特別研究報告

題目

遺伝的アルゴリズムを用いた自動プログラム修正における個体生成時間の計測とその分析

指導教員

楠本 真二 教授

報告者

皆森 祐希

令和 5 年 2 月 7 日

大阪大学 基礎工学部 情報科学科

令和 4 年度 特別研究報告

遺伝的アルゴリズムを用いた自動プログラム修正における個体生成時間の計測とその分析

皆森 祐希

内容梗概

ソフトウェア開発において，デバッグは開発工数の半分を占める作業といわれている．自動プログラム修正は，バグを含むプログラムに変更を加えることで用意したテストを通過するプログラムを出力する手法であり，デバッグの工数を削減することが期待されることから研究が盛んに行われている．自動プログラム修正の手法として，遺伝的アルゴリズムに基づいて修正を行うものがある．ここで，遺伝的アルゴリズムを用いた自動プログラム修正は，目的のプログラムが得られるまでプログラム文の挿入，削除，置換及び交叉を行う手法である．今回実験で使用する自動プログラム修正ツールの kGenProg も遺伝的アルゴリズムを採用している．現状の自動プログラム修正における課題の 1 つとして，バグの修正に長い時間がかかる点があげられる．しかし，kGenProg にはバグ修正時間のうち個体生成に対する計測機能が存在しない．そこで，本研究では kGenProg に各個体の生成にかかった時間を計測する処理を追加し，さらに得られた個体情報をビルドの成否および解であるかによって分類した．その後，全体の生成時間のうち修正に成功したプログラムの生成経路にかかった時間およびビルドに失敗した個体に費やした時間を定量的に計測することで，自動プログラム修正における無駄な経路や修正に役に立った経路の時間的コストを調査した．結果として，修正対象のバグによっては全個体の生成時間のうち 80% 前後がビルドに失敗する個体に費やされたことや，多くのプログラムで修正に成功した個体に費やした時間が全体の 10% を下回ったなどの結果を得た．

主な用語

自動プログラム修正， 時間計測，可視化， JSON

目次

図目次

表目次

1　はじめに

ソフトウェア開発におけるデバッグ作業は費用および時間的な点で多くのコストを必要とする．ある研究によると，ソフトウェア開発にかかるコストのうち，50% 以上をデバッグが占めるという結果が出ている [1, 2].

それらのコストを削減するための手段の 1 つとして，自動プログラム修正 (APR) があげられる．APR は，人の手を介さずにソースコード中に含まれるバグを自動的に取り除く技術であり，盛んに研究が進められている [3, 4].

APR に関する研究には意味論的な制約を抽出し，修正プログラム同士を合成することでバグを取り除く意味論をベースとしたものもある [5] が，本研究においては遺伝的アルゴリズムを用いた APR について述べる．APR の実用化に向けて，ここ 10 年で数多くの研究がなされている．例えば，生成と検証[*1]に基づく複数の APR ツールにおけるデザイン空間を調べ，それらを拡張することで新たな APR ツールを提案したり [6]，1 つのツールで複数の言語に対するプログラムのバグを取り除く研究 [7] などがある．それらの実用化に向けた研究のうち，コストを削減する取り組みも行われている [8, 9]．しかし，遺伝的アルゴリズムを採用した APR において，個体の生成に要する時間を調査した研究はされていない．そこで，本稿では，既存の生成と検証に基づく APR ツールを拡張して個体の生成時間を計測し，筆者が考案した指標を実際のバグに対して計算することで時間的コストを調査した．

この調査の目的として，既存の APR ツールの個体生成にかかる時間的コストを調べることがあげられる．また，調査の結果として，修正対象のバグによっては全個体の生成時間のうちおよそ 8 割ほどがビルドに失敗する個体に費やされたことや，多くのプログラムにおいて修正に成功した個体に費やされた生成時間が全体の 1 割に満たない程度であるといったような興味深い研究データが得られた．

以降，2 章では APR の課題および本研究を行う契機となった研究の説明および研究のための予備知識について説明する．3 章では本研究における提案手法について論じる．4 章では今回提案した評価指標とその具体例について論述する．5 章では実験の概要と結果を提示する．6 章では実験から得られた結果による考察について述べる．7 章では本研究の妥当性への脅威について論じる．8 章では今後の課題について説明し，最後に 9 章で総括を述べる．

[*1] はじめに候補となるパッチをいくつか生成した後にそれらの適合性をテストスイートを用いて検証する方法であり，プログラム修正におけるデザイン哲学の 1 つである [6]

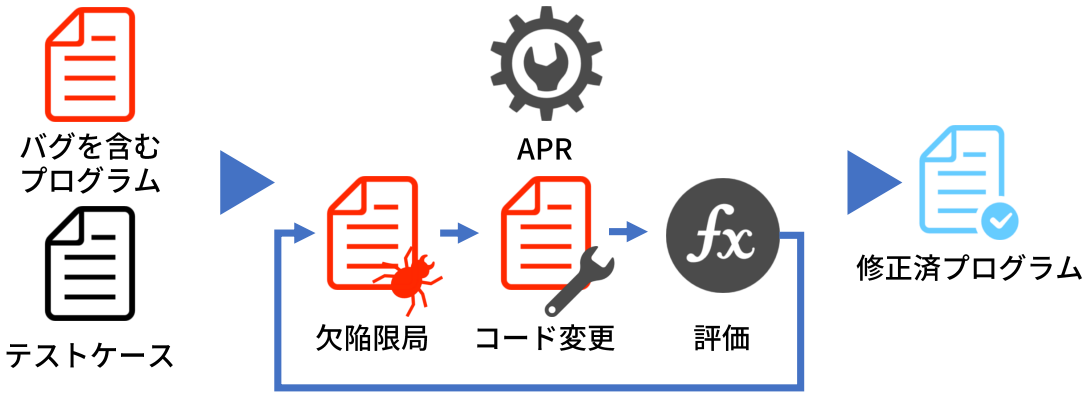

図 1　自動プログラム修正：APR

2　準備

2.1　自動プログラム修正 (APR)

