修　士　論　文

題　　目　　ベイズ推定による
優先度付き組み合わせテストの改良と
不具合発見傾向の評価

主任指導教員　　水野　修　教授

京都工芸繊維大学大学院 工芸科学研究科

情報工学専攻

学生番号　　15622042

氏　　名　　藤原　剛史

平成29年2月10日提出

様式 2 号

学位論文の要旨（和文）

平成 29 年 2 月 10 日

京都工芸繊維大学大学院
工芸科学研究科長　殿

工芸科学研究科　　情報工学専攻
平成 27 年入学
学生番号　　15622042
氏　　名　　藤原　剛史　㊞

（主任指導教員　　水野　修　㊞　）

本学学位規則第 4 条に基づき、下記のとおり学位論文内容の要旨を提出いたします。

1.　論文題目

ベイズ推定による優先度付き組み合わせテストの改良と不具合発見傾向の評価

2.　論文内容の要旨（400 字程度）

ソフトウェアテストを効率良く実施する技術は，ソフトウェアの品質向上や開発コストの削減につながるため，今後ますます重要になると考えられている．ソフトウェアのテストを行うにあたって，不具合を発見する可能性の高いテストケースをテストスイートの初期に位置するよう並べ替えると，より少ないテストケース数で多くの不具合を発見することが可能になると考えられる．実際のテスト環境では，時間的な制約や人的な制約によってテストスイート内のすべてのテストケースを実行できないということは多々あるため，こうした並べ替えによるテストケースの優先度付けが有望である．

本研究では，ベイズ推定を用いた組み合わせテスト最適化手法を不具合発見情報を利用して改良し，オープンソースソフトウェアを用いてその評価を行う．具体的には，以前の手法に対して，不具合を発見したテストケースに高い優先度を与えるための方法を提案し，実装した．また，ミューテーション解析を行うことでより実践に即した評価を行う．元の最適化手法と改良した最適化手法により並べ替えたテストを比較した結果，最初期における不具合の発見傾向が向上することがわかった．

Improvement of the Bayesian Inference Based Prioritized Combinatorial Testing and Assessment of the Tendency to Detect Faults

2017 15622042 *FUJIWARA Tsuyoshi*

Abstract

In software testing, it is important to assure the quality of software with limited time resource. Combinatorial testing(CT) is a widely-used technique to detect system interaction faults. To improve the test effectiveness of the CT, the prioritized CT has been proposed. The prioritized CT generates combinatorial test suites based on the priority weights of parameter-values.

We have been proposed a prioritized CT based on Bayesian inference. In this study, we improve the prioritized CT based on Bayesian inference by taking into account the information of faults in previous revisions and mutations in source code. For experiments, we applied the mutation analysis to the target open source software, and obtained more practical results than that in previous studies. We confirmed that the faults can be found at an early stage of the testing in the improved prioritized CT technique in some projects.

目　次

1. 緒言

近年，社会におけるソフトウェアの重要度が高まる一方，ソフトウェアの不具合が与える影響も大きくなっている．ソフトウェアの不具合は後から修正することもできるが，一度世に出したソフトウェアに大きな不具合が出ることは，顧客の機会損失や信頼の低下を招き，労力や資源以上の損害を起こすことも少なくない．ソフトウェアのリリース後に不具合を出さないようにするためにも，万全なテストを実施することが重要である．しかし，様々な制約により，十分なテストを実施できないという場合も起こり得る．そのような場合であっても，テストスイートを最適化して，より多くの不具合を発見できるテストケースをテストの初期段階で行うことができれば，より効率の良いテストが可能となり，その被害を抑えることができる．組み合わせテストの最適化技術の向上は，少ない時間でのテスト品質をより向上させると考えられる．

組み合わせテストの最適化手法はいくつか存在する．Qu ら [8] はコード網羅率を用いた手法を提案し，実験対象の 1 つ前のバージョンの重みを用いた回帰テストで評価している．Choi ら [2] は pricot という組み合わせテスト生成ツールに，順序重視，頻度重視，テストケース数重視の 3 つの戦略を与え，それらを組み合わせることを可能とした最適化手法を提案している．また，河端ら [6] は仕様書とバグレポートからベイズ推定を用いて求めた値により，組み合わせテストを並べ替える最適化手法を提案した．

本研究では，河端らが提唱したベイズ推定を用いた組み合わせテスト最適化手法に対して改良案を提案し，それを評価する．まず，この手法では不具合検出情報を扱うときに，テストケースがいずれかの不具合を検出するかしないかの情報しか扱わない．この点に対して改良案では，それぞれの不具合についての情報を保持するよう変更する．すなわち，テストスイート最適化のための重みの計算に利用する不具合情報を増やすということである．このようにすることで，重み付けをより強くすることができるため，パラメータが取る値と不具合発見の関係がより顕著に表れ，最適化が改善されることを期待する．さらに，重みの計算にどのバージョンの不具合検出情報を用いるかについて，3 つの決定方法を提案し，不具合の発見傾向を多面的に調査する．

また，従来手法では，実験・評価を行うためにSoftware artifact Infrastructure Repository(SIR)[注 1)]を用いているが，ここで用いられている不具合は，研究者によって人為的に埋め込まれたものである．これら人為的に埋め込まれた不具合は，検出されることを想定されているため，比較的検出しやすくあまり実践的ではない．そこで，本実験では，各パラメータの情報等に関してはSIRのデータを利用しつつ，機械的に変異させられたプログラムを検出するミューテーション解析を行う．

結果として，従来手法と本研究で提案した最適化手法により並べ替えたテストを比較したところ，最初期における不具合の発見傾向が向上することが判明した．

本報告書の構成を以下に示す．2章では本実験への準備として，前提知識として必要な事項を説明する．3章では，研究設問，本実験の対象となるプログラムおよび本実験の準備，提案手法について説明し，4章ではその結果を示す．5章では，本実験の結果に対して考察を行う．そして6章では，本研究のまとめと今後の課題について述べる．

(注 1)：http://sir.unl.edu/portal/index.php

2. 準備

本章では実験を行う前に，実験の前提となる事項を説明する．

2.1 組み合わせテスト

組み合わせテストは，パラメータが取る値の組み合わせに注目し，ある組み合わせ数のパラメータ間で取る値のパターンすべてを網羅するテストのことであり，組み合わせ数を t としたとき t-way テストや t-wise テストと呼ぶ．以後，パラメータが取る値のことをパラメータ値と表記する．例として表 2.1 のパラメータ仕様が与えられたときの 2-wise テストを生成する．まず総テストケースなら，3 パターンのパラメータ値を取るパラメータが 1 つ，2 パターンのパラメータ値を取るパラメータが 2 つなので，テストケース数は $3 \times 2 \times 2 = 12$ となる．対して 2-wise の組み合わせテストなら，表 2.2 のテストスイートになり，テストケース数は 6 となる．考えられるすべての 2 つの組み合わせ (X, Y)，(Y, Z)，(Z, X) のパラメータ値を確認すると，確かにすべてのパターンを網羅している．組み合わせテスト生成ツールには，Microsoft 社の Czerwonka ら [3] が開発した PICT (注 2)，産業技術総合研究所の Choi ら [2] が開発した pricot，Bryce ら [1] が開発した DDA などがある．

2.2 組み合わせテストの最適化

組み合わせテスト生成ツールにより生成されたテストスイートは，テストケース数が少なくなるため，テスト実行に必要な資源を減らすことができる．しかし，このテストケースの並びを，不具合を発見する可能性がより高いテストケースをテストスイートの初期に位置するような並びにすれば，実行するテストケース数が少ない場合でも，テストスイート初期の段階で多くのフォールト検出が可能になる．これを組み合わせテストの最適化と呼ぶ．組み合わせテスト生成ツールには，pricot のように生成だけでなく最適化を行うものもある．

(注 2)：https://github.com/Microsoft/pict

表 2.1　パラメータの仕様

parameter	parameter-value
$X =$	$1, 2, 3$
$Y =$	$1, 2$
$Z =$	$1, 2$

表 2.2　表 2.1 の 2-wise 組み合わせテスト

test	X	Y	Z
T_1	1	1	1
T_2	1	2	2
T_3	2	1	2
T_4	2	2	1
T_5	3	1	1
T_6	3	2	2

2.3 評価手法

2.3.1 Average Percentage of Faults Detected (APFD)

良いテストスイートかを示す指標として，Elbaum ら [4] は Average Percentage of Faults Detected (以後 APFD と呼ぶ) を提案した．APFD の計算方法を以下に示す．

$$APFD = 1 - \frac{TF_1 + TF_2 + ... + TF_m}{m \times n} + \frac{1}{2n} \tag{2.1}$$

