# 修正ソースコードの特徴が自動バグ修正に与える影響の分析

中村 司[1,a)]　池田 翔[1,b)]　亀井 靖高[1,c)]　佐藤 亮介[1,d)]　鵜林 尚靖[1,e)]

**概要**：自動バグ修正手法の 1 つに DeepFix がある．これは，C 言語プログラムをトークン列として Seq2Seq モデルで学習し，文法エラーに特化した修正を行う手法である．しかし，他の自動バグ修正手法と同様に，これですべてのバグが解決されるわけではなく，さらにソースコードのどのような性質が修正を妨げているのかも明らかになっていない．そこで本研究ではソースコードの特徴である，**(RQ1)** 行数，**(RQ2)** トークン数，**(RQ3)** 複雑度の 3 つが自動バグ修正手法の修正結果に影響を与えるのか，調査する．そのために，DeepFix によって修正できたソースコードと，そうでないソースコードとをそれぞれの観点から比較し，修正に影響を与える特徴について考察する．C 言語プログラムの入門用講義で学生が書いたソースコードに対して実験を行なったところ，修正できたソースコードの方が，トークン数や複雑度が小さいことが明らかになった．これにより，DeepFix にとって修正しやすいソースコードの特徴が明らかになり，自動バグ修正の支援ができると考える．

**キーワード**：自動バグ修正，DeepFix，ソースコードメトリクス，C 言語

# Studies on how characteristics of target source code affect automatic program repair

## 1. はじめに

ソフトウェア開発において，もっともコストのかかる作業の 1 つがデバッグである．デバッグには開発コスト全体の 50 %以上が費やされることもあるが，これは主に，バグを特定し除去するために人の手による尽力が必要となるからである．そのため，デバッグにかかるコストを少しでも軽減するための取り組みとして，自動バグ修正の研究が現在盛んに行われている．自動バグ修正とは，人の手ではなくコンピュータによってプログラムのエラーを解消する技術のことである [5].

自動バグ修正のツールの 1 つとして，DeepFix[6] がある．これは，C 言語プログラムを学習し，C 言語の文法的なエラーに特化した修正を行う手法である．最初に提案された論文の中で行われた実験では，27%のプログラムに対してすべてのエラーを修正し，19%のプログラムに対して一部のエラーを修正することができた，という結果が得られている．

しかし，DeepFix による修正の成功率を向上させる試みはあまり行われていない．提案論文では，修正箇所が長くなるほど修正が難しくなると述べられているが，修正の対象となるソースコードの特徴に着目した考察はされていない．そのため，DeepFix による修正に影響を与える要因について，深く調査されていないのが現状である．

本論文では，修正するソースコードのいくつかの特徴について，修正の成否との関係を調査した．インド工科大学および九州大学の学生が書いたソースコードのデータを DeepFix に適用し，文法エラーを修正できたソースコードとできなかったソースコードに有意な差があるのかを検定した．これによって DeepFix で修正しやすいソースコードの性質を明らかにし，自動バグ修正の支援をすることが，本研究の目的である．

[1] 九州大学
Kyushu University
a) t.nakamura@posl.ait.kyushu-u.ac.jp
b) ikeda@posl.ait.kyushu-u.ac.jp
c) kamei@ait.kyushu-u.ac.jp
d) sato@ait.kyushu-u.ac.jp
e) ubayashi@ait.kyushu-u.ac.jp

ⓒ 2020 Information Processing Society of Japan

## 2. 背景

### 2.1 自動バグ修正

デバッグにかかるコストを少しでも軽減するため，自動バグ修正の研究が近年盛んに行われている．自動バグ修正の中には，論理エラーを修正するための試みと，文法エラーを修正するための試みがある．論理エラーとは，間違った分岐条件などにより想定と異なる動作をしてしまうエラーである．また，文法エラーとは，変数宣言の欠如のような，プログラミング言語の持つ文法規則に従っていないことによるエラーである．

本研究では，文法エラーの修正に着目する．文法規則のようなプログラミング言語特有の性質は，特にプログラミング初学者にとっては，誤りやすく訂正しにくいものである [4]．文法エラーを自動的に修正することにより，デバッグにかかるコストをより大きく軽減することができると考えられる．

### 2.2 DeepFix

DeepFix は，2017 年に Gupta らによって提案された, C 言語の文法エラーを修正することに特化したツールである [6]．これは，C 言語で書かれたソースコードをトークン列として Seq2Seq モデル [11] に学習させ，エラーを含むソースコードに対して正しいトークン列を推論することで, 文法エラーの修正を行う．そのため，学習用データを生成するために，エラーを含まないソースコードも入力データに含まれる．