APR は，計算機が自動的にプログラムのバグ修正を行う技術であり，テストケースとバグを含むプログラムを入力とし，修正したプログラムを出力とする．APR は大きく探索ベース[*2]と意味論ベース[*3] [5] の 2 つの手法に分類することができる．

本研究では，探索ベース APR 分野のブレイクスルーとなった GenProg [10] の採用する生成と検証 [6] に基づく手法に重点を置く．

この手法では，図 1 のように，対象となるプログラムにおけるバグの位置を特定する手法である欠陥限局[*4]を行い，限局した箇所に変更を加えた後ビルドとテストを実行して修正尺度の評価を行う．

2.2　遺伝的アルゴリズム

遺伝的アルゴリズム (GA) は各世代で個体を選択し，それらに変異，交叉などの操作を加えることでより強い個体を生成する生物の進化に基づくアルゴリズムである．これらの個体から以下の遺伝子操作

[*2] Heuristics-based
[*3] Semantics-based
[*4] バグ限局ともいう

を行うことで，次の世代の個体の集合を生成する．

選択 …個体のうちから何らかの関数による適応度 (APR ではテストスイートの通過率) に応じて選択

変異 …個体の遺伝子を変異させる (APR ではコードの一部を変更)

交叉 …複数の遺伝子の一部分を交配させて新しい遺伝子を生成

これらの遺伝子操作のうち，変異において APR では以下の操作を対象のコードに対して行う．

挿入 …選択したコード近辺への別コードの追加

置換 …選択したコードの別コードへの書き換え

交叉 …選択したコードの削除

遺伝的アルゴリズムの具体的な説明として，図 2 のような遺伝について考える．なお，図中の数字はテストスイートの通過率 (以下，単に通過率と表す) を表し，もっとも左の個体群を第 1 世代として，右に行くほど世代が進んでいるものとする．

第 1 世代においては，通過率 0.5 の個体を選択したもの，通過率 0.3 の個体に変異を起こしたもの，そして通過率 0.4 と 0.3 の個体を交叉したものの 3 つの個体を第 2 世代の個体として生成している．同様に，第 3 世代の個体は，第 2 世代の個体のうち，通過率 0.7 の個体の選択およびこの個体と通過率 0.5，通過率 0.6 の個体との交叉による個体となっている．ここで，第 3 世代の個体に通過率 1.0 の個体が存在するのでここでアルゴリズムを終了する．

2.3 既存の APR ツールの課題

既存の APR ツールは多くの課題が解決されておらず [11]，実用には程遠い段階である．具体的には，修正後のプログラムの可読性が低い [12]，単一のバグの修正は行えてもバグが複数になると修正が困難になる [13] などがあげられる．

その中で今回主に取り上げる課題として，1 回の修正実行にビルドやテスト実行といった多くの時間的コストがかかる点 [14] があげられる．この課題の動機として，筆者が実際に APR ツールを実行したときに比較的単純なプログラムを対象とした修正において手動でバグを修正したときよりも時間的コストがかかったことがあげられる．また，ビルドとテストにかかる時間を削減する研究が過去に行われているものの [8] ，個体の生成にかかった時間を計測することに関する研究はまだ発展途上である．

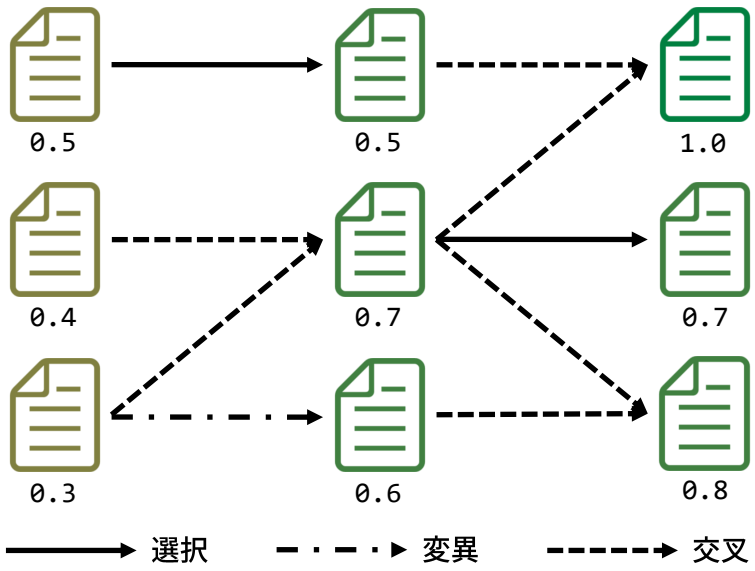

図 2　遺伝的アルゴリズムの例

2.4　時間的コスト調査の先行研究

Ghanbari [15] らは従来のソースコード解析に基づいた APR を実行した後に，バイトコードに APR を施す PraPR を提案し，既存の APR の性能を飛躍的に向上させた．この研究において，Ghanbari らは PraPR を Defects4J の Chart および Closure リポジトリに含まれるバグに対して修正を行うとともに時間的コストを計算した．結果として，Closure リポジトリのバグでは有効なパッチ (修正されたプログラム) の数が Chart リポジトリのバグに比べて 10 倍生成されたものの，時間的コストは 20 倍かかった．

また，古藤 [8] らは欠陥限局を用いて変更コード片を動的に切り替える手法を提案しビルドに費やす時間を従来手法に比べて 89% から 46% に削減することができ，結果として APR 全体の修正時間の削減につながった．これらの研究ではプログラム修正全体やビルド時間の観点から時間的コストの調査を行っていたが，自動プログラム修正の時間的コストについて別の観点から詳らかに調べてみようと思い，本研究をするに至った．