ここで，m は総テストケースにて検出できるフォールト数，n はテストケース数，TF_i はフォールト i を初めて検出したテストケース番号である．この式は x 軸をテスト ケース数，y 軸を検出したフォールトの累積数とした線グラフにて，縦が y = 0 から グラフの y 値まで，横が x = 0 からグラフの x 値までの面積を求めることを意味する．

例として，表 2.3 のテストスイートにおける APFD を計算すると式 (2.2) となり，値は 0.6 となる．

$$APFD = 1 - \frac{1+0+4}{2 \times 5} + \frac{1}{2 \times 5} = 0.6 \tag{2.2}$$

2.3.2 Normalized Average Percentage of Faults Detected (NAPFD)

APFD の問題として，値を比較したい場合，異なるテストケース数のテストスイート同士は比較できない点がある．例として，総テストケースである表 2.3 とその組み合わせテストである表 2.4 を比較したい場合，前者がテストケース数が 5 つであるのに対し，後者は 3 つと異なるため，単純に APFD を計算し比較することができない．この点に対し，Qu ら [8] は新メソッドである Normalized Average Percentage of Faults Detected(以後 NAPFD と呼ぶ) を提案した．NAPFD の計算方法を式 (2.3) に示す．

$$NAPFD = p - \frac{TF_1 + TF_2 + ... + TF_m}{m \times n} + \frac{p}{2n} \tag{2.3}$$

ここで，p は与えられた組み合わせテストが検出できるフォールト数を総テストケースが検出できるフォールト数で除算したものであり，その他の変数は APFD と同じである．変数 p の導入により APFD を正規化している．

上記の例の NAPFD を求める．表 2.3 は総テストケースであり $p = 1$ となるため APFD と変わらず値は 0.6，表 2.3 の組み合わせテストである表 2.4 の NAPFD は式

表 2.3　テストスイート例 1

test	F_1	F_2	F_3
T_1	✓		
T_2			
T_3			
T_4	✓		✓
T_5	✓		

表 2.4　テストスイート例 2

test	F_1	F_2	F_3
T_1	✓		
T_2			
T_3	✓		

(2.4) にて計算でき，値は 0.25 となる．

$$NAPFD = p - \frac{1+0+0}{2\times 3} + \frac{1/2}{2\times 3} = 0.25 \tag{2.4}$$

実験結果の評価は APFD ではなく NAPFD で評価していく．

2.4 ミューテーション解析

ミューテーション解析はソフトウェアテストにおける，テストスイートの十分さを測定するための手法である．

この手法では、テスト対象のプログラムの一部を機械的に書き換えることで，ミュータントと呼ばれる「人工的な誤りを含むプログラム」を生成する．テストスイートをミュータントに対して実行した結果と，元のプログラムに対して実行した結果が異なれば，テストスイートにはその誤りを発見するだけの鋭敏さが備わっていると考えられる (テストスイートはミュータントを kill すると表現する)．機械的に大量にミュータントを生成したとき，そのうちテストスイートが kill 出来るミュータントの割合を測定することで，テストスイートの「欠陥発見能力」の十分さを測定することが出来る．また，kill 出来なかったミュータントを kill するように追加のテストケースを作成することで，開発者はテストスイートの欠陥発見能力を高めることが出来ると期待される．

ミュータントを作成するためのプログラム書き換え方法をミューテーション操作と呼び，典型的なプログラムの誤り (演算子+(プラス) と-(マイナス) を逆にしてしまうなど）を模倣するものや、有意なテストケース生成を促すものなどが研究されている．ミューテーション解析によって既存のテストケース・テスト用入力データの十分さ・不十分さが測定でき，さらなるテストケースの作成・自動生成などに利用することが出来る．

2.5 ベイズ推定

ベイズ推定とは，観測事象から推定したい事柄を確率的な意味で推論することである．

事象 A が起こった条件下で事象 B が起こるという条件付き確率 $P(B|A)$ はベイズの定理によって以下のように表される．

$$P(B|A) = \frac{P(A|B)P(B)}{P(A)} \tag{2.5}$$

このとき，事象 B のベイズ確率について，$P(B)$ を事前確率，$P(B|A)$ を事後確率，$P(A|B)$ を尤度という．

2.6 ベイズ推定を用いた最適化

ベイズ推定を用いてテストスイートの最適化を行う際，以下のような流れで行われる．

1. 仕様書とバグレポートからベイズ推定によりパラメータ値の重み w_v を定義．
2. 重み w_v を用いて，１つのテストケースの重み w_t を定義．
3. 重み w_t により，テストケースを降順に並べ替える．

以降で，各ステップの詳細を説明する．

2.6.1 パラメータの重み定義

テストケースの重み定義に必要となるパラメータの値の重みを定義していく．実験対象には次のデータが含まれているとする．

- **各テストケースに関するバグレポート**

  テストケースを実行した結果，各フォールトを検出した，または検出しなかったかを記したデータ．
- **各テストケースの入力引数データ**

  テストにて使用する，パラメータ値の集合からなるデータ．
- **Test Specification Language (TSL)[7] で記述された実験対象に関する仕様書**

  各パラメータが取ることのできるパラメータ値のリストや，取る値の組み合わせにより生じる制約条件を記したデータ．

これら3つのデータから条件付き確率を求め，その値を重みとする．用いる確率は次の2通りとする．

$$P(B|p_i = x) \tag{2.6}$$

$$P(B|(p_i = x) \wedge (p_j = y)) \tag{2.7}$$

ここで，P は確率，B はフォールトを検出した事象，p_i，p_j はパラメータ，x，y はパラメータ値とする．式 (2.6) は1つのパラメータがあるパラメータ値を取る事象にてフォールト検出した確率を，式 (2.7) は2つのパラメータがそれぞれ特定のパラメータ値を取る共通事象にてフォールトを検出した確率を表す．これらの確率をパラメー タ値の重みと定義する．

2.6.2 テストケースの重み定義

定義したパラメータ値の重みにより，1つのテストケースの重みを定義していく．定義方法に関して以下の4つのメソッドが提案されている．[6]

- **1パラメータ-未定義値なし**

  式 (2.6) の重みを各パラメータ値へマッピング，その平均をテストケースの重みとする．このとき，指定のないパラメータ値へはマッピングしない．
- **1パラメータ-未定義値あり**

  メソッド1aを変更し，指定のないパラメータ値も1つの値として重みを計算しマッピングする．
- **2パラメータ-未定義値なし**

  式 (2.7) の重みを各パラメータ値の組み合わせへマッピングし，その平均をテストケースの重みとする．このとき，どちらかのパラメータ値が未指定ならば重みをマッピングしない．
- **2パラメータ-未定義値あり**

  メソッド2aを変更し，指定のないパラメータ値を含む組み合わせでも1つの値として重みを計算しマッピングする．

以上のメソッドで定義されたテストケースの重みにより，テストケースを降順に並べ替えることで，組み合わせテストの最適化を行う．

本研究では，先行研究にて最も性能が良かった”**1 パラメータ-未定義値なし**” のメソッドのみを用いて実験を行うこととする．

3. 不具合発見情報を用いた最適化手法

本章では，研究設問，実験対象，実験準備，提案手法，実験方法について説明する．

3.1 研究設問

実験を行うにあたって，以下のように研究設問を設定する．

- **RQ1**

  改良した手法によるテストは従来法に対して良いテストとなるか．

- **RQ2**

  どの時点の不具合情報を利用したテストが良いテストを生成するか．

3.2 実験対象

本実験では，C 言語で記述されたオープンソースプログラムである **flex**，**grep** の 2 つを実験対象とする．flex は字句解析処理系の生成ツールであり，grep は正規表現にマッチングするテキストを検索するプログラムである．これらのプログラムを機械的に書き換えたミュータントのデータセットが Henard ら [5] によって提供されており，これを使用している．

対象プログラムの詳細を表 3.1 に示す．LoC(Lines of Code) はプログラムのコード行数，#mutants は生成されたミュータントの個数，#tests はテストケースの個数を表している．

3.3 実験準備

3.3.1 基本事項

本研究では SIR リポジトリを用いて実験を行った．SIR は研究者によってオープンソースソフトウェア (OSS) から収集され，フォールトを挿入されたプロジェクトデータである．SIR の中には，次のテストデータが含まれている．

- **Universe**

  各テストケースを実行したとき，埋め込まれた各フォールトを検出した，ま

表 3.1　対象プログラムの詳細

program	ver.	LoC	#mutants	#tests
flex	v1(2.4.7)	9,470	32	525
	v2(2.5.1)	12,231	32	525
	v3(2.5.2)	12,249	20	525
	v4(2.5.3)	12,379	33	525
	v5(2.5.4)	12,366	32	525
grep	v1(2.2)	11,988	56	470
	v2(2.3)	12,724	58	470
	v3(2.4)	12,826	54	470
	v4(2.5)	20,838	59	470
	v5(2.7)	58,344	59	470

たは検出しなかったかを格納したファイル.

