修士学位論文

題目

プログラミング教育における提出プログラム間の機能差検出
-自動テスト生成と経路ベクトルを用いて-

指導教員

楠本 真二 教授

報告者

出田 涼子

令和 4 年 2 月 2 日

大阪大学 大学院情報科学研究科

コンピュータサイエンス専攻

令和 3 年度 修士学位論文

プログラミング教育における提出プログラム間の機能差検出
-自動テスト生成と経路ベクトルを用いて-

出田 涼子

内容梗概

プログラミング教育における一つの実施形態として，ソフトウェアテストを用いた方法が採用されつつある．この方法では，プログラムの仕様を単体，あるいは結合テストとして用意しておき，学生はそのテスト通過を目指してプログラムを作成する．客観的なプログラム仕様の確保，及び動作確認の自動化等様々な利点がある一方で，学生個々の独自仕様や機能拡張といった工夫の検出は難しい．さらには，与えられたテストには通過するが，その解決方法が十分に一般化されていないケースの検出も困難である．本研究の目的は，テストを用いたプログラミング教育における学生プログラム間の機能差検出の支援である．本稿では，この目的に対する 2 つの課題を整理し，その片方の実現方法を提案する．もう片方の課題の実現方法は実題材に対する実験において検討する．この提案手法では，学生のプログラムに対し自動テスト生成を適用し，生成テストを他の学生プログラムに対して相互に実行する．さらにテスト実行時の経路情報に基づき，テストをクラスタリングすることで，機能差検出能力を持つテストの集合を得る．ある大学の実題材への適用実験を行った結果，提案手法による機能差検出能力を持つテストの生成，及び同じ意味のテスト同士のクラスタリングの実現を確認した．

主な用語

自動評価，自動テスト生成，プログラミング教育，テスト実行経路

目次

図目次

表目次

1　はじめに

ソフトウェアテストを活用したプログラミング教育方法（以降，テストベース教育）が数多く提案されている [1, 2, 3, 4]．この手法では，課題の仕様を単体あるいは結合テストとして定義しておき，学生はその事前テストの通過を目指してプログラムを作成する．これにより，客観的な課題仕様の確保，及び課題達成確認の自動化等，学生と教員の両者に対する様々な利点が得られる．また，テスト駆動開発 [5] や継続的インテグレーション [6] をはじめとするアジャイルな開発手法との親和性も高く，プログラミング能力の取得のみならず，実践的なソフトウェア開発の経験を得るという副次的な効果も期待できる．

しかしながら，テストベース教育では学生個々の機能拡張や工夫の検出が困難という課題が存在する．拡張や工夫は事前テストの範囲外の実装であり，学生ソースコード間における機能差の一種であると解釈できる．この機能差の典型例はパラメタの妥当性確認処理であり，オブジェクトの null 確認や，整数値の範囲確認等が挙げられる．さらには，テストでは未確認の独自仕様や機能拡張といった機能差も考えられる．これら学生の努力に該当する部分を検出し，採点への反映や適切なフィードバックを実施することで学習効果の向上が期待できる．

さらに，肯定的な機能差のみならず，事前テストへの過剰適合という否定的な機能差も存在すると考えられる．例えば，総和計算という課題に対して `assert(sum(3,4,5), 12)` というテストが定義されているとする．ここで `for n in [3..5]` のような加算対象の値を固定した実装や， `return 12` 等加算処理すら行わない実装は，事前テストには通過するものの，一般化が不十分な実装であり課題内容を達成しているとはいえない．他にも，条件確認が甘く ArrayIndexOutOfBoundsException や NullPointerException といった動的例外が投げられうる実装も，事前テスト次第では通過するが後学のために意識的に修正してもらうのが望ましい．工夫と同様に検出し，その実装が抱える問題を適切にフィードバックするべきである．

機能差の検出方法としては，テストの充足が一つの選択肢として考えられるが，存在しうる全ての振る舞いをテスト化することは根本的に不可能である [7]．また過度なテストは，ソフトウェア品質という側面では効果的ではある一方で，教育の場ではいい戦略とはいえない．課題仕様の本質が曖昧になるだけでなく，プログラミングにおける学生の自由な発想を阻害する要因となるためである．例えば，イースターエッグ[*1]のようなプログラミングの楽しさに寄与するユーモアの実装は，テストで無機質に排除するよりも，何かしらの手法で検出し賞賛するべきであると著者らは考える．

本研究の最終的な目標は，テストベース教育における学生ソースコード間での機能差検出の支援であ

[*1] ソフトウェア内に含まれた隠し機能の意味．例えば，標準入力を受け付ける課題に対して，特殊な文字入力時に特殊な振る舞いをする等．

る．この目標に対し，図 1 に示す通り以下 2 つの課題を定める．

- 課題 1：機能差の検出能力を持つテストの生成と分類
- 課題 2：生成テストを用いた機能差の検出

課題 1 では学生の開発した複数のソースコード群から，事前テストのみでは実現できない機能差検出能力を持つテストを生成する．課題 2 では，得られたテストを各ソースコードに適用し，その機能差の有無を検出する．

本稿では自動テスト生成，及びテスト実行時の経路情報を活用したテストのクラスタリング手法を提案する．この手法では，まず全ての提出ソースコードに対して自動テスト生成を適用する．これにより機能差の検出能力を持つテストを得る．さらに，生成テストを全ての学生間で相互に適用し，その実行経路の計測により各テストの動作確認手順を抽出する．続いて，各テストの経路をベクトル化しクラスタリングすることで，同じ動作確認手順を持つテストを洗い出す．最後に，このクラスタリングに基づいて目視によって機能差を検出する．提案手法をある大学の実題材に適用した結果，適切に機能差検出能力があるテストが生成できていた．また，目視によりクラスタリングが同じ動作確認手順を持つテストをまとめられていること，機能差検出能力を持つテストを把握できることを確認した．

2 節では研究動機として，どのような機能差を検出したいかを実例を交えて説明し，3 節では提案手法について説明する．4 節では本実験の前に行った予備実験についての説明を行い，5 節で実題材に提案手法を適用した実験の内容と結果及び考察について述べる．最後に 6 節で本研究の関連研究を，7 節で本研究のまとめと今後の課題について述べる．

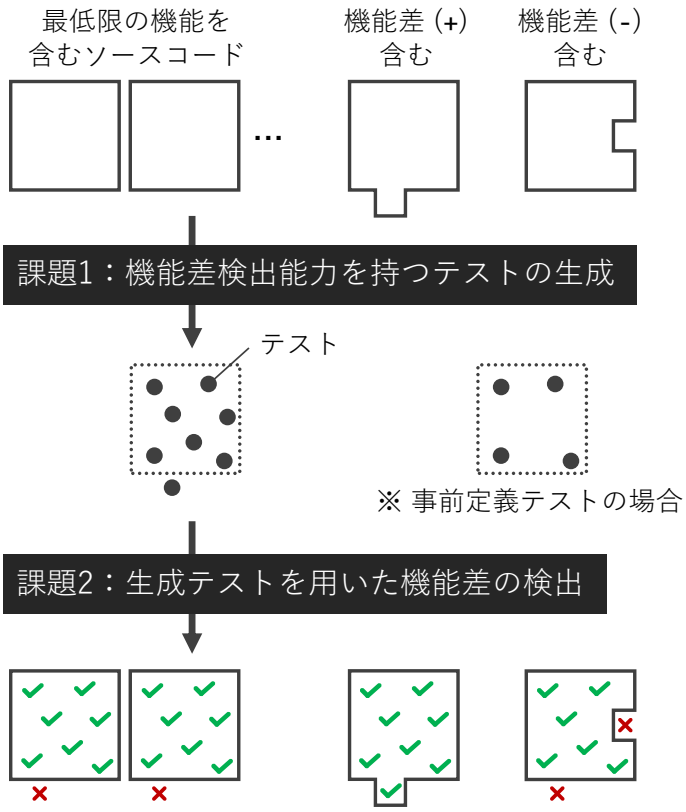

図 1: 機能差の自動検出における 2 つの課題

2 Motivating Example

数行程度のプログラミング題材に基づいて，本研究の動機を説明する．題材は TCP/IP プロトコルにおけるポート番号を表す PortNumber クラスである．当該クラスに対する模範解答，及びコンストラクタに対する事前テストを図 2 に示す．PortNumber クラスは，図 2 (a) に示すように，以下の要素から構成される．

- ポート番号を表す int 型フィールド `number`（2 行目）
- コンストラクタ（3〜5 行目）
- `number` に対する getter メソッド（6〜8 行目）
- `number` の文字列化メソッド（9〜13 行目）