DeepFix による学習および修正を行うためにトークン化される前のソースコードを**ソースコード 1** に，トークン化されたソースコードを**ソースコード 2** にそれぞれ示す．各トークンはその型とともに行番号の後ろに並べられるが, 学習や修正の性能を高めるため，文字列や数値などの，文法的に意味のないデータはその型のみが保持される．また，トークン化されたソースコードを 1 行ごとに分割し，文字列の羅列として扱うことで，さらに学習や修正を行いやすくしている．学習のために用意された，文法エラーを含まないソースコードは，このようにしてトークン化されたのちに文法エラーを埋め込まれることで学習用データとなる．

DeepFix は，「多くの場合，プログラムのエラーは，エンジニアの不注意や言語に対する知識不足などによって生じる文法エラーである．」という構想のもとで開発されたものであるため，論理的なエラーは修正の対象としていない．提案論文中の実験では全体の 27 %のプログラムを完全に，18 %のプログラムを部分的に修正することに成功している．

**ソースコード 1** トークン化される前のソースコードの一部

```
2    printf("hello,␣world!¥n");
```

**ソースコード 2** トークン化されたソースコードの一部

```
2 ~ _<APIcall>_printf _<op>_( _<string>
    _ _<op>_) _<op>_;
```

### 2.3 DeepFix による修正

DeepFix によって行われた修正の実例を**ソースコード 3** に示す．**ソースコード 3** では，12 行目の行末にセミコロンが欠けており，エラーメッセージにもそのような旨の記述がされていた．これに対し DeepFix による修正を行ったものが**ソースコード 4** である．12 行目を見ると修正前に欠けていたセミコロンが付与されていることがわかる．ただし，トークン化の段階で捨象された数値および文字列，インデント，空行のデータは手作業による復元を行なった．

一方，**ソースコード 5** も，同様に 9 行目の行末にセミコロンが欠けていることによるエラーである．エラーの数や種類，修正方法は**ソースコード 3** と変わりがないように予想されるが，DeepFix はこれに対して正しい修正を行うことができなかった．

これらのことから，エラーの数や種類以外にも，DeepFix による自動バグ修正に影響を与える要因が存在することが予想される．

### 2.4 研究の動機

DeepFix の提案論文では，修正の成否に関する考察は少なかった．エラーを解消するために必要な修正パッチのトークン数が多くなるほど修正が難しくなる，と述べられていたが，修正の対象となるソースコードについては分析がされていない．

本研究では DeepFix による修正が難しいソースコードの特徴を明らかにする．これにより，その特徴に対するリファクタリングを修正前にあらかじめ行うことで, DeepFix による修正を支援することができると考えられる．

DeepFix による修正に影響を与えるソースコードの性質を調べるため，本研究では次の 3 つの RQ について調査した．

**(RQ1)** 修正ソースコードの行数は修正に影響を与えるのか

**(RQ2)** 修正ソースコードのトークン数は修正に影響を与えるのか

**(RQ3)** 修正ソースコードの複雑度は修正に影響を与えるのか

ⓒ 2020 Information Processing Society of Japan

**ソースコード 3** 修正できたプログラムの例 (修正前)

```
1  #include <stdio.h>
2
3  int main( void )
4  {
5      float a, b;
6      double c;
7      int d;
8
9      a=62.5;
10     b=23.3;
11     c=a*b;
12     d=9
13
14     d*=5;
15     printf("%d\n", d);
16     d/=2;
17     printf("%d\n", d);
18     return 0;
19 }
```

**ソースコード 4** 修正できたプログラムの例 (修正後)

```
1  #include <stdio.h>
2
3  int main ( void )
4  {
5      float a , b ;
6      double c ;
7      int d ;
8
9      a = 62.5 ;
10     b = 23.3 ;
11     c = a * b ;
12     d = 9 ;
13
14     d *= 5 ;
15     printf ( "%d\n" , d );
16     d /= 2 ;
17     printf ( "%d\n" , d );
18     return 0 ;
19 }
```

**ソースコード 5** 修正できなかったプログラムの例

```
1  #include <stdio.h>
2  #include <math.h>
3
4  int main(){
5      int a,b;
6      printf("整数値を入力してください:");
7      scanf("%d",&a);
8
9      b = a % 2
10     if(b==0){
11         printf("偶数です");}
12     if(b==1){
13     printf("奇数です");}
14     return 0;
15 }
```

**表 1** 各データセットに含まれるソースコードの件数

| データセット | 問題数 | エラーあり | エラーなし | 合計 |
|---|---|---|---|---|
| IITK | 93 | 6,978 | 46,500 | 53,478 |
| ED | 47 | 3,157 | 7,339 | 10,496 |
| LA | 21 | 1,861 | 3,936 | 5,797 |

