DEIM Forum 2023 5b-6

# 識別パターンマイニングを用いた自動バグ限局

杉森　裕斗†　　中村　篤祥††

† 北海道大学工学部情報エレクトロニクス学科　〒 060–0814 札幌市北区北 14 条西 9 丁目
†† 北海道大学大学院情報科学研究院　〒 060–0814 札幌市北区北 14 条西 9 丁目
E-mail: †{sugimori,atsu}@ist.hokudai.ac.jp

**あらまし**　ソフトウェアに含まれるバグの原因箇所を自動で特定するため、自動バグ限局手法の研究が盛んに行われている。既存手法として、テストケース実行の結果と実行経路を利用する方法が提案されている。同手法ではテストケース実行においてどの行を通過したかという情報のみを使用しており、その順番は考慮されていない。本論文ではバグ限局効率向上のため、実行順序を記録した実行トレースを使用し、識別パターンマイニングを適用する手法を提案する。識別シーケンシャルパターンマイニングはクラス分けされたシークエンスデータの中からクラスを特徴づけるパターンを発見するデータマイニング技術である。失敗したテストケース実行を特徴づけるパターンにはバグが含まれている可能性が高いと考えられる。実験の結果、提案手法によるバグ限局効率は全体として低下したが、バグの種類によっては向上するものがあることがわかった。

**キーワード**　ソフトウェア開発,　バグ限局, データマイニング, 識別パターンマイニング

## 1　はじめに

デバッグは、ソフトウェア開発において多大なコストが掛かる作業である [3]。デバッグの中でもバグの原因箇所を特定する作業をバグ限局と呼ぶ。この作業は開発経験とプログラムに対する理解を持った技術者が行う必要がある。手動のデバッグはコード全体の再確認や、コードの複数箇所にブレークポイントをはり、変数の値を確認する作業が必要になるため、手間と費用がかかる。

そこで、バグ限局コスト削減のため自動バグ限局の研究が盛んに行われている [1, 4, 7, 8, 9, 10, 12]。その手法はプログラムの静的解析やバグレポートを利用したもの、テストケース実行時の通過ステートメント情報を利用したものなど、様々である。本論文ではテストケース実行におけるプログラム内のステートメントの実行順序であるトレースを使用し、識別パターンマイニングを用いた自動バグ限局手法を提案する。

識別パターンマイニングとはクラスづけされたデータにおいて、そのクラスを特徴づけるようなパターンを発見するデータマイニング技術である。識別パターンマイニングは様々な分類問題に応用されている [5, 6]。バグの含まれているプログラムにおいて、失敗したテスト実行トレースを特徴づけるパターンにはバグの原因となっているステートメントが含まれている可能性が高いと考えられる。このようなパターンを抽出することでバグ限局に利用できる。

### 関連研究

実行トレースに対するデータマイニング技術を応用したバグ限局手法として N グラムを用いた手法が提案されている [8]。N グラムとはシークエンスの長さ N の連続部分列である。この手法では、まずテスト実行におけるステートメントのトレースを同論文で提案されている線形実行ブロックのトレースに変換される。これにより計算量が削減される。全てのトレースから N グラムを生成し、その中から失敗トレースにおいて頻出なものを抽出する。抽出された N グラムは失敗トレースにおける信頼度によってランキングされ、ランキングが高い N グラムに含まれるステートメントから順に検査する。

機械学習を応用した手法としてニューラルネットワークを用いたバグ限局手法も提案されている [9]。入力には各テストケースで通過したステートメントを表したカバレッジベクトル $c$、出力にはその結果を表した結果ベクトル $r$ を用いる。カバレッジベクトルの要素 $c_i$ は、そのテストケースにおいてステートメント $s_i$ が実行された場合に 1、実行されなかった場合には 0 となる。結果ベクトルの要素 $r_m$ は、$m$ 番目のテストが失敗した場合に 1、成功した場合 0 となる。このデータから、実行されたステートメントとその結果の関係を学習する。その後、仮想テストケースとして一つのステートメントのみを通過したケースを入力する。つまり、$c_i = 1$ それ以外は 0 としたベクトルを入力する。これを全てのステートメントに対して行う。この時の出力値は 0 以上 1 未満の値を取る。その値がステートメント $s_i$ の疑惑値となり、この値が大きい順から検査する。この手法はさらに RBF ニューラルネットワークを用いた手法へと拡張されている [13]。