ここで，本クラスの最初の実装課題として，図 2 (b) に示すような，コンストラクタに対する 2 つの事前テストが与えられたとする．この事前テストを通過する 3 種類のコンストラクタの例を図 3 に示す．図 3 (a) の学生 A は模範解答と全く同じコンストラクタであり，事前テストを通過する最低限の機能のみを実装している．他方，図 3 (b) (c) の学生 B と C は，ポート番号の妥当性確認処理を実装しており，いずれも負の値を受け付けないコンストラクタである．異常系処理という工夫を含んでいると解釈できる．ただし，負の値指定時の異常系の処理内容は異なっており，B は代替値ゼロを代入し，C は実行時例外を発生させている．

B と C は，模範解答には含まれない肯定的な機能を含む実装であり，検出し適切な評価を与えるべきである．しかし，図 2 (b) の事前テストは最低限のコンストラクタ処理内容のみを確認しており，この工夫を検出する能力を持たない．目視による確認は一つの選択肢ではあるが，大学の演習をはじめとする提案手法の適用シーンを考えると受講生は 10〜100 人であり，現実的ではない．本節の例のような単純な課題であれば，キーワードベースやファイル比較，静的なソースコード解析技術を用いた検出も可能であるが，課題内容が複雑な場合その検出能力の低下が避けられない．

```
public PortNumber {
  int number;
  public PortNumber(int n) {
    this.number = n;
  }
  public getNumber() {
    return number;
  }
  public toString() {
    if (number == 22) return "ssh";
    if (number == 80) return "http";
    ...
  }
}
```

(a) 模範解答

```
@Test public void test1() {
  int number = new PortNumber(22).getNumber();
  assertEquals(22, number);
}

@Test public void test2() {
  int number = new PortNumber(80).getNumber();
  assertEquals(80, number);
}
```

(b) コンストラクタに対する事前テスト

図 2: PortNumber クラスの模範解答と事前テスト

```
public PortNumber(int n) {
  this.number = n;
}
```

(a) 学生 A のコンストラクタ

```
public PortNumber(int n) {
  if (n < 0) this.number = 0; // 機能差
  else this.number = n;
}
```

(b) 学生 B のコンストラクタ

```
public PortNumber(int n) {
  if (n < 0) throw new IllegalArgumentException(); // 機能差
  this.number = n;
}
```

(c) 学生 C のコンストラクタ

図 3: 事前テストを通過するコンストラクタの実装例

3　提案手法

3.1　概要

本研究の目的は，学生の提出した複数ソースコード間での機能差検出の支援である．本研究における機能差検出のキーアイデアは，自動テスト生成 [8] の活用にある．自動テスト生成ではソースコードを入力とし，その振る舞いを確かめる多数のテストを自動的に生成する．この技術の適用により，複数の学生ソースコードから振る舞いの違い，すなわち機能差を検出しうるテストを生成し，そのテストを再度ソースコードに適用して機能差を洗い出す．

この手法の実現に際して，以下 2 つの課題を定める．

- 課題 1：機能差の検出能力を持つテストの生成と分類．単純なテスト生成技術の適用では大量のテストが生成される．この生成テストの中には全ソースコードで共有する，すなわち課題要件の範疇のテストが大量に含まれる．機能差検出能力を持つテストも得られるが，複数の学生が同様の機能を実装していた場合，複数の同機能を試すテストが生成される．最終的な機能差検出処理においては，目視によるテスト内容の意味づけが必要なため，これら多数のテストを適切に分類する必要がある．
- 課題 2：生成テストを用いた機能差の検出．自動テスト生成では，同じテスト類（例えば null 確認テスト）であっても，その期待値の確認処理（assert 部分）が矛盾するテストが多数生成される．分類されたテストから適切な assert 部を抽出し，対象となるソースコードにテストを再適用することで機能差の検出を試みる．

本稿では，課題 1 の達成を目指して，自動テスト生成を活用したテストのクラスタリング手法を提案する．また，実題材への適用実験において目視による課題 2 の達成方法を検討する．

課題 1 に対する提案手法の全体の流れを図 4 に示す．手法は 4 つの手順から構成されており，入力は学生の提出したソースコード，出力は生成テストのクラスタリング結果である．本図は 2 節と同様，PortNumber クラスを想定している．2 人の学生が提出したソースコード A と X が課題の最低限の実装（図 3 (a)）を，B のみが負の値の指定という異常系を考慮した実装（図 3 (b)）を行っている．以降の節では，図の流れに従って 4 つの手順それぞれについて説明する．

3.2　手順 1．自動テスト生成

まず，自動テスト生成技術を適用して，全ての提出ソースコードから単体テストを生成する．自動テスト生成はこれまで数多くの研究が行われており [8, 9, 10]，EvoSuite[11] や Randoop[12] 等のオープンソースツールも公開されている．自動テスト生成では，遺伝的アルゴリズム等の探索的メタアルゴリ

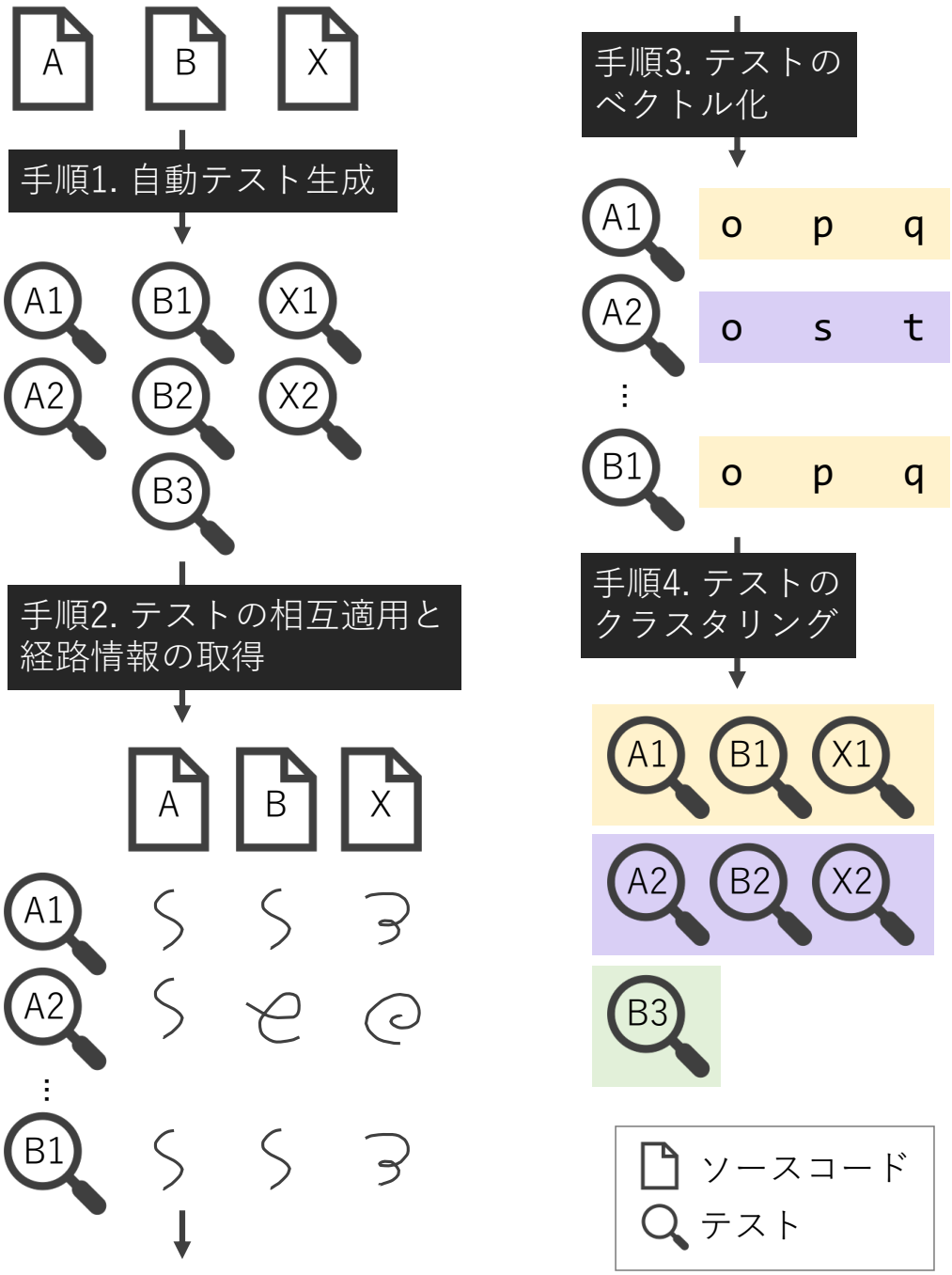

図 4: 提案手法の流れ

ズムや実行パスの解析等を用いて，網羅率を最大化するようにテストを生成する．

この手順 1 の目的は，機能差の検出能力を持つテストを得る点にある．テストベース教育においては，教員の用意した事前テストも存在するが，その動作確認は課題の要件の範疇であることが多く，機

