修　士　論　文

確率密度関数を用いた回帰テストのためのテストケース優先順位付け戦略

A Strategy of Test Case Prioritization for Regression Testing Using Probability Density Function

指導教員　佐藤 周行 准教授

東京大学工学系研究科
電気系工学専攻

氏　名　**37-176440 齋藤 雄太**

提　出　日　平成 31 年 1 月 30 日

概 要

ソフトウェアテスティングは、テストを実行することでシステムが意図したとおりに動作するかどうかを検証する手法である。中でも既存のソフトウェアが変更された際に実行されるテストを回帰テストと呼ぶ。ソースコードに頻繁に生じる変更の度に回帰テストが行われるため、そのコストが膨大になるという問題がある。回帰テストのコストを削減するための手法の一つとしてテストケース優先順位付け (TCP) が研究されている。TCP では不具合を早く発見できる確率を高めるようにテストケースの実行順序を並び替える。それぞれのテストがコードのどの部分を網羅したかを表すカバレッジ情報を用いて優先順位付けを行うカバレッジベースの TCP が主に用いられているが、既存の優先順位付け戦略では対象によって最適な戦略やパラメータが異なるという問題がある。

本論文では、TCP における新しい戦略アルゴリズムとして、実行した際に新たに検出される不具合の個数の期待値が最も高くなるテストケースを順に選んでいくような優先順位付け戦略を提案する。不具合の期待値はあるユニットを実行した際にある不具合が検出される確率の分布を表す確率密度関数を用いて計算できるため、この確率密度関数をデータセットを用いて推定し、その分布がベータ分布に従うという仮定の下で不具合の期待値を計算した。

提案した戦略と既存の戦略について、40 個の C 言語のプログラムと 27 個の Java のプログラムに対してそれぞれ実験を行い、T 検定により比較を行った結果、statement カバレッジを用いた場合、提案した戦略によって TCP の性能が有意に向上する結果が得られた。

目 次

図 目 次

表 目 次

アルゴリズム，擬似コード

第1章　序論

1.1　背景

ソフトウェアテスティングとは、ソフトウェアシステムの実行を観察し、意図したとおりに動作するかどうかを検証することである [6]。テストを実行した結果を意図していた結果と照らし合わせ、一致しない場合そのシステムに不具合があることが分かる。不具合がないことを証明することは出来ないが、複数のテストを行うことでそのシステムが正しく動くことを保証することができる。一般に、数学的に厳密な検証を行うモデルチェッキングと比較して計算量が小さいため、ソフトウェアテスティングはソフトウェア開発に不可欠であり、品質保証のために広く用いられている。中でも、既存のソフトウェアが変更された際に実行されるテストを回帰テストと呼ぶ。回帰テストの目的は、新しい変更が既存の変更されていない部分の動作を妨げていないことを検証することである [27]。ソースコードは新しい機能の追加や既知のバグの修正によって頻繁に変更されており、その度にテストを実行する必要があるため、回帰テストにかかるコストは膨大であり [18]、多くのシステムの開発においてテストに費やされるコストは全体の 50 %を超える [19]。したがって、回帰テストのコストを削減することはソフトウェア開発の効率の向上に大きく貢献する。

ソフトウェアテスティングにかかるコストを削減するために進められている研究分野の一つとして、テストケース優先順位付け (TCP) が挙げられる。回帰テストにおける TCP とは、不具合を早期に検出できるようにテストケースの実行順序を並び替えることである [18]。より少ないテストで不具合を検出できれば残りのテストを実行しなくても修正が必要であることが分かるため、テストにかかる時間を減らすことが出来る。また、上位に順位付けされたテストケースのみを実行することによってもコストを削減できる。TCP の多くはカバレッジベースであり、テストケースがソースコードのどの部分を実行するかを表すカバレッジ情報を用いたものである [24]。カバレッジベースの TCP の基礎的な戦略は Total Strategy と Additional Strategy であり、多くの TCP は 2 つの戦略をベースにカバレッジ以外の情報を用いて順位付けを行っている。しかし、Total Strategy と Additional Strategy にはそれぞれに長所と短所があり、どちらの戦略を使うべきかは場合によって異なるという問題がある [9]。Hao らは 2 つの戦略を統合した戦略として Unified Strategy を提案しており、両戦略に劣ることはない性能を得られている [9] が、パラメータのチューニングが必要であり、この問題の本質的な解決には至っていない。

1.2 本論文の提案

本論文では、TCP の性能を向上させること、汎用的に有効な戦略を求めることを目標として、Unified Strategy に代わる新しい戦略を提案する。この戦略はユニットを実行した際に不具合が見つかる確率を確率密度関数の形で表現するものであり、Unified Strategy の拡張になっている。Unified Strategy と同様に、新たに不具合が見つかる個数の期待値が最大となるテストケースから順に選択していく戦略であるが、優先順位付けに用いる期待値は確率密度関数から計算される。戦略に用いる確率密度関数は、各ユニットを実行した際に不具合が検出される確率をデータセットから最尤推定し、その分布によって推定した。

また、提案手法の応用として複数の metric から確率密度関数のパラメータを推定することで、ユニットによって異なる確率密度関数で優先順位付けを行う戦略を生成した。

1.3 本論文の貢献

本論文の貢献は以下のとおりである。

1. テストケース優先順位付けの戦略に確率密度関数の概念を取り入れ、新たな戦略を考案した。既存手法との比較実験を行った結果、一部の条件において既存手法より優位に良い結果が得られた。
2. 不具合の検出される確率密度関数をデータセットから最尤推定によって推定し、ベータ分布によって近似できることを発見した。
3. metric から確率密度関数のパラメータを推定する手法を提案し、TCP に機械学習を適用できる可能性を示した。

1.4 本論文の構成

本論文の構成は以下のとおりである。

第 2 章 TCP と優先順位付け戦略

本研究の研究対象である TCP と優先順位付け戦略の概念や既存手法、評価指標について述べる。

第 3 章 提案手法

本研究の提案手法である確率密度関数を用いた優先順位付け戦略と、データセットから確率密度関数を推定する手法について具体的に説明する。

第 4 章 実験

提案した戦略の性能を評価するために行った確率密度関数の推定、近似と既存戦略との比較についての実験の内容とその結果、考察をまとめる。

第 5 章 関連研究

本研究に関連する、様々なカバレッジベースの TCP の研究についてまとめる。

第 6 章 結論

本論文のまとめと、今後の課題を述べる。

第2章　TCPと優先順位付け戦略

この章では本研究の研究対象であるTCPと優先順位付け戦略の概念や既存手法、評価指標について述べる。

2.1　Test Case Prioritization (TCP)

テストケース優先順位付け (TCP) とは、何らかの基準に従ってテストケースを実行順に並び替えることであり、その目的は障害を早い段階で検出することやシステムの信頼性を早期に高めることなどである [20]。特に回帰テストにおいては、不具合が発見された時点で以降のテストをする必要がなくなるため、テストケースの実行順序を最適化することで不具合をより早く発見し、コストを削減することがTCPの目標となる [8]。

TCPが行うことは、テストスイート (テストケースの集合) T の可能な全ての置換の集合を PT、目的関数を F として、

$$(\forall P'')\,(P'' \in PT)\,(P'' \neq P')\ ,\ F\,(P') \geq F\,(P'') \tag{2.1}$$

を満たすような置換 P' を求めることである [21]。

2.1.1　目的関数

TCPの目的関数として用いられている指標の1つにAverage Percentage Faults Detected (APFD) [20] がある。これは不具合がどれだけ早期に発見されているかを表す指標である。

APFDは、並び替えたテストケースの n 番目までで発見できる不具合の数 $\mathcal{F}_n$ とテストケースの総数 N を用いて

$$APFD = \frac{\sum_{n=1}^{N-1} \mathcal{F}_n}{N \times \mathcal{F}_N} + \frac{1}{2N} \tag{2.2}$$

で与えられる。APFDは全ての不具合が最初のテストケースで見つかる場合に $\mathcal{F}_1 = \mathcal{F}_2 = ... = \mathcal{F}_N$ となるため最大値 $1 - \frac{1}{2N}$ をとり、全ての不具合が最後のテストケースで見つかる場合に $\mathcal{F}_1 = \mathcal{F}_2 = ... = \mathcal{F}_{N-1} = 0$ となるため最小値 $\frac{1}{2N}$ をとる。

2.1.2 カバレッジベースの TCP

TCP の多くはカバレッジベースの TCP と呼ばれ、カバレッジ情報を用いて優先順位付けを行う [24]。カバレッジ情報とは、一定のユニットに分けられたソースコードに対して、各テストケースが実行された時にそれぞれの部分が実行されるかという情報のことである。情報として、ユニットが実行されたか否かという 2 値の情報を用いる場合と、何回実行されたかという回数を用いる場合がある。ソースコードを区切る粒度の単位として statement、method、class 等が用いられる。カバレッジ情報の取得にはコールグラフが用いられ、静的にコードを分析する静的コードカバレッジと、以前のバージョンのテスト実行時に得られるカバレッジ情報を用いる動的コードカバレッジに大別される [15]。

2.2 優先順位付け戦略

TCP における優先順位付け戦略とは、カバレッジなどの情報を入力としテストケースの実行順を出力する関数を指す。言い換えると、各テストケースのカバレッジなどから、どのテストケースを何番目に実行するかを決定するものが優先順位付け戦略である。カバレッジベースの TCP の優先順位付け戦略のうち、最も古典的な戦略は Total Strategy と Additional Strategy であり、多くの TCP は 2 つの戦略をベースにカバレッジ以外の情報を用いて順位付けを行っている [9]。

2.2.1 Total Strategy

Total Strategy は、カバーするユニットの多いテストケースから順に選択していく戦略である。T、P、$Cover\,[n,m]$、$Sum\,[n]$、$Selected\,[n]$、$Priority\,[i]$ をそれぞれ

- $T = \{t_1, t_2, ..., t_N\}$:$N$ 個のテストケース t_n を含むテストスイート
- $P = \{u_1, u_2, ..., u_M\}$:$M$ 個のユニット u_m を含むプログラム
- $Cover\,[n,m]$:テストケース t_n がユニット u_m をカバーしているかを表すカバレッジ情報
- $Sum\,[n]$:テストケース t_n に関する優先順位付けに用いる和
- $Selected\,[n]$:テストケース t_n が既に選択されたかどうかを表すブール値
- $Priority\,[i]$:i 番目にどのテストケースが順序付けされたかを表す

と定義すると、アルゴリズムは Algorithm 2.1 と書ける。

1-7 行目では、各テストケース t_n について、変数 $Selected\,[n]$ の初期化とカバレッジの総和

$$Sum\,[n] = \sum_{1 \le m \le M} Cover\,[n,m] \tag{2.3}$$

の計算を行っている。

Algorithm 2.1 Prioritization in the Total Strategy

Require: coverage information $cover\,[n, m]$

Ensure: proritization $Priority\,[i]$

```
for each n  (1 ≤ n ≤ N) do
    Selected [n] ← false
    Sum [n] ← 0
    for each m  (1 ≤ m ≤ M) do
        Sum [n] ← Sum [n] + Cover [n, m]
    end for
end for
for each i  (1 ≤ i ≤ N) do
    n ← 1
    while Selected [n] do
        n ← n + 1
    end while
    for each n′  (n + 1 ≤ n′ ≤ N) do
        if not Selected [n′] then
            if Sum [n′] > Sum [n] then
                n ← n′
            end if
        end if
    end for
    Priority [i] ← n
    Selected [n] ← true
end for
```

8-21 行目ではテストケースを 1 つずつ選択し、$Priority[i]$ に格納していく。9-12 行目でまだ選択されていないテストケースのうち、最も n が小さい t_n を取得し、13-19 行目で他の選択されていないテストケースと $Sum[n]$ を比較して、これが最大となる t_n を得る。20 行目で $Priority[i]$ に n を格納し、21 行目では t_n が選択済みであることを示すフラグ $Selected[n]$ を $true$ に変更する。