## 3. データセット

本研究では，次に示す 3 つのデータセットを使用した．各データセットに含まれるソースコードの件数など，数値データについては**表 1** に示す．

### 3.1 IITK(Indian Institute of Technology Kanpur)

DeepFix を提案した論文で用いられたデータセットである．インド工科大学カーンプル校において Prutor[2] と呼ばれる個別指導システムを用いて収集されたものであり，C 言語プログラミングの入門用講義で課された 93 の課題と，それに対する学生の解答をまとめたものとなっている．

### 3.2 ED(Education)

2017 年に九州大学で開講された，基幹教育科目の講義によるものである．講義で課された C 言語プログラム初級者向けの課題に対し，教育学部の学生が解答として書いたソースコードを，まとめたものとなっている．九州大学では，TERA-TERM を通じて接続できる作業サーバーが学生に提供されている．そのサーバーには GCC がインストールされているため，学生はコンパイラをインストールすることなくコンパイル作業に取り組める．本データセットは，そのサーバーから SFTP（SSH File Transfer Protocol）を通じてコンパイル時のログを収集したものである [3]．

### 3.3 LA(Law)

九州大学の基幹教育科目の講義によるものである．ED データセットと異なり 2015 年に開講された際のものであり，受講した学生も法学部の学生が主となっている．

## 4. 実験と結果

実験手法の概要を**図 1** に示す．

**手順 1.** 各データセットに対して，エラーを含まないソースコードから学習データを生成し，エラーを含むソースコードに修正を適用した．この際，IITK データセットにおいてはトークン数が 100 以上 400 以下のもの，ED および LA データセットにおいてはトークン数が 25 以上 400 以下のソースコードに限定した．この選別は，DeepFix の提案論文で述べられていた実験の設定を踏襲したものであるが，ED および LA データセットについてはデータの大きさを確保するために下限を小さく設定した．

**手順 2.** DeepFix は，生成した修正パッチだけでなく，そ

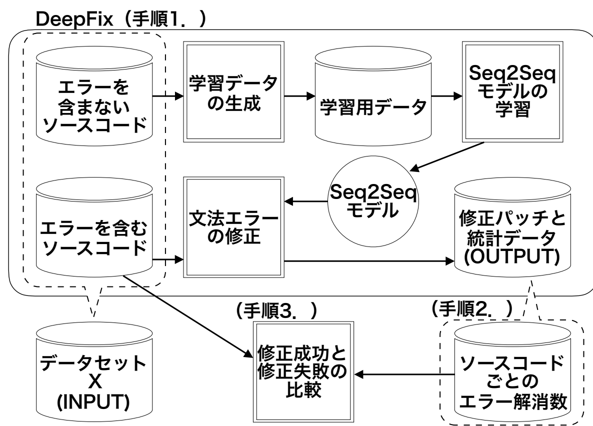



**図 1** DeepFix の修正過程と本研究のアプローチ

れぞれの修正ソースコードに対して解消することのできた文法エラーの数も出力する．これをもとに，修正を施す前のソースコードを，修正に成功したソースコードと失敗したソースコードに分類した．ただし，ここでは 1 つ以上エラーを解消することができたものを修正成功，1 つもエラーを解消できなかったものを修正失敗と定義する．

**手順 3．** DeepFix による自動バグ修正とその対象となるソースコードの特徴との関係性を調べるため，ソースコードの規模と複雑さという 2 つの方面から，次のような分析を行なった．

## 4.1 (RQ1) 規模-行数

### 4.1.1 動機

DeepFix による自動バグ修正が難しいソースコードの特徴として，ソースコードの規模が大きいということが考えられる．そこで，DeepFix が修正に成功したソースコードの集団と，修正に失敗したソースコードの集団に対し，行数の比較を行った．

### 4.1.2 アプローチ

DeepFix による修正を施す前のソースコードに対し，構造解析ツール Understand [*1] を用いていくつかのメトリクスを測定し，そのうちの 1 つである行数[*2]を比較した．また，帰無仮説および有意水準を

**帰無仮説** このデータセットにおいて，修正に成功したソースコードと失敗したソースコードは，同じ母集団から抽出されたものである．

**有意水準** $\alpha = 0.05$

と設定してマン・ホイットニーの U 検定を行い，比較によって見られた差が有意なものであるのかをデータセットごとに評価した．ただし，このとき検定の多重性に関する問題を解消する必要がある．本研究では，検出力を落とさないためにホルム法を採用し，$p$ 値を有意水準と比較する前に，有意水準にはホルム法による補正をかけた．

### 4.1.3 結果

データセットごとに修正に成功したソースコードと失敗したソースコードに対して行数を測定した結果の比較を**図 2** に示す．さらに，それぞれのデータセットに対してマンホイットニーの U 検定を行なったところ，$p$ 値はそれぞれ