- **Frame**

  各テストケースにおいて，各パラメータが取る値 (文字列) を格納したファイル.

- **TSL**

  各パラメータの使用を TSL 形式にて格納したファイル.

しかし，SIR で挿入されている不具合は人為的なものであり，検出させることを想定して埋め込まれたものである．そのため，実際の現場での適用環境と大きく異なってしまうことが懸念される．よって，本研究では，より実践的な評価を行うために，ミューテーション処理が行われた Henard のデータセットを用いて実験を行った．ミューテーション解析を用いて実験を行うため，SIR の Universe ファイルは用いずにミュータントが kill されたかどうかを調べた結果を用いる．以降，便宜上，このミュータントが kill されたかどうかを調べた結果を格納したファイルを DETECT ファイルと呼ぶ.

3.3.2 従来最適化手法

最適化手法に改良を施すにあたって，従来最適化手法の流れを説明する.

1. **DETECT ファイルの生成**

   元のプログラム，またミュータントにテストスイートの各テストケースを適用し，各テストケースが各ミュータントを kill したか，kill しなかったかを示す情報を DETECT ファイルに出力する.

2. **BUG ファイルの生成**

   DETECT ファイルから各ミュータントを kill したか，kill しなかったかを示す情報を抽出し，BUG ファイルへ出力する．各テストケースについて，全ミュータントのうちどれか 1 つでも kill すれば 1，1 つも kill しなければ 0 と記録する.

3. **TS ファイルの生成**

   各テストケースの各パラメータ値 (文字列) を扱いやすくするために名義尺度の数値へマッピングし，TS ファイルへ出力する．このとき，パラメータ値

の情報を仕様書である TSL ファイルから確認する．

4. **パラメータ値の重みの計算**

   1 パラメータのみの条件下でベイズ推定を行うため，式 2.6 を用いて BUG, TS ファイルからパラメータ値の重みを求める．

5. **テストケースの重みの計算**

   前章で示したように，テストケースの重みは **”1 パラメータ-未定義値なし”** のメソッドを用いて定義する．テストケースの各パラメータに先ほど求めたパラメータ値の重みをマッピングし，それらの平均を取ることでテストケースの重みを計算する．

6. **テストケースの重みによりテストスイートを並べ替える**

   各テストケースのパラメータ値に対して重みを付け，テストスイートを降順に並べ替えることで最適化が完了する．

以降，この最適化手法を **original** と表記する．

3.4 提案手法

3.4.1 最適化手法の改良

本研究で提案する，ベイズ推定を用いた最適化の改良手法について説明する．従来手法とそれを改良した手法を比較することで，RQ1 に対する実験を行う．先ほど示した手法である original を次のように変更する．

- **BUG ファイルの仕様の変更**

  original の BUG ファイルでは，全ミュータントのうちどれか 1 つでも kill すれば 1，1 つも kill しなければ 0 と記録するものであったが，これを各ミュータントごとに，kill すれば 1，kill しなければ 0 と記録するように変更する．

  変更前と変更後の BUG ファイルの表記例を表 3.2，表 3.3 に示す．このとき，n をテストケースの個数，m をミュータントの個数とすると，$T_i(1 \leq i \leq n)$ はテストスイートの各テストケース，$M_j(1 \leq j \leq m)$ は対象プログラムの各ミュータントを表す．そして，T_i と M_j が交わった点にある数字が 1 ならば，テスト T_i がミュータント M_j を kill したことを示し，0 ならば，kill しなかった

ことを示す．

- **パラメータ値の重みの計算の変更**

  パラメータ値の重みを計算する方法は，変更前と同じように式 2.6 を用いて求める．ただし，BUG ファイルの 1 列ごとにパラメータ値の重みを計算できるので，m をミュータントの個数とすると，パラメータ値の重みは m 個求められる．そこで，パラメータ値の重みはこれら m 個の重みを総和したものを用いるように変更する．

以降，改良した最適化手法を **modified** と表記する．

3.4.2 利用する不具合情報

実際の現場での適用環境では，これからテストを行おうとするバージョンに不具合情報は存在しない．よって，パラメータ値の重みの計算には以前のバージョンの不具合情報を用いるのが通常であり，どの時点の不具合情報を用いるかの決定方法は複数考えられる．

RQ2 に対する実験を行うために，以下の 3 つの不具合情報の決定方法を用いた．

- **v1**

  対象プログラムの v1 の BUG ファイルをもとにパラメータ値の重みを計算する．

- **pre**

  対象プログラムの 1 つ前のバージョンの BUG ファイルをもとにパラメータ値の重みを計算する．

  v2 以降のバージョンに対するテストケースのみを対象とする．

- **hist**

  対象プログラムの以前のバージョンの BUG ファイルすべてをもとにパラメータ値の重みを計算する．例えば flex の v4 に対するテストケースの並べ替えを考えるとき，flex の v1，v2，v3 の BUG ファイルすべてから重みを計算し，その総和をパラメータ値の重みとする．

  v2 以降のバージョンに対するテストケースのみを対象とする．

表 3.2　変更前の BUG ファイルの表記例

test	fault
T_1	1
T_2	1
T_3	1
T_4	0
T_5	1

表 3.3　変更後の BUG ファイルの表記例

test	M_1	M_2	M_3	M_4
T_1	0	1	0	0
T_2	1	0	1	1
T_3	1	0	0	0
T_4	0	0	0	0
T_5	1	0	1	0

3.5 実験方法

本実験の方法，流れを以下に示す．

3.5.1 テストスイートの並べ替え

original，modified の 2 つの最適化手法のそれぞれに対して，v1，pre，hist の 3 つの不具合情報の決定方法を組み合わせたものを用いて，各プログラム各バージョンの並べ替えたテストスイートを作成する．

さらに，最適化手法の評価をより客観的に行うため，テストケースを無作為に並べ替える．並べ替えに用いる擬似乱数はメルセンヌ・ツイスター法 (MT 法) (注 3) により生成し，100 通りのテストスイートを作成する．

以降, original と v1, pre, hist を組み合わせた手法をそれぞれそのまま **v1，pre，hist** と，modified と v1，pre，hist を組み合わせた手法をそれぞれ **v1_m，pre_m，hist_m** と呼び，無作為に並べ替える手法を **rnd** と呼ぶ．

これら **v1，v1_m，pre，pre_m，hist，hist_m，rnd** の 7 つの手法に対して評価を行う．

3.5.2 NAPFD 値の計算

各手法を評価する指標として，v1，v1_m，pre，pre_m，hist，hist_m，rnd によって並べ替えられたテストスイートの，それぞれの NAPFD 値を計算する．

NAPFD 値の計算は，式 2.3 を用いて行う．

3.5.3 ミュータント kill 数の計算

各手法の不具合発見傾向を調査するために，v1，v1_m，pre，pre_m，hist，hist_m，rnd によって並べ替えられたテストスイート内のテストケースを，一番目から実行した場合の累計ミュータント kill 数を計算する．複数のテストケースが同一のミュータントを kill する場合，そのミュータントが最初に kill されたときのみをカウントする．