能差の検出能力を持たない．また一般的なプログラミングにおいては，教員やユーザが想定していない工夫やバグも数多く発生するため，その全体を事前に把握してテスト化することは現実的ではない．

それに対し，自動テスト生成技術は網羅率の最大化を目的としてテストを生成する．ソースコード間における機能差は分岐等の違い，すなわち実行パスの違いとして現れることが多い．よって，網羅率最大化を目的関数としたテスト生成によって，その機能差を検出しうるテストが得られると考えられる．

図 4 では，3 人の学生の提出ソースコード ABX から，それぞれ複数のテストが生成されている．ソースコード AX と比べて B のみ生成テスト数が多い．これは B が負の値の指定という異常系を考慮しており，他のソースコードと比べて実行パスが多いためである．

A と B から生成された具体的な生成テストの一例を図 5 に示す．この例は実際に図 3 (a) と図 3 (b) のソースコードに EvoSuite を適用して得られたテストである．4 つのテストはいずれも PortNumber クラスに特定のポート番号を指定してインスタンス化し，それが適切に代入されているかを確認している．まずテスト A1 と B1 に着目する．この 2 つのテストは指定するポート番号自体は異なるものの，いずれもポート番号として正常の範囲（0〜65535）内であり，課題の範疇のテストであるといえる．他方，A2 と B2 は負の値の指定という異常系のテストであると解釈できる．ただし，A2 ではコンストラクタで指定した負の値がそのまま代入されることを期待しているのに対し，B2 は負の値の代わりにゼロが代入されることを期待している．

自動テスト生成では，真のテストオラクルは未知である [13, 14] ため，assert 節にはそのテストを題材に適用して得られた結果がそのまま用いられることが多い．ここでテストオラクルとは，ポート番号として負の値を受け付けないという普遍的な事実であるといえる．よって，バグを含むソースコードからはバグを期待する assert 節が生成される．テスト A2 は異常系を考慮していない，バグを含むソースコードから生成されたバグを期待するテストであるといえる．

3.3 手順 2. テストの相互適用と経路情報の取得

手順 2 では，まず手順 1 で生成されたテストを全てのソースコードに対して相互に適用する．N 個のソースコードから，計 M 個のテストが生成された場合，$N \times M$ のテスト実行が行われる．さらに各テスト実行において，その実行経路を記録する．テストの相互実行と経路情報の記録は，最終的なテストのクラスタリングのための前処理である．

ここで最終的なテストクラスタリングにおいて，どのような基準でテストの類似度を算出するべきかについて考える．本研究の最終目標は機能差の自動検出であり，各テストがどのような機能を試すテストであるか，という情報を類似度とするべきだといえる．例えば，図 5 の 4 つのテストの場合，A1 と B1 は具体的なポート番号自体は異なるものの，いずれも 0〜65535 の範囲の正常系のテストであると見なせる．同様に，A2 と B2 はポート番号の違いや assert 節の違いはあるものの，0〜65535 の範囲外の

```
1  @Test public void testA1() {
2    PortNumber portNumber0 = new PortNumber(992);
3    int int0 = portNumber0.getNumber();
4    assertEquals(992, int0);
5  }
6  @Test public void testA2() {
7    PortNumber portNumber0 = new PortNumber(-1);
8    int int0 = portNumber0.getNumber();
9    assertEquals(-1, int0);
10 }
```

(a) 学生 A から生成されたテスト

```
1  @Test public void testB1() {
2    PortNumber portNumber0 = new PortNumber(654);
3    int int0 = portNumber0.getNumber();
4    assertEquals(654, int0)
5  }
6  @Test public void testB2() {
7    PortNumber portNumber0 = new PortNumber(-4824);
8    assertEquals(0, portNumber0.getNumber());
9  }
```

(b) 学生 B から生成されたテスト

図 5: 自動テスト生成ツールによる生成テストの一例

一種の異常系テストであるといえる．すなわち，A1 と B1，さらに A2 と B2 を同一集合として分類するための類似度の基準を考える必要がある．

基準の一つの選択肢は，テスト実行時の成否（pass/fail）という情報の利用である．しかし，A2 のようなバグ時の挙動を期待するバグを含むテストも生成されるため，類似度の基準としては利用が困難である．例えば，A2 と B2 は同じ集合として扱うべきであるが，その assert 節は互いに矛盾しており，テストの成否も異なってしまう．またこの題材において，B2 は負の値にゼロを代入する，という異常系処理に限定されている一方で，負の値の場合に例外を投げる，標準エラーにエラーの旨を通知する，等異常時の対策は多種多様であり，その対策に応じて様々な assert 節が生成されてしまう．そのためテスト成否，すなわち assert 部を含めてテストをクラスタリングした場合，テスト分類結果の過剰な細分化を引き起こす可能性がある．

よって提案手法では，テストを，動作確認の手順を示した実行部，及び実行結果と期待値との比較を行う assert 部，2 つに分解し，実行部のみの情報に基づいてテストをベクトル化する．この実行部のみの情報として，実行時の経路情報を採用する．経路情報の利用により，assert 部を排除した各テストの動作確認手順という観点からのクラスタリングが可能となる．

3.4 手順 3. テストのベクトル化

手順 2 で得られた経路情報を ID 化し，各テストのベクトルを算出する．図 4 の手順 2 では各テストがソースコードの何行目を通過したか，という経路を記録するため，ソースコード間で経路の重複が発生する．しかし，ソースコード間での経路情報の比較は成り立たない．例えば，テスト A1 をソースコード A に適用した場合（以降，A1⇒A と記載）S の文字のような経路を取っており，A1⇒B も同経路 S を辿っている．しかし，2 つのソースコードの処理の内容が異なれば，同じ経路でもその経路の意味が異なるため同一視できない．

よって，経路情報のベクトル化の際には，学生 x のパス ID 情報 P_x は任意の学生 y に対して，以下の式が成り立つ必要がある．

$$|P_x| \cap |P_y| = \phi \tag{1}$$

本手法では，テスト実行時の各行の通過情報を文字列として直列化して学生 ID を付与する．これを MD5 等の処理によってハッシュ化することで学生間の重複を回避する．例えば，学生 x にあるテストを実行した際の経路情報が 11101（4 行目のみ通過しなかった，という意味）だった場合，x11101 という文字列を生成してハッシュを得る．このようなハッシュ化処理を全ての経路情報に適用することで，式 (1) を満たしつつテストのベクトルが得られる．図 4 ではハッシュ値を簡略しているが，テスト A1 と B1 は `[o,p,q]`，テスト A2 は全く異なるベクトル `[o,s,t]` を持つことが確認できる．なお，ベクトルの成分（ハッシュ値）は ID，すなわち名義尺度でありその大小の比較には意味を持たない．

3.5 手順 4. テストのクラスタリング

最後に，手順 3 で作成したベクトルに基づいてテストをクラスタリングする．図の場合，手順 2 で生成した計 7 個のテストは最終的に 3 種類のテストに分類できている．テスト A1，B1，X1 は同じ集合として，さらに A2，B2，X2 が同じ集合として分類されている．手順 2 でも説明した通り，経路情報を活用することで，テスト内での期待値との比較（assert 部）を排除した，テストの動作確認手順を基準とした分類が可能となっている．

本節概要でも述べたとおり，提案するテストクラスタリング手法は受講者 100 人規模のソースコードに対しても，人手を要さない完全自動化が可能であり，大量の生成テストをその確認手順の類似度という観点から分類が可能である．なお，本図での題材は数行程度であり，ベクトルの完全一致のみで分類が可能であるが，より実践的な題材ではこの限りではない点に注意されたい．

4 予備実験

本予備実験の目的は，前節で述べた提案手法により機能差の検出能力を持つテストを生成し，適切に分類できるかの確認である．そのために数行程度のプログラミング題材に対して提案手法を適用した．

4.1 実験題材

実験に用いた題材は 2 節，3 節に引き続き PortNumber クラスである．想定する機能差は 2 つの要素の組み合わせで構成される．一つはポート番号の範囲確認の有無であり，もう一つは異常時処理の種類である．範囲確認は，下限確認（`n < 0`）と上限確認（`n > 65535`），及びその両方の 3 種類である．異常時の処理は，指定ポート番号が範囲外だった際に代替値（ゼロ）を代入するか，例外を発生させる（`throw new Exception`）かの 2 種類である．