**IITK** $pvalue = 2.08 \times 10^{-6}$
**ED** $pvalue = 8.11 \times 10^{-4}$
**LA** $pvalue = 1.07 \times 10^{-4}$

となった．これらをホルム法による補正をかけた有意水準と比較すると，

**IITK** $pvalue = 2.08 \times 10^{-6} < \alpha/3$
**LA** $pvalue = 1.07 \times 10^{-4} < \alpha/2$
**ED** $pvalue = 8.11 \times 10^{-4} < \alpha$

となり，3 つすべてのデータセットにおいて帰無仮説が棄却された．これにより，3 つのデータセットすべてにおいて，修正されるソースコードの行数について有意差が認められた．

### 4.1.4 考察

3 つすべてのデータセットにおいて有意差が認められたため，修正に失敗したソースコードの方が行数が大きいと言える．しかし，例えば次節で考察するトークン数のような，行数と相関のある他のデータから影響を受けていることも考えられる．

## 4.2 (RQ2) 規模-トークン数

### 4.2.1 動機

RQ1 では，修正するソースコードの規模を表す指標としてソースコードの行数を採用した．しかし，実際のソースコードには空行やコメントなどプログラム実行上では意味を持たない行も含まれる．そこで，同様に規模が反映される指標のうち，よりソースコードの意味に着目したものとして，本節ではトークン数を採用した．

### 4.2.2 アプローチ

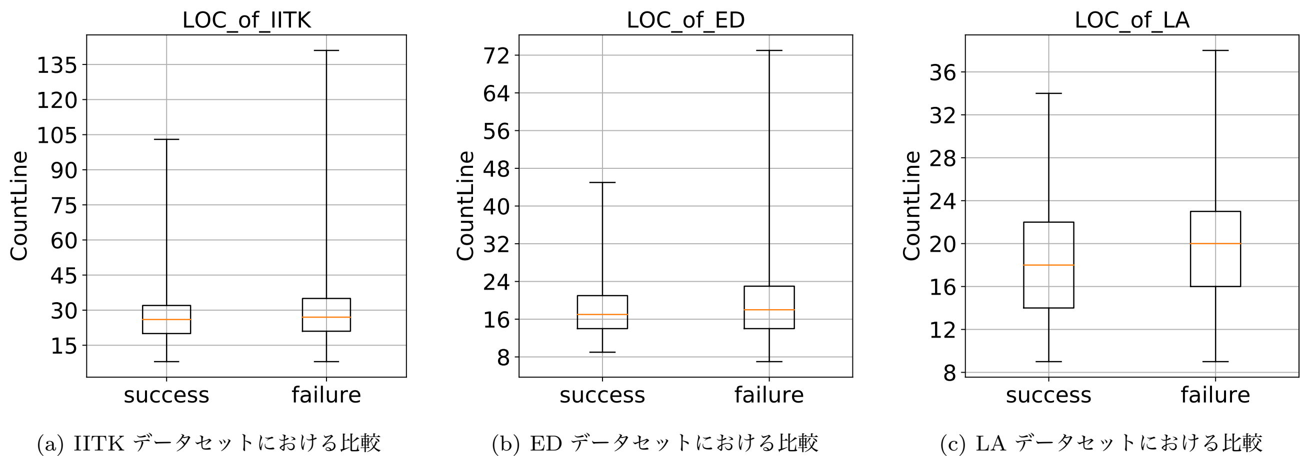
DeepFix では，学習・修正のためのソースコードはトークン化された状態で扱われる．このとき，トークン化されたソースコードとそのトークン数は出力データベースに保存されるため，これらを抽出して統計的な処理を行なった．また，それぞれのデータセットに対し，ホルム法による補正をした U 検定を RQ1 と同様に行なった．

### 4.2.3 結果

修正に成功したソースコードと失敗したソースコードに対してデータセットごとにトークン数を測定した結果の比較を**図 3** に示す．さらに，それぞれのデータセットに対して RQ1 と同様の U 検定を行なったところ，$p$ 値はそれぞれ