Total Strategy の最悪時間計算量は $O(MN)$ である [20]。

2.2.2 Additional Strategy

Additional Strategy はそれまでに選択したテストケースがカバーしていないユニットをより多くカバーするテストケースを順に選択していく戦略であり、そのアルゴリズムは Algorithm2.2 となる。

1-3 行目では $Selected[n]$ を、4-6 行目では $Covered[m]$ を $false$ に初期化する。ここで $Covered[m]$ は、ユニット u_m がこれまでに選択されたテストケースによってカバーされたかどうかを表すブール値の変数である。

7-30 行目でテストケースを順に選択し、$Priority[i]$ に格納していく。ループ内の 8 行目で変数の初期化を行い、9-22 行目で次に選択するテストケース t_n を計算する。これまでに選択されていないテストケース $t_{n'}$ について、これまでにカバーされておらず、$t_{n'}$ によってカバーされるユニットの総数が最大となるテストケースを計算する。23-24 行目で $Priority[i]$、$Selected[n]$ をそれぞれ更新し、25-29 行目では選択された t_n がカバーする全てのユニット u_m について、$Covered[m]$ を $false$ に更新する。

Additional Strategy の最悪時間計算量は $O\left(MN^2\right)$ である [20]。

2.2.3 Unified Strategy

Total Strategy はユニットのカバー回数を最大化するように優先順位付けができる一方、それまでに選択したテストケースのカバレッジ情報がその後の選択に影響しないため、カバーするカバレッジ情報の少ないテストケースのみでカバーされる部分の検証が遅くなる恐れがある。Additional Strategy はカバレッジの網羅率を最大化するように優先順位付けができる一方、一度カバーされたユニットについてのカバレッジ情報はその後の選択に影響しないため、一度不具合が検出されなかった部分の検証が不十分になる恐れがある。したがって、2 つの戦略のうちどちらが最適かは場合によって異なる。

この問題を解決するため、Hao らは Total Strategy と Additional Strategy を統合した戦略として Unified Strategy を提案した [9]。

概要

Unified Strategy の特長は、区間 $(0,1)$ で定義されるパラメータ c の値によって Total Strategy と Additional Strategy のどちらに近い戦略かを調整することができるようになっている点である。ここでパラメータ c は 1 つのユニットを実行した際に不具合が検出される確率を意味し、ユニット u_m

Algorithm 2.2 Prioritization in the Additional Strategy

Require: coverage information $cover\,[n, m]$

Ensure: proritization $Priority\,[i]$

```
for each n  (1 ≤ n ≤ N) do
    Selected [n] ← false
end for
for each n  (1 ≤ n ≤ N) do
    Covered [m] ← false
end for
for each i  (1 ≤ i ≤ N) do
    n ← 0 ,  sum ← −1
    for each n′  (1 ≤ n′ ≤ N) do
        if not Selected [n′] then
            s ← 0
            for each m  (1 ≤ m ≤ M) do
                if Covered [m] = false and Cover [n′, m] then
                    s ← s + 1
                end if
            end for
            if s > sum then
                sum ← s
                n ← n′
            end if
        end if
    end for
    Priority [i] ← n
    Selected [n] ← true
    for each m  (1 ≤ m ≤ M) do
        if Cover [n, m] then
            Covered [m] ← true
        end if
    end for
end for
```

が未発見の不具合を含んでいる確率を表す変数 $Prob\,[m]$ を更新し、その重みづけ和によって優先順位の決定を行う。

アルゴリズム

Unified Strategy の戦略を Algorithm2.3 に示す。

1-6 行目では $Prob\,[m]$ を 1 に、$Selected\,[n]$ を 0 にそれぞれ初期化している。$Prob\,[m]$ を 1 に初期化するのは、テストケースを選択していない状態においてはどのユニットからも不具合は検出されないためである。

7-26 行目ではテストケースを 1 つずつ選択し、$Priority\,[i]$ に格納していく。ループ内の 8-22 行目では、まだ選択されていないテストケースのうち、カバーするユニットが未検出の不具合を検出する確率の合計が最大となるようなものを選択する。

ここでテストケース t_n を実行した際にユニット u_m が新たに不具合を検出する確率について考える。ユニットを実行した際に不具合が検出される確率は全ての試行において一定値 c を取るため、あるユニットを x 回実行した際に不具合が検出される確率は $1-(1-c)^x$ である。したがって、あるユニット u_m が未知の不具合を含んでいる確率が $Prob\,[m]$ であるとき、u_m を $Cover\,[n,m]$ 回実行した際に新たに不具合が検出される確率は

$$Prob\,[m] \times \left(1-(1-c)^{Cover[n,m]}\right) \tag{2.4}$$

となる。

以上の計算から、

$$Sum\,[n] = \sum_{m=1}^{M} \left(Prob\,[m] \times \left(1-(1-c)^{Cover[n,m]}\right)\right) \tag{2.5}$$

が最大値をとるような t_n を選べばよい。

23-25 行目では、$Prob$ の値を更新する。ユニット u_m が未検出の不具合を含んでいる確率 $Prob\,[m]$、テストケース t_n の実行により $Cover\,[n,m]$ 回カバーされた際に不具合が検出されない確率 $(1-c)^{Cover[n,m]}$ より、更新式は

$$Prob\,[m] \leftarrow Prob\,[m] \times (1-c)^{Cover[n,m]} \tag{2.6}$$

となる。

古典的戦略との関連性

c が 0 に限りなく近い場合について考えると、

$$(1-c)^{Cover[n,m]} \simeq 1 - Cover\,[n,m] \times c \simeq 1 \tag{2.7}$$

Algorithm 2.3 Prioritization in the Unified Strategy [9]

Require: coverage information $cover\,[n, m]$

Ensure: proritization $Priority\,[i]$

```
for each m (1 ≤ m ≤ M) do
    Prob[m] ← 1
end for
for each n (1 ≤ n ≤ N) do
    Selected[n] ← false
end for
for each i (1 ≤ i ≤ N) do
    n ← 0 , sum ← −1
    for each n′ (1 ≤ n′ ≤ N) do
        if not Selected[n′] then
            s ← 0
            for each m (1 ≤ m ≤ M) do
                s ← s + Prob[m] × (1 − (1 − c)^Cover[n,m])
            end for
            if s > sum then
                sum ← s
                n ← n′
            end if
        end if
    end for
    Priority[i] ← n
    Selected[n] ← true
    for each m (1 ≤ m ≤ M) do
        Prob[m] ← Prob[m] × (1 − c)^Cover[n,m]
    end for
end for
```

より、式 (2.6) は

$$\begin{aligned} Prob\,[m] \leftarrow & Prob\,[m] \times (1-c)^{Cover[n,m]} \\ \simeq & Prob\,[m] \end{aligned} \tag{2.8}$$

となるため、常に $Prob\,[m] \simeq 1$ となる。したがって、$c \simeq 0$ のとき

$$1-(1-c)^{Cover[n,m]} \simeq Cover\,[n,m] \times c \tag{2.9}$$

が成り立つため、式 (2.5) は

$$\begin{aligned} Sum\,[n] &= \sum_{m=1}^{M} \left(Prob\,[m] \times \left(1-(1-c)^{Cover[n,m]} \right) \right) \\ &\simeq c \times \sum_{m=1}^{M} Cover\,[n,m] \end{aligned} \tag{2.10}$$

と近似できる。式 (2.3) との比較から、この戦略は $c \to 0$ のとき Total Strategy に限りなく近づく。

また、c が 1 に限りなく近いとき、

$$\lim_{c \to 1} (1-c)^{Cover[n,m]} = \begin{cases} 1 & (Cover\,[n,m]=0) \\ 0 & (Cover\,[n,m]>0) \end{cases} \tag{2.11}$$

より、式 (2.6) は

$$Prob\,[m] \leftarrow Prob\,[m] \times (1-c)^{Cover[n,m]} \simeq \begin{cases} Prob\,[m] & (Cover\,[n,m]=0) \\ 0 & (Cover\,[n,m]>0) \end{cases} \tag{2.12}$$

となるため、既に選択したテストケースが u_m をカバーしている場合は $Prob\,[m] \simeq 0$、カバーしていない場合は $Prob\,[m] \simeq 1$ となる。したがって、式 (2.5) は $U_n = \{u_m | Cover\,[n,m] > 0\}$ を用いて

$$\begin{aligned} Sum\,[n] &= \sum_{m=1}^{M} Prob\,[m] \times \left(1-(1-c)^{Cover[n,m]} \right) \\ &\simeq \sum_{m \in U_n} Prob\,[m] \end{aligned} \tag{2.13}$$

となるため、$Sum\,[n]$ は既に選択したテストケースによってカバーされず、t_n がカバーするユニットの個数とほぼ等しい。以上より、この戦略は $c \to 1$ のとき Additional Strategy に限りなく近づく。

そして $0 < c < 1$ のときは、c が 0 に近いほど Total Strategy に、1 に近いほど Additional Strategy に近い、二つを複合した戦略となる。この戦略の最悪時間計算量は Additional Strategy と同じ $O\left(MN^2\right)$ である。

Metric を用いた Unified Strategy

Hao らは Unified Strategy の発展として、カバレッジ以外の metric によってパラメータ p の値を変更する戦略も提案している。ここで metric とはプログラム等から計算される尺度であり、例として各ユニットの行数などが挙げられる。ユニット u_m に対応する metric を $Metric\,[m]$ としたとき、ユニット u_m を実行した際に不具合が検出される確率 c_m を

$$c_m = p_{high} - (p_{high} - p_{low}) \times \frac{Metric\,[m] - Metric_{min}}{Metric_{max} - Metric_{min}} \tag{2.14}$$

もしくは

$$c_m = p_{high} - (p_{high} - p_{low}) \times \frac{\log_{10}(Metric\,[m] + 1) - \log_{10}(Metric_{min} + 1)}{\log_{10}(Metric_{max} + 1) - \log_{10}(Metric_{min} + 1)} \tag{2.15}$$

とする。ただし、p_{high}、p_{low} はそれぞれ c_m の最大値、最小値を表すパラメータで、$Metric_{max}$、$Metric_{min}$ はそれぞれ

$$Metric_{max} = \max_{1 \le m \le M} Metric\,[m] \tag{2.16}$$

$$Metric_{min} = \min_{1 \le m \le M} Metric\,[m] \tag{2.17}$$

である。

第3章　提案手法

この章では本研究の提案手法である確率密度関数を用いた優先順位付け戦略と、データセットから確率密度関数を推定する手法について具体的に説明する。

3.1　確率密度関数による戦略の生成

3.1.1　概要

Unified Strategy では、ユニットを実行した際に不具合が検出される確率を定数 c で表現していたが、実際にはこの確率はテストの実行前に推定することはできず、ユニットによってそれぞれ異なる値 p_m をとると考えられる。p_m の分布を $f(p)$ とおくと、これは不具合が検出される確率についての確率密度関数である。この確率密度関数を用いて新たに不具合が検出される確率を計算する。

3.1.2　アルゴリズム

不具合が検出される確率 p_m の分布が $f(p)$ に従うとき、u_m が x 回カバーされた時に 1 回以上不具合が検出される確率は

$$\begin{aligned} P(x) &= \int_0^1 \left(1-(1-p)^x\right) f(p)\, dp \\ &= 1 - \int_0^1 (1-p)^x f(p)\, dp \end{aligned} \tag{3.1}$$