上記 2 つの要素の組み合わせの結果，表 1 に示す 7 種類の機能差を持つ題材を作成した．題材名称は範囲確認と異常時処理の組み合わせとなっている．例えば，一番上の CheckNone は一切の範囲確認処理を行っておらず，指定された値をそのまま代入している．課題要件の最低限の実装であるといえ，図 3 (a) に相当する．CheckLow-SetAltValue では下限確認の後，下限を下回っていた場合に代替値を代入している．上限確認は行っていない．これは図 3 (b) に相当し，続く CheckLow-ThrowException は図 3 (c) に相当する．

本予備実験で想定する機能差は，2 種類の範囲確認処理を行っているか否かであり，異常時処理の内容は無視されるべきである．例えば，表 1 の下 4 行（CheckUpp-*と CheckBoth-*）は異常時の振る舞い自体は異なるものの，いずれも上限確認という機能差を持っている，と検出されることが望ましい．この機能差の定義に関しては 3.3 節を確認されたい．

表 1: 実験題材

題材名称	`n` < 0 時	`n` > 65535 時
CheckNone	`n` を代入	`n` を代入
CheckLow-SetAltValue	0 を代入[*a]	`n` を代入
CheckLow-ThrowException	例外発生[*a]	`n` を代入
CheckUpp-SetAltValue	`n` を代入	0 を代入[*b]
CheckUpp-ThrowException	`n` を代入	例外発生[*b]
CheckBoth-SetAltValue	0 を代入[*a]	0 を代入[*b]
CheckBoth-ThrowException	例外発生[*a]	例外発生[*b]

[*a] 想定する機能差 1，[*b] 想定する機能差 2

4.2 実験手法

実験としてはまず，著者らが手動で 7 種類の題材ソースコードを作成した．次に，自動テスト生成ツール EvoSuite を用いて題材ソースコードのテストを自動生成した．EvoSuite の各種パラメタは初期設定とした．続いて，生成された各テストを各ソースコードに対して相互に実行した．テスト実行時には経路情報を取得するために，実行経路計測ツール JaCoCo[*2]を使用した．最後に，JaCoCo によって得られた経路情報に各ソースコードの題材名称を付与し，`md5sum` コマンドを用いて経路 ID となるハッシュ値を算出した．

4.3 結果と考察

実験により得られた各テストの経路 ID を図 6 に示す．列がソースコード，行がテストを表しており，値は経路のハッシュ値である．視認性向上のため，列ごとの同一ハッシュを同一色で塗りつぶしている．テストの名称は，そのテストの生成元となったソースコード名に連番を付与した名前となっている．なお，縦方向のテストの順序は EvoSuite が生成した順序をそのまま採用しており，意味を持たないことに注意されたい．表の右列の集合 ID は，実行経路 ID から算出された各テストのクラスタリング結果である．この場合，経路の完全一致に基づいてクラスタリングを行っている．さらに，テストの各集合 ID のテスト内容を表 2 に，集合 ID ごとの具体的な生成テストの例を図 8 に示す．以降では，これらの結果に基づき，自動テスト生成とテストクラスタリングの 2 つそれぞれの結果について説明する．

4.3.1 自動テスト生成

まず図 6 の結果を自動テスト生成の観点から説明する．生成テストの数に関しては，CheckNone と CheckLow-*が 3 個，CheckUpp-*が 4 個，CheckBoth が 5 個と機能差を含むソースコードほど多くのテストが生成される傾向にある．CheckNone と CheckLow-*でテスト数に差がない理由は，EvoSuite のテスト生成戦略に起因する．EvoSuite は int 引数に負の値を渡すというヒューリスティックを採用しており，下限確認の有無がテスト数として表れなかったといえる．

ソースコードごとの経路の種類（ハッシュ値の種類）という観点では，やはり機能差が多いほど様々な経路を取る傾向にある．CheckNone が 2 種類のみに対し，CheckLow-SetAltValue は 4 種類，最大では CheckBoth-SetAltValue の 5 種類である．ただし，*-ThrowException は*-SetAltValue と比べて経路種類が少ない．これは例外を期待するテストが getter メソッドを呼び出さないことに起因している．例えば，生成テストの具体例図 8 (b) と図 8 (c) は，下限以下の値をコンストラクタに渡すという点では同じテストであるが，その後に getter メソッドを使って値を確認するか，例外発生を期待する

[*2] `https://www.jacoco.org/jacoco/`

ソースコード テスト	ChkNone	ChkLow-SetAltV	ChkLow-ThrowE	ChkUpp-SetAltV	ChkUpp-ThrowE	ChkBoth-SetAltV	ChkBoth-ThrowE	集合ID
ChkNone1	765d	cc29	4e8e	a0e6	4f09	cc48	e794	1
ChkNone2	765d	cc29	4e8e	a0e6	4f09	cc48	e794	1
ChkNone3	765d	8daa	c8a0	a0e6	4f09	214f	e04b	2
ChkLow-SetAltV1	765d	cc29	4e8e	a0e6	4f09	cc48	e794	1
ChkLow-SetAltV2	765d	8daa	c8a0	a0e6	4f09	214f	e04b	2
ChkLow-SetAltV3	765d	cc29	4e8e	a0e6	4f09	cc48	e794	1
ChkLow-ThrowE1	765d	cc29	4e8e	a0e6	4f09	cc48	e794	1
ChkLow-ThrowE2	258a	9a26	c8a0	e831	e087	54db	e04b	3
ChkLow-ThrowE3	765d	cc29	4e8e	a0e6	4f09	cc48	e794	1
ChkUpp-SetAltV1	765d	cc29	4e8e	a0e6	4f09	cc48	e794	1
ChkUpp-SetAltV2	765d	8daa	c8a0	a0e6	4f09	214f	e04b	2
ChkUpp-SetAltV3	765d	cc29	4e8e	849f	7e7a	dc18	08f1	4
ChkUpp-SetAltV4	765d	cc29	4e8e	a0e6	4f09	cc48	e794	1
ChkUpp-ThrowE1	765d	cc29	4e8e	a0e6	4f09	cc48	e794	1
ChkUpp-ThrowE2	765d	8daa	c8a0	a0e6	4f09	214f	e04b	2
ChkUpp-ThrowE3	258a	55aa	f528	2da1	7e7a	25a9	08f1	5
ChkUpp-ThrowE4	765d	cc29	4e8e	a0e6	4f09	cc48	e794	1
ChkBoth-SetAltV1	765d	cc29	4e8e	a0e6	4f09	cc48	e794	1
ChkBoth-SetAltV2	765d	cc29	4e8e	a0e6	4f09	cc48	e794	1
ChkBoth-SetAltV3	765d	cc29	4e8e	849f	7e7a	dc18	08f1	4
ChkBoth-SetAltV4	765d	8daa	c8a0	a0e6	4f09	214f	e04b	2
ChkBoth-SetAltV5	765d	cc29	4e8e	a0e6	4f09	cc48	e794	1
ChkBoth-ThrowE1	765d	cc29	4e8e	a0e6	4f09	cc48	e794	1
ChkBoth-ThrowE2	765d	cc29	4e8e	a0e6	4f09	cc48	e794	1
ChkBoth-ThrowE3	258a	55aa	f528	2da1	7e7a	25a9	08f1	5
ChkBoth-ThrowE4	258a	9a26	c8a0	e831	e087	54db	e04b	3
ChkBoth-ThrowE5	765d	cc29	4e8e	a0e6	4f09	cc48	e794	1
生成テスト数	3	3	3	4	4	5	5	
経路種類の数	2	4	3	4	3	5	3	

図 6: 生成テストと実行時経路の一覧

かが異なる．前者のみ getter メソッドを呼び出しているため，後者と経路が異なる．

4.3.2 経路の計測とクラスタリング結果

クラスタリングの結果，多くのテストが集合 ID1 に割り振られた．表 2 に示す各集合のテスト内容，及び図 8 (a) のテストの実例を確認すると，n の値自体は様々ではあるものの，いずれも正常範囲（$0 \leq$

ソースコード テスト	ChkNone	ChkLow- SetAltV	ChkLow- ThrowE	ChkUpp- SetAltV	ChkUpp- ThrowE	ChkBoth- SetAltV	ChkBoth- ThrowE	集合 ID
ChkNone3	765d	8daa	c8a0	a0e6	4f09	214f	e04b	2
ChkLow-SetAltV2	765d	8daa	c8a0	a0e6	4f09	214f	e04b	2
ChkUpp-SetAltV2	765d	8daa	c8a0	a0e6	4f09	214f	e04b	2
ChkUpp-ThrowE2	765d	8daa	c8a0	a0e6	4f09	214f	e04b	2
ChkBoth-SetAltV4	765d	8daa	c8a0	a0e6	4f09	214f	e04b	2
ChkLow-ThrowE2	258a	9a26	c8a0	e831	e087	54db	e04b	3
ChkBoth-ThrowE4	258a	9a26	c8a0	e831	e087	54db	e04b	3
ChkUpp-SetAltV3	765d	cc29	4e8e	849f	7e7a	dc18	08f1	4
ChkBoth-SetAltV3	765d	cc29	4e8e	849f	7e7a	dc18	08f1	4
ChkUpp-ThrowE3	258a	55aa	f528	2da1	7e7a	25a9	08f1	5
ChkBoth-ThrowE3	258a	55aa	f528	2da1	7e7a	25a9	08f1	5