図 3　kGenProg における入力と出力

3　提案手法

APR ツールにおける従来の研究においては，個体の生成にかかった時間を計測し，得られた値に基づいて分析を行うといった研究はなされていない．そこで，本研究では遺伝的アルゴリズムを採用した GenProg [10] 系の APR ツールの 1 つである kGenProg [16] の個体処理部分に時間計測用のコードをはさみ，記録された生成時間を独自の指標で評価する手法を提案する．図 3 は kGenProg におけるプログラム修正の流れを表す．本研究においては kGenProg の生成処理のソースコードに個体の生成時間の情報を追加した，また，kGenProg には，個体情報を JSON 形式で出力するオプションが用意されている．本研究では，kGenProg で同一のバグ修正を複数回実行し，それら 1 つ 1 つに対して実行結果を出力する．

図 4 はこれらの JSON ファイルを Python で記述されたファイルに入力し，各個体の ID や生成時間，テストスイート通過率の情報および全体の生成時間，ビルドに失敗した個体の生成時間および解となる個体の生成時間とそれらから得られる指標を記載した各修正におけるテキストファイルと対象のバグすべての指標を記載したテキストファイルを出力として得る．

工夫点として，1 回ごとの修正に対する個体情報を記載したファイルと対象のバグ全体に対する指標を記載したファイルを 1 つの Python プログラムで実装した点があげられる．

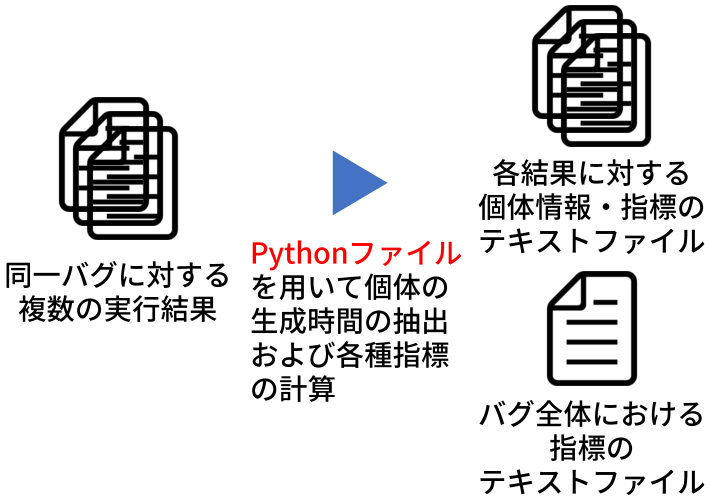

図 4　実行結果の解析：入力と出力

4　評価指標とその実装

4.1　評価指標

4.1.1　概要

独自の指標は

1. 解の生成に関係する個体を特定し，それらの生成時間を計算した後あらかじめ求めた全個体の生成時間で割る
2. ビルドに失敗する個体の生成時間を特定・計算した後，1 と同様に善個体の生成時間で割る

で定義され，これらは APR ツールの時間的コストの傾向を知るためのものである．

4.1.2　STR

APR を実行するにあたり，修正に成功したバグに対して，その解となる個体[*5]に関係する経路の全体の生成時間に占める割合を求める指標として，**STR**(Solution Time Ratio, 解時間比率) を次の式で定義する．

$$\mathrm{STR} = \frac{\text{解となる個体の経路の総生成時間}}{\text{すべての個体の総生成時間}} \tag{1}$$

ここで，解となる個体の経路の集合は

1. まず解である個体を選択し，集合に入れる
2. 集合内の全個体に対してその親を求め，それらを集合に入れる
3. 集合に変化がなくなるまで 2 を繰り返す

のように求められる．STR を計算する目的として，プログラム修正において成功に必要な個体の生成が時間的にどの程度の割合を占めているのかを定量的に求めることがあげられる．そのため，STR の値は大きい方が好ましい．

4.1.3　FTR

一方で，すべてのバグに対してビルドに失敗する個体がすべての個体の生成時間に占める割合を求める指標として，**FTR**(Failure Time Ratio, 失敗時間比率) を次の式で定義する．

$$\mathrm{FTR} = \frac{\text{ビルドに失敗した個体にかかった総生成時間}}{\text{すべての個体の総生成時間}} \tag{2}$$