また、プログラムスライスを利用したバグ限局手法も提案されている [12]。プログラムスライスとはある変数の値に関係するステートメントの集合である。この手法は、「テストが失敗した時、出力に関係するスライスにバグが含まれている」という考えに基づいている。テストが失敗した時のスライスとテストが成功したときのスライスの差を考えることで、バグが含まれる可能性のあるステートメントの範囲を絞り込むことができる。

識別パターンマイニングはバグ検知にも応用されている。バグ検知とは、プログラムの実行履歴からそのプログラムにバグ

が含まれているかを判断する技術である。バグが検知された場合、テストを追加しバグ限局を行う。[11] ではテストトレースにおける頻出パターンを特徴として分類器を学習する手法を提案している。まず実行トレースから反復頻出パターンを抽出する。反復頻出パターンとはそれぞれのトレースにおいて出現した回数を考慮した頻出パターンである。このパターンからさらに識別パターンを抽出する。実行トレースにおける識別パターンの出現回数からそのプログラムにバグが含まれているかどうかを判断する。バグ検知においては抽出されたパターンに対応するコードの行にバグが含まれているかではなく、プログラムのどこかにバグが含まれているかを判断する。本稿では識別パターンマイニングをバグ限局に用いるため、パターンに対応するコードの行にバグが含まれるようなパターンを抽出する手法の開発を目的とする。

## 2 問題設定

$n$ 行からなるプログラムを $P$ とする。$P$ 内のソースコードの実行可能ステートメントの集合を $S = \{s_1, s_2, ..., s_k\}$ とする。実行可能ステートメントとは、ソースコードにおいて変数宣言や空行を除いた、実行可能なステートメントである。このプログラムに対するテスト集合を $T = \{t_1, t_2, ..., , t_m\}$ とする。各テスト $t_i$ は入力 $I_i$ と期待される出力 $Y_i$ の組 $t_i = (I_i, Y_i)$ で表される。テスト $t_i$ において、入力 $I_i$ を与えられた際の実行されたステートメントを実行順に並べた列を $X_i = (x_1, x_2, ..., x_l)(x_j \in S)$ とし、これをトレースと呼ぶ。$t_i$ における実際の出力を $A_i$ とする。このテストの出力が期待出力 $Y_i$ と異なっていた場合、つまり $A_i \neq Y_i$ のときテストは失敗したとする。パターン $b$ は実行可能ステートメント列で表され、トレース $X_i$ に $b$ が連続部分列として現れるとき、パターン $b$ はトレース $X_i$ に出現するという。$b$ が出現するトレースの集合を $Occ(b)$ で表現すると、パターン $b$ のトレース $X_i$ における出現は $X_i \in Occ(b)$ で表現される。

それぞれのトレース $X_i$ に、そのテストが成功したか失敗したかを表すクラス $c_i \in \{0, 1\}$ が与えれているものとする。ただし、$t_i$ が成功した場合 $c_i = 0$、失敗した場合 $c_i = 1$ とする。それぞれのテストのトレース $X_i$ とそのテストのクラス $c_i$ を $d_i = (X_i, c_i)$ のように順序対で表現すると、全てのテストにおけるトレースとそのテスト結果は $D = \{d_1, d_2, ..., d_m\}$ で表される。

表 1 に示すサンプルプログラム $P$ は自然数 $n$ に対して $\lfloor \frac{n}{2} \rfloor$ を求めるプログラムである。このプログラムに対してテストケース $T = \{t_1, t_2, t_3\}$ を実行したとする。この時各テストケースにおけるトレースは $X_1 = (s_1, s_2, s_4, s_5, s_6), X_2 = (s_1, s_2, s_3, s_6), X_3 = (s_1, s_2, s_3, s_6)$ であり、結果を組み合わせた順序対で表現すると、$d_1 = (X_1, 1), d_2 = (X_2, 0), d_3 = (X_3, 1)$ となる。すべてのテストにおけるトレースとその結果は $D = \{d_1, d_2, d_3\}$ である。

本稿では、実行可能ステートメント集合 $S$、トレースとテスト結果の集合 $D$ が与えられたとき、$S$ に属するステートメントをバグを含む可能性が高い順に並べる問題を考える。