(注 3)：http://www.math.sci.hiroshima-u.ac.jp/ m-mat/MT/mt.html

そして，このミュータント kill 数を用いて，RQ1 に対する実験として，**original**，**modified** を比較し，RQ2 に対する実験として，**v1**，**pre**，**hist** を比較する．

4. 実験結果

本章では，実験によって得られた結果を示す．

4.1 NAPFD 値

表 4.1 に各手法によって並べ替えられたテストスイートの NAPFD 値とその平均値を, 図 4.1，図 4.2 に表 4.1 をプロジェクトごとに箱ひげ図にしたものを示す．ただし，表 4.1 の rnd の列の値は 100 個ある NAPFD 値の平均値である．

表 4.1 を見ると，flex では modified の NAPFD 値が original の値を概ね上回っている．特に，v1_m の値が高くいことが確認できる．一方，grep では modified の NAPFD 値が original の値を下回る．特に grep の v2 における値は 3 つの重みの決定方法とも同じになるのだが，そのときの modified の NAPFD 値が 0.5053 と著しく低くなっていることがわかる．

図 4.1，図 4.2 について説明する．これらの図の縦軸は NAPFD 値を示しており，横軸は各手法の名称を示している．rand を除いて箱ひげ図が 2 つずつ並んでいるが，これは左側が original の，右側が modified の NAPFD 値の箱ひげ図を示す．

図 4.1 を見ると，flex では original に比べて modified の方が概ね優れていることが確認できる．しかし，やはり grep では modified は original に劣る．先ほどの表 4.1 に見た，grep の v2 での NAPFD 値 0.5053 が大きく影響しているように見える．

4.2 ミュータント kill 数の推移

4.2.1 original，modified における比較

図 4.3，図 4.4，図 4.5，図 4.6，図 4.7，図 4.8 に各プロジェクトの不具合情報の各決定方法におけるミュータント kill 数の推移の，original と modified の比較を示す．

図 4.3，図 4.4，図 4.5，図 4.6，図 4.7，図 4.8 について説明する．これらの図の縦軸は kill されたミュータントの数を，横軸はテストスイート先頭から実行されたテストケースの数を示している．そして，青い線グラフは original の，緑色の線グラフは modified の結果を示している．また，赤い線グラフは 100 通りの rnd の結果の平均値

表 4.1　各手法の NAPFD 値

program	ver.	rnd	v1	v1_m	pre	pre_m	hist	hist_m
flex	v2	0.8715	0.8266	0.8344	0.8266	0.8344	0.8266	0.8344
	v3	0.8993	0.8504	0.8619	0.8181	0.8287	0.8306	0.8491
	v4	0.8739	0.8261	0.8631	0.8108	0.7601	0.8012	0.8198
	v5	0.8760	0.8711	0.8645	0.8385	0.8721	0.8146	0.8201
	avg.	0.8746	0.8434	0.8560	0.8235	0.8238	0.8183	0.8309
grep	v2	0.8861	0.9831	0.5053	0.9831	0.5053	0.9831	0.5053
	v3	0.9989	0.9989	0.9989	0.9989	0.9989	0.9989	0.9989
	v4	0.9472	0.9122	0.8036	0.9426	0.9552	0.9122	0.9558
	v5	0.8783	0.7867	0.7679	0.7689	0.7250	0.7867	0.7259
	avg.	0.9217	0.9202	0.7689	0.9234	0.7961	0.9202	0.7965
all	avg.	0.8982	0.8687	0.8118	0.8735	0.8100	0.8692	0.8137

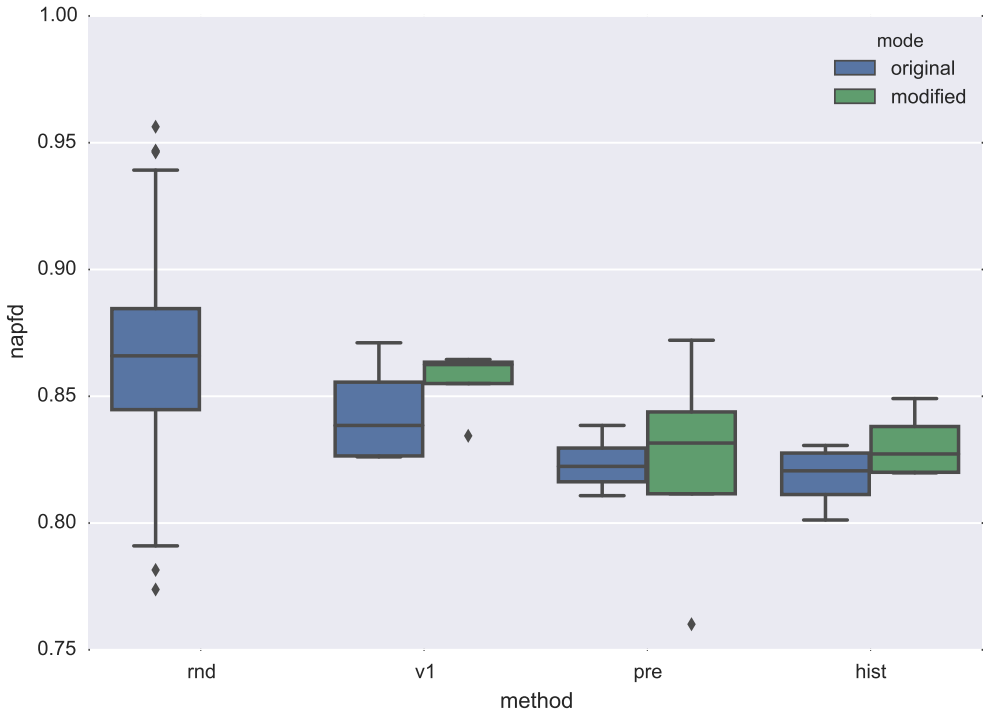

図 4.1　flex における各手法の NAPFD 値

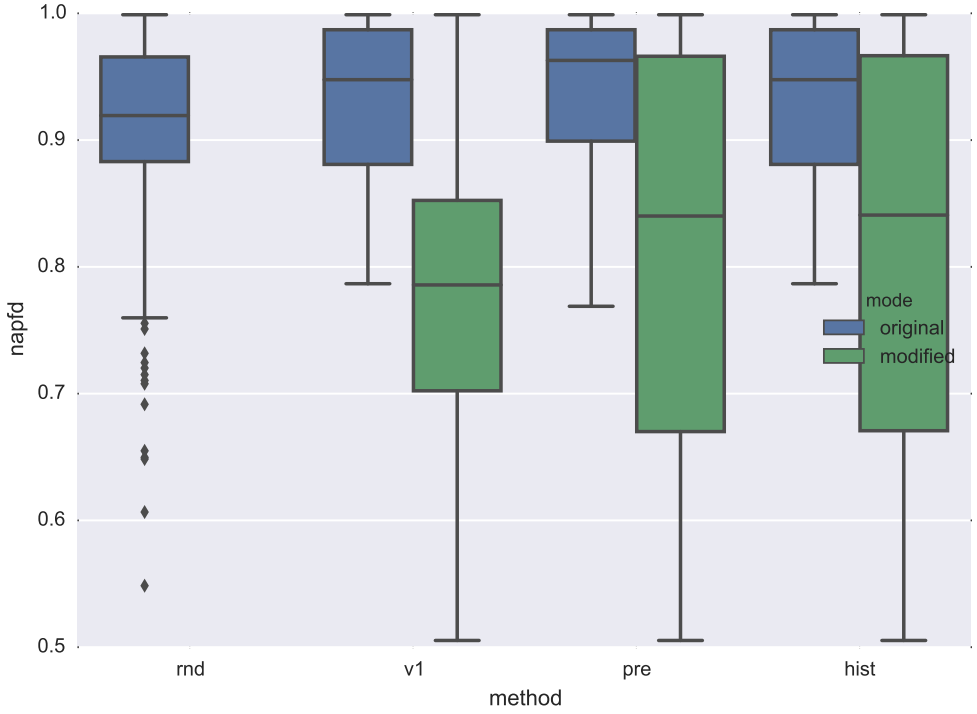

図 4.2　grep における各手法の NAPFD 値

を示している．よって，縦軸は kill されたミュータントの個数を表しているが，赤い線グラフの値は整数とは限らない．

結果を flex に限って見ると，序盤は modified の結果が優れており，中盤になると original の結果が勝るというパターンが多く見られる．また，modified は序盤に至っては，rnd の平均値の kill 傾向を上回ることがしばしば確認できる．grep の結果を見ると，ほぼすべての kill 数がひどく不安定であり，特に grep の v2 と v3 ではかなり極端なグラフが確認できる．これは，ある一部のテストケースが非常に多くのミュータントを kill 可能な場合に起こる．その一部のテストケースを偶然にでもテストスイートの先頭付近に並べ替えた場合には極端に結果が良くなり，そうしなかった場合には極端に結果が悪くなるためである．grep の v4，v5 に限って見ても，modified は初期の 5%〜10%こそは良い性能であるが，その後の kill 数の傾向はあまり結果は芳しくない．