[*5] ここでは「修正に成功しすべてのテストスイートを通過した個体」と定義する，以下同様

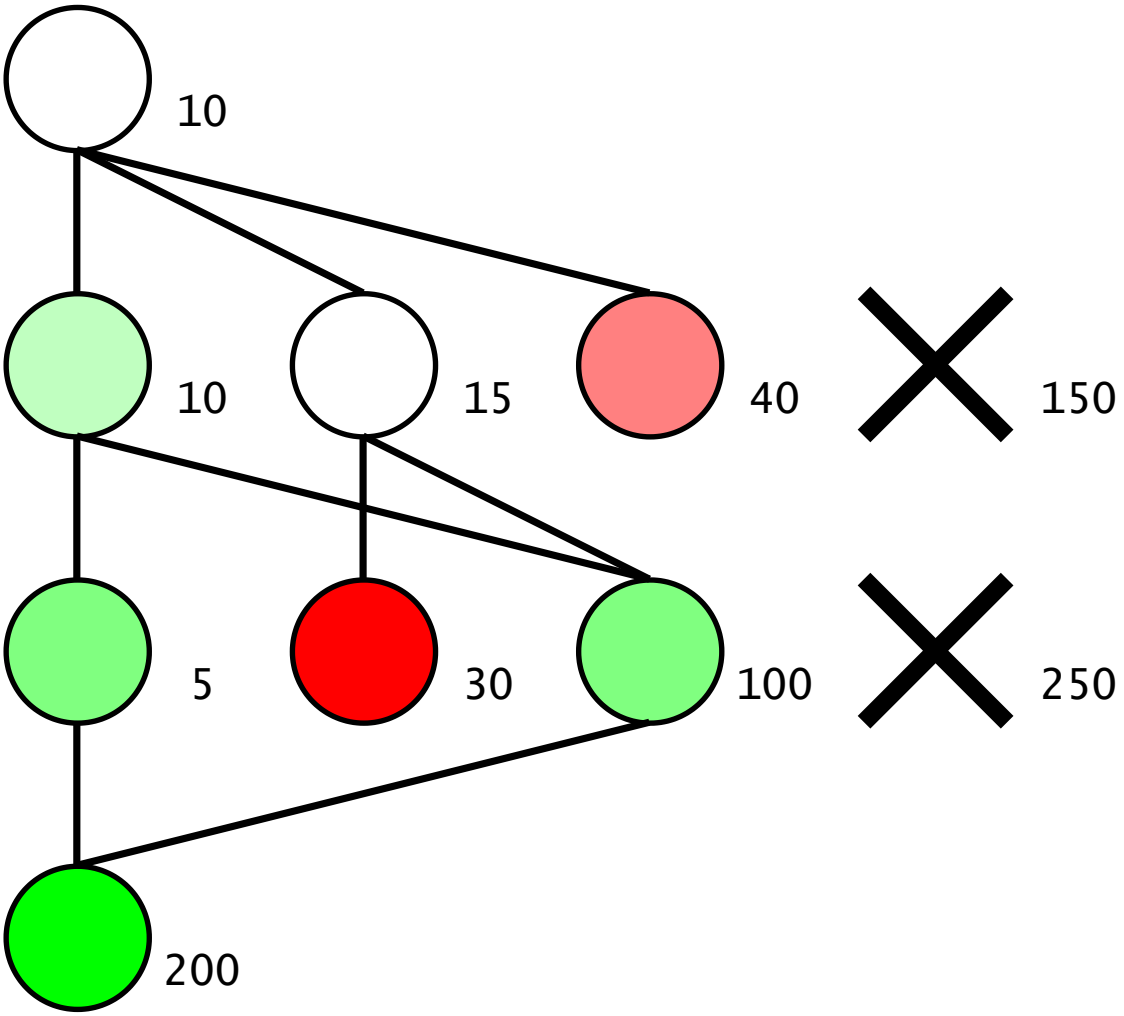

図 5　生成結果のツリー表示例

FTR を計算する目的として，プロジェクト内のバグを含むプログラムの修正にかかる時間のうちビルドに成功せずに終わった個体がどの程度の割合を占めるかを定量的に測定し，その傾向を知ることがあげられる．そのため，一般的に FTR の値は小さい方が望ましいとされる．

4.2　STR・FTR の具体的な計算例

先ほど定義した値を求めるための具体的な例として，図 5 の生成木をもつ修正結果について考える．この生成木は，kGenProg の実行結果を記した JSON ファイルをツリー状に表示する Macaw [17] を参考に描画した[*6]．ここで，縦方向は世代を表しており，下に進むにつれてより新しい世代を表す．また，

[*6] ビルドに失敗した個体集合を表す X 印における数字の意味合いとして，この例では生成時間の和を表すが Macaw においては個体の数を指すなどの細かい違いがある点に注意

