}$$

表 2: Ochiai を用いた SBFL で検出できない欠陥を含む行の原因の内訳

	行数
失敗テストケースで実行されず検出できない欠陥を含む行	61 行
対象の Math プロジェクトが実行できず検出できない欠陥を含む行	15 行

Tarantula

$$suspicious(s_i) = \frac{\frac{ef^i}{ef^i + nf^i}}{\frac{ef^i}{ef^i + nf^i} + \frac{ep^i}{ep^i + np^i}} \tag{11}$$

4.5 評価指標

評価指標は次の 2 つである．

1. TopN%
2. 実行時間

1 つ目の評価指標は，TopN% である．TopN% とは，欠陥を含む行の疑惑値の順位が疑惑値がついた行全体の上位何パーセントに含まれるかを表す指標である．値が小さいほど欠陥を含む行を正しく推定できていることを意味する．なお，TopN% は異なる算出式での比較はできないことに注意されたい．算出式によって，疑惑値がついた行全体の数が変わるためである．例えば，Ochiai と Ample の TopN% を比較して Ample の TopN% の方が小さい値であっても，このことは Ochiai の算出式よりも Ample の算出式の方が優れていることは意味しない．

2 つ目の評価指標は，実行時間である．提案手法では重みを計算していることから，実行時間は増加することが予想される．既存手法と比べ，どの程度実行時間が増加するかを評価する．

4.6 重み付け，ブロック化の有無，算出式の違いによる欠陥限局の精度の変化

Ochiai の算出式を用いた場合の各閾値における TopN% を表 3 に記す．表中の太字は，既存手法よりも NonBSBFL，あるいは BSBFL の TopN% が良いことを表す．本実験において，重み付け (3) の閾値は，0.1-0.9 の区間を 0.1 区切りで取る．疑惑値の算出式として Ochiai を用いた場合の既存手法，NonBSBFL，BSBFL における TopN% の箱ひげ図を図 8 に示す．箱ひげ図では，NonBSBFL，BSBFL おいて，それぞれの手法が最も良い TopN% を取る時の閾値を用いた．表 3 に示したように，NonBSBFL，BSBFL において最も TopN% の結果が良かった閾値は，NonBSBFL においては 0.8，BSBFL においては 0.9 であった．

既存手法と BSBFL の比較

BSBFL の重み付けにより既存手法よりも良い精度で欠陥限局できるか確かめるために，既存手法と BSBFL の比較を行う．図 8 の箱ひげ図からわかるように，BSBFL は既存手法よりも第一四分位数，中央値，第三四分位数，平均値において優れている．各閾値ごとの，既存手法の方が高い順位をつけた欠陥を含む行の数，BSBFL の方が高い順位をつけた欠陥を含む行の数を表 4 に記す．既存手法の方が

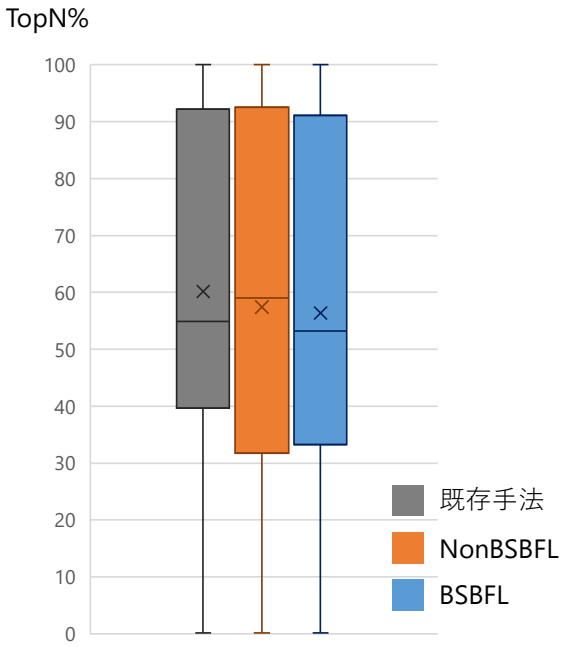

図 8: 各手法における TopN% を表した箱ひげ図

高い順位をつけた欠陥を含む行の数に対し，BSBFL の方が高い順位をつけた欠陥を含む行の数の割合が，閾値が高くなるほど大きくなる傾向にあることがわかる．閾値が高いということは，実行経路が失敗テストケースにより近い成功テストケースにのみ大きな重み付けがなされるということである．このことは，3.2 節にて述べた「失敗テストケースの実行経路との類似度が低い成功テストケースへ重み付けを行うことは，かえってノイズになりかねないため類似度が高い成功テストケースにのみ重み付けを