となる。したがって、あるユニット u_m について、これまでに選択したテストケースが合計 $Count[m]$ 回、あるテストケース t_n が $Cover[n,m]$ 回カバーする t_n によってはじめて u_m の不具合が検出される確率は

$$P(Count[m] + Cover[n,m]) - P(Count[m]) \tag{3.2}$$

となるため、t_n によってはじめて検出される不具合の合計数の期待値

$$Sum[n] = \sum_{m=1}^{M} \left(P(Count[m] + Cover[n,m]) - P(Count[m])\right) \tag{3.3}$$

が最大となるテストケースを順に選択する戦略が考えられる。

以上を踏まえて、提案手法のアルゴリズムを Algorithm 3.1 に示す。この戦略の最悪時間計算量は Additional Strategy や Unified Strategy と同じ $O\left(MN^2\right)$ である。

Algorithm 3.1 Prioritization in the proposed method

Require: coverage information $cover\,[n, m]$

Ensure: proritization $Priority\,[i]$

```
for each m (1 ≤ m ≤ M) do
    Cover [m] ← 0
end for
for each n (1 ≤ n ≤ N) do
    Selected [n] ← false
end for
for each i (1 ≤ i ≤ N) do
    n ← 0 , sum ← −1
    for each n′ (1 ≤ n′ ≤ N) do
        if not Selected [n′] then
            s ← 0
            for each m (1 ≤ m ≤ M) do
                if Cover [n′, m] then
                    s ← s + P(Count [m] + Cover [n′, m]) − P(Count [m])
                end if
            end for
            if s > sum then
                sum ← s
                n ← n′
            end if
        end if
    end for
    Priority [i] ← n
    Selected [n] ← true
    for each m (1 ≤ m ≤ M) do
        Count [m] ← Count [m] + Cover [n, m]
    end for
end for
```

3.1.3 Unified Strategy との関連

Unified Strategy [9] の $Prob\,[m]$、$Sum\,[n]$ はそれぞれ $Count\,[m]$ を用いて

$$Prob\,[m] = (1-c)^{Count[m]}\ , \tag{3.4}$$

$$\begin{aligned} Sum\,[n] &= \sum_{m=1}^{M} Prob\,[m] \times \left(1-(1-c)^{Cover[n,m]}\right) \\ &= \sum_{m=1}^{M} \left((1-c)^{Count[m]} - (1-c)^{Count[m]+Cover[n,m]}\right) \end{aligned} \tag{3.5}$$

と表せる。一方、提案手法の戦略において $f\,(p) = \delta\,(p-c)$ とおくと

$$P\,(x) = 1-(1-c)^{x} \tag{3.6}$$

より、

$$\begin{aligned} Sum\,[n] &= \sum_{m=1}^{M} \left(P\,(Count\,[m] + Cover\,[n,m]) - P\,(Count\,[m])\right) \\ &= \sum_{m=1}^{M} \left((1-c)^{Count[m]} - (1-c)^{Count[m]+Cover[n,m]}\right) \end{aligned} \tag{3.7}$$

となるため式 (3.5) と (3.7) から、Unified Strategy は提案手法において不具合の見つかる確率を c に固定した場合と等しいと言える。$f\,(p) = \delta\,(p-c)$ の形で表される確率密度関数が最良であるか否かについては検証する必要があり、その上で最適な確率密度関数を予測することが本研究の目的となる。

3.2 確率密度関数の推定

L 個の不具合の集合 $F = \{f_1, f_2, ..., f_L\}$ のうちのある不具合 f_l について、ユニット u_m を 1 回カバーした際にこれが検出される確率が各試行に対して常に一定値 $p_{m,l}$ をとるとする。この $p_{m,l}$ が確率密度関数 $f\,(p)$ に基づいて生成されていると考えると、反対に $f\,(p)$ は $p_{m,l}$ を用いて

$$f\,(p) = \frac{1}{ML} \sum_{m=1}^{M} \sum_{l=1}^{L} \delta\,(p - p_{m,l}) \tag{3.8}$$

という形で離散的に近似できる。

以上の議論から、$p_{m,l}$ の値をデータセットから推定し、連続な分布によって近似を行うことで連続的な確率密度関数を得ることを考える。

3.2.1 最尤推定

最尤推定とは、データの従う分布が分かっているがそのパラメータ θ が分かっていない状態において、観察されたデータが得られる確率が最も大きくなるような θ を探す手法である。

観測されたデータが得られる確率を尤度関数 $L(\theta)$ と呼び、これは観測されたデータを定数、θ を変数とする関数である。尤度関数を最大化する θ は尤度方程式

$$\frac{\partial}{\partial\theta}\log L(\theta) = 0 \tag{3.9}$$

の解として求められる。

TCP のデータセットからはカバレッジ情報 $Cover[n,m]$ と、実行した際に不具合 f_l が検出されるようなテスト t_n の集合 T_{f_l} が得られる。これらを用いて $p_{m,l}$ を最尤推定することを考える。

ある不具合 f_l について考えると、テスト t_n を実行した際に不具合が検出される確率は、

$$P_{n,l} = 1 - \prod_m (1 - p_{m,l})^{Cover[n,m]} \tag{3.10}$$

となる。f_l についての尤度関数は T_{f_l} に含まれるテストケースから不具合が検出され、それ以外のテストケースから検出されない確率であるため、

$$L(\boldsymbol{p}_l) = \prod_{t_n \in T_{f_l}} P_{n,l} \times \prod_{t_n \notin T_{f_l}} (1 - P_{n,l}) \tag{3.11}$$

となる。ただし、$\boldsymbol{p}_l = (p_{1,l}, p_{2,l}, ..., p_{M,l})$ である。したがって、

$$\begin{aligned}\frac{\partial}{\partial p_{m,l}}\log L(\boldsymbol{p}_l) =& \frac{\partial}{\partial p_{m,l}}\left(\sum_{t_n \in T_{f_l}} \log P_{n,l} + \sum_{t_n \notin T_{f_l}} \log(1 - P_{n,l})\right) \\ =& \sum_{t_n \in T_{f_l}} \frac{P_{n,l} - 1}{P_{n,l}} \frac{Cover[n,m]}{p_{m,l} - 1} + \sum_{t_n \notin T_{f_l}} \frac{Cover[n,m]}{p_{m,l} - 1}\end{aligned} \tag{3.12}$$

より、尤度方程式は

$$\sum_{t_n \in T_{f_l}} \frac{P_{n,l} - 1}{P_{n,l}} \frac{Cover[n,m]}{p_{m,l} - 1} + \sum_{t_n \notin T_{f_l}} \frac{Cover[n,m]}{p_{m,l} - 1} = 0 \tag{3.13}$$

となる。

尤度方程式を解く手法の一つに最急降下法がある。最急降下法は勾配法の一種で、関数の傾きのみから極小値もしくは極大値を探索するアルゴリズムであり、更新式

$$\boldsymbol{w}_{t+1} = \boldsymbol{w}_t - \alpha \boldsymbol{\nabla} f(\boldsymbol{w}_t) \tag{3.14}$$

によってパラメータを更新する。全体としての手順は

1. パラメータ $\boldsymbol{w}$ の初期値 $\boldsymbol{w}_0$ をランダムに設定し、$t = 0$ とする。

2. 勾配 $\boldsymbol{\nabla} f(\boldsymbol{w}_t)$ を求める。
3. $|\boldsymbol{\nabla} f(\boldsymbol{w}_t)|$ が一定値を下回った場合、終了する。
4. $t \leftarrow t+1$ とし、更新式 (3.14) によって w_t の値を更新する
5. 2 に戻る

となる。

α は学習率と呼ばれ、更新が進み収束に近づくに従って値を小さくしていくことが一般的である。収束を速める学習率として代表的なものが Adam [11] である。パラメータは $\alpha = 0.001$、$\beta_1 = 0.9$、$\beta_2 = 0.999$、$\epsilon = 10^{-8}$ が推奨されており、$\boldsymbol{m}_0 = \boldsymbol{v}_0 = \boldsymbol{0}$ として、更新式 (3.14) の代わりに

$$\begin{aligned}
\boldsymbol{m}_t =& \beta_1 \boldsymbol{m}_{t-1} + (1-\beta_1)\boldsymbol{\nabla} f(\boldsymbol{w}_{t-1}) \\
\boldsymbol{v}_t =& \beta_2 \boldsymbol{v}_{t-1} + (1-\beta_2)\boldsymbol{\nabla} f(\boldsymbol{w}_{t-1}) \circ \boldsymbol{\nabla} f(\boldsymbol{w}_{t-1}) \\
\hat{\boldsymbol{m}}_t =& \frac{\boldsymbol{m}_t}{1-{\beta_1}^t} \\
\hat{\boldsymbol{v}}_t =& \frac{\boldsymbol{v}_t}{1-{\beta_2}^t} \\
\boldsymbol{w}_t =& \boldsymbol{w}_{t-1} - \alpha \frac{\hat{\boldsymbol{m}}_t}{\sqrt{\hat{\boldsymbol{v}}_t} + \epsilon}
\end{aligned} \tag{3.15}$$

によって更新する。ただし、$\boldsymbol{u} \circ \boldsymbol{v}$ はアダマール積であり、ベクトルの要素ごとの積である。

本研究の最尤推定に当てはめる場合、尤度関数 $L(\boldsymbol{p}_l)$ を最大化するため、関数 $f(\boldsymbol{w})$ にあたるのは $-L(\boldsymbol{p}_l)$ となる。

3.2.2 ベータ近似

ベータ分布は範囲 $(0,1)$ で定義され、正の実数 α、β を用いて以下の式で表される。

$$f(p) = \frac{p^{\alpha-1}(1-p)^{\beta-1}}{B(\alpha,\beta)} \tag{3.16}$$

ただし、$B(\alpha,\beta)$ はベータ関数

$$B(\alpha,\beta) = \int_0^1 p^{\alpha-1}(1-p)^{\beta-1}\,dp \tag{3.17}$$

であり、ガンマ関数を用いて

$$B(\alpha,\beta) = \frac{\Gamma(\alpha)\Gamma(\beta)}{\Gamma(\alpha+\beta)} \tag{3.18}$$

と表される。この時、x 回の試行で 1 回以上不具合が見つかる確率 $P(x)$ は

$$\begin{aligned} P(x) =& 1 - \int_0^1 (1-p)^x \frac{p^{\alpha-1}(1-p)^{\beta-1}}{B(\alpha,\beta)} dp \\ =& 1 - \frac{B(\alpha,\beta+x)}{B(\alpha,\beta)} \end{aligned} \tag{3.19}$$

となる。

ベータ関数の期待値 $E[X]$ と分散 $var[X]$ はそれぞれ

$$\begin{aligned} E[X] =& \frac{\alpha}{\alpha+\beta} \\ var[X] =& \frac{\alpha\beta}{(\alpha+\beta)^2(\alpha+\beta+1)} \end{aligned} \tag{3.20}$$

と表せるので、逆に期待値と分散を用いてパラメータ α、β を

$$\begin{aligned} \alpha =& E[X]\left(\frac{E[X](1-E[X])}{var[X]} - 1\right) \\ \beta =& (1-E[X])\left(\frac{E[X](1-E[X])}{var[X]} - 1\right) \end{aligned} \tag{3.21}$$

と推定することができる。

3.2.3 Metric によるパラメータ調整

3.2.2 節により、$p_{m,l}$ の平均と分散から確率密度関数のパラメータが推定できることが導かれた。したがって、metric から平均、分散の値を推定するモデルを学習することで、metric によって異なる確率密度関数を適用させることが可能になる。

第4章　実験

この章では、提案した戦略の性能を評価するために行った確率密度関数の推定、近似と既存戦略との比較についての実験の内容とその結果、考察をまとめる。

4.1　データセット

実験には、Hao らの公開するデータセット[1]を用いた。このデータセットは 40 個の C 言語オブジェクトと 28 個の Java オブジェクトの method、statement 単位の動的カバレッジ情報と不具合の情報からなる。C 言語オブジェクトは GNU Core Utilities のバージョン 6.11 から選択され、MutGen [4] により不具合を、KLEE [7] によってテストケースをそれぞれ自動生成されている。Java オブジェクトは Software artifact Infrastructure Repository (SIR [2]) から取得され、Javalanche [22] もしくは MuJava [16] によって不具合を自動生成されたものである。