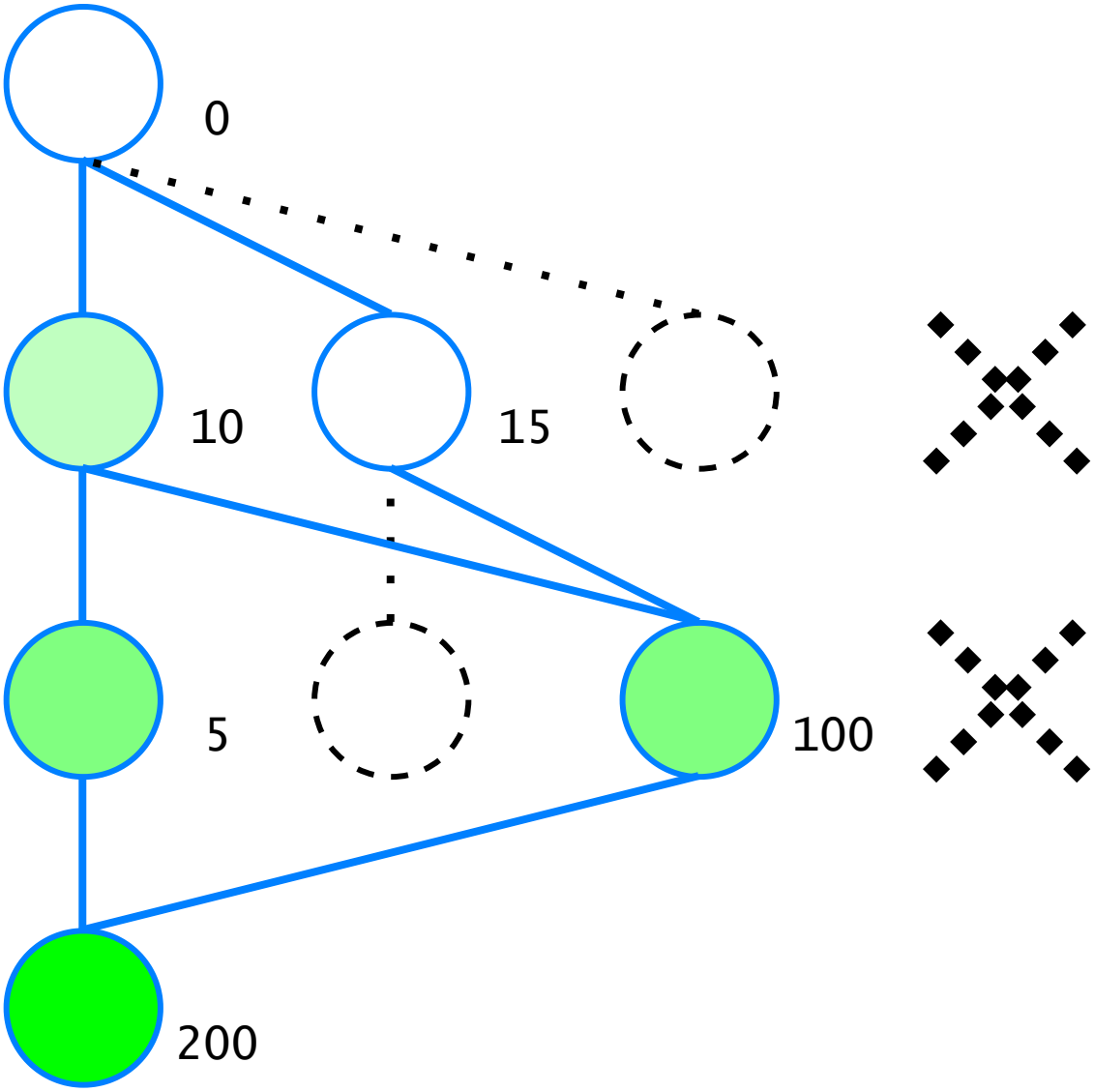

図 6　生成結果の例：解となる個体の経路

横の列は一つの世代におけるすべての個体を表す．円及び X 印はそれぞれビルドに成功した 1 つの個体，その世代でビルドに失敗したすべての個体を表す．円の色は個体の Fitness(全体のテストケースに占める期待通りのテスト結果が得られた割合) を表し，緑に近いほど高い Fitness を右下の数字はその個体あるいは個体の集合の生成時間を表す．なおこの例におけるプログラムの総生成時間は 800 である．この時，STR は図 6 で表される部分の時間 $(10+15+5+100+200) \div 800 = 0.4125$ となり，FTR は X 印で示されたすべての時間を足した値を総生成時間で割ったもの，すなわち $(150+250) \div 800 = 0.5$ と求められる．

4.3 APR ツールへの時間計測機能実装

提案手法では，既存の APR ツールである kGenProg [16] を拡張して実装する．ソースコード内部のふるまいは変えずに，`Mutation` クラスと `RandomCrossover` クラスに時間計測を行うコードを挿入した．具体的には，`org.apache.commons.lang3.time.StopWatch` クラスをインポートし，各個体を生成するループの最初に `StopWatch.createStarted()` メソッドを呼び出すことによって時間計測を開始，処理を終えた後に `getTime()` メソッドを呼び出すことで生成時間を取得し，それを個体情報に格納する．この際，kGenProg に付属している JSON ファイルの出力オプションをオンにすることで対象のバグにおける個体の解析を可能にする．

次に，JSON ファイルを Python で記述したプログラムを用いて処理し，各個体の ID(通し番号)・生成時間・Fitness(ただしここでは ID に対応する個体がビルドに失敗した場合-1 を格納する) の情報を取得した後，その情報をもとに STR と FTR を計算する．

具体的には，出力となる JSON ファイルを読み込み，バグの修正過程から生成された個体の時間を 1 つずつ取得する．この時，Fitness の値で追加の処理を行う．例えば Fitness が-1 であればビルドに失敗した個体であるので失敗時間 (FTR を計算する際の分子) に加算する．また，Fitness が 1 であれば，テストケースを満たす解となる個体であるので 4.1.2 で挙げた手順で解となる個体の親を求める．プログラム中では再帰的なアルゴリズムを用いている．

5　実験

5.1　概要

本章では，GA を採用した APR ツールである kGenProg [16] を用いて，Defects4J [18] の Lang バグの Lang1〜Lang44 を対象に自動プログラム修正を行った．そのうち，ビルドに成功し，かつ解を得ることができた Lang6, Lang22, Lang25 および Lang39 の 4 つのバグを対象として先ほど定義した STR と FTR を計算し，その値を確認する．なお，生成時間には不確定性があるため各バグごとに APR を複数回実行している．表 1 に APR ツール実行時の設定を，表 2 に実行回数や解に至るまでの総個体数など，各バグに対する実験の条件を示す． ここで，サンプル数がバグによって異なるのは，1 回のプログラム修正にかかる総時間が異なるためである．

5.2　実験結果

5.2.1　Lang6

図 7 に Lang6 における STR と FTR の箱ひげ図を，表 3 に STR と FTR の平均 (以降 AVG)，最小値 (以降 MIN)，第 1 四分位数（以降 1Q)，中央値 (以降 MED)，第 3 四分位数 (以降 3Q)，最大値 (以

表 1　APR ツールの設定

項目	値
実験題材	Defects4J Lang6, Lang22, Lang25, Lang39
題材数	4
1 世代ごとの変異個体数	70
1 世代ごとの交叉個体数	30
乱数シード	2
実験環境	Corei5-1240P 16GB mem

表 2　各バグの詳細

バグ名	Lang6	Lang22	Lang25	Lang39
到達世代数	1	7	1	4
個体数	8	624	38	300
ビルド失敗個体数	7	511	36	274
サンプル数	100	15	70	40