表 3: Ochiai における各手法での疑惑値

閾値	既存手法	NonBSBFL	BSBFL
0.1	60.05091	62.27870	60.40415
0.2	60.05091	62.35625	60.37689
0.3	60.05091	62.38185	60.37779
0.4	60.05091	62.35625	60.36919
0.5	60.05091	62.14986	**59.30508**
0.6	60.05091	**59.69354**	**56.76153**
0.7	60.05091	**57.72750**	**56.63470**
0.8	60.05091	**57.48240**	**58.80798**
0.9	60.05091	60.42033	**56.50219**

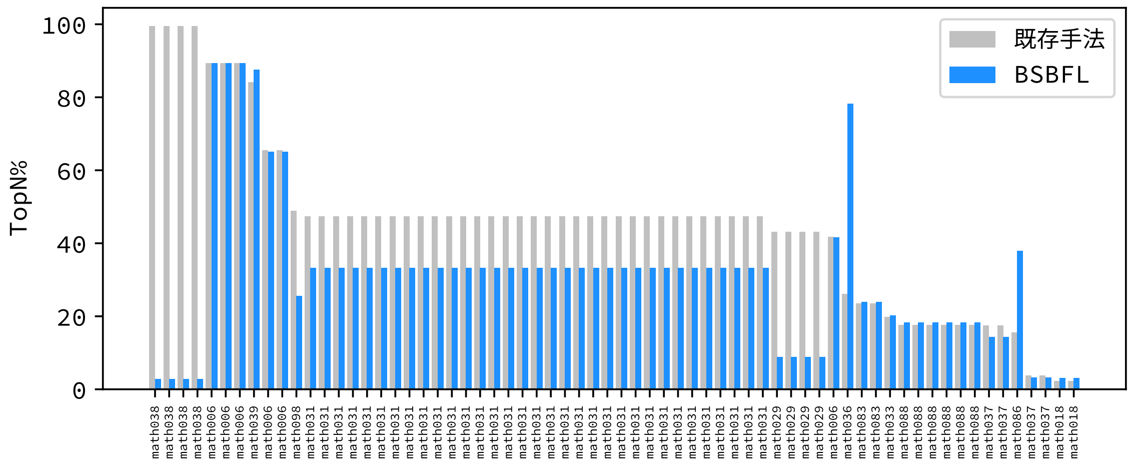

図 **9:** 既存手法，BSBFL における TopN% を表した棒グラフ

するべきである」とするアイデアを裏付ける結果であると捉える．

閾値 0.9 において，既存手法と BSBFL 間で TopN% に差異が生じた合計 66 個の欠陥を含む行の TopN% を図 9 に記す．BSBFL のほうが優れている部分では TopN% の既存手法からの上がり幅が，既存手法のほうが優れている部分の下がり幅よりも大きい傾向にあることが読み取れる．この傾向は，math038 や math088 といったプロジェクトで顕著である．

BSBFL では，閾値を設け失敗テストケースとの類似度が高い成功テストケースにのみ大きな重みを付与する．失敗テストケースで実行され，大きな重みのついた成功テストケースで実行されなかった行

表 **4:** 既存手法と BSBFL がそれぞれに対しより高い順位をつけていた欠陥を含む行の数

閾値	既存手法の方が高い順位をつけていた欠陥を含む行の数	BSBFL の方が高い順位をつけていた欠陥を含む行の数
0.1	71	19
0.2	70	19
0.3	71	19
0.4	71	24
0.5	75	19
0.6	40	53
0.7	28	47
0.8	47	23
0.9	14	52