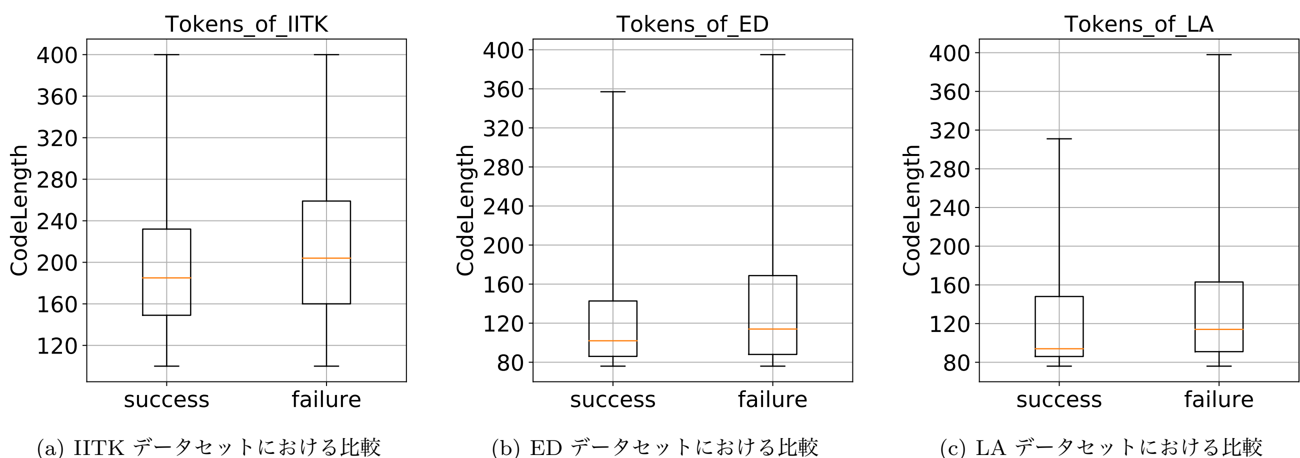
**IITK** $pvalue = 2.11 \times 10^{-25}$
**ED** $pvalue = 2.07 \times 10^{-4}$

[*1] http://understand.techmatrix.jp/
[*2] Understand における表記は CountLine．コメントのみの行や空白の行も行数に含む．




(a) IITK データセットにおける比較　(b) ED データセットにおける比較　(c) LA データセットにおける比較

**図 2**　修正に成功/失敗したソースコードにおける行数の比較




(a) IITK データセットにおける比較　(b) ED データセットにおける比較　(c) LA データセットにおける比較

**図 3**　修正に成功/失敗したソースコードにおけるトークン数の比較

**LA**　$pvalue = 1.21 \times 10^{-6}$

となった．これらをホルム法による補正をかけた有意水準と比較したところ，3 つのデータセットすべてにおいて帰無仮説が棄却された．これにより，3 つのデータセットすべてにおいて，修正されるソースコードのトークン数について有意差が認められた．

#### 4.2.4　考察

3 つ全てのデータセットで有意差が認められており，修正に失敗したソースコードの方がトークン数が大きいと言える．DeepFix ではソースコードはトークン化された形で扱われるため，行数による修正への影響は，トークン数を介した間接的なものであったことが考えられる．

**ソースコード 6** は，行数および文法エラーの内容に関しては**ソースコード 5** と概ね一致するが，DeepFix によって文法エラーを修正することができた．要因の 1 つとして，変数の使い方や分岐条件の違いにより，トークン数が小さくなっていることが考えられる．

**ソースコード 6**　トークン数の小さいプログラムの例

```
1  #include <stdio.h>
2  #include <math.h>
3
4  int main (){
5      int a;
6
7      printf("整数を入力してください=");
8      scanf("%d",&a);
9
10     if(a%2==0){
11         printf("%d は偶数です \n",a)
12     } else {
13         printf("%d は奇数です \n",a)
14     }
15     return 0
16     }
```

### 4.3　(RQ3) 複雑度

#### 4.3.1　動機

RQ1 および 2 では，修正するソースコードの規模に着目した分析を行なった．しかし，自動バグ修正の障害となり

うるものとして，規模以外にもソースコードの複雑さが予想される．そこで，本節ではソースコードの複雑さに着目した分析を行なった．

ここでは，ソースコードの複雑さを表す指標として循環的複雑度，Halstead のソフトウェアサイエンス [7] の 2 つを採用した．これにより，動作をフローチャートで表した際の分岐の多さ，ソースコードに含まれるオペランドやオペレータの多様さから，それぞれソースコードを評価する．

#### 4.3.2 アプローチ

循環的複雑度については RQ1 と同様に Understand による測定[*3]を行い，Halstead のソフトウェアサイエンスについては commentedCodeDetector [*4] による測定を行なった．それぞれの指標に対して，RQ1 および RQ2 と同様の U 検定を行なった．

#### 4.3.3 結果

実験結果を**図 4** および**図 5** にそれぞれ示す． また U 検定における $p$ 値は，循環的複雑度については

**IITK** $pvalue = 1.93 \times 10^{-2}$

**ED** $pvalue = 3.90 \times 10^{-1}$

**LA** $pvalue = 5.89 \times 10^{-6}$

となり，Halstead のソフトウェアサイエンスについては

**IITK** $pvalue = 1.60 \times 10^{-16}$

**ED** $pvalue = 8.83 \times 10^{-3}$

**LA** $pvalue = 1.07 \times 10^{-4}$

となった．これらをホルム法による補正をかけた有意水準と比較したところ，循環的複雑度については IITK および LA データセットにおいて，Halstead のソフトウェアサイエンスについては 3 つのデータセットすべてにおいて，それぞれ帰無仮説が棄却された．これにより，循環的複雑度については IITK および LA データセットにおいて有意差が認められ，Halstead のソフトウェアサイエンスについては 3 つのデータセットすべてにおいて有意差が認められた．