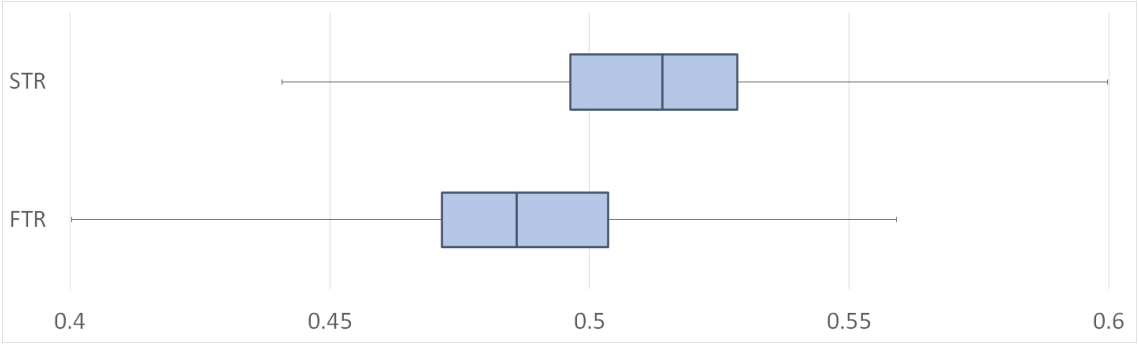

図 7　箱ひげ図：Lang6

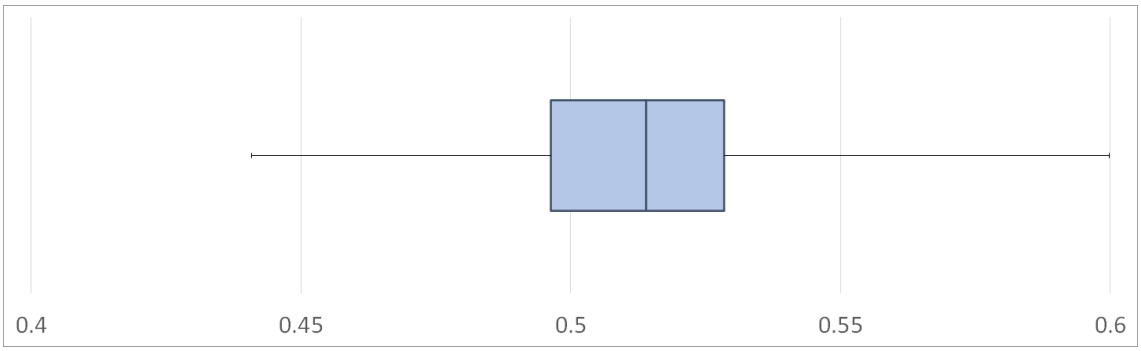

図 8　STR の箱ひげ図：Lang6

降 MAX) の各データを示す.

データからわかることとして，修正を完了するまでに生成した個体の数が 8 つと比較的少ないこともあってか，STR，FTR の値はいずれも 0.5 程度となった．この値は，解となる個体に関係する生成時間およびビルドに失敗した個体の生成時間が全体の生成時間のおよそ半分程度であることを意味する．また，このバグにおいては，他の 3 つのバグとは異なり，STR の値が FTR を上回るという結果を得た.

5.2.2　Lang22

次いで，図 10 に Lang22 における STR と FTR の箱ひげ図を示す．この結果からわかることとして，他のバグに比べてパッチの生成に多くの時間および操作を要した．そのため，他のバグに比べる

表 3　Lang6 バグの STR, FTR

評価指標	AVG	MIN	1Q	MED	3Q	MAX
STR	0.5117	0.4409	0.4964	0.5140	0.5284	0.5997
FTR	0.4883	0.4003	0.4716	0.4860	0.5036	0.5591

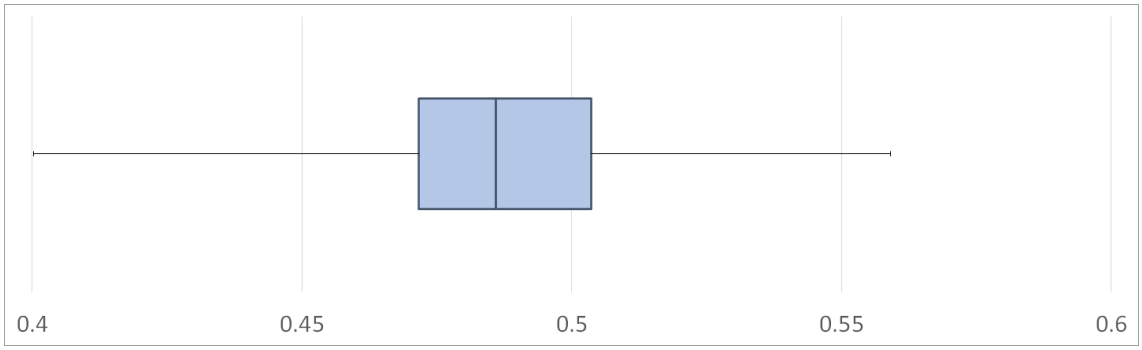

図 9　FTR の箱ひげ図：Lang6

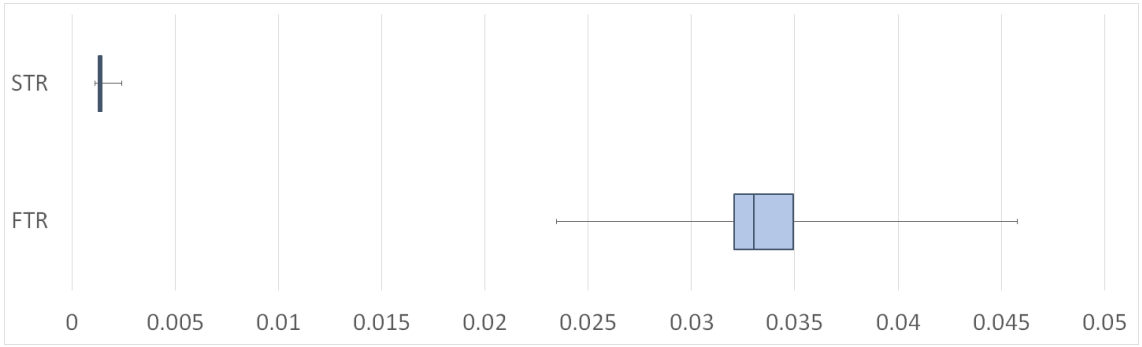

図 10　箱ひげ図：Lang22

と STR，FTR のいずれの値も小さくなっている．とりわけ FTR の値が 0.03 程度と非常に小さい値を得た．

5.2.3　Lang25

次に，図 13 に Lang25 における STR と FTR の箱ひげ図を示す．このバグにおける特徴として，FTR の値が平均して 0.8 程度と高い数値を示している点があげられる．一方で，STR の値は 0.1 程度にとどまった．