## 3 従来手法 SBFL

自動バグ限局とは、プログラムのバグの位置を特定する、デバッグ支援技術である。spectrum based fault localization (SBFL) はテストの結果と実行経路からソースコード上でバグの含まれている位置を特定する手法である [4, 10]。具体的な手法は、まずそれぞれのテストの実行結果の記録と実行されたソースコード上の行番号の記録を収集する。その記録をもとに、各ステートメント $s$ にバグが含まれている可能性となる「疑惑値」を算出し、その値が高い順にコードを検査していく。疑惑値の計算方法は様々だが、最も一般的である tarantula[4] の指標 $suspiciousness(s)$ は以下の通りである

$$suspiciousness(s) = \frac{\frac{failed(s)}{totalfailed}}{\frac{passed(s)}{totalpassed} + \frac{failed(s)}{totalfailed}} \quad (1)$$

ただし、$totalfailed$ は失敗テストケースの総数を表し、$totalpassed$ は成功テストケースの総数を表す。また、プログラム上のあるステートメント $s$ を実行し、かつ失敗したテスト数を $failed(s)$、成功したテスト数を $passed(s)$ と表す。

SBFL は「そのステートメントを通過し、失敗したテストの割合が高いほどバグの原因となっている可能性が高い。」という直感に基づき、指標が考案されている。

表 1 の $suspiciousness(s)$ の値は tarantula の指標を用いて疑惑値を計算したものである。この場合、検査の優先順位が最も高いステートメントは $s_4, s_5$、次に $s_1, s_2, s_6$、最後に $s_3$ となる。バグが含まれている $s_5$ の優先順位が高くなっていることがわかる。

## 4 提案手法

従来手法では、テスト実行経路の情報として、どのステートメントを通ったかという情報のみが使用されており、その順番は考慮されていなかった。本研究では、精度向上のため、実行経路をシークエンスデータとして扱う自動バグ限局手法を提案する。

### 4.1 識別シークエンスパターンマイニング

識別シークエンスパターンマイニングとは、クラスづけされたシークエンスデータが与えられた際に、そのクラスを特徴づけるようなパターンを発見するデータマイニング技術である。シークエンスのパターンに対し、識別力を表す指標（識別スコア）を計算することでそのパターンがクラスの分類においてどれだけ有用であるかを計算することができる。頻出パターンに対し識別スコアを計算し、ある閾値以上のスコアを持つパターンを識別パターンとして抽出する手法が提案されている [2]。指標には情報利得やフィッシャースコアが使用される。テストに対するトレースと結果の集合 $D$ に対して、パターン $b$ がトレースに出現するとき 1、出現しないとき 0 の値をとる変数を $X_b$、

表 1　サンプルプログラム $P$

| ステートメント | プログラム $P$ | $t_1(1,0)$ | $t_2(4,2)$ | $t_3(5,2)$ | suspiciousness(s) |
|---|---|---|---|---|---|
| $s_1$ | scanf(%$d$, &$n$) | ○ | ○ | ○ | 0.5 |
| $s_2$ | if(a%2 == 0){ | ○ | ○ | ○ | 0.5 |
| $s_3$ | ans = n/2} | | ○ | | 0 |
| $s_4$ | else if(a%2 == 1){ | ○ | | ○ | 1 |
| $s_5$ | ans = (n+1)/2} #bug | ○ | | ○ | 1 |
| $s_6$ | return(ans) | ○ | ○ | ○ | 0.5 |
| result(failed:1, passed:0) | | 1 | 0 | 1 | |

テスト結果 (0 または 1) を表す変数を $C$ とした場合、パターン $b$ に対する情報利得 $IG_b$ は以下のように定義される。

$$IG_b = H(C) - H(C|X_b) \tag{2}$$

ただし、$H(C)$ は $C$ のエントロピー、$H(C|X)$ は $X$ で条件づけた $C$ の条件付エントロピーである。

$D$ におけるパターン $b$ に対するフィッシャースコア $Fr_b$ の定義を次に示す。

$$Fr_b = \frac{\sum_{i=0}^{1} n_i(\mu_i - \mu)^2}{\sum_{i=0}^{1} n_i \sigma_i^2} \tag{3}$$

ただし、$n_i$ はクラス $i$ におけるデータ数、$\mu_i$ はクラス $i$ における $X_b$ の値の平均、$\sigma_i$ はクラス $i$ における $X_b$ の値の標準偏差、$\mu$ は全データにおける $X_b$ の値の平均を表す。