#### 4.3.4 考察

循環的複雑度について，IITK および LA データセットの 2 つにおいては有意差が認められた．また，ED データセットにおいては有意差こそ認められなかったが，図 4 によるとデータの範囲に差があり，循環的複雑度が 8 より大きいものは修正できていないことがわかる．これらのことから，循環的複雑度も修正に失敗したソースコードの方が大きいと言える．

また，Halstead のソフトウェアサイエンスについては，3 つすべてのデータセットにおいて有意差が認められた．すなわち，Halstead のソフトウェアサイエンスも修正に失敗したソースコードの方が大きいと言える．これは，ソースコードに含まれるトークンの種類が多くなることにより，正しいソースコードの推論が難しくなるためであると考えられる．

*3 Understand における表記は AvgCyclomatic であり，各関数の循環的複雑度をソースコードごとに平均したメトリクスである．
*4 https://github.com/dborowiec/commentedCodeDetector

### 4.4 結果のまとめと考察

修正ソースコードの行数，トークン数，複雑度の 3 つすべてにおいて，修正に失敗したソースコードの方が大きいという結果が得られた．そのなかでも特にトークン数と Halstead のソフトウェアサイエンスについては，DeepFix の構造上の特徴からも，修正に失敗したソースコードの特徴であると考えられる．

## 5. 関連研究

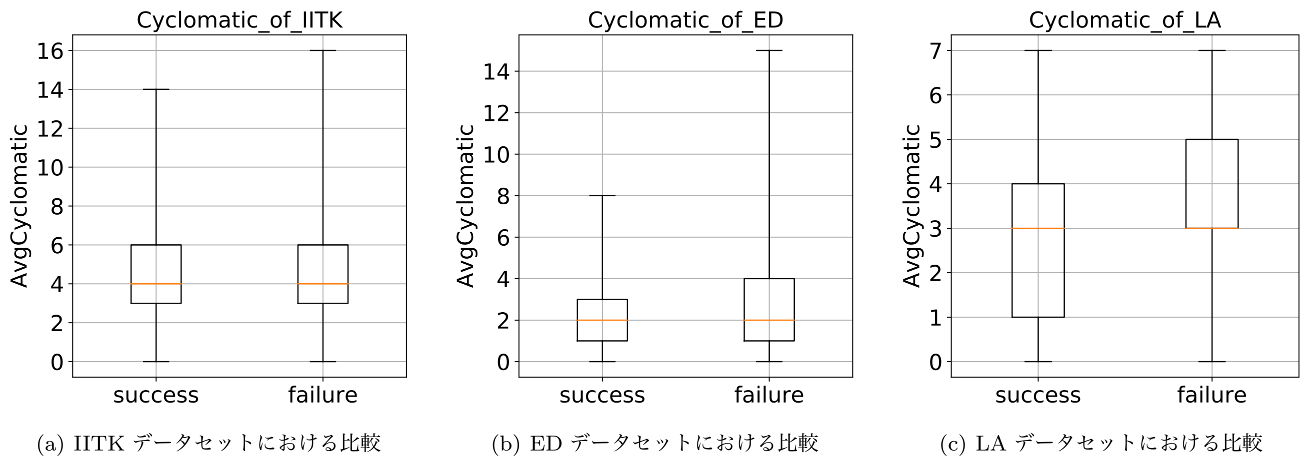
自動バグ修正の研究は近年盛んに行われており，DeepFix 以外にも様々な手法が提案されている．

Long らの提案した Prophet は，エラーのあるプログラムに対して修正パッチを生成する [9]．まず，テストスイートに含まれる入力でプログラムを実行し，エラーの原因となり得る命令を絞り込む．続いて，絞り込まれたそれぞれの命令に対応する修正パッチを生成し，それらを修正が期待できる順に序列づける．この序列づけには，オープンソースソフトウェアの修正履歴を学習したモデルが利用されている．この修正パッチの序列から，テストスイートに対して正しい出力ができたものだけを選んで出力する．提案論文中の実験では，実際のエラー 69 件に対して適用した結果，19 件に対して正しい修正パッチを出力することに成功している．