図 7: 生成テストと実行時経路の一覧（ID1 を削除）

$n \leq 65535$）のテストであり，課題要件の範疇内のテストであるといえる．この集合 ID1 のテストは全てのソースコードから生成される，共有機能であると解釈できる．

視認性向上のため図 6 の結果に対し，集合 ID1 のテストを削除し集合 ID で並び替えた結果を図 7 に示す．図 8 (b) に示す集合 ID2 のテストは，負の値を指定してその代入値を確認するテストである．ここで代入値の期待値は負の値そのものである場合と，代替値ゼロである場合の両方が存在していた．assert 部の内容に依存することなく，負の値を試すというテストが適切に分類されていたといえる．この結果は図 8 (d) に示す集合 ID4 の，上限確認のテストでも同様であった．

他方，集合 ID3 と 5 は過剰に細分化されたテスト群であった．ID3 は $n \leq 0$ のテストという観点では ID2 と同一，ID5 は $n \geq 65535$ のテストという観点で ID4 と同一と分類されることが期待される結果であった．その理由について考察する．図 8 (c)，及び図 8 (e) を確認すると，いずれも例外発生を期待するテストである．加えて，assert 節における getter メソッドの呼び出しが含まれていない．getter の呼び出しはテスト実行時の経路の一種であり，この有無が経路ベクトルの違いとクラスタリング結果の過剰な細分化に繋がっていると考えられる．実際に，図 7 の CheckNone の列に着目すると，CheckNone は最低限の機能実装であるにも関わらず，765d と 258a の 2 種類の経路を取っている．765d は getter の通過を含んだテスト実行を，258a は含まないテスト実行であると読み取れる．図 7 の集合 ID を確認しても，258a の場合に集合 ID3 と 5 を取っていることが確認できる．このように例外を期待するテストは過剰に細分化される傾向にあり，対策が必要であると考えられる．

```
@Test public void CheckNone2() {
  PortNumber portNumber0 = new PortNumber(992);
  int int0 = portNumber0.getNumber();
  assertEquals(992, int0);
}
```

(a) 集合 1 のテストの例

```
@Test public void CheckNone3() {
  PortNumber portNumber0 = new PortNumber(-1);
  int int0 = portNumber0.getNumber();
  assertEquals(-1, int0);
}
```

(b) 集合 2 のテストの例

```
@Test public void CheckLowThrowException2() {
  PortNumber portNumber0 = null;
  try {
     portNumber0 = new PortNumber(-394);
     fail("Expecting␣exception:␣IllegalArgumentException");
  } catch(IllegalArgumentException e) {
    verifyException("PortNumber", e);
  }
}
```

(c) 集合 3 のテストの例

```
@Test public void CheckUppSetAltValue3() {
  PortNumber portNumber0 = new PortNumber(65545);
  assertEquals(0, portNumber0.getNumber());
}
```

(d) 集合 4 のテストの例

```
@Test public void CheckUppThrowException3() {
  PortNumber portNumber0 = null;
  try {
     portNumber0 = new PortNumber(65558);
     fail("Expecting␣exception:␣IllegalArgumentException");
  } catch(IllegalArgumentException e) {
      verifyException("PortNumber", e);
  }
}
```

(e) 集合 5 のテストの例

図 8: 各集合のテストの例

表 2: テストの集合 ID とその内容

集合 ID	代入する n の範囲	期待値
1	$0 \leq \mathtt{n} \leq 65535$	`n`
2	$\mathtt{n} < 0$	`n` か代替値 0
3	$\mathtt{n} < 0$	例外発生
4	$65535 < \mathtt{n}$	`n` か代替値 0
5	$65535 < \mathtt{n}$	例外発生

5　実題材への適用実験

本章では，提案手法を評価するために行った実験について述べる．実題材に対して提案手法を適用し，提案手法の目的である機能差検出が可能であるかを確認した．

5.1　実験題材

実験題材には，ある大学のプログラミング演習で作成された `Card.java` を用いた．この演習は簡単なトランプゲームの開発を目的としている．本実験ではこの演習の最初期の課題を題材として取り上げる．`Card.java` はトランプ一枚を表すデータクラスであり，与えられた仕様を元に学生が実装したものである．実験には 59 人の学生から提出されたソースコードを用いた．

課題の仕様は次の通りである．1 つの `Card` オブジェクトが 1 枚のトランプを表している．Card クラスは 2 つの int 型フィールド suit と number を持つ．それぞれトランプカードのスートと数字を表している．例えば，suit=0, number=10 のカードはスペードの 10 である．suit はそれぞれ 0 がスペード，1 がダイヤ，2 がハート，3 がクラブに対応している．また，number は 1 から始まり，13 までとなる．ジョーカーは suit=-1, number=0 で表現される．Card クラスに用意されているメソッドはコンストラクタ，ゲッター，セッターの他に，toString メソッドがある．toString メソッドは 2 つの int 型フィールド（suit, number）からなる内部表現を文字列に変換するメソッドである．例えば，suit=0, number=11 のカードに対しては “スペード J” を返す．

課題の達成条件は上記仕様を満たすプログラムの開発であり，仕様外の振る舞いは未定義である．例えば，suit が取り得る値は-1〜3，number が取り得る値は 0〜13 となっており，これ以外の値は異常値となる．異常値に対する振る舞いは定義されていないため，どのように対応するかは学生個人に委ねられる．ここに機能差があると想定する．

5.2　実験手法

以下の手順に基づいて実験を行った．

手順 0　題材の事前処理

手順 1　自動テスト生成

手順 2　テストの相互実行及び経路取得

手順 3　テストのベクトル化

手順 4　階層クラスタリング

以降の節で詳細を説明する．

5.2.1 手順 0. 題材の事前処理

初めに，前処理として題材の事前処理を行った．受講生が提出したソースコードには，提案手法の前提を満たしていない部分が存在した．この処理の意図は課題の仕様を統一してノイズを減らすことである．題材の仕様は 5.1 節で述べたとおりだが，学生に与えられた仕様はここまで厳密ではなかった．例えば，学生によって number の開始を 0 とするか 1 とするかで仕様解釈の揺れが存在した．この揺らぎはテスト生成以降にノイズとなってしまうため，著者らで 5.1 節で述べた仕様に添うように統一した．

また，事前準備として学生の匿名化を行った．この際，学生や大学を特定しうるパッケージ名や author タグ，及びコメントを削除した．また，md5sum コマンドを用いてファイル名の学籍番号をハッシュ値に変換した．ハッシュ値を用いるのは，学生を唯一に指定できるという学籍番号の特徴を可能な限り生かしながら匿名化を行うためである．

5.2.2 手順 1. 自動テスト生成

次に，自動テスト生成ツール EvoSuite を用いて，各提出ソースコードからテストを自動生成した．EvoSuite の各種パラメタは初期設定のままである．また，EvoSuite の実行には乱択が含まれているが，実行は 1 回のみ行った．これは EvoSuite に網羅率を最大化する戦略があり，それによって乱択による非確実性を補えるからである．

5.2.3 手順 2. テストの相互実行及び経路取得

続いて，生成された各テストを各提出ソースコードに対して相互に実行した．テスト実行時には経路情報を取得するために，実行経路計測ツール JaCoCo[*3]を使用した．

5.2.4 手順 3. テストのベクトル化

JaCoCo によって得られた経路情報に各提出ソースコードのハッシュ値を付与し，md5sum コマンドを用いて経路 ID となるハッシュ値を算出した．なお，このベクトルは各要素が名義尺度であるため，ハッシュ値をそのまま数値として比較することはできない．しかし，手順 4 で使用するクラスタリング手法では数値ベクトルを与える必要がある．そのため，各列に対する One-Hot エンコーディングによって 0 と 1 のみの数値ベクトルに変換した．実装としては，Python の pandas パッケージ[*4]の `pandas.get_dummies()` 関数を利用した．