4.2.2 v1，pre，hist における比較

また，図 4.9，図 4.10，図 4.11，図 4.12 に各プロジェクト各最適化手法におけるミュータント kill 数の推移の，v1 と pre と hist の比較を示す．

図 4.9，図 4.10，図 4.11，図 4.12 について説明する．これらの図の縦軸は kill されたミュータントの数を，横軸はテストスイート先頭から実行されたテストケースの数を示している．そして，緑色の線グラフは v1 の，青い線グラフは pre の，紫色の線グラフは hist の結果を示している．また，赤い線グラフは 100 通りの rnd の結果の平均値を示している．よって，縦軸は kill されたミュータントの個数を表しているが，赤い線グラフの値は整数とは限らない．

flex での結果では，NAPFD 値の結果に見たように，全体的には v1 のミュータントの kill 傾向が優れている．しかし，その他の手法が常に劣っているということではなく，ミュータントの kill 数の順位が複雑に入れ替わっていることが確認できる．

また，結果を grep に限って見ると，original，modified における比較にて見たように，やはり grep の v2 と v3 ではかなり極端なグラフであった．grep の v4 と v5 のグラフを見ると，flex の結果とは逆に v1 の kill 数が低いことが確認できる．pre と hist に関しては，グラフを見る限りではどちらが優れているかの判断がつかない．