また、Java オブジェクトの method カバレッジについては 2 種類の metric が提供されている。McCabe [17] は循環的複雑度を表し、MLoC はコードの行数を意味する。

以上のデータセットから、40 個全ての C 言語のオブジェクトと、再現性が得られなかった jmeter1.0 を除いた 27 個の Java オブジェクトを実験に用いた。

4.2　確率密度関数の推定

3.2.1、3.2.2 節で述べた手法により、確率密度関数を離散的に推定する実験を行った。

4.2.1　実験設定

確率 $p_{m,l}$ には区間 $[0,1]$ で定義されるという制約条件があるため、$p_{m,l}$ をパラメータ $\theta_{m,l}$ を用いて

$$p_{m,l} = 1 - \frac{1}{1 + e^{\theta_{m,l}}} \tag{4.1}$$

とし、$\theta_{m,l}$ を推定した。

最急降下法の終了条件として、勾配の絶対値の最大値が ϵ を下回ることとし、ϵ の値はそれぞれのデータセットについて表 4.2 のようにした。

[1] https://sites.google.com/site/unifiedtestprioritization
[2] http://sir.unl.edu/

表 4.1. Parameter of learning termination condition

		ϵ
C	method	10^{-12}
C	statement	10^{-11}
Java	method	10^{-9}
Java	statement	10^{3}

表 4.2. Parameters of beta approximation

		α	β
C	method	1.75×10^{-2}	1.35×10^{-1}
C	statement	4.44×10^{-3}	3.84×10^{-2}
Java	method	2.61×10^{-4}	9.59×10^{-3}
Java	statement	3.87×10^{-5}	9.37×10^{-4}

また、推定された離散的な確率密度関数に対して、平均 $E[X]$ と分散 $var[X]$ を

$$E[X] = \frac{1}{ML} \sum_{m=1}^{M} \sum_{l=1}^{L} p_{m,l}$$

$$var[X] = \frac{1}{ML} \sum_{m=1}^{M} \sum_{l=1}^{L} (p_{m,l} - E[X])^2 \tag{4.2}$$

のように計算し、ベータ近似を行った。

4.2.2 実験結果

最尤推定を行い求めた確率密度関数の累積分布を図 4.1 に示す。ここで累積分布関数とは、確率密度関数 $f(p)$ を用いて

$$F(p) = \int_0^p f(p')\, dp' \tag{4.3}$$

で定義され、分布が 0 から p までの間の値をとる確率を表す。累積分布関数を用いる理由は離散的な確率密度関数の場合、概形を書くことが出来ないためである。

累積分布関数の形状より、確率 $p = 0, 1$ とその付近に分布が集中していることが分かる。どの関数についても 80% 以上が $p = 0$ 付近に分布しており、ソフトウェアの不具合の 80%はソースコードの 20%の部分に分布する [29] という経験則と矛盾しない結果が得られた。また、C オブジェクトと比較して Java オブジェクトの方が分布が $p = 1$ 側に、method カバレッジと比較して statement カバレッジの方が $p = 0, 1$ の両側に寄っている。

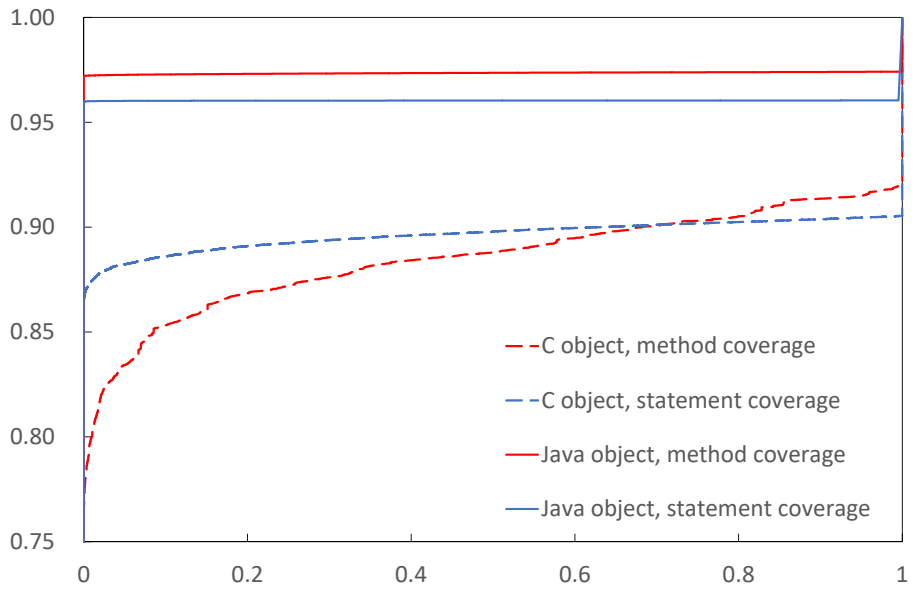

図 4.1. Cumulative distribution functions calculated by maximum likelihood estimation

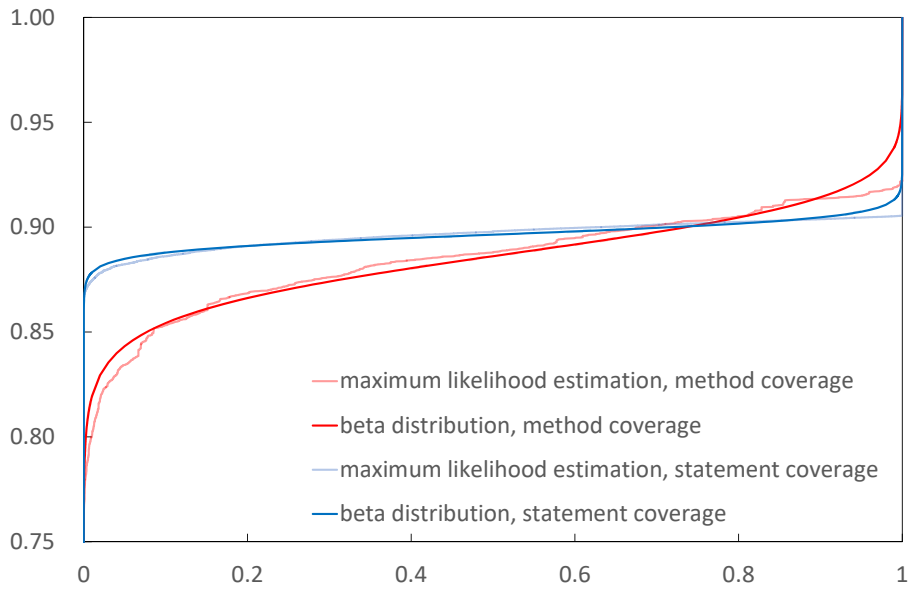

図 4.2. Cumulative distribution functions of C programs for test suites

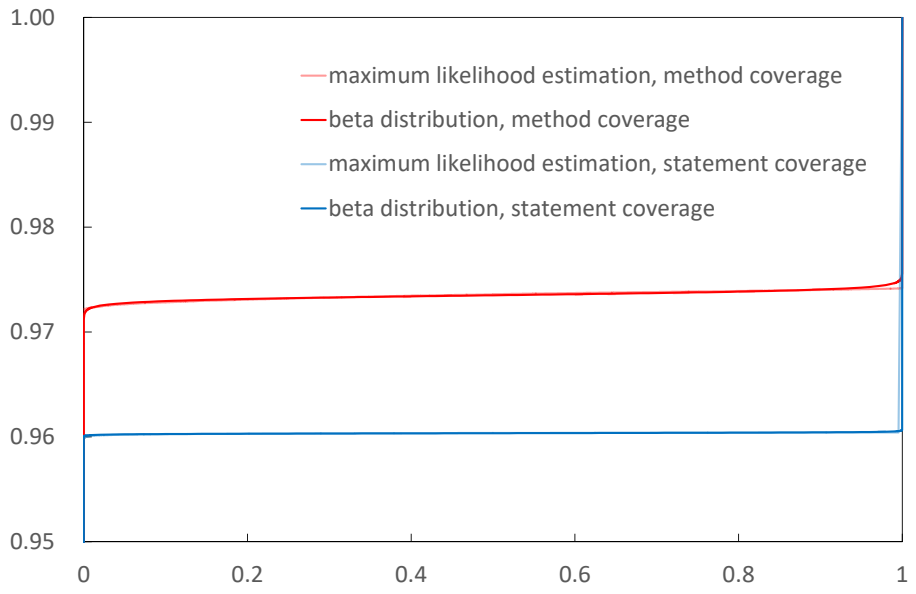

図 4.3. Cumulative distribution functions of Java programs for test suites

C 言語オブジェクトの method カバレッジについて、最尤推定によって求めた確率密度関数と、それを近似したベータ分布の累積分布を図 4.2 と 4.3 に、パラメータ α、β の値を表 4.2 に示す。各関数の形状の比較より、最尤推定で求めた離散的な確率密度関数を連続なベータ分布で近似できていることが分かる。

4.2.3 考察

実験で得られた確率密度関数を用いた戦略がどのような特徴を持つかを考察する。

不具合が x 回目に初めて検出される確率 $Q(x)$ は x 回目までに不具合が検出される確率 $P(x)$ を用いて

$$Q(x) = P(x) - P(x-1) \tag{4.4}$$

と表せる。式 (3.6) より、Unified Strategy において $Q(x)$ は

$$\begin{aligned} Q(x) =& P(x) - P(x-1) \\ =& (1-c)^{x-1} - (1-c)^{x} \\ =& c(1-c)^{x-1} \end{aligned} \tag{4.5}$$

となる。したがって、不具合が新たに検出される確率は $1-c$ 倍ずつ減少していく。この減少率が小さいほど確率を上昇させるためにカバーする回数が重要になり、大きいほどこれまでにカバーした回数が少ないことが重要になる。

一方、確率密度関数 $f(p)$ がベータ分布に従うとき、$P(x)$ は式 (3.18) 、 (3.19) より

$$\begin{aligned} P(x) =& 1 - \frac{B(\alpha, \beta+x)}{B(\alpha,\beta)} \\ =& 1 - \frac{\Gamma(\alpha)\Gamma(\beta+x)}{\Gamma(\alpha+\beta+x)} \times \frac{\Gamma(\alpha+\beta)}{\Gamma(\alpha)\Gamma(\beta)} \\ =& 1 - \frac{\Gamma(\alpha+\beta)}{\Gamma(\alpha+\beta+x)} \times \frac{\Gamma(\beta+x)}{\Gamma(\beta)} \\ =& 1 - \prod_{i=0}^{x-1} \frac{\beta+i}{\alpha+\beta+i} \end{aligned} \tag{4.6}$$

となる。ただし、

$$\Gamma(x+1) = x\Gamma(x) \tag{4.7}$$

を用いた。したがって $Q(x)$ は

$$\begin{aligned} Q(x) =& P(x) - P(x-1) \\ =& \prod_{i=0}^{x-2} \frac{\beta+i}{\alpha+\beta+i} - \prod_{i=0}^{x-1} \frac{\beta+i}{\alpha+\beta+i} \\ =& \frac{\alpha}{\alpha+\beta} \prod_{i=1}^{x-1} \frac{\beta+i-1}{\alpha+\beta+i} \end{aligned} \tag{4.8}$$