の疑惑値の順位は，相対的に非常に高くなる．この順位が高まる行に，欠陥を含む行が含まれる場合，欠陥を含む行の順位は大きく向上する．逆に，この部分に欠陥を含む行が存在しない場合，欠陥を含む行の順位は，失敗テストケースで実行され大きな重みのついた成功テストケースで実行されなかった行数分だけ下がる．失敗テストケースと成功テストケースの類似度が高いということは，失敗テストケースで実行され成功テストケースで実行されないブロックの数が少ないことを意味する．そのため，類似度が高い時，失敗テストケースで実行され大きな重みのついた成功テストケースでは実行されない行数は少ないことが期待される．したがって，順位が低下する場合の下がり幅は，比較的小さくなる．

BSBFL と NonBSBFL の比較

ブロック化の有効性を確かめるために，BSBFL と NonBSBFL の比較を行う．NonBSBFL と BSBFL の比較を行い，ブロック化の有効性を確かめる．図 8 に記した箱ひげ図に，2 つの手法で最も良い TopN% を取る時の結果を示した．BSBFL は，NonBSBFL と比べ，第一四分位数では劣るが，中央値，平均値，第三四分位数で優れている．特に，中央値で BSBFL は 5.802% と大きな差をつけて NonBSBFL よりも優れている．表 3 を見ると，BSBFL では閾値 0.5 以上のすべての閾値で既存手法よりも良い TopN% を出す．しかし，NonBSBFL で既存手法よりも良い TopN% を出す閾値は，0.6，0.7，0.8 の 3 つのみである．より広い閾値の範囲において既存手法よりも良い TopN% の結果を示す BSBFL のほうが優れていると解釈できる．

これらの実験結果から，NonBSBFL よりも BSBFL の方が欠陥を含む行の推定をより正確に行えると結論づける．よって，ブロック化を行うことは，より正確に欠陥を含む行を推定する上で有効である．

さまざまな疑惑値の算出式における既存手法，BSBFL，NonBSBFL の比較

算出式によってどのような変化が生じるかを調査するため，算出式として Jaccard[7]，Zoltar[18]，Ample[18]，Tarantula[19] を用いた場合の既存手法，NonBSBFL，BSBFL における TopN% を比較する．結果を表 3，表 5，表 6，表 7，表 8 に記す．表中の太文字は，既存手法よりも良い TopN% であったことを表す．なお，提案手法のキーアイデアが，「実行経路が失敗テストケースに近い成功テストケースに大きな重み付けを行う」であるため，重み付け (3) の閾値として 0.4 以降のみを記した．

Ochiai，Jaccard，Zoltar，Ample，Tarantula の算出式は 4.4 節において示した通りである．なお，Ochiai の時と同様に，提案手法において疑惑値を算出する際には ep^i，np^i をそれぞれ ep'^i，np'^i で置き換える．

各手法における最も良い TopN% とその時の閾値，BSBFL と既存手法の TopN% の差を表 9 に記す．表 9 からわかるように，Ochiai，Jaccard，Zoltar，Tarantula の算出式を用いた時，適切な閾値を選ぶと，BSBFL は既存手法よりも TopN% が良くなる．しかし，Ample の算出式を用いた時には，全

ての閾値において BSBFL は既存手法よりも TopN% が悪くなる．Ample で TopN% が悪くなる原因は，4.6 節にて考察する．

Jaccard，Zoltar，Tarantula の算出式では，Ochiai の式と同様に閾値が高くなるにつれ TopN% の結果が良くなる傾向が見られる．類似度が低いテストケースへの重み付けは，かえってノイズになることを示す結果だと考えられる．

表 5: Jaccard における各手法での疑惑値

閾値	既存手法	NonBSBFL	BSBFL
0.4	62.68231	64.33053	**60.55796**
0.5	62.68231	64.31797	**59.46270**
0.6	62.68231	64.52492	**59.33604**
0.7	62.68231	**62.30274**	**59.21052**
0.8	62.68231	**60.53649**	**58.98406**
0.9	62.68231	**60.70188**	**58.82356**

表 6: Zoltar における各手法での疑惑値

閾値	既存手法	NonBSBFL	BSBFL
0.4	58.79702	61.99813	60.03897
0.5	58.79702	61.98556	58.87003
0.6	58.79702	62.02654	**58.75032**
0.7	58.79702	59.87069	**58.41994**
0.8	58.79702	59.90939	**58.51958**
0.9	58.79702	60.05751	**58.42319**