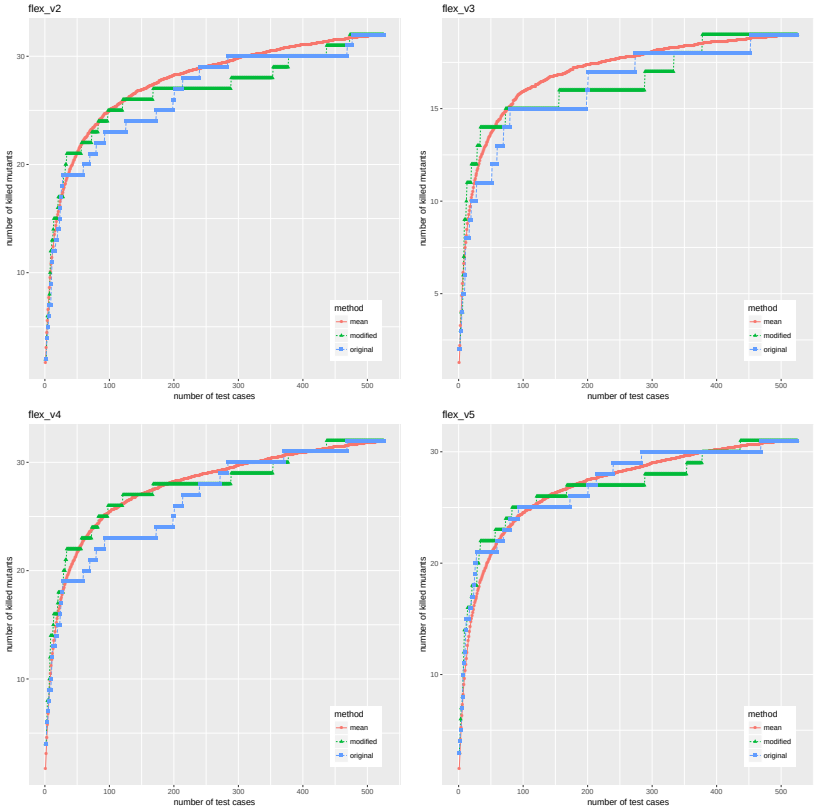

図 4.3　flex における v1 のミュータント kill 数の推移

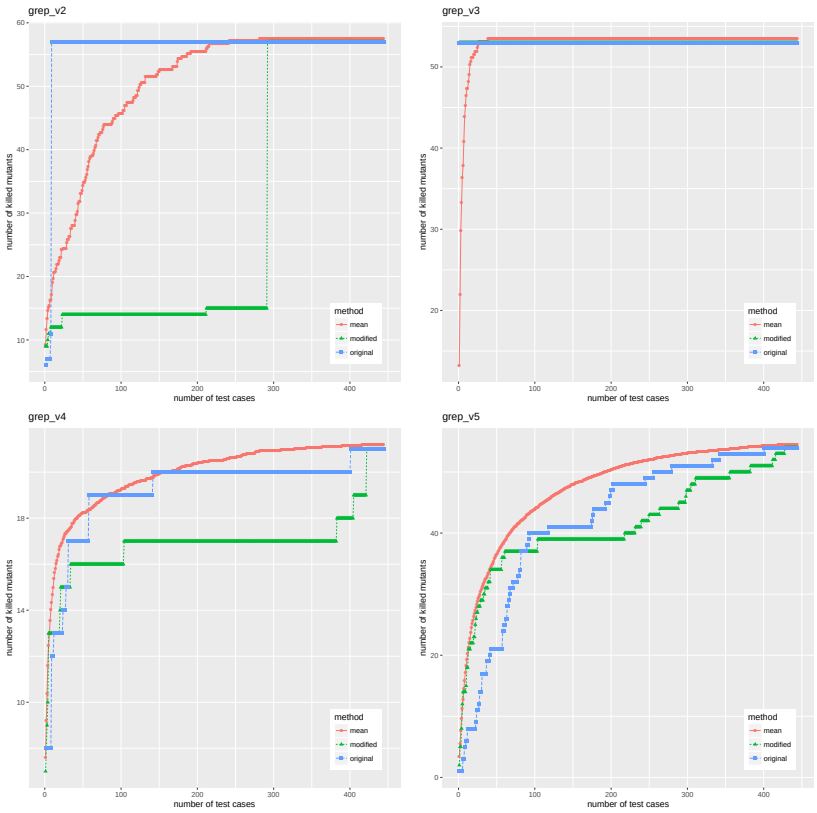

図 4.4　grep における v1 のミュータント kill 数の推移

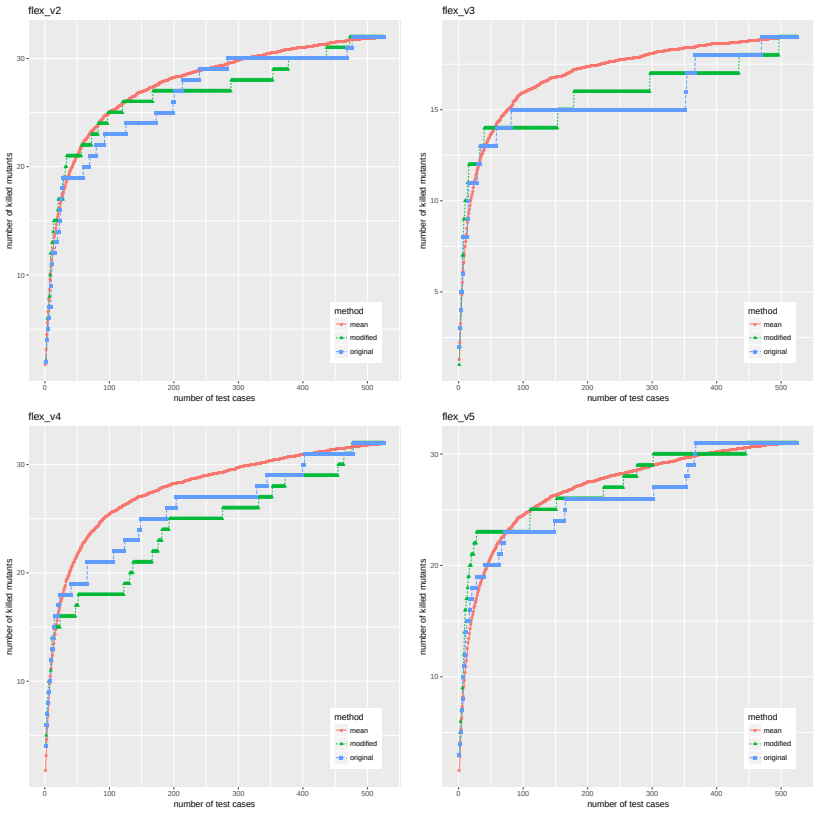

図 4.5　flex における pre のミュータント kill 数の推移

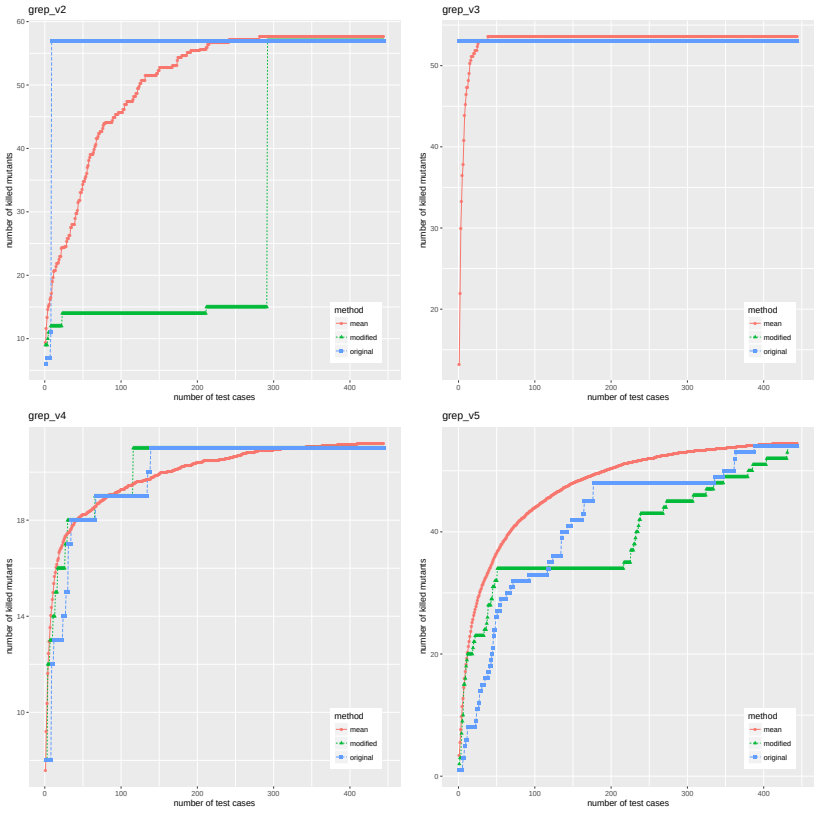

図 4.6　grep における pre のミュータント kill 数の推移

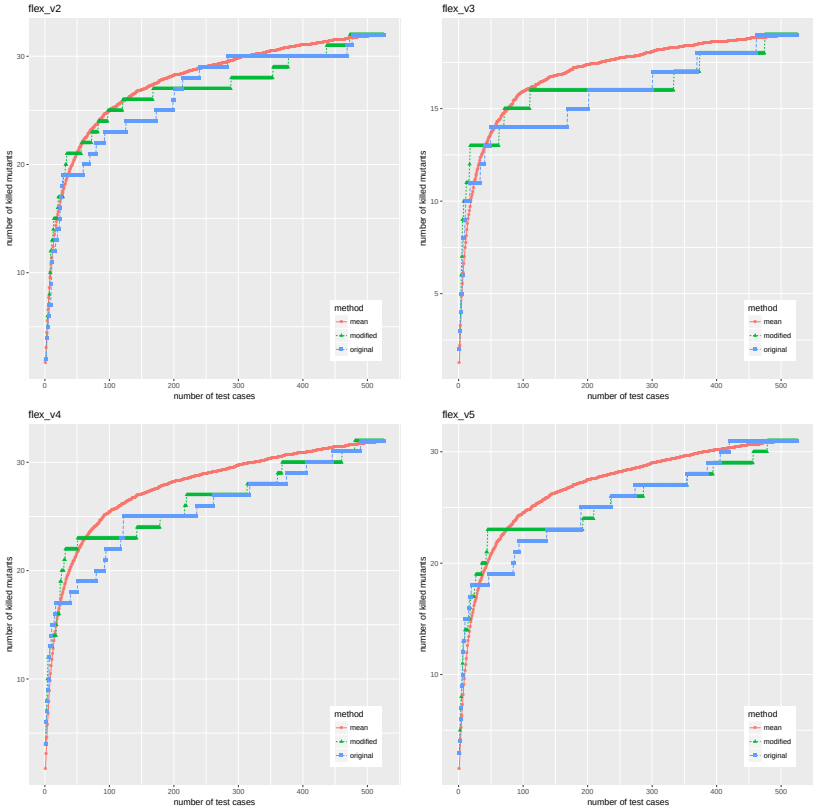

図 4.7　flex における hist のミュータント kill 数の推移

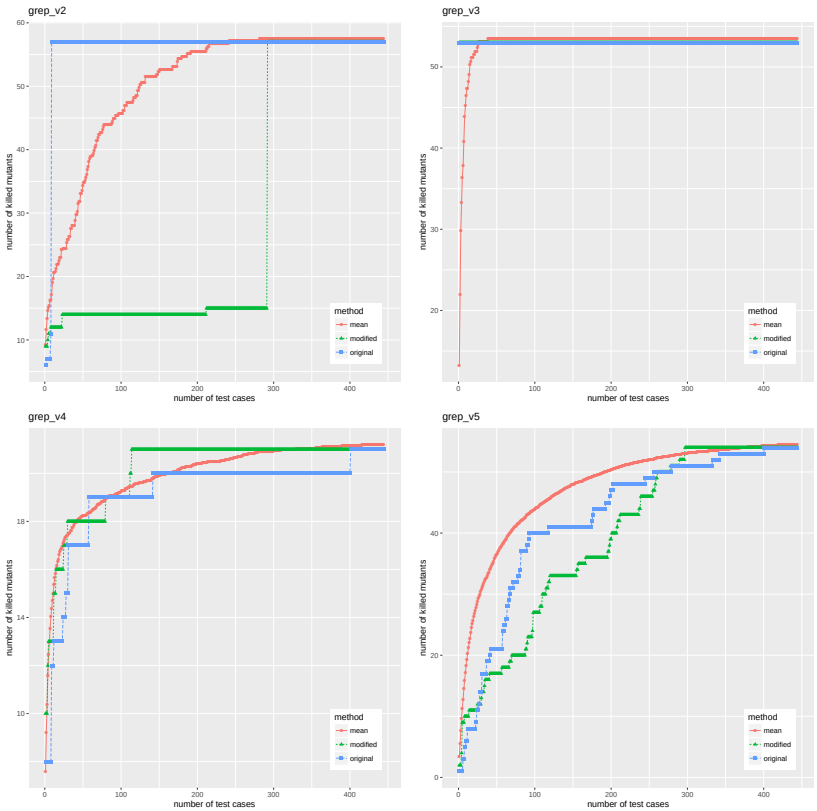

図 4.8　grep における hist のミュータント kill 数の推移

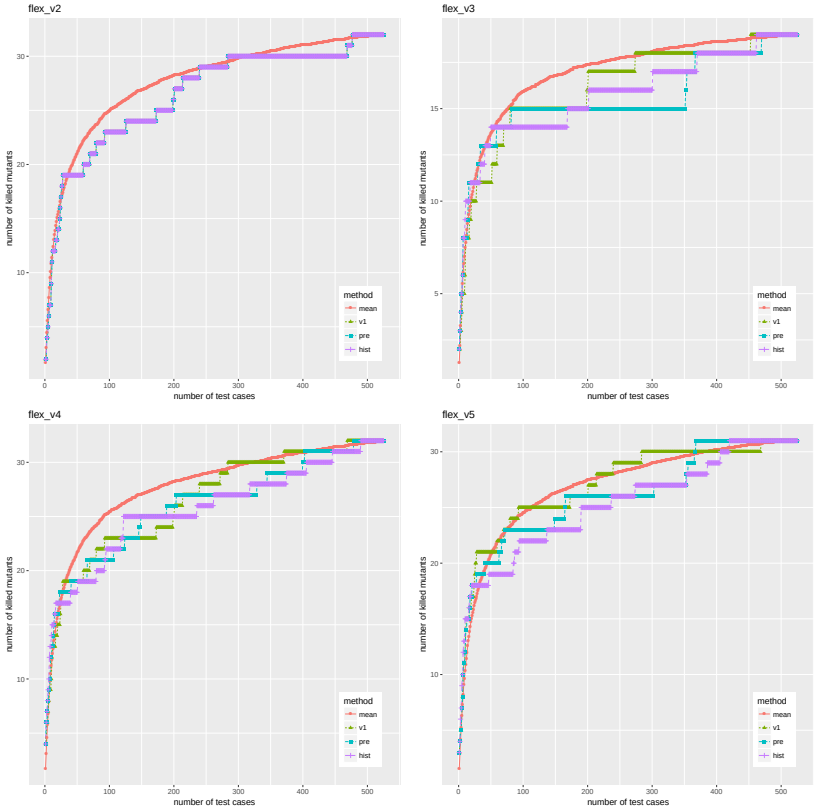

図 4.9　flex における original のミュータント kill 数の推移