より、

$$Q(x+1) = \frac{\beta+x-1}{\alpha+\beta+x} Q(x) \tag{4.9}$$

となる。したがって、不具合が検出される確率は $\frac{\beta+x-1}{\alpha+\beta+x}$ 倍ずつ減少していく。

ここで、今回の実験で用いたベータ分布は表 4.2 より、$\alpha, \beta \ll 1$ であるため、

$$\frac{\beta+x-1}{\alpha+\beta+x} \simeq \begin{cases} 0 & x = 1 \\ 1 & x \geq 2 \end{cases} \tag{4.10}$$

より、

$$Q(1) \gg Q(2) \simeq Q(3) \simeq \cdots \tag{4.11}$$

となり、不具合が検出される確率は 1 回目から 2 回目にかけて大きく減少し、それ以降は減少する割合が小さくなる。

各条件について、$Q(x)$ の推移を図 4.4 に示す。縦軸を $Q(x)$、横軸を x とした。図 4.4 からも、ここまでの議論が正しいことが分かる。

以上のような戦略の場合、まだカバーされていないユニットが存在する場合はそれをカバーするテストケースの重みづけが大きくなり、Additional Strategy に近い戦略となる。一方、全てのユニットが 1 回以上カバーされた場合について考えると、$x \geq 2$ について $Q(x)$ はほぼ等しい値を取るため、Total Strategy に近い戦略となる。

4.3 優先順位付け戦略の評価

3.1 節で提案した戦略について、既存戦略との性能の比較評価を行った。

4.3.1 実験設定

- Total Strategy

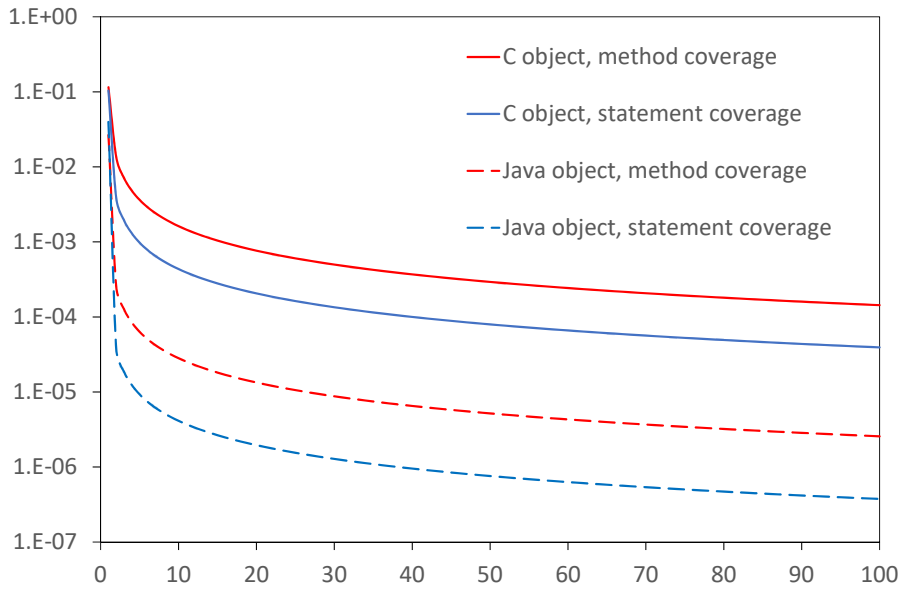

図 4.4. $Q(x)$ of each probability density function

- Additional Strategy
- Unified Strategy の $c = 0.05$ から 0.95 まで、0.05 刻みで 19 通り
- 提案手法による戦略

という複数の戦略について、データセットの各テストスイートに対して優先順位付けを行い、APFD を計算した。提案手法の確率密度関数には、実験 4.2 で得られたものを用いた。提案手法と既存手法のそれぞれについて APFD を計算し、提案手法と既存手法の APFD の差をそれぞれ T 検定で検証した。APFD を計算する際、テストケースを 20 通りにランダムに並び替え、全ての戦略について APFD が同じ値となったものを除いたものを評価に用いた。

4.3.2 実験結果

実験結果をそれぞれ図 4.5〜4.8 に示す。図 4.5 と図 4.6 は C 言語オブジェクト、図 4.7 と図 4.8 は Java オブジェクトに対して、method カバレッジと statement カバレッジを用いて APFD を計算した結果である。横軸は戦略の種類を表しており、Beta は提案手法、Tot は Total Strategy、Add は Addirional Strategy、Uxx は Unified Strategy の $c = \frac{xx}{100}$ の場合を表している。縦軸が APFD の値を表し、箱の上下はそれぞれ第 3 四分位点と第 1 四分位点、中央の線は中央値、上下のひげはそれぞれ 95 パーセンタイルと 5 パーセンタイル、丸印は平均値、上下の×印はそれぞれ最大値と最小値を意味する。

図 4.5 と図 4.6 を比較すると、C 言語オブジェクトでの実験では method カバレッジと比較して statement カバレッジを用いた方がより APFD の値が高くなっていることが分かる。一方、図 4.7 と図 4.8 を見ると Java オブジェクトではカバレッジの粒度に依らず APFD の値は Total Strategy を除いて 0.9 近辺に多く分布している。いずれの条件でも Unified Strategy は Total Strategy を上回り、Additional Strategy と最低でも競合するという先行研究 [9] の結論を支持する結果が得られた。

提案戦略の APFD は図 4.5 の結果では中央値、平均値共に Additional Strategy や c の大きい Unified Strategy に劣っていることが分かるが、他の結果では図 4.6〜 4.8 を見た限りでは少なくとも劣っているとはいえない。

提案戦略と既存戦略の APFD の差の分布はそれぞれ図 4.9〜4.12 のようになった。図 4.9 と図 4.10 は C 言語オブジェクト、図 4.11 と図 4.12 は Java オブジェクトに対しての結果である。縦軸はそれぞれのプログラムに対する提案戦略の APFD から各既存戦略の APFD を引いた差を意味する。

いずれの条件でも Total Strategy と Additional Strategy との差が Unified Strategy との差より広い分布をとっていることが分かる。また、図 4.9, 4.10 と図 4.11, 4.12 を比較すると、C 言語より Java オブジェクトの方が APFD の差が分散している。同様に図 4.9, 4.11 と図 4.10, 4.12 を比較すると、method カバレッジの方が statement カバレッジよりも APFD の差が分散している。

T 検定の結果を表 4.3 に示す。各オブジェクト、ユニットについて、既存手法のそれぞれとの APFD 値の差を母集合とし、提案手法の方が有意に大きいことを “+”、有意に小さいことを “-” で表す。有意差があると判断する基準は、p 値が 0.05 を下回ることとした。