また，本研究のように，プログラムの分析や改善を行うツールの持つ, 性質や特性を調査した研究も多数存在する．

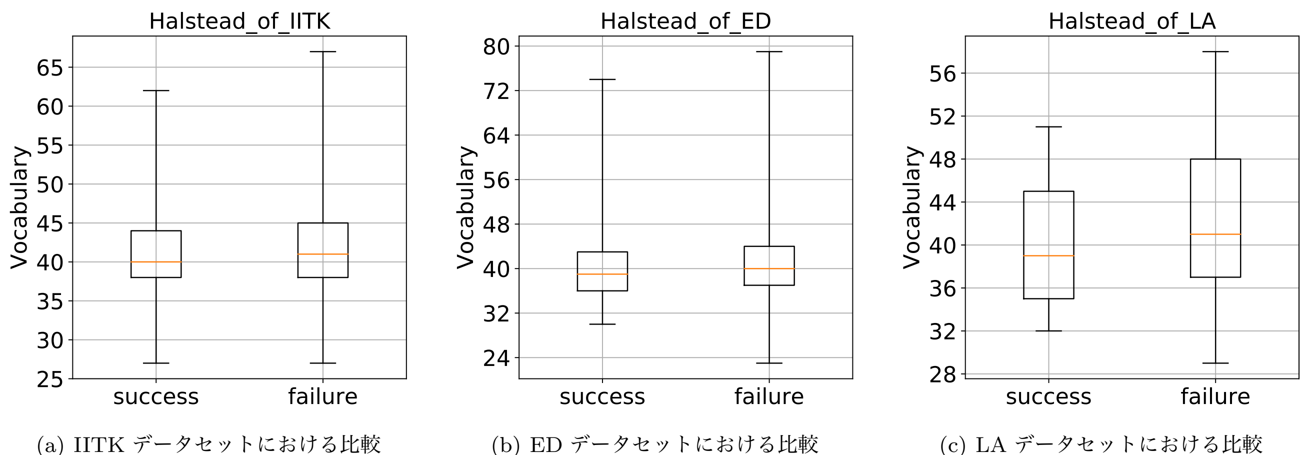
Yang らは，コードカバレッジツールの信頼性を高めるため，コードカバレッジツールの性能を効率よく評価するための手法を提案した [13]．彼らは，コードカバレッジツールの検証を効率化するために **(1)** Csmith[12] を用いてテストスイートの重複を防ぐこと，**(2)** カバレッジレポートに現れやすい矛盾を分類しテストの数を絞ること，**(3)** バグを含んでいるコードカバレッジツールを自動的に特定することの 3 点が必要だと考え，それらを行うツールとして C2V を実装した．さらに，現在広く使われているコードカバレッジツールとして gcov と llvm-cov を挙げ，これらに C2V を適用することで，gcov においては 42 件，llvm-cov においては 28 件のバグをそれぞれ確認した．このことから，コードカバレッジツールは想定されるよりも信頼性が高くない，と結論づけている．

Papadakis らは，ミューテーションスコアとバグ検出との間に本当に相関があるのかを調査した [10]．ミューテーションスコアとバグ検出の関係について調べた先行研究は数多く存在していたが，テストスイートの大きさによる影響を考慮に入れて調査した研究は少なかった．そこで，CoREBench[1] と Defects4J[8] の 2 つのデータセットを使




(a) IITK データセットにおける比較　(b) ED データセットにおける比較　(c) LA データセットにおける比較

**図 4**　修正に成功/失敗したソースコードにおける循環的複雑度の比較




(a) IITK データセットにおける比較　(b) ED データセットにおける比較　(c) LA データセットにおける比較

**図 5**　修正に成功/失敗したソースコードにおける Halstead のソフトウェアサイエンスの比較

用し，テストスイートの大きさを固定した場合としなかった場合で，それぞれ相関の分析を行なった．その結果，テストスイートの大きさを固定するとミューテーションスコアとバグ検出の相関が小さくなることが明らかになり，この相関は単にテストスイートの大きさによる効果だったと結論づけている．

これらに対し，提案されて以来あまり分析の進んでいなかった DeepFix について取り上げた点が，本研究の特徴である．

## 6.　妥当性への脅威

**内的妥当性．** 本研究で利用したデータセットのうち，ED と LA の 2 つについては，九州大学で開講された基幹教育科目の講義に基づくものであった．これらはコンパイル時の記録を集めたものであるが，学生が同じソースコードを繰り返しコンパイルしていることもある．そのため，同じ内容のソースコードが複数存在している場合がある．このようにして重複して存在するソースコードが，極端に修正しやすいものであったり極端に修正しにくいものであったりすると，実験結果に影響を与えている可能性がある．

また，本研究ではトークン数が極端に大きいソースコードや極端に小さいソースコードはデータセットから除外して実験を行なったため，これらを含めると結果が変わる可能性がある．