[*3] https://www.jacoco.org/jacoco/
[*4] https://pandas.pydata.org

5.2.5　手順 4. 階層クラスタリング

提案手法のクラスタリングを実現するため，Ward 法にて階層クラスタリングを行った．実装としては Python の SciPy パッケージ[*5]を利用した．予備実験で確認された過剰な細分化を抑制するため，階層クラスタリングを行う．

5.3　RQ

提案手法の評価を行うにあたり，以下 3 つの Research Question（RQ）を設定した．

RQ1: 機能差検出能力を持つテストを生成できているか？

機能差が網羅率重視の自動テスト生成でテストできるかを確かめるために，生成されたテストの平均命令網羅率を調査する．自動生成された全てのテストを各提出ソースコードに対して適用する．各提出ソースコードの命令網羅率がどうなるかを調査する．仮に網羅率が 100 %ならば全ての機能を呼び出しており，コード内に存在する機能差も呼び出して検出できるとみなせる．網羅率が 100 %でない場合は，網羅できていないコード片を調査する．該当するコードが到達不能なコードかどうか，もしくは機能差に関係ないテストかどうかを示せれば，生成されているテストでは適切に機能差を検出できることを確かめられる．

RQ2: 経路情報を用いてテストの適切な階層的クラスタが構築できているか？

同値類のテストは同じ実行経路を通るかどうかを確かめるために，クラスタリングの結果を目視で調査する．RQ1 で機能差検出能力を持つテストが生成できていたとしても，数多くのテストに埋もれてしまい，目視で見つけるには多大な手間がかかる．そこで，目視の母数を減らすために適切に同値類をまとめたクラスタリングを行う必要がある．クラスタリングが適切であるかの確認として，特定のクラスタに注目し，そのクラスタが他のクラスタと統合される過程を粒度が細かい方から順に確認する．この時，統合されたクラスタ同士が統合されるべき同値類かどうかを目視で判定する．また，クラスタリングによって目視の母数がどの程度減っているかどうかも調査する．

RQ3: 機能差検出能力を持つテストを特定できるか？

これまでの結果を受けて，提案手法自体の出力として目的である機能差検出支援が達成されているかどうかを調査する．

[*5] https://scipy.org

```
1 if(suit < -1 || 3 < suit) {
2   System.out.println(suit + "is␣invalid.Try␣again");
3   System.out.println("0:spade,1:diamond,2:heart,3:club,-1:joker");
4   suit = k.inputNumber();
5   setSuit(suit);
6 }
```

(a) 標準入力待機中にタイムアウト

```
1 private static String getString(final int suit, final int number) {
2   if (!validate(suit, number))
3     throw new IllegalArgumentException("invalid␣card");
4   else if (suit == -1 && number == 0) return Number.getName(number);
5   else return Suit.getName(suit) + Number.getName(number);
6 }
```

(b) コンストラクタでもそれ以外でも異常系判断

```
1 if (suit < -1 && suit > 3) {
2   System.out.println("絵柄エラー");
3 } else {
4   this.suit = suit;
5 }
```

(c) 分岐条件にバグ

図 9: 自動テスト生成で網羅できなかった 3 種類の到達不能なソースコードの例

5.4 結果

RQ1: 機能差検出能力を持つテストを生成できているか？

自動テスト生成を行ったところ，全部で 965 個のテストが生成された．提出ソースコードごとの網羅率を調査したところ，高い網羅率を実現していた．網羅率 100 %の提出ソースコードは 59 個のうち 50 個存在した（85 %）．また，提出ソースコードごとの網羅率の平均は 98.59 %であった．

網羅率が 100 %でない提出ソースコードを目視で内容を確認したところ，いずれもこの実験のテスト実行では到達不能なソースコードであることが確認できた．到達不能なソースコードの内容は 3 種類に大別される．標準入力待機中にタイムアウトするソースコード（図 9 (a)），コンストラクタでもそれ以外でも異常系判断するソースコード（図 9 (b)），分岐条件にバグが存在するソースコード（図 9 (c)）である．

図 9 (a) に示す標準入力待機中にタイムアウトする事例は，異常値を検出した時の対処として標準入力を受け付けるソースコードである．この実験のテスト実行環境では標準入力への入力を用意していないため，入力が与えられることはなく，テストはタイムアウトで失敗する．そのため，4 行目と 5 行目は実行されない．

図 9 (b) に示すコンストラクタでもそれ以外でも異常系判断する事例は，丁寧に異常系処理を行っているソースコードである．コンストラクタでも同様の異常処理を行い，各種呼び出しメソッドでも異常系確認を行っている．しかし，全ての異常系を必ず通るコンストラクタで処理できているため，各種呼び出しメソッド内では異常系処理は起こらず，3 行目が実行されない．

図 9 (c) に示す分岐条件にバグが存在する事例では，そもそも分岐の条件文が間違っている．絶対に達成されない条件となっているため，2 行目は実行されない．

このように，網羅できていないコードはこの実験設計では到達不能である．それ以外のコードは網羅できているため，RQ1 の結論としては十分な網羅率を誇っており，機能差検出能力を持つテストが生成できているとみなせる．

なお，この網羅率の調査によって，到達不能な機能差も存在することが確認できた．例えば，事例 1 は異常があった場合に再入力を標準入力に求めている．今回のテストケースでは標準入力を与えていないためタイムアウトでテスト失敗となってしまうが，実践においては有効な手である．よって，これは可能ならばフィードバックをすべき機能差である．テストを実行して通過可否を確認するだけでは見つからないが，網羅率に注目することで発見できた．

RQ2: 経路情報を用いてテストの適切な階層的クラスタが構築できているか？

クラスタリングの結果，965 個のテストの中には全く同じ経路を通過するテストが存在した．異なるテスト同士でも，同じ提出ソースコードに対しては実行経路が同じになるテストである．提案手法における階層クラスタリングは経路ベクトルの距離に基づいて行うため，完全一致のテスト同士は距離が 0 となり，同じクラスタとなる．この完全に一致する経路ベクトルを持つテスト同士だけをまとめると，965 個のテストが 189 個のクラスタに削減された．このクラスタは 965 個のテストを母集団として階層クラスタリングをした際，同じクラスタになる．この類似度を用いる限り常に同じクラスタになるので，クラスタの統合度合いの検討においてはあらかじめ統合されているとして良い．そのため，以降では階層クラスタリングの母集団には 965 個の自動生成されたテストではなく，189 個の経路ベクトルに基づいたクラスタを用いる．

189 個のクラスタに対して階層クラスタリングを実行した結果を図 10 に示す．図の上半分が階層クラスタリングの過程を示すデンドログラムである．横に各クラスタが並んでおり，縦がクラスタ同士の類似度を示している．類似度が小さいほどクラスタ同士の距離が近く似ているクラスタである．デンドログラムの横線の高さがその統合の類似度を示している．図の下半分がクラスタのサイズを示す棒グラフである．棒が長いほど多くのテストが同じ経路を通過し，その棒が示すクラスタに統合されていることを示す．

189 個のクラスタのうち，一部のクラスタを選び，そのクラスタが他のクラスタと統合されていく過

程を目視で確認した．目視するクラスタの選定は，目視するクラスタサイズが偏らないように，及びデンドログラム内での場所が偏らないように行った．目視したクラスタは図 10 内で薄黄色の背景色によって示してある．デンドログラムの一番下から確認を始め，一つずつデンドログラムを上に辿り，統合の内容を確認した．統合されるクラスタがすでに複数のクラスタが統合されたクラスタならば，先により末端のクラスタの内容を確認してから該当の統合の内容を確認した．閾値よりも類似度が大きい統合以降は目視確認を行っていない．閾値は 20 に設定した．類似度が閾値以上となった場合はそのクラスタを目視を終了し，次の目視確認へと移った．

目視をした結果，意味ごとに適切なクラスタ統合が行われていることが確認できた．クラスタ統合が適切であるかは以下の基準を共に満たしているかどうかで判定した．ただし，後述する値差し替えの例のように例外を認めることはある．

基準 1　1 つ以上の共通するメソッドを呼び出している

基準 2　suit と number はそれぞれ同値類になっている

基準 1 は，例えば同じクラスタだと判定されたテストのうちの 1 つが getSuit と toString を呼び出しており，もう 1 つが toString だけを呼び出している場合は，toString メソッドが共通しているため満たされる．基準 2 はテストの同値分割として同値であるかを確認している．例えば，異常系を調べているテストと正常系を調べているテストが統合される場合は満たされない．

目視した結果，同じ意味のクラスタであると判定した範囲を図 10 の赤枠で示す．また，各クラスタの内容，含まれてる総テスト数，同じクラスタに許容した統合の最大類似度，及び同じクラスタになることを許容しなかった統合の最小類似度を表 3 に示す．表の ID は図 10 内の赤枠に添えられているアルファベットを指す．例えば，図 10 の左端にある ID:A のクラスタは正常系の toString を呼び出しているテストが合計 182 個統合されたクラスタとなっている．また，類似度 14.07 の統合までは同じ意味であったが，次の類似度 17.67 の統合は意味が異なると判断した．なお，非許容類似度が 20 を超えるクラスタについては次の統合自体を確認していない．