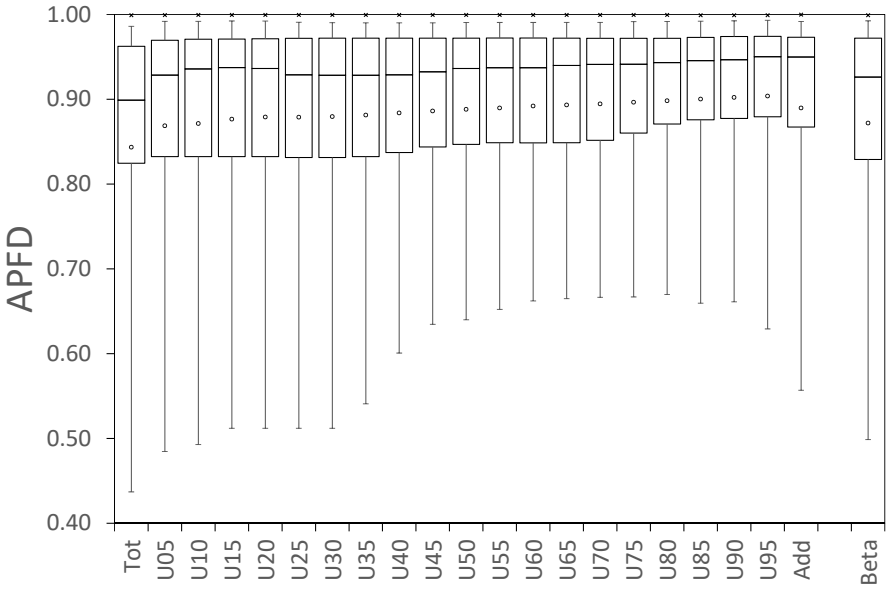

図 4.5. Results of C programs for test suites with dynamic method coverage

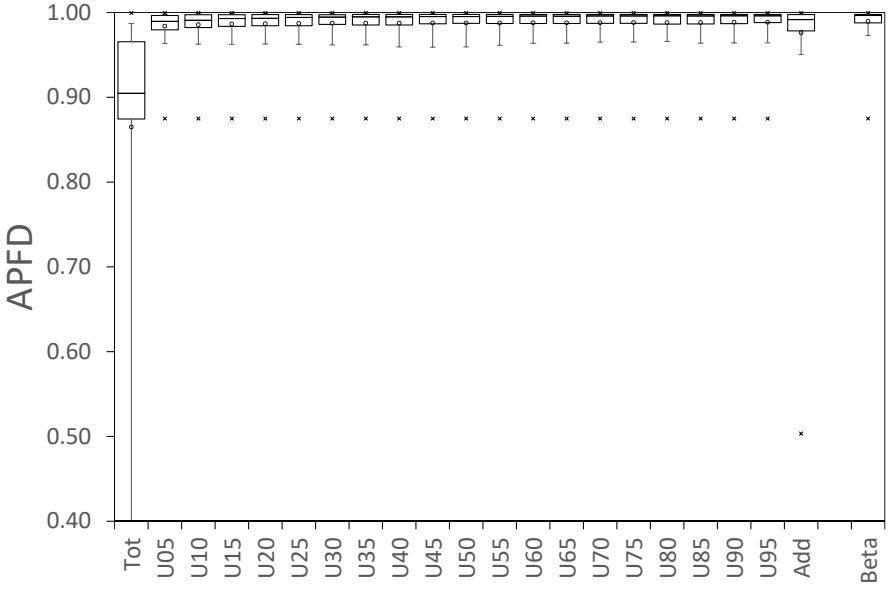

図 4.6. Results of C programs for test suites with dynamic statement coverage

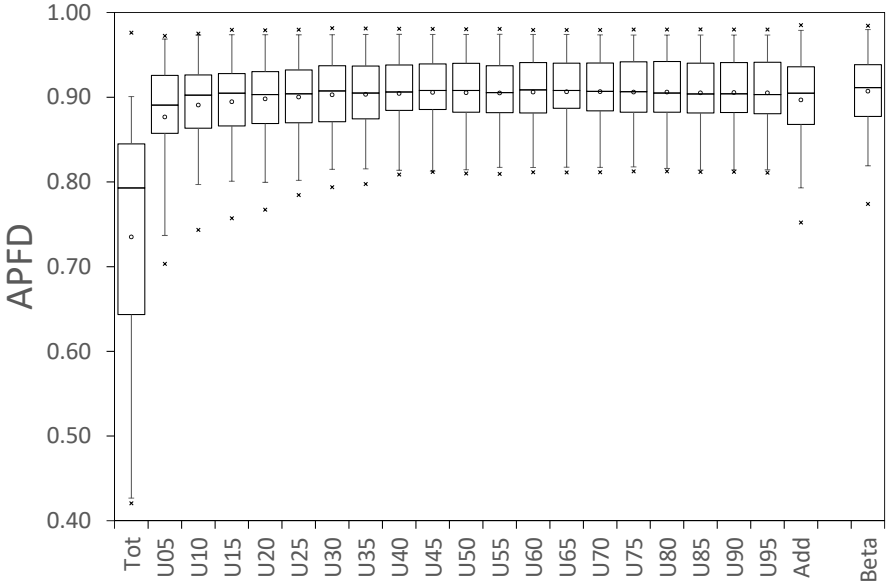

図 4.7. Results of Java programs for test suites with dynamic method coverage

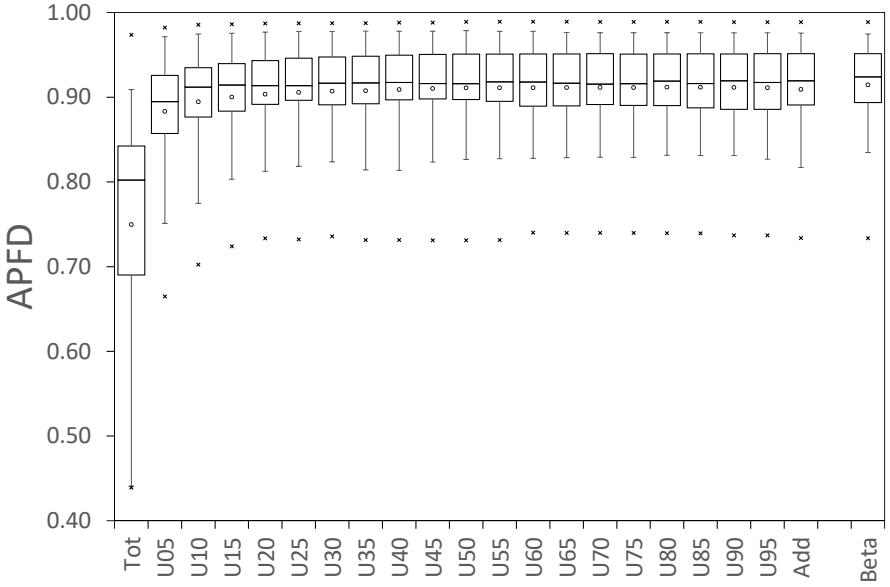

図 4.8. Results of Java programs for test suites with dynamic statement coverage

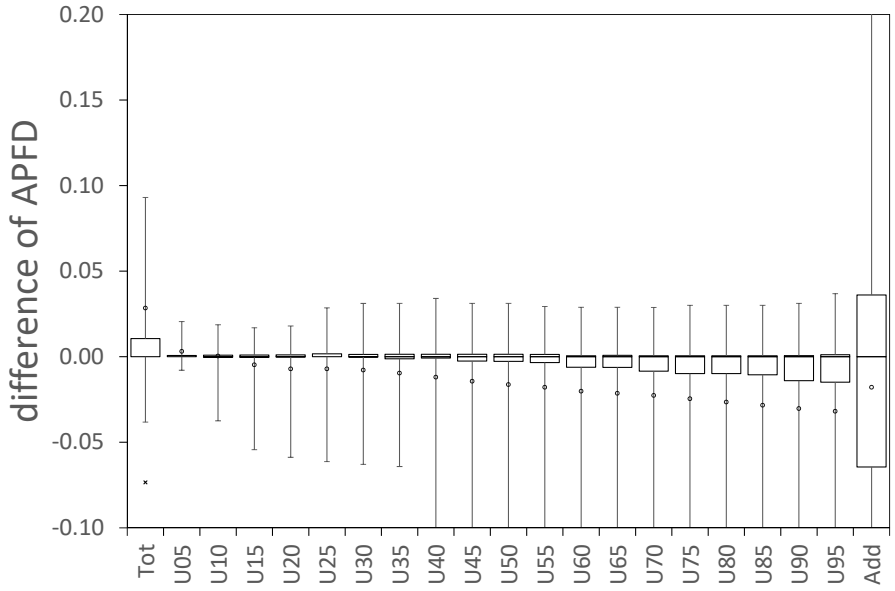

図 4.9. Differences of APFD of C programs for test suites with dynamic method coverage

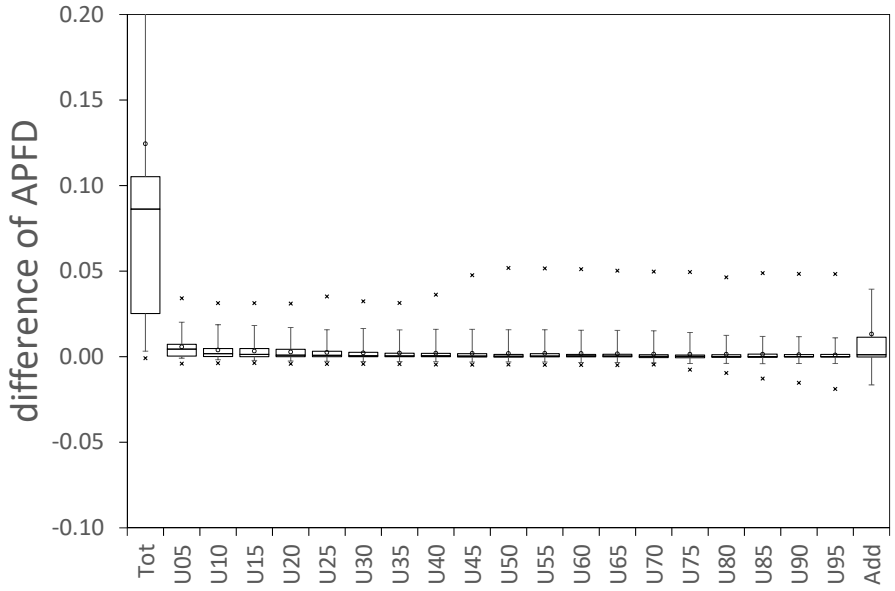

図 4.10. Differences of APFD of C programs for test suites with dynamic statement coverage

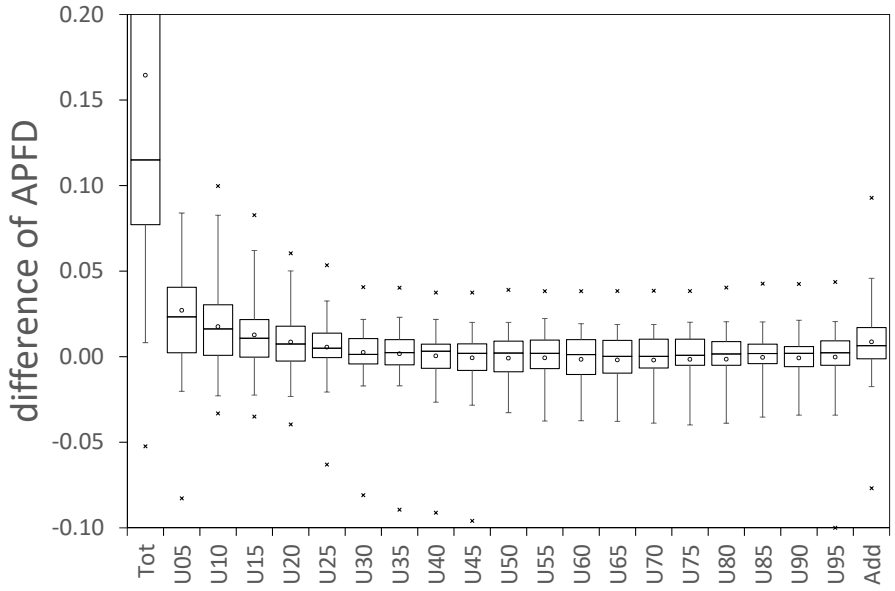

図 4.11. Differences of APFD of Java programs for test suites with dynamic method coverage

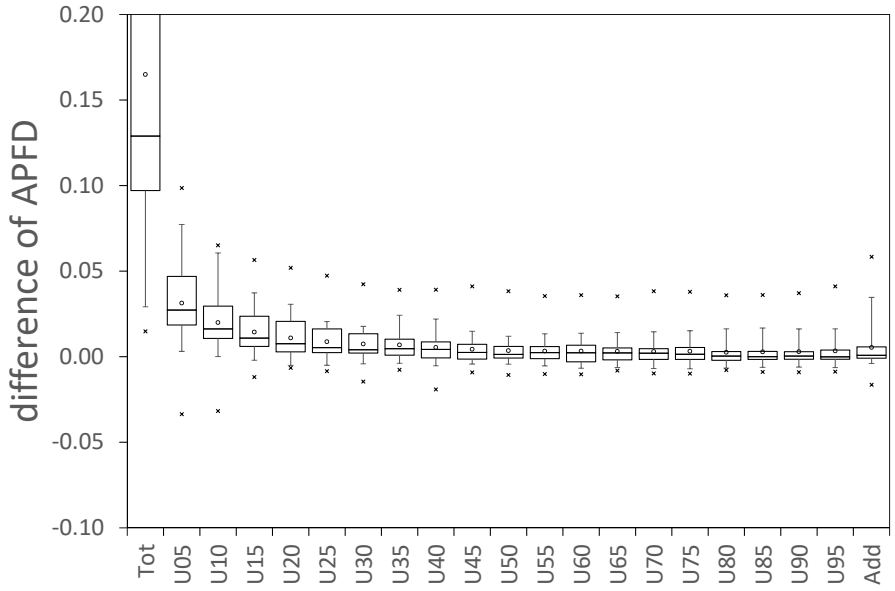

図 4.12. Differences of APFD of Java programs for test suites with dynamic statement coverage

表 4.3. Results of Student's t-test of proposed strategy and existing strategy

	Tot	U05	U10	U15	U20	U25	U30	U35	U40	U45	U50	U55	U60	U65	U70	U75	U80	U85	U90	U95	Add
C method	+	+		-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
C statement	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+
Java method	+	+	+	+	+	+	+	+													+
Java statement	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+

表 4.3 が示すように、statement カバレッジを用いた実験では、C 言語と Java の両方のオブジェクトについて、提案手法が全ての既存手法を有意に上回った。このことから、statement カバレッジを用いる場合、提案手法は既存手法と比較して有用であると言える。一方で、method カバレッジ、C 言語オブジェクトによる実験では $c \geq 0.10$ の場合について既存の戦略を有意に下回った。

4.3.3 考察

いずれの条件においても提案した戦略と Total Strategy や Additional Strategy との差は Unified Strategy と比較して分布が分散している。これは Total Strategy や Additional Strategy では優先順位付けに用いる和の計算式が簡単であるため多くのテストケースで同じ値を取り、その初期順序によって結果が大きく異なるためだと考えられる。

C 言語オブジェクト、method カバレッジの条件において提案した戦略が既存戦略と比較して良い結果が得られなかった原因について考える。まず、図 4.9 と図 4.10〜 4.12 を比較すると、提案した戦略と Additional Strategy の APFD の差が C 言語オブジェクト、method カバレッジにおいてのみ非常に大きい分散をとっていることが分かる。このことから、C 言語オブジェクト、method カバレッジの条件では全てのユニットをカバーして網羅率が 100%になった状態においても多くの未検出の不具合が残っていると推測できる。4.2.3 節で考察した通り、提案した戦略では全てのユニットをカバーするまでは Additional Strategy に、カバーした後は Total Strategy に近い戦略となる。したがって、網羅率が 100%になって以降、多くの不具合を Total Strategy に近い戦略で優先順位付けした結果、Unified Strategy の $c \simeq 0$ に近い戦略に近い実験結果が得られたと考えられる。実際に図 4.9 より、U05、U10 の結果の半数以上が提案した戦略の APFD とほぼ等しくなっていることが、この仮説を支持する。

また、既存戦略と比較して大きく APFD の値が劣る結果となったプログラムである readlink, groups, pathchk, touch のうち、pathchk を除く 3 つでは最尤推定の結果 $p_{m,l} = 1$ となる m, l は存在せず、pathchk においても $p_{m,l} \leq 0.5$ を満たすものが $p_{m,l} > 0$ となるものの 9 割を占めた。したがってこれらのプログラムで提案手法の APFD が低くなった理由は、網羅率が 100%になっても検出されていない不具合が多くあったことだと考えられる。

以上を踏まえると、データのサイズが小さい C 言語オブジェクト、粒度の粗い method カバレッジの実験では、ユニットをカバーした際に不具合が検出される確率密度関数の形状が全てのプログラムで一致すると仮定したために良い結果が得られなかったといえる。