### 4.2　自動バグ限局アルゴリズム

提案するアルゴリズムを Algorighm 1 に示す。このアルゴリズムは、テストトレースとその実行結果のデータ $D = \{d_1, d_2, ..., d_m\}$、実行可能ステートメント集合 $S = \{s_1, s_2, ..., s_k\}$、頻出パターンマイニングにおける頻度の閾値 $\theta_0$ を入力としてバグが含まれている可能性の高いステートメントのランキングを出力する。収集されたテストトレースはまずプログラム基本ブロックのトレースに変換される (step1)。プログラム基本ブロックとは、内部に分岐や合流を持たないステートメントの列である。基本ブロック内の最初のステートメントが実行されると、同じブロックに含まれるステートメントも必ず順次実行される。次に失敗トレースから頻出パターンを抽出する (step2)。失敗トレースに頻出なパターンにはバグが含まれている可能性が高いと考えられるが、成功トレースにおいても頻出である可能性もある。失敗トレースと成功とレースの両方において頻出なパターンは単にプログラムの実行時に必ず実行されるだけであり、バグが含まれている可能性が高いとは考えられない。そこで、失敗トレースと成功トレースそれぞれにおける頻度にどれだけの差があるかを表す指標として (情報利得またはフィッシャースコア) を計算する (step3)。この値が大きいパターンは失敗トレースを代表するようなパターンであると考えられる。最後にステートメントに対するスコアをこの指標を用いて計算し、ランキングする (step4)。このランキング順にステートメントを検査することで早いバグ発見を可能にする。

**Algorithm 1** Fault Localization

**Require:** S: 実行ステートメント集合,
D: トレースと結果の集合 $\theta_0$ : 頻度の閾値

#step1 テストトレースをブロックトレース変換
**for** $(X_i, c_i) \in D$ **do**
  Convert statement trace $X_i'$ to block trace
**end for**
#step2 失敗トレース頻出パターン抽出
$F \leftarrow \{b | sup_f(b) \geqq \theta_0\}$
#step3 識別スコア計算
**for** $f \in F$ **do**
  $score_b \leftarrow calc\_score(b, D)$
**end for**
#step4 ステートメントランキング
Sort $s$ in descending order of $max\{score_b | s_i \in f\}$
Examine s from rank 1

#### step1 テストトレースを基本ブロックのトレースへ変換

テストにおけるステートメントの実行順序を記録したトレースをプログラム基本ブロックのトレースへと変換する。基本ブロック内には分岐を含まないので同じブロック内のステートメントは必ず同じ順序で実行される。そのため、ステートメントトレースをブロックトレースに変換することで実行順序の情報を損なうことなくトレースの長さを小さくすることができ。これにより、パターン抽出における計算量を削減することができる。

ブロックトレースへの変換を厳密に行うためにはプログラムの構文解析が必要となる。プログラムの構文解析は言語依存であり、言語ごとの解析ツールが必要となる。そこで実験では、トレースのシーケンスにおいて各ステートメントの前後に実行されるステートメントが常に同じであるか否かでブロックを推定する。この推定により実際より荒いブロック分割が得られるが、頻度分析における影響はない。

#### step2 失敗テストトレース頻出パターン抽出

失敗したテスト実行ブロックトレース $X_i$ に対し、頻出パターンマイニングを行う。本論文においては、抽出されたパターンにバグが含まれている必要がある。そのため、頻出パターン抽出は失敗したテストの実行トレースに対して行い、成功したテストのトレースは用いない。失敗トレースにおけるパターン $b$ の頻度を表す関数 $sup_f(b)$ は次のように表される。

$$sup_f(b) = \frac{|\{d_j \in D | c_j = 1, X_j' \in Occ(b)\}|}{|\{d_j \in D | c_j = 1\}|} \tag{4}$$

ただし $|\cdot|$ は集合'$\cdot$' の要素数を表す。頻度の閾値を $\theta_0$ とすると、$sup_f(b) \geqq \theta_0$ を満たすパターン $b$ を頻出であるとし、頻出パターンの集合を $F$ を求める。