表 7: Ample における各手法での疑惑値

閾値	既存手法	NonBSBFL	BSBFL
0.4	46.78844	52.09677	47.47852
0.5	46.78844	52.50156	47.27867
0.6	46.78844	52.95457	47.90712
0.7	46.78844	53.41038	48.80310
0.8	46.78844	51.83798	49.38491
0.9	46.78844	52.73496	48.47011

表 8: Tarantula における各手法での疑惑値

閾値	既存手法	NonBSBFL	BSBFL
0.4	64.27574	64.28749	**63.97079**
0.5	64.27574	64.34203	**63.95078**
0.6	64.27574	64.41642	**62.65841**
0.7	64.27574	**64.18703**	**62.55332**
0.8	64.27574	**62.25333**	**62.68193**
0.9	64.27574	**62.36277**	**62.34650**

表 9: 各手法の最も良い TopN% とその時の閾値，既存手法の TopN% との差

手法	最も良い TopN%	閾値	既存手法との差
Ochiai	56.38650	0.9	-3.54874
Jaccard	58.82356	0.9	-3.85875
Zoltar	58.41994	0.7	-0.37708
Ample	47.27867	0.5	+0.4902
Tarantula	62.34650	0.9	-1.92924

		ta	tb	tc	td	te	疑惑値
1:	`if(n>5)`	●	●		●		0.17
2:	`n -= 6;`		●	●	●		0.17
3:	`if(n<0){`	●			●	●	0.17
4:	`n += 1;`				●	●	1.00
⋮	⋮	⋮	⋮		⋮	⋮	
	テスト結果	✗	✗	✗	✓	✓	

図 10: Ample の算出式における問題点

4.7 Ample では BSBFL の TopN% が既存手法よりも悪くなる原因

Ample の算出式 (10) を用いると TopN% が悪化した理由を考察する．Ample の式では，$\frac{ef^i}{ef^i+nf^i}$ よりも，$\frac{ep^i}{ep^i+np^i}$ の値が大きくなると，成功テストケースでより多く実行され，失敗テストケースで実行されなかった行の疑惑値が高まる．このことは，失敗テストケースで実行された行は欠陥である可能性が高く，成功テストケースで実行された行は欠陥を含まない可能性が高いという SBFL のアイデアに矛盾する．そのため，$\frac{ef^i}{ef^i+nf^i} < \frac{ep^i}{ep^i+np^i}$ となる場合に Ample の算出式を用いると正確に欠陥限局を行うことができない．

例えば，図 10 は，Ample を算出式として用いた時の，重み付けを行わない既存 SBFL の実行結果を抜粋した図である．t_a，t_b，t_c が失敗テストケース，t_d，t_e が成功テストケースである．4 行目は，成功テストケース t_d，t_e によってのみ実行される行である．SBFL のアイデアに基づくと，2 行目は成功テストケースによってのみ実行されているため，欠陥を含む可能性が低い行となる．しかし，Ample の算出式を用いると疑惑値は 1.0 となり，失敗テストケースで実行された 1-3 行の疑惑値よりも高い値を示す．これは，$\frac{ef^i}{ef^i+nf^i}$ よりも，$\frac{ep^i}{ep^i+np^i}$ の値が大きくなるため，成功テストケースでより多く実行された行に高い疑惑値がつくためである．SBFL のアイデアに反した疑惑値を算出することは問題である．

本提案手法では，失敗テストケースと類似度の高い成功テストケースにのみ大きな重み付けを行う関係上，$\frac{ep^i}{ep^i+np^i}$ の値が $\frac{ef^i}{ef^i+nf^i}$ よりも大きくなりやすい．例えば，図 10 の，1，2 行目では，重み付けを行わない場合には $\frac{ef^i}{ef^i+nf^i} > \frac{ep^i}{ep^i+np^i}$ だった．この関係式が成り立つ時，失敗テストケースで実行された行は欠陥である可能性が高く，成功テストケースで実行された行は欠陥でない可能性が高いという SBFL のアイデアに沿った疑惑値が算出されるため，問題はない．しかし，例えば w_c に 4 の重みを，w_d に 1 の重みをつけると，重み付け前には $\frac{ef^i}{ef^i+nf^i} > \frac{ep^i}{ep^i+np^i}$ であった 1，2 行目において，$\frac{ef^i}{ef^i+nf^i} < \frac{ep^i}{ep^i+np^i}$ となる．先に述べたように，このような大小関係の時，成功テストケースで実行され，失敗テストケースで実行されなかった行ほど欠陥である可能性が高くなり，問題となる．Ample の算出式は，成功テストケースへの重み付けを行う提案手法との相性が悪いといえる．