目視したクラスタの多くは同じ意味として統合できる．例えば，ID: F, J, K では目視した範囲である距離 20 までの全ての統合が同じ意味の統合であった．それ以外でも多くの同じ意味のテストを同じクラスタにまとめられているため，目視の必要があるテスト数を大幅に削減できている．

目視した範囲で類似度が小さい段階から統合を許容できなかったクラスタは B と C，D と E，G と H と I である．例えば D と E は共に getNumber の呼び出しを確認しているが，ID: D は number が異常値の場合を確認している．一方で，ID: E は正常系の場合を確認している．これは基準 2 に抵触している．

同じクラスタに別の意味のテストが統合されていないだけでなく，別のクラスタに同じ意味のテストが分断された例もほとんどない．目視した結果では B と G のみが同じ「suit 異常のゲッター 2 種」を

呼び出した結果を確認している．この 2 つが別のクラスタになってしまった，すなわち経路が分かれてしまった原因としては，複数 assert によるテスト途中終了が考えられる．EvoSuite によって生成されたテストには，1 つのテスト内に assert が複数あるものも含まれている．テスト実行に当たってソフトアサートの指示は行っていないため，assert が 1 つでも失敗すればテストはその時点で終了する．B のクラスタのテストは少数派な値が来ることを想定しているため，多くの学生は 1 つ目の assert でテスト失敗し，2 つ目の assert で呼び出しているゲッターが呼ばれず，通過する経路が異なると考えられる．

どの統合まで許容できるかはクラスタによって大きく異なる．例えば，ID: K のクラスタは類似度 19.08 の統合も許容できているが，ID: B では類似度 4.69 の統合でさえ許容できなかった．よって，一概にこの値で区切ると全体を同じ意味の過不足ないクラスタリングにできると結論づけることはできない．例えば，一番低かった非許容類似度 4.69 で区切るとクラスタ数は 89 となり，目視の負担はまだ大きい．許容類似度の平均値である 7.46 で区切ると 66 個，非許容類似度の平均値である 15.42 で区切ると 19 個となる．これらは大きく削減できているが，ID: G と H のような事例が統合されてしまう．例えば G と H が統合されてしまうと，suit の異常系を確認できているのか number の異常系を確認できているのかが判断できなくなってしまう．よって，意味を考慮しながら目視の数を効果的に減らせる 1 つの基準を全体で統一して設けることは難しい．

以上のことより，階層クラスタリングは効果的に働いているが，どこまでを 1 つのクラスタとして扱うかの統一された基準を設けることは難しいことがわかった．

表 3: 目視したクラスタの内容

ID	内容	テスト数	許容類似度	非許容類似度
A	正常系の toString	182	14.07	17.67
B	suit 異常のゲッター 2 種	2	1.41	4.69
C	number に直接代入	3	4.47	17.72
D	number 異常の getNumber	7	5.54	7.05
E	正常系の getNumber	4	3.06	7.05
F	suit=-1 の toString	56	13.49	33.85
G	suit 異常のゲッター 2 種	65	3.76	5.83
H	number 異常のゲッター 2 種	59	3.74	5.83
I	正常系のゲッター 2 種	149	2.71	8.14
J	suit 異常，number=12 の toString	35	10.67	22.54
K	suit 異常，number が 2〜10 の toString	53	19.08	39.25

RQ3: 機能差検出能力を持つテストを特定できるか？

RQ1 と RQ2 の結果から，少なくとも機能差を検出する能力があるテストが同じクラスタにまとめられている．よって，機能差検出能力を持つテストを特定するためには，そのクラスタを見つけ出せれば良い．以降機能差検出能力を持つテストを検出テストと表現する．

ここではシナリオベースで検出テストの特定方法について議論する．まずは提案手法に従って，図 10 のようなデンドログラムを作成する．そして閾値を定めてクラスタを決定する．例えば，粗めの目視ということで閾値を 7 に設定する．クラスタ数は 69 になるので，これらを目視していく．著者が目視した際は．1 つのテストに対してテストの確認，内容の把握，及びその機能差に対応する学生の確認で 1 分程度かかった．よって，全体では 70 分程度かかると想定される．クラスタの目視によって検出テストが見つけられる．検出テストが発見できれば，そのコードを確認することでブラックボックス的に機能差の内容を把握できる．例えば，ID: D の中に図 11 に示す検出テストがある．これは異常値である number=670 を正常値である number=0 に置き換えることを想定しているテストである．これは異常系処理の一例であり，実装している学生にはその努力を褒めるフィードバックを提供すべきである．そのためには，検出テストに対する各提出ソースコードのテスト通過可否を確認すれば良い．機能差があることを想定しているテストに対しては通過している提出ソースコードが，ないことを想定しているテストに対しては未通過の提出ソースコードが機能差を実装している．それらのソースコードを必要に応じて目視し，そのソースコードを提出した学生に実装を褒めるフィードバックを提供すれば良い．例えば，図 11 のテストは機能差があることを想定しているため，このテストを通過しているソースコードを提出した学生にポジティブなフィードバックを提供する．

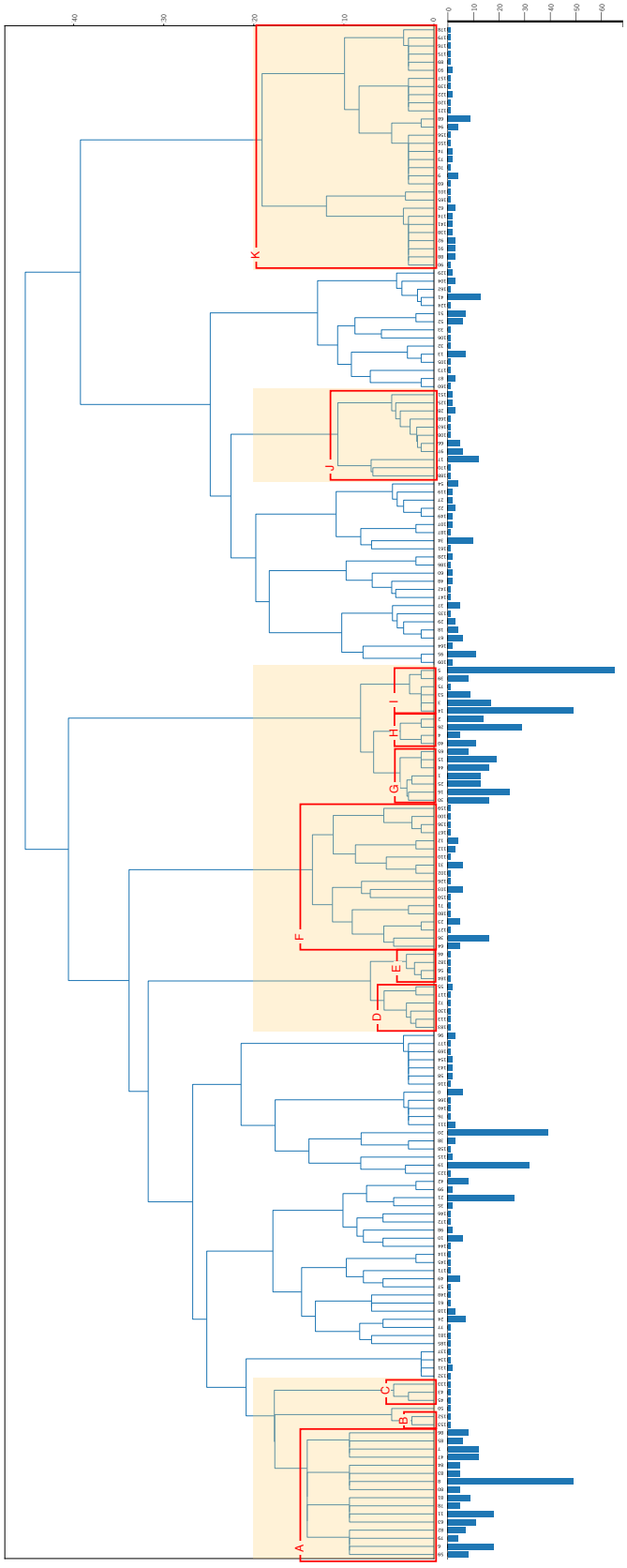

図 10: 階層クラスタリングの結果

```
1  public void test00() throws Throwable {
2      Card card0 = new Card(0, 670);
3      assertEquals(1, card0.getNumber());
4      assertEquals(0, card0.getSuit());
5  }
```