連続頻出パターンは一般化接尾辞木を用いると総入力長の線形時間で求めることができる。

**step3 識別スコア計算**

頻出パターンマイニングで発見されたパターン集合 $F$ に含まれるパターン $b$ に対して識別スコアを計算する。本論文では指標として情報利得またはフィッシャースコアを使用し、自動バグ限局における有用性を比較する。パターン $b$ において、

$$\theta = \frac{|\{d_j \in D | X'_j \in Occ(b)\}|}{|D|}$$
$$p = \frac{|\{d_j \in D | c_i = 1|}{|D|}$$
$$q = \frac{|\{d_j \in D | c_j = 1, X'_j \in Occ(b)\}|}{|\{X'_j \in Occ(b)|}$$

とする。$\theta$ はパターン $b$ を含むトレースの割合、$p$ は失敗したテストの割合、$q$ はパターンを通過したテストにおいて失敗したテストの割合を表す。頻出パターンマイニングにおいては失敗したテストのトレースのみを使用したが、本手順においては成功したテストのトレースを含めた全てのトレースを使用する。これらの定義を用いると、(2) 式よりパターン $b$ における情報利得は

$$\begin{aligned} IG_b(C|X) = &-p\log p - (-\theta q \log q - \theta(1-q)\log(1-q) \\ &+ (\theta q - p)\log\frac{p-\theta q}{1-\theta} \\ &+ (\theta(1-q) - (1-q))\log\frac{(1-p)-\theta(1-q)}{1-\theta}) \end{aligned} \tag{5}$$

と計算され、フィッシャースコアは

$$Fr_b = \frac{\theta(p-q)^2}{p(1-p)(1-\theta) - \theta(p-q)^2} \tag{6}$$

と計算される [2]。

tarantula の指標 (1) では通過頻度が小さくても通過した時の失敗割合が高ければスコアは高くなる。情報利得やフィッシャースコアでは通過頻度が小さいパターンは分類に寄与しないため、小さくなるようにできている。

### step4 ステートメントランキング

頻出パターン $f$ の要素であるブロックのいずれかがステートメント $s$ を要素として持つとき、$s_i$ が $f$ に含まれるとし、$s_i \in f$ と表記する。各ステートメント $s_i$ のスコアを、それを含むパターンの識別スコアの最大値とし、そのスコアでランキングする。

## 5 実　　験

本章では実験に用いたデータと、各手法を比較するための評価方法について述べる。

### 5.1 対象プログラム

実験には、多くのバグ限局手法の実験において使用されている Siemens suite を使用した。このプログラムセットには、C を用いて作成された 7 つのプログラムがあり、それぞれのプログラムには、正しいバージョンと、バグを含んだ複数のバージョンが作成されている。バグを含むバージョンはそれぞれ一つのバグを含んでいる。Siemens suite に関する詳細な情報は [7] を参照されたい。

このプログラムセットには様々な種類のバグが含まれているが、コードの欠落によって引き起こされるバグ、定数および変数定義によるバグを含むバージョンは対象外とした。SBFL および提案手法ではこの種のバグの位置を推定することができないためである。また、GDB を用いて正常にトレースを取得することができないテストも対象外とした。結果として、6 プログラム、71 バージョンにおいて実験を行った。

正しいバージョンにおける各プログラムの総実行可能ステートメント数を表 2 に示す。

表 2　各プログラムの実行可能ステートメント数

| プログラム | 実行可能ステートメント数 |
|---|---|
| print tokens | 159 |
| print tokens2 | 171 |
| schedule | 128 |
| schedule2 | 117 |
| tot info | 55 |
| tcas | 112 |

### 5.2 評 価 指 標

評価指標として、EXAM を用いる。これは、バグの原因箇所を発見するまでに検査する必要のあるコードの割合を表す。この値が小さいと優れたバグ限局手法であると言える。

実験に使用したプログラムは正しいバージョンを改変することでバグを含んだバージョンを作成している。その多くはある一つのステートメントを変更したものであるが、複数ステートメントに渡って変更されているバージョンも存在する。この場合は、変更されたステートメントの集合に含まれるステートメントのうちいずれかに到達した時点でバグを発見したものとする。

場合によっては、複数の異なるステートメントに対して同じ疑惑値が与えられる。バグが含まれるステートメントと同じ疑惑値が与えられたステートメントが複数存在する時、そのステートメントの集合を $S_F$ とおくと、$\lfloor \frac{|S_F|}{2} \rfloor$ 行目でバグを含んだステートメントに到達したものとする。

### 5.3 実行トレースの収集

実行トレースの収集には GDB を用いた。収集されたトレースは計算量削減のため、ステートメントのトレースからプログラム基本ブロックのトレースに変換し、実験に使用した。基本ブロックへの変換は次のように行なった。