4.4 Metric を用いた戦略の比較

3.2.3 節で説明したように、metric からベータ分布のパラメータを推定することで metric に応じて確率密度関数を変化させることが出来る。この metric を利用した戦略を、同じように metric を利用して Unified Strategy のパラメータを変化させる戦略と比較し、評価する。

4.4.1 実験設定

- metric に応じてパラメータを変化させる Unified Strategy
- パラメータを変化させない提案手法の戦略
- metric からパラメータを推定する提案手法の戦略

という複数の戦略について、データセットの各テストスイートに対して優先順位付けを行い、APFD を計算し、提案手法と既存手法の APFD の差をそれぞれ T 検定で検証した。

Unified Strategy は式 (2.14) (Linear) 、 (2.15) (Log) によってパラメータを調整する。metric として McCabe と MLoC のうち一方を用いるため、合計 4 通りの戦略がある。c の範囲を表す p_{low}、p_{high} はそれぞれ 0.5、1 とした。

提案手法では、2 つの metric から平均と分散を線形回帰することで metric からパラメータを推定する。ユニット u_m について、これを実行した際にそれぞれの不具合が検出される確率 $p_{m,l}$ の平均と分散を求め、$Metric[m]$ との組をサンプルとした。

metric から推定した平均と分散はそれぞれ

$$E[X] = -3.16 \times 10^{-4} \times Metric_{McCabe} + 2.76 \times 10^{-4} \times Metric_{MLoC} + 0.24926 \times 10^{-2}$$

$$var[X] = -2.89 \times 10^{-4} \times Metric_{McCabe} + 1.81 \times 10^{-4} \times Metric_{MLoC} + 0.24328 \times 10^{-2} \quad (4.12)$$

となった。

4.4.2 実験結果と考察

実験結果を図 4.13 に示す。横軸は戦略の種類を表しており、左側が既存の戦略、右側の Beta は実験 4.3 の提案戦略、Beta Metric が本実験の提案戦略を表している。図 4.13 から、metric を用いた提案戦略が最も APFD の分散が小さいことが分かる。また、T 検定の結果、metric を用いた提案戦略は全ての既存手法を有意に上回ったが、metric を用いない提案戦略と比較して有意差は見られなかった。提案戦略と他の戦略の APFD の差の分布はそれぞれ図 4.14 のようになった。metric を用いない提案手法との差は、既存戦略との差と比較して分散が小さいことが分かる。

metric を活用しても有意差が生じなかった原因として、metric の影響が小さかったことが考えられる。metric の種類や数を変更し、教師データ数を増やすことで性能が向上する可能性がある。

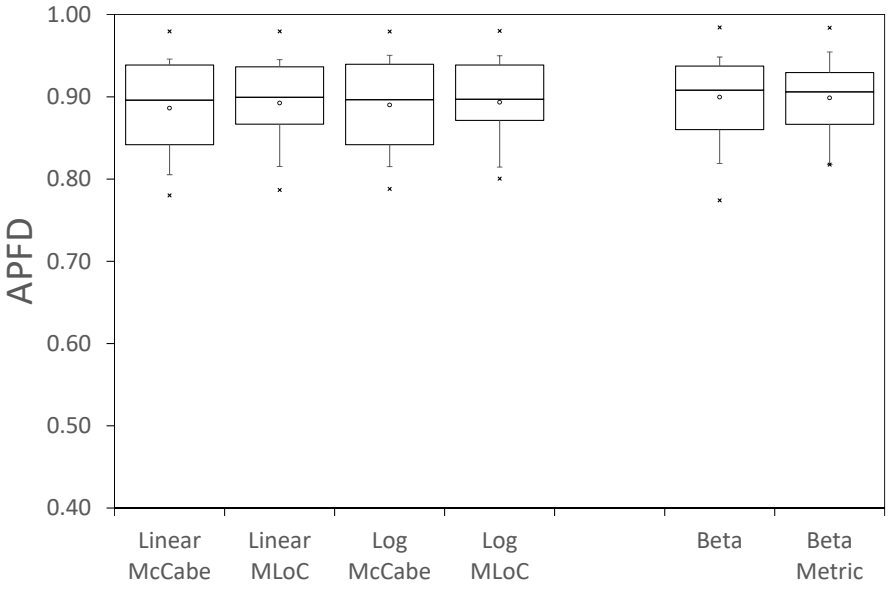

図 4.13. Results for strategy used metrics

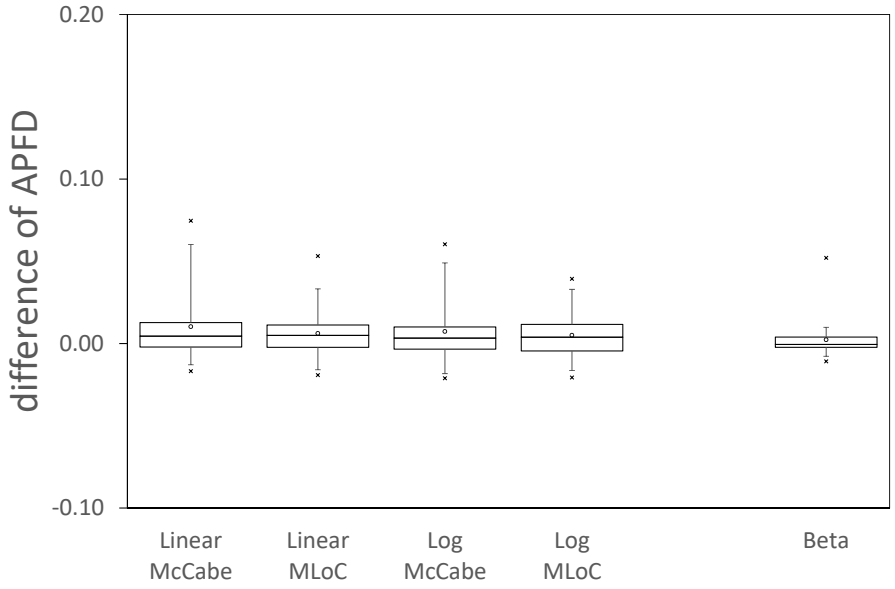

図 4.14. Differences of APFD for strategy used metrics

第5章　関連研究

カバレッジベースのテストケース優先順位付けに関しては様々な研究がなされており、それらをカテゴリー別にまとめる。

5.1　カバレッジベースの TCP

5.1.1　貪欲な戦略

Rothermell らはテストケース優先順位付けの戦略として Total Strategy と Additional Strategy を定義した [20]。Total Strategy は、カバーするユニットの多いテストケースから順に選択していく戦略であり、Additional Strategy はそれまでに選択したテストケースがカバーしていないユニットをより多くカバーするテストケースを順に選択していく戦略で、いずれも優先度の高いテストケースから貪欲に選択していく戦略である。

カバレッジベースの貪欲な戦略の殆どは

1. カバレッジやそれ以外の情報に基づいてそれぞれのテストケースの優先度を数値化し、それが高いものから順に選択していく戦略
2. カバレッジ以外の情報からユニット毎に重みづけを行い、Total Strategy もしくは Additional Strategy を適用する戦略

のいずれかに分類できる。前者の戦略はそれまでに選択したテストケースのカバレッジを考慮していないという点で Total Strategy の応用だと考えられるため、どちらに分類される戦略もこの 2 つの戦略をベースとしていると言える。

カバレッジやそれ以外の情報に基づいてそれぞれのテストケースの優先度を数値化し、それが高いものから順に選択していく戦略

Zhang らは、カバーする変更された部分の数、変更された部分を実行する回数の多い順に選択する優先順位付け戦略を提案した [31]。Yadav らは、カバレッジなどの metric を入力として、非線形の fuzzy 関数の出力に基づいて優先順位付けを行った [26]。Noor らは、類似性に基づく metric などを用い、ロジスティック回帰モデルを使って不具合が検出される確率を計算した [18]。カバレッジや以前のバージョンで不具合を検出した回数、テストのサイズの他に、以前のバージョンで不具合を出

したテストとの類似性の metric を用いている。類似性はメソッド呼び出しの合致数、編集距離、ハミング距離を用いている。metric からロジスティック回帰モデルによって計算される不具合の検出される確率が高い順に優先順位付けを行う。Alazzam らは、複数の粒度のカバレッジ情報の和によって優先順位付けを行った [2]。

カバレッジ以外の情報からユニット毎に重みづけを行い、Total Strategy もしくは Additional Strategy を適用する戦略

Aggrawal らは、コードの変更された部分、削除された部分の情報を利用してカバレッジが他のテストケースと完全に重複するテストケースを削除し、変更された部分のみを対象に Additional Strategy を行う戦略を提案した [1]。Agawa らは、Additional Strategy で同率の場合、過去のテスト結果から計算される故障重大度が高い方を選択する戦略を提案した [25]。Wang らは、静的コード解析によって不具合が検出される確率が高い部分を推定し、重みづけを行った [24]。パターンからコーディングエラーを検出する静的バグファインダーと機械学習による欠陥予測モデルによって静的コード解析を行い、不具合があると予測された部分の重みづけを変更して Total Strategy、Additional Strategy の要領で優先順位付けを行っている。Aziri らは、レコメンドシステムによって過去の不具合の情報から新たに不具合が検出される確率が高い部分を推定し、重みづけを行った [5]。

それ以外の戦略

Li らは k 個のテストケースの組を貪欲に選択していく戦略を紹介し、k=2 の場合について実験を行っている [13]。この戦略の $k = 1$ の場合が Total Strategy や Additional Strategy にあたるので、この戦略はこれらの発展形であるといえる。また、Jiang らは、Adaptive Random Testing (ART) をテストケース優先順位付けに応用した [10]。ランダムに抽出したテストケースの中から、それまでに選択したテストケースとカバレッジが最も離れているものを順に加えていくという戦略である。2 つのカバレッジの距離を表す指標として、2 つの集合の共通要素数の割合によって計算される Jaccard 距離を用いている。

5.1.2 貪欲でない戦略

貪欲でない手法では、多くの場合より早期にカバレッジの網羅率を高めるような戦略が研究されている。カバレッジの網羅率についての目的関数の一つに Average Percentage Statement Coverage (APSC) があり、

$$APSC = 1 - \frac{\sum_{m=1}^{M} S_m}{M \times N} + \frac{1}{2N} \tag{5.1}$$

と定義される [13]。ただし、N はテストケースの総数、M は statement の総数、S_m は statement s_m を始めてカバーするテストケースが何番目であるかである。APSC の最大化は NP 困難であり、

最適な優先順位付けは Additional Strategy では求めることが出来ないため、近似解を求める研究がなされている。Li らは山登り法 [13]、Konsaard は遺伝的アルゴリズム [12]、Lu らは ant colony system [14] によって優先順位付けを行う戦略をそれぞれ提案している。

また、各テストケースの実行時間が推定できるという前提で、時間あたりの不具合検出率を最大化する場合においても、貪欲でない戦略が用いられる。これは一定時間内の不具合検出率を最大化する優先順位付けがナップサック問題にあたり、NP 完全であるためである。Walcott らは遺伝的アルゴリズム [23]、Alspaugh らはナップサックソルバ [3]、Zhang らは動的計画法 [30] を用いた優先順位付け戦略をそれぞれ提案したが、Yoo らは Total Strategy や Additional Strategy との比較実験の結果、時間の要素は TCP において重要な要素ではないと結論付けている [28]。

5.2 カバレッジの選択

ソースコードのカバレッジはユニットの粒度 (method、statement など) やその取得方法 (動的、静的など) によって異なることから、テストケース優先順位付けにおいてどのカバレッジが有効であるかという研究も進められている。Elbaum らは複数の戦略、条件に付いて対照実験を行い、カバレッジの粒度を細かくすることで TCP の性能は僅かに向上するが、収集のコストが小さい粗い粒度のカバレッジにもメリットがあるとした [8]。また、Zhou らは実験の結果から、動的カバレッジは静的カバレッジと比較して良い性能が得られていると結論付けた [32]。