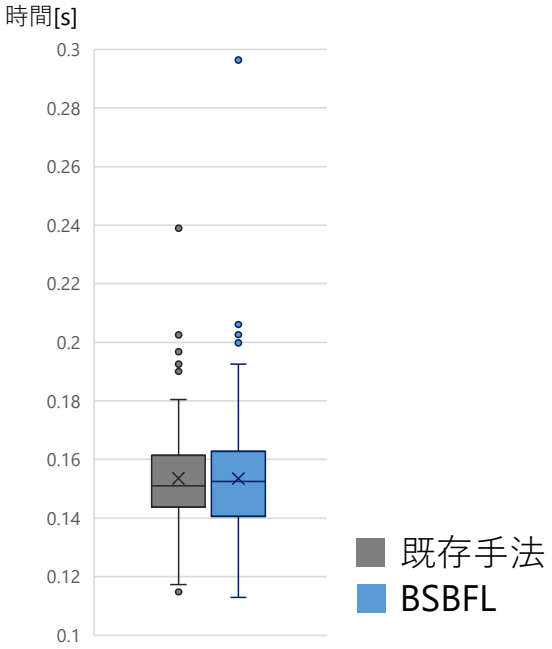

図 11: 既存手法と BSBFL における実行時間

4.8 実行時間の変化

既存手法，BSBFL 間で実行時間にどのような変化があるかを確かめた．既存手法と BSBFL は，どちらも次の 2 つの処理からなる．

1. 欠陥を含むプログラムをテストスイートに通し実行経路情報を取得
2. 取得した実行経路情報から疑惑値を算出

実行経路情報を取得する部分の処理は全く同じであり，取得した実行経路情報から疑惑値の算出を行う部分のみが既存手法と BSBFL で異なる．したがって，実行時間の変化を比べる上で，疑惑値の算出部分のみ比較する．各手法における実行時間を図 11 に記す．

既存手法における実行時間の中央値は 0.151[s]，BSBFL における実行時間の中央値は 0.152[s] であった．実行時間の増加は微小であることを確認した．

5　妥当性の脅威

5.1　実験対象

本実験では，Defects4J[13] に含まれる Math プロジェクトを実験対象とした．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．

5.2　重み付け

成功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

本実験では，SBFL の精度を高めるためにブロック化を行い，実行経路の類似度から成功テストケースに重みを持たせる手法を提案した．重み付けを行わない既存の SBFL と BSBFL を実行時間と TopN% の観点から比較した．実行時間の増加は微増でありながら，TopN% が向上することを確認した．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，ブロック化により精度が上昇することを確認した．以上の結果から，BSBFL が有効である可能性を示した．

今後の課題としては以下が考えられる．

Defects4J の Math プロジェクト以外を実験対象とした実験

Defects4J の Math プロジェクト以外にも BSBFL を適用し，他プロジェクトにおいても既存手法よりも高い精度で欠陥を含む行を推定できているかを調査する．

BSBFL が既存手法より優れた結果を出すプロジェクトの特徴の分析

BSBFL が既存手法より良い結果を示したプロジェクトに見られる傾向を，行数などの観点から調査し，どのようなプロジェクトに BSBFL が有効である傾向にあるのか調査する．

より適切な重み付けを行う重み付け関数の調査

重み付けの算出式として利用した式の数が少ないため，より多くの算出式を考案し，どのような算出式が BSBFL に適しているのかを調査する．

謝辞

本研究を行うにあたり，研究の場を与えてくださり，研究成果を熱心に聞いていただきました楠本真二教授に深く感謝申し上げます．

楠本研究室に配属されてから今日に至るまで，研究の全過程を通じて熱心にご指導をしていただき，親身に相談に乗っていただきました肥後芳樹准教授に心より感謝申し上げます．

本研究において，随所で的確な助言をしていただき，kGenProg に関する見識を教えていただきました柗本真佑助教に心より感謝申し上げます．

研究活動を行うにあたり，時に慰め，時に励ましていただいた楠本研究室の先輩，そして同回の皆様に心より感謝申し上げます．

本研究に至るまでに，大阪大学基礎工学部情報科学科の授業でお世話になりました先生方に，この場を借りて心から御礼申し上げます．

最後に，22 年間様々な側面から支えていただきお世話になりました家族にこの場を借りて心から御礼申し上げます．

参考文献

[1] B. Hailpern and P. Santhanam. Software debugging, testing, and verification. *IBM Systems Journal*, Vol. 41, No. 1, pp. 4–12, 2002.