（1） GDB のログからソースコードにおける行番号の実行順序

を抜き出し、ステートメントトレースとする。

（2） 全テストのステートメントトレースを走査し、各ステートメントの直前、直後に実行されるステートメント集合を求める。

（3） 次の条件 a または b を満たすステートメントをブロックの入り口とする。

a プログラムの最初に実行される

b 直前に実行されるステートメントが分岐点となっている。

（4） ブロックの入り口から直後に実行されるステートメントをたどっていき、分岐点に到達したところで終了する。

（5） 入り口から走査してきたステートメント列を一つのブロックとする。

（6） 全てのステートメントトレースをもう一度走査し、ステートメントとブロックを対応づけ、ブロックトレースとする。

バグを含んだバージョンと正しいバージョンの出力を比較し、正しいバージョンの出力と異なるテストを失敗テストとした。

### 5.4 頻度の閾値設定

頻出パターンマイニングに用いる頻度の閾値 $\theta_0$ は、0.25 から 1.0 まで 0.25 刻みで設定し、それぞれの結果を比較した。その結果、情報利得では 0.25 および 0.5、フィッシャースコアでは 0.25 に設定すると最も良い結果が得られた。提案手法におけるスコアは頻度が低いとその値も低くなるため、これ以上小さい頻度をにする必要はない。これ以降の結果は $\theta_0$ を 0.25 に設定したときのものである。

## 6 結　　果

提案手法および既存手法の実験結果を示す。提案手法と既存手法 tarantula の結果を比較した結果を図 1 に示す。青の線が情報利得、赤の線がフィッシャースコアを指標とした結果であり、黒の線は既存手法 tarantula の結果である。横軸は EXAM を表し、縦軸は発見したバグの累積の割合を表している。

例えば、tarantula における横軸の値 40%、縦軸の値が 75%の点は、全体の 40%のステートメントを検査することで全てのバグのうち 75%を発見できることを表している。

この図から、検査ステートメントの割合が約 40%に至るまでは既存手法である tarantula の方がバグを早く発見できていると言える。しかし、40%以降は提案手法の方が早くバグを発見できている。

## 7 考　　察

既存手法と提案手法のバグ限局効率の違いについて議論する。

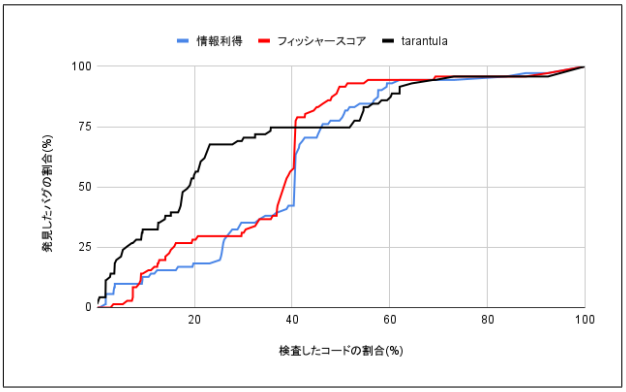


図 1　提案手法と既存手法 tarantula の比較

### 7.1 自動バグ限局指標

表 3 に、情報利得とフィッシャースコアを比較し、より早くバグを発見できたバージョン数を示す。情報利得の方が早くバグを見つけられたバージョン数は多いが、その差は全体の 7%である。よって自動バグ限局においてはこの二つの指標の有用性は同程度であると考えられる。

さらに自動バグ限局におけるスコアの有用性を比較するため、ステートメントではなく頻出パターンに対して tarantula のスコアを適用した場合を考える。パターン $b$ に対する tarantula スコアは式 (1) より

$$suspiciousness(b) = \frac{\frac{failed(b)}{totalfailed}}{\frac{passed(b)}{totalpassed} + \frac{failed(b)}{totalfailed}} \qquad (7)$$

と表される。ただし、

$$failed(b) = |\{d_j \in D | c_j = 1, X'_j \in Occ(b)\}|$$

$$passed(b) = |\{d_j \in D | c_j = 0, X'_j \in Occ(b)\}|$$

図 2 は提案手法と tarantula スコアを頻出パターンに適用した場合の結果である。この図から、提案手法と tarantula スコアの大きな差は見られない。つまり、本研究において従来手法と比べてバグ限局効率が改善しなかった理由はスコアではなくパターン抽出にあると考えられる。