第6章　結論

6.1　本論文のまとめ

ソフトウェアに対してテストを実行し、その出力と意図した出力を照らし合わせることで不具合がないかを検証することをソフトウェアテスティングと呼ぶ。回帰テストはソフトウェアテスティングに分類され、ソフトウェアが変更される度にテストを実行し、加えた変更が既存の機能に影響を与えていないかを検証する手法である。頻繁に生じるソフトウェアの変更の度にテストを行うため、回帰テストのコストは膨大であり、その削減が課題である。TCP はテストケースの実行順序を並び替えることでより少ないテストで不具合を検出できる確率を高める手法である。少ないテストで不具合を検出することで実行するテストの回数を減らすことができるため、回帰テストのコスト削減に有効である。

TCP の中ではカバレッジを用いたものが主に用いられており、基本的な優先順位付け戦略として Total Strategy と Additional Strategy がある。これらの発展として様々な戦略が提案されているが、2 つの戦略にはそれぞれ長所と短所があり、どちらの戦略が有効であるかは場合によって異なるという問題がある。2 つの戦略を統合する戦略として Unified Strategy が提案されたが、パラメータによって 2 つの戦略のどちらに近いかを調整するという性質上、この問題を根本的に解決したとは言えない。

以上を踏まえ、本論文ではカバレッジ情報のみを利用した TCP の戦略として、Unified Strategy に代わる新しい戦略を提案した。これはユニットを実行した際に不具合が検出される確率を確率密度関数で表すことで、既存の戦略を拡張したものになっている。提案した戦略では、確率密度関数を用いて計算される、新たに検出される不具合の期待値が最大となるテストケースを順に選択していく。Unified Strategy ではパラメータを調整する手法について言及されていなかったが、本研究では確率密度関数をデータセットから推定する手法を提案した。これはカバレッジ情報と不具合の情報を用いて最尤推定した、各ユニットを実行した際に不具合が検出される確率の分布を確率密度関数とするというものである。推定した離散的な確率密度関数は、ベータ分布を用いて連続的な関数として近似できることを示した。C 言語と Java のプログラムに対して提案手法と既存手法の戦略を適用し実験を行った結果、statement カバレッジを用いた場合について、提案手法が既存手法を有意に上回るという結論が得られた。

6.2 今後の課題

今後の課題として、より多くのデータセットから確率密度関数の推定を行い、対象の言語やカバレッジの粒度、metric によってその分布がどのように変化するかを調べることが挙げられる。様々な条件と確率密度関数の関連性を調べることは特定の条件においてどの戦略、パラメータを用いるべきかを決める際に大きな貢献をすると考えられる。また、metric と確率密度関数の関連性を調べることでその metric の TCP に対する有用性をはかることが可能である。

更に、本研究ではデータセットの量を考慮して metric によるパラメータ推定は線形回帰にとどめたが、膨大なデータセットを用いることでより複雑なモデルに対する学習も可能になる。

参考文献

[1] K. K. Aggrawal, Yogesh Singh, and A. Kaur. Code coverage based technique for prioritizing test cases for regression testing. *SIGSOFT Softw. Eng. Notes*, Vol. 29, No. 5, pp. 1–4, September 2004.

[2] Iyad Alazzam and Khalid M. O Nahar. Combined source code approach for test case prioritization. In *Proceedings of the 2018 International Conference on Information Science and System*, ICISS '18, pp. 12–15, New York, NY, USA, 2018. ACM.

[3] Sara Alspaugh, Kristen R. Walcott, Michael Belanich, Gregory M. Kapfhammer, and Mary Lou Soffa. Efficient time-aware prioritization with knapsack solvers. In *Proceedings of the 1st ACM International Workshop on Empirical Assessment of Software Engineering Languages and Technologies: Held in Conjunction with the 22Nd IEEE/ACM International Conference on Automated Software Engineering (ASE) 2007*, WEASELTech '07, pp. 13–18, New York, NY, USA, 2007. ACM.

[4] J. H. Andrews, L. C. Briand, and Y. Labiche. Is mutation an appropriate tool for testing experiments? In *Proceedings of the 27th International Conference on Software Engineering*, ICSE '05, pp. 402–411, New York, NY, USA, 2005. ACM.

[5] Maral Azizi and Hyunsook Do. A collaborative filtering recommender system for test case prioritization in web applications. In *Proceedings of the 33rd Annual ACM Symposium on Applied Computing*, SAC '18, pp. 1560–1567, New York, NY, USA, 2018. ACM.

[6] Antonia Bertolino. Software testing research: Achievements, challenges, dreams. In *2007 Future of Software Engineering*, pp. 85–103. IEEE Computer Society, 2007.

[7] Cristian Cadar, Daniel Dunbar, and Dawson Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pp. 209–224, Berkeley, CA, USA, 2008. USENIX Association.

[8] Sebastian Elbaum, Alexey G Malishevsky, and Gregg Rothermel. Test case prioritization: A family of empirical studies. *IEEE transactions on software engineering*, Vol. 28, No. 2, pp. 159–182, 2002.

[9] Dan Hao, Lingming Zhang, Lu Zhang, Gregg Rothermel, and Hong Mei. A unified test case prioritization approach. *ACM Trans. Softw. Eng. Methodol.*, Vol. 24, No. 2, pp. 10:1–10:31, December 2014.

[10] Bo Jiang, Zhenyu Zhang, Wing Kwong Chan, and TH Tse. Adaptive random test case prioritization. In *2009 IEEE/ACM International Conference on Automated Software Engineering*, pp. 233–244, Nov 2009.

[11] Diederik Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[12] Patipat Konsaard and Lachana Ramingwong. Total coverage based regression test case prioritization using genetic algorithm. In *2015 12th International Conference on Electrical Engineering/Electronics, Computer, Telecommunications and Information Technology (ECTI-CON)*, pp. 1–6, June 2015.

[13] Zheng Li, Mark Harman, and Robert M Hierons. Search algorithms for regression test case prioritization. *IEEE Transactions on Software Engineering*, Vol. 33, No. 4, pp. 225–237, April 2007.

[14] Chengyu Lu and Jinghui Zhong. An efficient ant colony system for coverage based test case prioritization. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, GECCO '18, pp. 91–92, New York, NY, USA, 2018. ACM.

[15] Qi Luo, Kevin Moran, and Denys Poshyvanyk. A large-scale empirical comparison of static and dynamic test case prioritization techniques. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, pp. 559–570, New York, NY, USA, 2016. ACM.

[16] Yu-Seung Ma, Jeff Offutt, and Yong Rae Kwon. Mujava: an automated class mutation system. *Software Testing, Verification and Reliability*, Vol. 15, No. 2, pp. 97–133, 2005.

[17] T. J. McCabe. A complexity measure. *IEEE Trans. Softw. Eng.*, Vol. 2, No. 4, pp. 308–320, July 1976.

[18] Tanzeem Bin Noor and Hadi Hemmati. Studying test case failure prediction for test case prioritization. In *Proceedings of the 13th International Conference on Predictive Models and Data Analytics in Software Engineering*, PROMISE, pp. 2–11, New York, NY, USA, 2017. ACM.

[19] Pandimurugan, M. parvathi, and A. jenila. A survey of software testing in refactoring based software models. In *International Conference on Nanoscience, Engineering and Technology (ICONSET 2011)*, pp. 571–573, Nov 2011.

[20] Gregg Rothermel, Roland H Untch, Chengyun Chu, and Mary Jean Harrold. Test case prioritization: An empirical study. In *Software Maintenance, 1999.(ICSM'99) Proceedings. IEEE International Conference on*, pp. 179–188. IEEE, 1999.

[21] Gregg Rothermel, Roland H. Untch, Chengyun Chu, and Mary Jean Harrold. Prioritizing test cases for regression testing. *IEEE Transactions on Software Engineering*, Vol. 27, No. 10, pp. 929–948, Oct 2001.

[22] David Schuler and Andreas Zeller. Javalanche: efficient mutation testing for java. In *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pp. 297–298. ACM, 2009.

[23] Kristen R. Walcott, Mary Lou Soffa, Gregory M. Kapfhammer, and Robert S. Roos. Timeaware test suite prioritization. In *Proceedings of the 2006 International Symposium on Software Testing and Analysis*, ISSTA '06, pp. 1–12, New York, NY, USA, 2006. ACM.

[24] Song Wang, Jaechang Nam, and Lin Tan. Qtep: Quality-aware test case prioritization. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, pp. 523–534, New York, NY, USA, 2017. ACM.

[25] Yiting Wang, Xiaomin Zhao, and Xiaoming Ding. An effective test case prioritization method based on fault severity. In *2015 6th IEEE International Conference on Software Engineering and Service Science (ICSESS)*, pp. 737–741, Sep. 2015.

[26] Dharmveer Kumar Yadav and Sandip S. Dutta. Test case prioritization technique based on early fault detection using fuzzy logic. In *2016 3rd International Conference on Computing for Sustainable Global Development (INDIACom)*, pp. 1033–1036, March 2016.

[27] S. Yoo and M. Harman. Regression testing minimization, selection and prioritization: a survey. *Software Testing, Verification and Reliability*, Vol. 22, No. 2, pp. 67–120, 2012.

[28] Dongjiang You, Zhenyu Chen, Baowen Xu, Bin Luo, and Chen Zhang. An empirical study on the effectiveness of time-aware test case prioritization techniques. In *Proceedings of the 2011 ACM Symposium on Applied Computing*, SAC '11, pp. 1451–1456, New York, NY, USA, 2011. ACM.

[29] Hongyu Zhang. On the distribution of software faults. *IEEE Transactions on Software Engineering*, Vol. 34, No. 2, pp. 301–302, March 2008.

[30] Lu Zhang, Shan-Shan Hou, Chao Guo, Tao Xie, and Hong Mei. Time-aware test-case prioritization using integer linear programming. In *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis*, ISSTA '09, pp. 213–224, New York, NY, USA, 2009. ACM.

[31] Z. Zhang, Y. Mu, and Y. Tian. Test case prioritization for regression testing based on function call path. In *2012 Fourth International Conference on Computational and Information Sciences*, pp. 1372–1375, Aug 2012.

[32] Jianyi Zhou and Dan Hao. Impact of static and dynamic coverage on test-case prioritization: An empirical study. In *2017 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pp. 392–394, March 2017.

対外発表

齋藤雄太, 佐藤周行, “確率密度関数を用いたソフトウェアテスティングのためのテストケース優先順位付け戦略”, 情報処理学会・第 122 回プログラミング研究会, 2019

謝辞

本研究を進めるにあたり、多くの方々にお世話になりました。

指導教員である佐藤周行准教授には修士課程の 2 年間、研究や発表、論文執筆等の際に数々のご指導をいただきました。研究のテーマ決めから論文の書き方に至るまで、研究を進める上で必要となる助言を何度もいただきました。ここに深謝の意を表します。研究室の先輩である清野喜裕さんは私の進捗を大変気にかけてくださり、研究に関する助言をいただくなど、本当にお世話になりました。同輩である Cosmas Krisna Adiputra 君とはお互いの研究進捗の確認や雑談を行うなど、充実した研究生活を共に送ることができました。異なる研究分野ではありましたが、すぐ隣で努力し続ける同期の存在は研究を進める上で大きな励みとなりました。名前を挙げた方々にとどまらず、コンピュータネットワーク研究室の皆様には、ミーティングや日々の研究生活においてお世話になりました。

最後に、現在に至るまで学生である私を支えてくれた家族を含め、お世話になりました全ての方々にこの場を借りて感謝の意を表します。