表 4　Lang22 バグの STR, FTR

評価指標	AVG	MIN	1Q	MED	3Q	MAX
STR	0.001476	0.001096	0.001291	0.001354	0.001458	0.002418
FTR	0.03432	0.02346	0.03208	0.03304	0.03492	0.04578

図 11　STR の箱ひげ図：Lang22

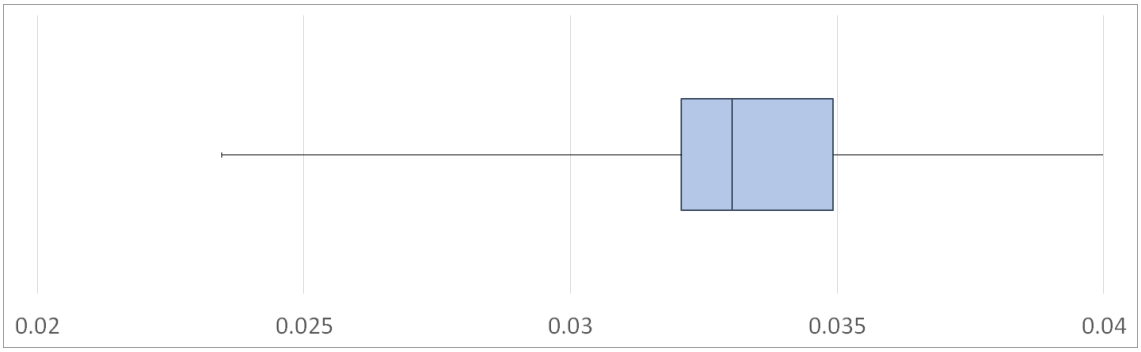

図 12　FTR の箱ひげ図：Lang22

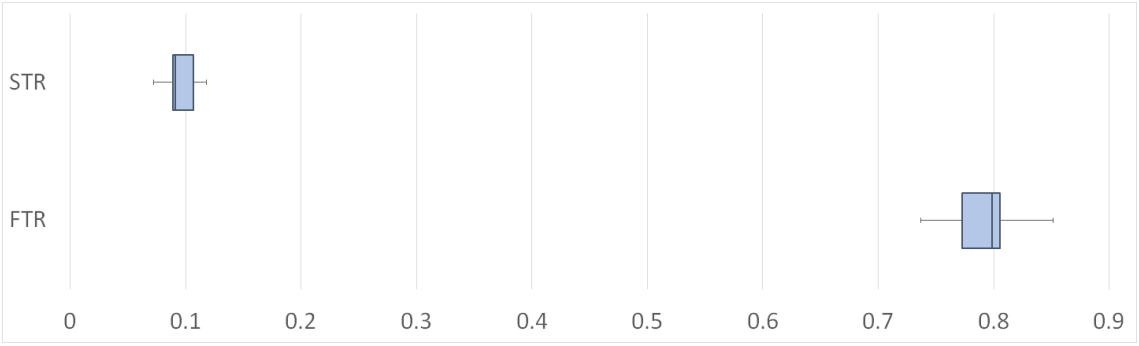

図 13　箱ひげ図：Lang25

5.2.4　Lang39

次に，図 16 に Lang39 における STR と FTR の箱ひげ図を示す．このバグにおいても，Lang25 ほどではないものの，FTR が 0.6 付近と高い値を得た．一方で，STR の値は 0.02 程度と FTR よりも著しく低い値となった．

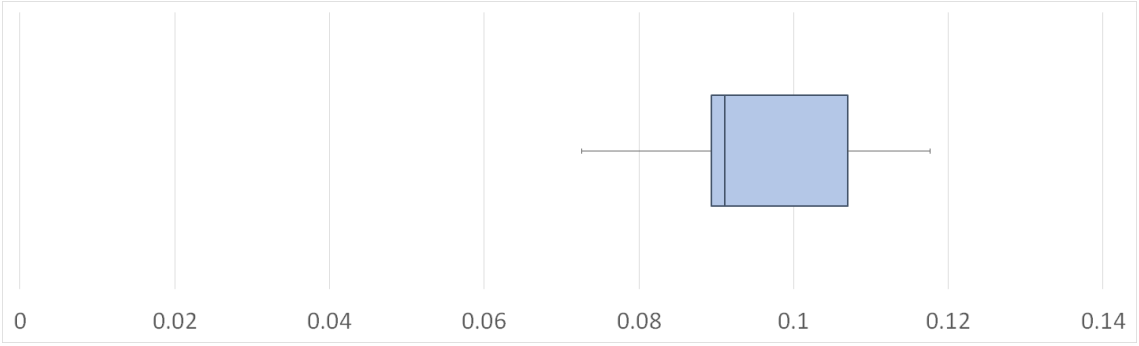

図 14　STR の箱ひげ図：Lang25

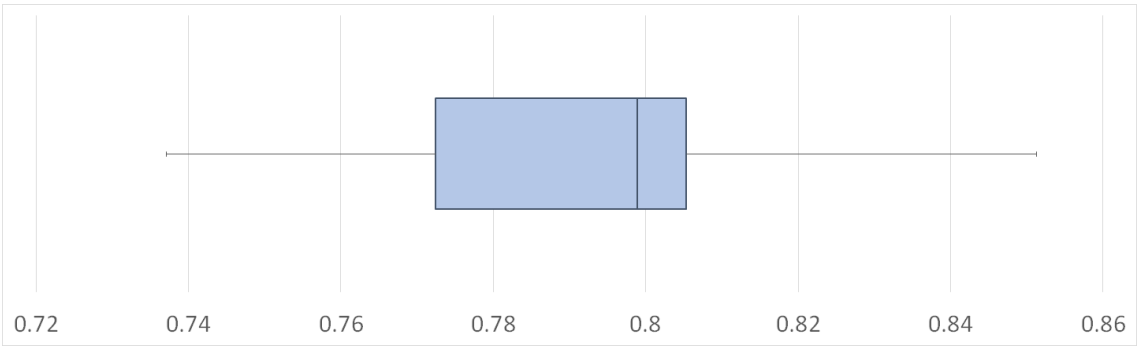

図 15　FTR の箱ひげ図：Lang25

5.2.5　実験結果の総括

最後に，今回実験の対象とした 4 つのバグについて，得られた結果を比較しながら論述する．まず，STR(図 19) について，先述したとおり Lang6 においては 0.5 を超える値を記録している．一方で，そ