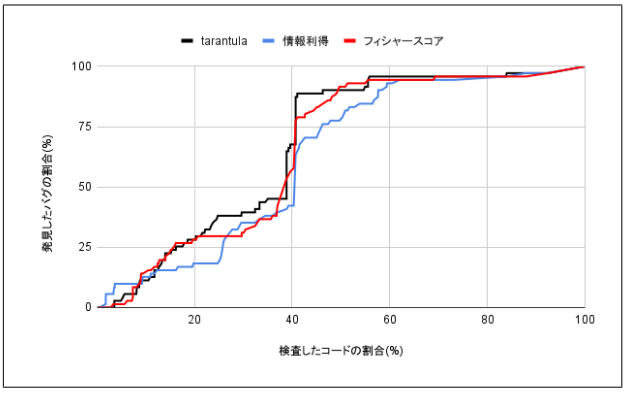


図 2　提案手法とパターンに対する tarantula スコアの比較

表 3　EXAM を用いた情報利得とフィッシャースコアの比較

| 指標 | バージョン数 |
|---|---|
| 情報利得 | 32 |
| フィッシャースコア | 27 |
| 同じ | 12 |

### 7.2　パターンの長さ

次に、パターン抽出方法のバグ限局効率への影響を考える。提案手法ではステートメントのトレースをブロックトレースに変換し、ブロックトレースからパターンを抽出している。同じパターンに含まれるステートメントは同じスコアを与えられるため、大きなパターンのスコアが高いと、検査の優先順位が同じステートメントの数が多くなってしまう。そのため、たとえバグが含まれているパターンに高いスコアが与えられたとしても、そのパターンが大きいと、EXAM の値は大きくなる。そのため、提案手法が少ないステートメント検査数でバグを発見できなかった要因として、ステートメントとパターンの粒度の違いが考えられる。

粒度の差によるバグ限局効率への影響を考察するために、best EXAM と worst EXAM の結果を図 2 に示す。best EXAM とはバグが含まれるステートメントと同じスコアのステートメントを検査する際に一番最初にバグの入ったステートメントが検査されるとした時の EXAM の値である。worst EXAM は反対にバグが入ったステートメントが一番最後に検査されるとした時の EXAM の値である。つまり、バグが含まれるステートメントと同じスコアが与えられたステートメントが他に存在する時の最良の結果と最悪の結果である。IG は情報利得、Fr はフィッシャースコア、tar は既存手法 tarantula の結果であり、それぞれの best EXAM を実線、worst EXAM を破線で表している。best と worst の差が大きいとき、同じスコアが与えられるステートメント数が多いと言える。

図 3 から、提案手法は best EXAM と worst EXAM の値の差が大きく、既存手法 tarantula はその差が提案手法より小さい。また、フィッシャースコアの best EXAM と tarantula の bestEXAM を比べると、フィッシャースコアの方が精度が良い。情報利得の場合も、tarantula と同程度の精度であることがわかる。

つまり、バグを含むパターンが高スコアになっているもののその長さがひじょうに大きいため、その中間で割合では提案手法は既存手法よりも良いバグ限局効率とならなかったと言える。抽出されるパターンが長いと、同じスコアが与えられるステートメント数が多くなるため精度が低下する。

### 7.3　バグの種類

提案手法が既存手法より早く見つけられたバグの種類について分析する。提案手法はテスト実行情報をシークエンスとして捉えていることが既存手法との相違点である。これにより、if 文による条件分岐のような実行経路制御におけるバグを見つけやすいと考えられる。if 文の条件式を通過したあと、どの分岐先に進んだかが記録されているためである。従来手法ではどの

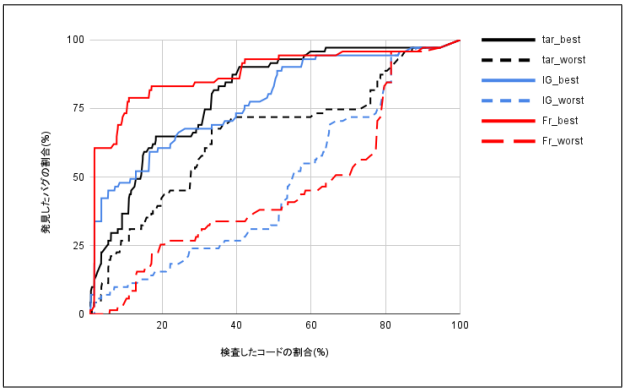


図 3　best EXAM と worst EXAM の比較

ステートメントが通過したかという記録しか使用されないため、一方の分岐のみに誤りがあっても、もう一方の分岐の正常な動作時も通るため分岐条件ステートメントの tarantula のスコアは低下する。