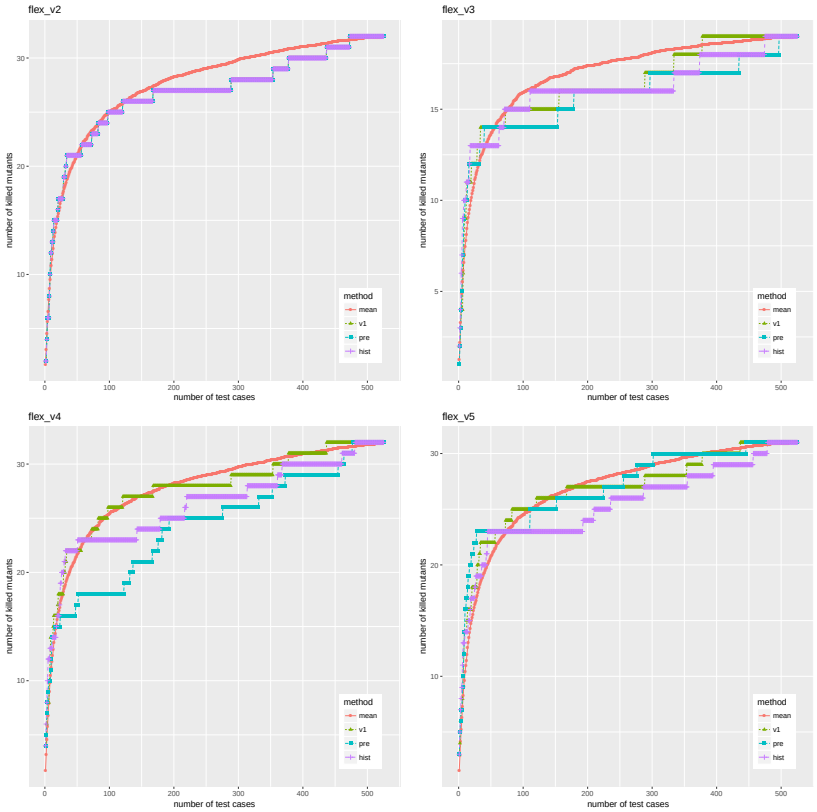

図 4.10　flex における modified のミュータント kill 数の推移

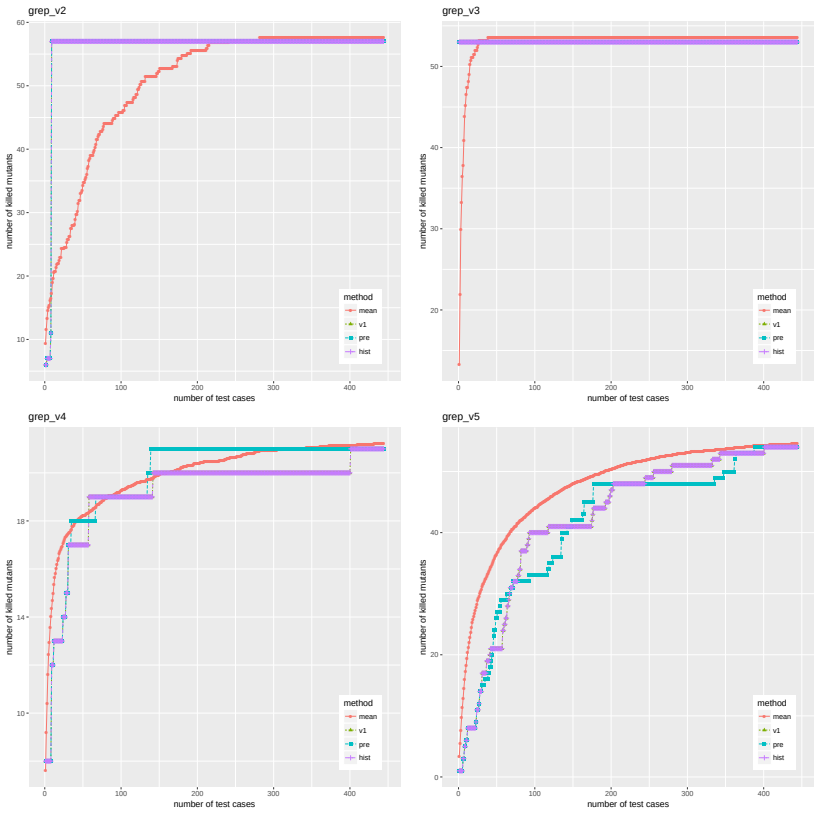

図 4.11　grep における original のミュータント kill 数の推移

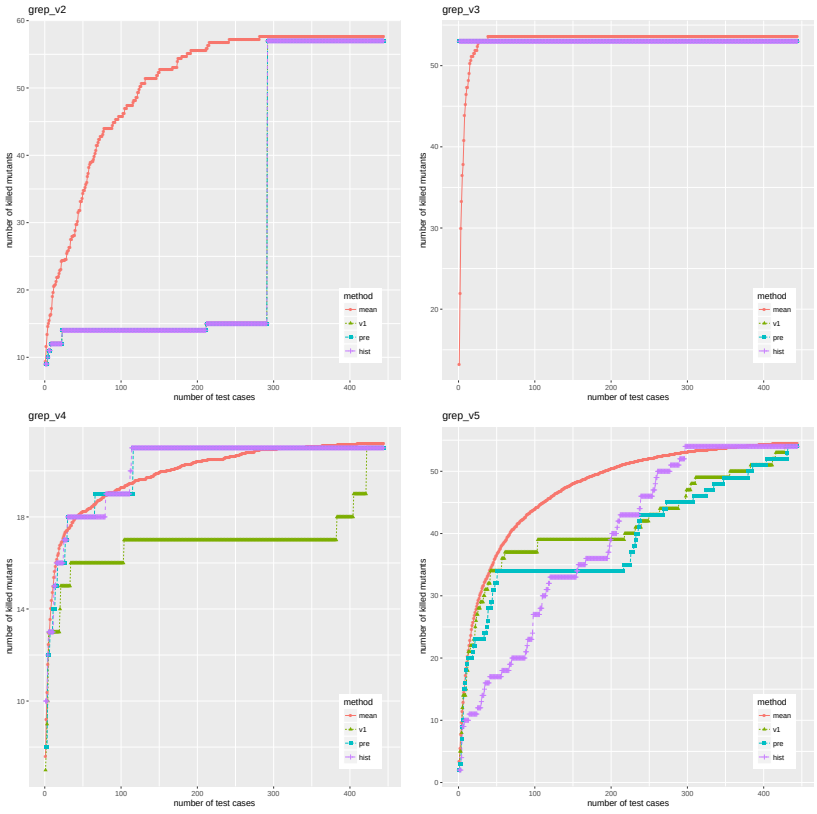

図 4.12　grep における modified のミュータント kill 数の推移

5. 考察

本章では，実験に対する考察を行う．

5.1 実験結果の考察

5.1.1 NAPFD 値

表 4.1 に見られるように，flex においては全体的に modified の方が original より NAPFD 値が高くなっている．このことは，図 4.1 を見ても明白である．ただし pre に関しては，第一四分位数と第三四分位数の間隔が比較的大きく，modified の方が優れているとは言えない．また，rnd が一番優れているように見えるが，実際はばらつきが大きいため，同程度の NAPFD 値を持ちながらばらつきの小さい v1_m は，十分に優れていると言えるであろう．