[2] Tom Britton, Lisa Jeng, Graham Carver, and Paul Cheak. Quantify the time and cost saved using reversible debuggers. Technical report, Cambridge Judge Business School, 2012.

[3] B. Korel. Pelas-program error-locating assistant system. *IEEE Transactions on Software Engineering*, Vol. 14, No. 9, pp. 1253–1260, 1988.

[4] Wei Jin and Alessandro Orso. F3: Fault localization for field failures. In *Proc. International Symposium on Software Testing and Analysis*, pp. 213–223, 2013.

[5] Chao Liu, Xifeng Yan, Long Fei, Jiawei Han, and Samuel P. Midkiff. Sober: Statistical model-based bug localization. In *Proc. Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pp. 286–295, 2005.

[6] W. Eric Wong, Ruizhi Gao, Yihao Li, Rui Abreu, and Franz Wotawa. A survey on software fault localization. *IEEE Transactions on Software Engineering*, Vol. 42, No. 8, pp. 707–740, 2016.

[7] Rui Abreu, Peter Zoeteweij, and Arjan J.C. van Gemund. On the accuracy of spectrum-based fault localization. In *Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION*, pp. 89–98, 2007.

[8] Aritra Bandyopadhyay and Sudipto Ghosh. Proximity based weighting of test cases to improve spectrum based fault localization. In *Proc. International Conference on Automated Software Engineering*, pp. 420–423, 2011.

[9] 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. In *Proc. Asia-Pacific Software Engineering Conference*, pp. 697–698, 2018.

[10] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. Genprog: A generic method for automatic software repair. *IEEE Transactions on Software Engineering*, Vol. 38, No. 1, pp. 54–72, 2012.

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

[12] Westley Weimer, ThanhVu Nguyen, Claire Le Goues, and Stephanie Forrest. Automatically finding patches using genetic programming. In *Proc. International Conference on Software*

Engineering, pp. 364–374, 2009.

[13] René Just, Darioush Jalali, and Michael D. Ernst. Defects4j: A database of existing faults to enable controlled testing studies for java programs. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, pp. 437–440, 2014.

[14] Jeongju Sohn and Shin Yoo. Fluccs: Using code and change metrics to improve fault localization. In *Proc. International Symposium on Software Testing and Analysis*, pp. 273–283, 2017.

[15] 藤本章良, 肥後芳樹, 柗本真佑, 楠本真二, 安田和矢. 自動テスト生成を利用した自動プログラム修正出力パッチ分類の試み. 電子情報通信学会技術研究報告, 第 121 巻, pp. 124–129, 1 2022.

[16] 肥後芳樹, 柗本真佑, 内藤圭吾, 谷門照斗, 楠本真二, 切貫弘之, 倉林利行, 丹野治門. 設定ファイルを考慮した fault localization の拡張. 情報処理学会論文誌, Vol. 61, No. 4, pp. 884–894, 4 2020.

[17] 九間哲士, 肥後芳樹, 柗本真佑, 楠本真二. 欠陥限局に適したテストスイートに関する考察. 電子情報通信学会技術研究報告, Vol. 119, No. 362, pp. 019–024, 1 2020.

[18] Simon Heiden, Lars Grunske, Timo Kehrer, Fabian Keller, André van Hoorn, Antonio Filieri, and David Lo. An evaluation of pure spectrum-based fault localization techniques for large-scale software systems. *Software: Practice and Experience*, Vol. 49, pp. 1197–1224, 05 2019.

[19] James A. Jones and Mary Jean Harrold. Empirical evaluation of the tarantula automatic fault-localization technique. In *Proc. International Conference on Automated Software Engineering*, pp. 273–282, 2005.