今回実験に用いた 71 バージョンのバグうち、20 バージョンが if 文の分岐条件の改変によるバグであった。この 20 バージョンで提案手法と既存手法を比較すると、情報利得またはフィッシャースコアを用いた手法が tarantula より優れた結果を出したバージョン数は 14 であった。

このことから、提案手法は分岐条件にバグを含む場合に既存手法よりも早くバグを発見することができると考えられる。

## 8　今後の課題

本論文では識別パターンマイニングを用いたバグ自動限局手法を提案した。テスト実行情報をシークエンスデータとして扱うことによるバグ限局効率向上が期待された。

しかし、提案手法では多くのバージョンで既存手法と比べたバグ限局効率の向上はなされなかった。そのため、バグ限局効率改善のためパターンステートメントのスコアの付け方を改良する必要がある。

提案手法はバグを含むパターンに高いスコアを付けるが、その長さが大きいため、検査範囲を絞り込むことができていない。そこで、バグの含まれる可能性の低いステートメントの除外や、ステートメントのスコアをそれを含むパターンのスコアの最大値とするのではなく平均値にするような、ステートメントのスコアに差がつくスコアの付けによってバグ限局効率の改善が期待できる。

## 文　献

[1] R. Abreu, P. Zoeteweij, et al. An evaluation of similarity coefficients for software fault localization. 2006 12th Pacific Rim Int ' l Symposium on Dependable Computing (PRDC ' 06), pp. 39–46, 2006.

[2] H. Cheng, X. Yan, J. Han, and C. Hsu. Discriminative frequent pattern analysis for effective classification. In ICDE, 2007

[3] B. Hailpern and P. Santhanam. Software debugging, testing, and verification. IBM Systems Journal, 41(1):4–12, 2002.

[4] Jones, J.A., Harrold, M.J., Stasko, J.: Visualization of test

information to assist fault localization. In: Proceedings of the 24th International Conference on Software Engineering, 2002.

[5] B. Liu, W. Hsu, and Y. Ma. Integrating classification and association rule mining. In Proc. of KDD, pages 80–86, 1998.

[6] H. Lodhi, C. Saunders, J. Shawe-Taylor, N. Cristianini, and C. Watkins. Text classification using string kernels. Journal of Machine Learning Research, 2:419– 444, 2002.

[7] M. Hutchins, H. Foster, T. Goradia and T. Ostrand, Experiments on the effectiveness of dataflow- and controlflow-based test adequacy criteria, Proceedings of the 16th International Conference on Software Engineering, Sorrento, Italy, pp. 191–200, 1994

[8] S. Nessa, M. Abedin, W. Eric Wong, L. Khan, and Y. Qi, " Fault localization using N-gram analysis, " in Proc. Int. Conf. Wireless Algorithms, Syst, pp. 548–559, 2009.

[9] W. E. Wong and Y. Qi, "BP Neural Network-based Effective Fault Localization, " International Journal of Software Engineering and Knowledge Engineering 19(4):573-597, June 2009.

[10] W. E. Wong, V. Debroy, Y. Li, and R. Gao, "Software fault localization using DStar (D*) " in Proceedings of 6th International Conference of Software Security Rel., Washington, D.C., USA, pp. 21–30, 2012.

[11] D. Lo, H. Cheng, J. Han, "Classification of Software Behaviors for Failure Detection: A Discriminative Pattern Mining Approach,"Proceedings of the 15th ACM SIGKDD, pp. 557-566, 2009.

[12] H. Agrawal, J. R. Horgan, S. London, and W. E. Wong, " Fault Localization using Execution Slices and Dataflow Tests, " in Proceedings of the 6th IEEE International Symposium on Software Reliability Engineering, pp. 143-151, Toulouse, France, October 1995.

[13] W. E. Wong, Y. Shi, Y. Qi, and R. Golden, "Using an RBF Neural Network to Locate Program Bugs, " in Proceedings of the 19th IEEE International Symposium on Software Reliability Engineering, pp. 27-38, Seattle, Washington, USA, November 2008.