しかし，grep での結果を見ると，すべてのメソッドで modified の NAPFD 値が original より低くなっているのが分かる．図 4.2 を見てもその様子は歴然であり，grep においては改良は良い結果を示さなかったようである．modified の grep の v2 の結果が極端に低いうえに，rnd を除いた各箱ひげ図の標本数が 4 つまたは 5 つしかなく，下方に大きく引っ張られているため実際は見た目ほど酷くはないはずであるが，grep の v2 の結果を除いたとしても，original に少し劣ってしまっている．

全体の平均を見ると，modified よりも original の方が優れて見える．しかし，grep の v2 の modified の極端に低い結果が大きく足を引っ張っているのは明白である．そこで，平均から grep の v2 の結果を除いてみたところ，大差のない結果となった．

また，v1 と pre と hist について全体の平均に注目すると，これらもまた大差はなかった．プログラムごとに有利不利はあるが，どの手法が優れているかということが無いように推測される．

5.1.2 ミュータント kill 数の推移

(1) original，modified における比較

図 4.3〜図 4.8 を見たところ，grep の v2 と v3 におけるグラフはすべて極端な例となっている．これは，先ほど述べたように，ある一部のテストケースが非常に多く

の不具合を検出可能な場合に起こる．その一部のテストケースを偶然にでもテストスイートの先頭付近に並べ替えた場合には極端に結果が良くなり，そうしなかった場合には極端に結果が悪くなるためである．

grep の v2 と v3 以外のプログラムについて考察する．これらの図の modified の線グラフについて，全メソッド全プログラムについて概ね共通している点に気付いた．それは，立ち上がりが早いことである．modified はテストケースの初期 5%あるいは 10%近くまでの間に，多くの不具合を発見しており，rnd の平均値を超えていることがしばしば見受けられる．すなわち，改良した最適化手法は，より多くの不具合を発見できるテストケースをテストの初期段階で行うことができる．しかし，テストケースの 10%を超えたあたりから，その性能を大きく落としてしまう．その理由として，一度発見した不具合を重複して発見することが多くなったのであると考える．改良によって多くの不具合を検出するテストケースの重みを強くした結果，特定のパラメータ値の重みが極端に高くなってしまったと推定する．もし，そのパラメータ値が特定の不具合の発見と強い関係を持っているとすれば，その不具合が検出されるテストケースの重みが強くなる．そうすると，多くの不具合を検出するテストケースが初期に置かれた後は，その検出された不具合を検出するテストケースが優先的に置かれやすくなってしまう．

(2) v1，pre，hist における比較

図 4.9〜図 4.12 を見たところ，v1 は，flex では一貫して優れているが，grep ではあまり性能が良くない．pre は，バージョンによってまったく違う不具合情報を参照するためか，結果が良いときもあれば悪いときもある，といったように見られる．そして，hist であるが，結果が安定してはいるがテスト性能はそれほど高くないように見える．

以上より，「v1 はプロジェクトによって一貫した結果が出るが優れているとは限らない」，「pre は結果が良いときもあれば悪いときもあり，テストするバージョンによっては最適な手法であることもある」，「hist は結果が凡庸であるが安定している」と推測する．データが不足しており推測の域を出ないため，更なる研究が必要である．

5.2 研究設問への回答

5.2.1 RQ1:改良した手法によるテストは従来法に対して良いテストとなるか

改良した最適化手法は初期の不具合発見傾向が強いため，制約が大きく，多くのテストが行えない環境では良いテストとなり得る．

しかし，中期以降は同じ不具合が検出される可能性が高く，多くのテストを行うためには，より高度な重みのコントロールが必要であると考える．

5.2.2 RQ2:どの時点の不具合情報を利用したテストが良いテストを生成するか

プロジェクトによって得手不得手があり，どれが優れているかは一概には言えない．それぞれの結果の平均値はほとんど同じであるため，どれを選んでも大差はないと考える．

5.3 妥当性の検証

本実験で得られた結果の妥当性を検証する．

- **対象プロジェクトの不足**

  本実験にて対象としたプログラムは flex，grep の 2 種類であり，そのバージョンは計 10 個であった．今回の結果が一般的であると主張するにはサンプルの数が不足している．また，対象としたプログラムは 2 つとも C 言語で記述された OSS であり，結果に信頼性を持たせるにはより多様なプログラムを対象にするべきである．

- **データセットの不備**

  本実験で使用した SIR のデータセットや，Henard らのミュータントのデータセットは第三者が作成したデータであるため，不備がある可能性がある．

- **プログラムの不具合**

  実験を行うにあたって，複数のプログラムを作成したが，これらのプログラムが完全であるとは限らない．十分にテストをしているが，不具合が含まれている可能性がある．

また，一部のプログラムは第三者が作成した物を再利用している．出力データの整合性を確認しているため，問題なく動作していると考えるが，確実ではない．

6. 結言

本研究では，ベイズ推定を用いた最適化手法に対して改良を行い，元の最適化手法と改良した最適化手法とを用いてミューテーション解析を行った．そして，これらの最適化手法によって並べ替えられたテストスイートによる不具合の発見傾向を調査・評価した．

実験の結果，改良した最適化手法により並べ替えられたテストスイートが，初期段階において非常に高い不具合発見傾向を持つことを示した．また，重みとして利用する不具合情報の決定方法について，一般的に最適な方法はないことを推測した．

今後の課題としては，重みによりテストケースの並べ替えを行う際，テスト中盤においても強くなるよう，最適化手法を改良することが考えられる．また，さらにより多くのプログラムに対して実験を行い，より多くの結果を得て，今回の結果や推測を一般的に言えるようにすることを今後行うべきであると考える．

謝辞

本研究を行うにあたり，研究課題の設定や研究に対する姿勢，本報告書の作成に至るまで，全ての面で丁寧なご指導を頂きました，本学情報工学部門 水野 修 教授，加えて産業技術総合研究所情報技術研究部門ソフトウェアアナリティクス研究グループ 崔 銀惠 研究員に厚く御礼申し上げます．本報告書執筆にあたり貴重な助言を多数頂きました，本学ソフトウェア工学研究室の皆さま，学生生活を通じて著者の支えとなった家族や友人に深く感謝致します．

参考文献

[1] R. Bryce and C. Colbourn. Prioritized intersction testing for pair-wise coverage with seeding and constraints. *Information and Software Technology*, Vol. 48, No. 10, pp. 960–970, 2006.

[2] E. Choi, T. Kitamura, C. Artho, A. Yamada, and Y.Oiwa. Priority integration for weighted combinatorial testing. *Proc. of 39th Annual International Computer Software and Applications Conference*, pp. 242–247, 2015.

[3] J. Czerwonka. Pairwise testing in the real world: Practical extensions to test case generators. *Microsoft Corporation Software Testing Technical Articles*, 2008.

[4] Sebastian Elbaum, Alexey G. Malishevsky, and Gregg Rothermel. Test case prioritization: A family of empirical studies. *IEEE Trans. on Software Engineering*, Vol. 28, No. 2, pp. 159–182, February 2002.

[5] C. Henard, M. Papadakis, M. Harman, Y. Jia, and Y.L. Traon. Comparing white-box and black-box test prioritization. *Proc. of 38th International Conference on Software Engineering*, pp. 523–534, 2016.

[6] S. Kawabata, E. Choi, and O. Mizuno. A prioritization of combinatorial testing using bayesian inference. *IEICE Technical Report SS2015-95*, pp. 115–120, 2016.

[7] T.J. Ostrand and M.J. Balcer. The category-partition method for specifying and generating functional tests. *Communications of the ACM, Vol.31, No.6*, pp. 676–686, 1988.

[8] X. Qu, M.B. Cohen, and K.M. Woolf. Combinatorial interaction regression testing: A study of test case generation and prioritization. *2007 IEEE International Conference on Software Maintenance*, pp. 255–264, 2007.