図 11: 発見できた検出テストの例

6 関連研究

6.1 自動テスト生成

3.2 節でも述べたように，自動テスト生成はこれまで数多くの研究が行われている [15]．3.2 節で紹介した遺伝的アルゴリズムを使用する EvoSuite[11] やランダムテスティングを使用する Randoop[12] の他にも，遺伝的アルゴリズムに加えて記号実行を行う Sushi[16] というツールなどもある．EvoSuite を用いたが，これらの別のツールを使うことによって異なる結果が得られる可能性がある．

6.2 テストケース優先度決定

今回行った自動生成テストのクラスタリングは，テストケース優先度決定の一環である．テストケース優先度決定の他の研究 [17, 18, 19] では，経路情報ではなくオブジェクトやメソッドの数などを利用している．

6.3 自動プログラミング評価

自動でプログラミング課題を評価する研究としては AUTOGRADER[20] などが存在する．こちらの研究では教師が想定している機能についてのみ評価しているため，本研究のように教師にとっても未知の機能差には対応できない．

7　おわりに

本研究では，テストベース教育におけるソースコードのより良い採点を目的として，提出ソースコード間の機能差の自動検出に向けた課題を整理した．さらに，自動テスト生成，及びテスト実行時の経路情報を活用したテストクラスタリング手法を提案した．また，予備実験として行った数行程度のプログラミング題材に対する手法の適用結果について報告し，実題材に対しても適用実験を行った．実験結果から，提案手法は機能差検出能力を持つテストを生成できること，及びテストの動作確認手順を基準としてテストを分類できることを確認した．また，クラスタリングしたテストから機能差を検出する手法を考察した．

今後の課題としては，テストの優先度決定や機能差の内容の判定が挙げられる．現在出力としてはクラスタリングした結果を提供するが，どのクラスタを優先的に目視すればいいのか，そのクラスタの中で機能差を認識しやすいテストがどれなのかも提供できれば，より教師の採点の助けになる．さらに，検出した機能差が肯定的な内容か，否定的な内容かを推測する技術についても検討中である．特に否定的な機能は多くの場合，事前テストへの過剰適合に起因しており，いくつかのヒューリスティックにより検出可能であると考えている．

謝辞

本研究は多くの方々に支えられてきました．その方々にこの場を借りてお礼申し上げます．

まず，B4 の時からずっと面倒を見てきてくださった担当教員の柗本真佑助教．なかなか実験や論文執筆が進まない中，いつも私が進捗を生めるようになるにはどうすれば良いかを考えアドバイスしてくださりました．学術的な助けももちろん多くいただいたのですが，そういった研究姿勢の面で多くのことを教わりました．

同じく，研究室で多くの面倒を見てきてくださった肥後芳樹准教授と楠本真二教授．輪講や中間報告の場で多くの貴重な意見をいただきました．また，困った時に雑談したら助けていただけたりと，本当にお世話になりました．

共同研究を行った他大学の井垣先生，福安先生，佐伯先生．報告ミーティングの際いつも真摯な意見をくださり，とても白熱した議論をしていただきました．

事務の橋本さんと神谷さん．煩雑な事務手続きを一手に担っていただけるおかげで，研究に集中することができました．

同期の友人方．同じ研究室で切磋琢磨し，困った時に相談したら真面目に考えてくれました．研究の真面目の話からちょっとした確認事項，息抜きの雑談など多くの場で頼ってきました．本当に心の支えでした．

同じ研究室の先輩と後輩．同期と同じく，困った時に相談したら暖かく手伝ってくれました．

両親．大学院に通うお金を出してくれ，家でも相談に乗ったり家事をしてくれたり，多くのことをしていただきました．

皆様本当にありがとうございました．心よりお礼申し上げます．

参考文献

[1] Felipe Restrepo-Calle, Jhon J. Ramírez Echeverry, and Fabio A. González. Continuous assessment in a computer programming course supported by a software tool. *Computer Applications in Engineering Education*, Vol. 27, No. 1, pp. 80–89, 2019.

[2] Petri Ihantola, Tuukka Ahoniemi, Ville Karavirta, and Otto Seppälä. Review of recent systems for automatic assessment of programming assignments. In *Proc. Koli Calling International Conference on Computing Education Research*, pp. 86–93, 2010.

[3] Stephen H. Edwards and Manuel A. Pérez-Quiñones. Experiences using test-driven development with an automated grader. *Journal of Computing Sciences in Colleges*, Vol. 22, No. 3, pp. 44–50, 2007.

[4] Christopher Douce, David Livingstone, and James Orwell. Automatic test-based assessment of programming: A review. *J. Educ. Resour. Comput.*, Vol. 5, No. 3, pp. 4–es, 2005.

[5] Chetan Desai, David Janzen, and Kyle Savage. A survey of evidence for test-driven development in academia. *SIGCSE Bulletin*, Vol. 40, No. 2, pp. 97–101, 2008.

[6] Brian P. Eddy, Norman Wilde, Nathan A. Cooper, Bhavyansh Mishra, Valeria S. Gamboa, Keenal M. Shah, Adrian M. Deleon, and Nikolai A. Shields. A pilot study on introducing continuous integration and delivery into undergraduate software engineering courses. In *Proc. Conference on Software Engineering Education and Training*, pp. 47–56, 2017.

[7] Cem Kaner, Copyright Cem, and Kaner All. The impossibility of complete testing, 1997.

[8] Saswat Anand, Edmund K. Burke, Tsong Yueh Chen, John Clark, Myra B. Cohen, Wolfgang Grieskamp, Mark Harman, Mary Jean Harrold, Phil McMinn, Antonia Bertolino, J. Jenny Li, and Hong Zhu. An orchestrated survey of methodologies for automated software test case generation. *Journal of Systems and Software*, Vol. 86, No. 8, pp. 1978–2001, 2013.

[9] Zahra Sadri-Moshkenani, Justin Bradley, and Gregg Rothermel. Survey on test case generation, selection and prioritization for cyber-physical systems. *Software Testing, Verification and Reliability*, p. e1794, 2022.

[10] Phil McMinn. Search-based software test data generation: A survey: Research articles. *Software Testing, Verification and Reliability*, pp. 105–156, 2004.

[11] Gordon Fraser and Andrea Arcuri. Evosuite: automatic test suite generation for object-oriented software. In *Proc. ACM SIGSOFT Symposium and European Conference on Foundations of Software Engineering*, pp. 416–419, 2011.

[12] Carlos Pacheco and Michael D. Ernst. Randoop: Feedback-directed random testing for java. In *Proc. ACM SIGPLAN Conference on Object-Oriented Programming Systems and Applications Companion*, pp. 815–816, 2007.

[13] Fabrizio Pastore, Leonardo Mariani, and Gordon Fraser. Crowdoracles: Can the crowd solve the oracle problem? In *Proc. International Conference on Software Testing, Verification and Validation*, pp. 342–351, 2013.

[14] Earl T. Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. The oracle problem in software testing: A survey. *IEEE Transactions on Software Engineering*, pp. 507–525, 2015.

[15] Fitsum Kifetew, Xavier Devroey, and Urko Rueda. Java unit testing tool competition: Seventh round. In *Proc. International Workshop on Search-Based Software Testing*, pp. 15–20, 2019.

[16] Pietro Braione, Giovanni Denaro, Andrea Mattavelli, and Mauro Pezzè. Combining symbolic execution and search-based testing for programs with complex heap inputs. In *Proc. ACM SIGSOFT International Symposium on Software Testing and Analysis*, pp. 90–101, 2017.

[17] Muhammad Khatibsyarbini, Mohd Adham Isa, Dayang N.A. Jawawi, and Rooster Tumeng. Test case prioritization approaches in regression testing: A systematic literature review. *Information and Software Technology*, Vol. 93, pp. 74–93, 2018.

[18] Jinfu Chen, Lili Zhu, Tsong Yueh Chen, Dave Towey, Fei-Ching Kuo, Rubing Huang, and Yuchi Guo. Test case prioritization for object-oriented software: An adaptive random sequence approach based on clustering. *Journal of Systems and Software*, Vol. 135, pp. 107–125, 2018.

[19] Ryan Carlson, Hyunsook Do, and Anne Denton. A clustering approach to improving test case prioritization: An industrial case study. In *Proc. IEEE International Conference on Software Maintenance*, pp. 382–391, 2011.

[20] Xiao Liu, Shuai Wang, Pei Wang, and Dinghao Wu. Automatic grading of programming assignments: An approach based on formal semantics. In *Proc. International Conference on Software Engineering: Software Engineering Education and Training*, pp. 126–137, 2019.