**外的妥当性．** 本研究で利用したデータセットは，すべて講義の課題に対する学生の解答をまとめたものであった．そのため，学生の習熟度の影響を受けることが考えられ，どのような学生の集団に対しても必ず同様の結果が得られるとは言い切れない．

**構成概念的妥当性．** 本研究では，DeepFix によって修正できなかったソースコードの特徴を，ソースコードの規模と複雑さ (量と質) という 2 面から予測した．しかし，RQ として挙げた **(RQ1)** 行数，**(RQ2)** トークン数，**(RQ3)** 循環的複雑度の 3 つで規模と複雑さのあらゆる側面を検証できているとは言い切れず，調査すべき性質が他にも存在する可能性がある．

## 7.　まとめ

本研究では，DeepFix による自動バグ修正に対して，その対象となるソースコードの持つ性質がどのように影響し

ⓒ 2020 Information Processing Society of Japan

ているのかを調査した．まずはソースコードの規模に着目し，ソースコードの行数とトークン数から分析を行なった．どちらも修正を難しくしている要因ではあるものの，より直接的に作用しているのはトークン数であることが明らかになった．続いてソースコードの複雑さに着目し，循環的複雑度や Halstead のソフトウェアサイエンスについて分析を行なった．U 検定の結果やデータの分布から，ソースコードの複雑さも修正を難しくしていることが明らかになった．

今後の課題として，同様の調査を他のデータセットに対しても行うことで分析をより正確にすることや，本研究の結果を利用してソースコードを DeepFix にとって修正しやすい形式へと整形するツールを実装する試みが挙げられる．

**謝辞** 本研究の一部は JSPS 科研費 JP18H04097・JP18H03222・JP19H04086 の助成を受けた．

## 参考文献

[1] Böhme, M. and Roychoudhury, A.: CoREBench: studying complexity of regression errors, *International Symposium on Software Testing and Analysis, ISSTA 2014, San Jose, CA, USA - July 21 - 26, 2014*, pp. 105–115 (2014).

[2] Das, R., Ahmed, U. Z., Karkare, A. and Gulwani, S.: Prutor: A System for Tutoring CS1 and Collecting Student Programs for Analysis, *CoRR*, Vol. abs/1608.03828 (2016).

[3] Fu, X., Shimada, A., Ogata, H., Taniguchi, Y. and Suehiro, D.: Real-time learning analytics for C programming language courses, *Proceedings of the Seventh International Learning Analytics & Knowledge Conference, Vancouver, BC, Canada, March 13-17, 2017*, pp. 280–288 (2017).

[4] Fu, X., Yin, C., Shimada, A. and Ogata, H.: Error log analysis in c programming language courses, *Doctoral Student Consortium (DSC) - Proceedings of the 23rd International Conference on Computers in Education, ICCE 2015*, pp. 641–650 (2015).

[5] Gazzola, L., Micucci, D. and Mariani, L.: Automatic Software Repair: A Survey, *IEEE Trans. Software Eng.*, Vol. 45, No. 1, pp. 34–67 (2019).

[6] Gupta, R., Pal, S., Kanade, A. and Shevade, S. K.: DeepFix: Fixing Common C Language Errors by Deep Learning, *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence, February 4-9, 2017, San Francisco, California, USA*, pp. 1345–1351 (2017).

[7] Halstead, M. H. M. H.: *Elements of software science*, New York : Elsevier (1977). "Elsevier computer science library.".

[8] Just, R., Jalali, D. and Ernst, M. D.: Defects4J: a database of existing faults to enable controlled testing studies for Java programs, *International Symposium on Software Testing and Analysis, ISSTA 2014, San Jose, CA, USA - July 21 - 26, 2014*, pp. 437–440 (2014).

[9] Long, F. and Rinard, M.: Automatic patch generation by learning correct code, *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, pp. 298–312 (2016).

[10] Papadakis, M., Shin, D., Yoo, S. and Bae, D.: Are mutation scores correlated with real fault detection?: a large scale empirical study on the relationship between mutants and real faults, *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, pp. 537–548 (2018).

[11] Sutskever, I., Vinyals, O. and Le, Q. V.: Sequence to Sequence Learning with Neural Networks, *Advances in Neural Information Processing Systems 27: Annual Conference on Neural Information Processing Systems, NIPS 2014, December 8-13 2014, Montreal, Quebec, Canada*, pp. 3104–3112 (2014).

[12] Yang, X., Chen, Y., Eide, E. and Regehr, J.: Finding and understanding bugs in C compilers, *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*, pp. 283–294 (2011).

[13] Yang, Y., Zhou, Y., Sun, H., Su, Z., Zuo, Z., Xu, L. and Xu, B.: Hunting for bugs in code coverage tools via randomized differential testing, *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*, pp. 488–498 (2019).

ⓒ 2020 Information Processing Society of Japan