表 5　Lang25 バグの STR, FTR

評価指標	AVG	MIN	1Q	MED	3Q	MAX
STR	0.09386	0.07261	0.08936	0.09106	0.1070	0.1177
FTR	0.7986	0.7371	0.7725	0.7989	0.8053	0.8512

表 6　Lang39 バグの STR, FTR

評価指標	AVG	MIN	1Q	MED	3Q	MAX
STR	0.02406	0.01645	0.02105	0.02326	0.02510	0.02873
FTR	0.6218	0.5847	0.6137	0.6224	0.6299	0.6644

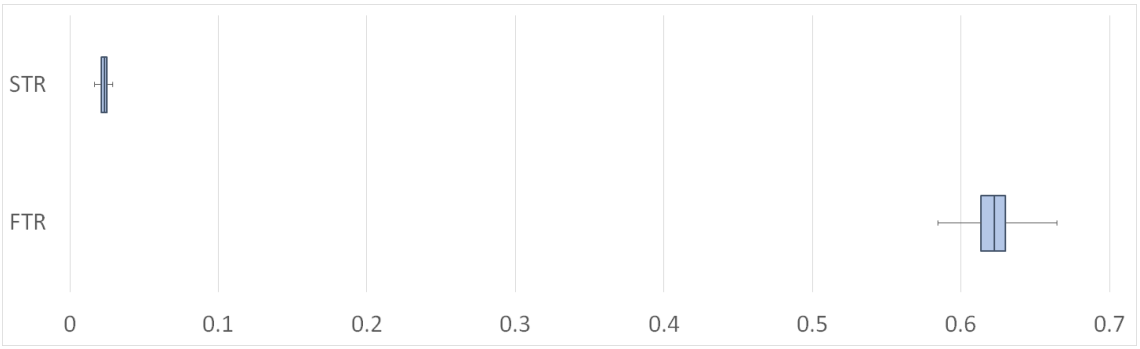

図 16　箱ひげ図：Lang39

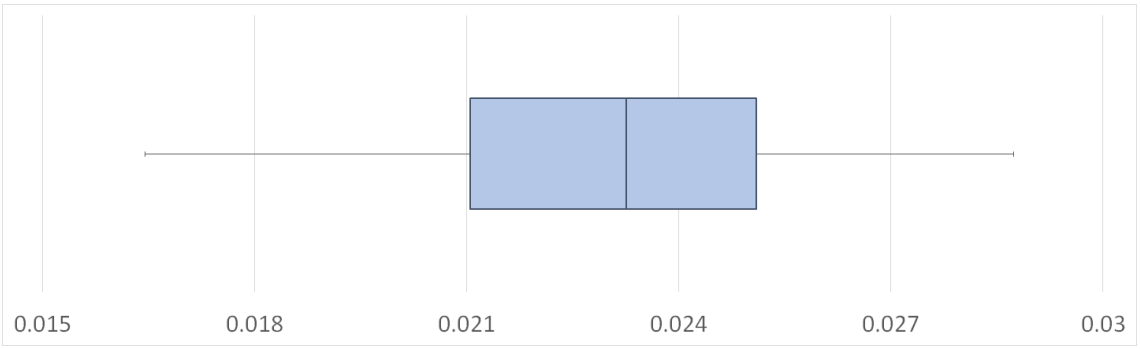

図 17　STR の箱ひげ図：Lang39

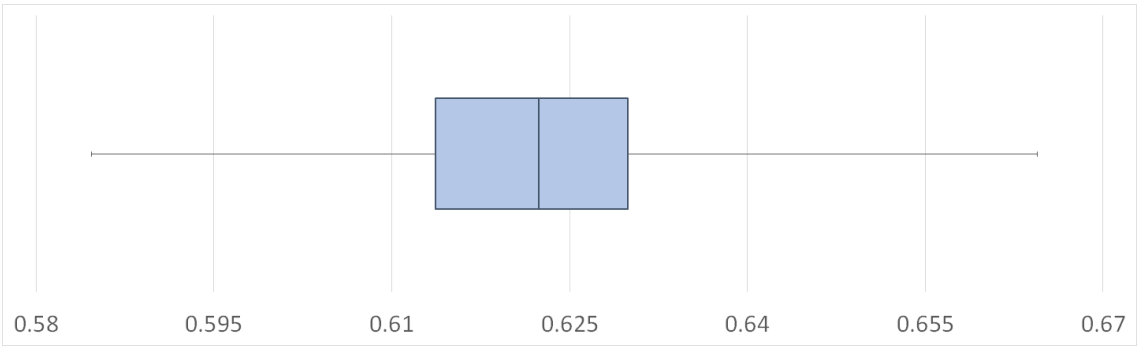

図 18　FTR の箱ひげ図：Lang39

の他のバグにおいてはいずれにおいても平均値が 0.1 を下回った．このことは，解に関係する個体の全体に占める生成時間が 1 割未満であることを意味する．次に，FTR(図 20) について，0.03 程度の値を得た Lang22 を除くと，5 割〜8 割程度の値に落ち着いた．特に Lang25 においてはビルドに失敗した個体の全体に占める生成時間が 8 割を超えることもあることが分かった．

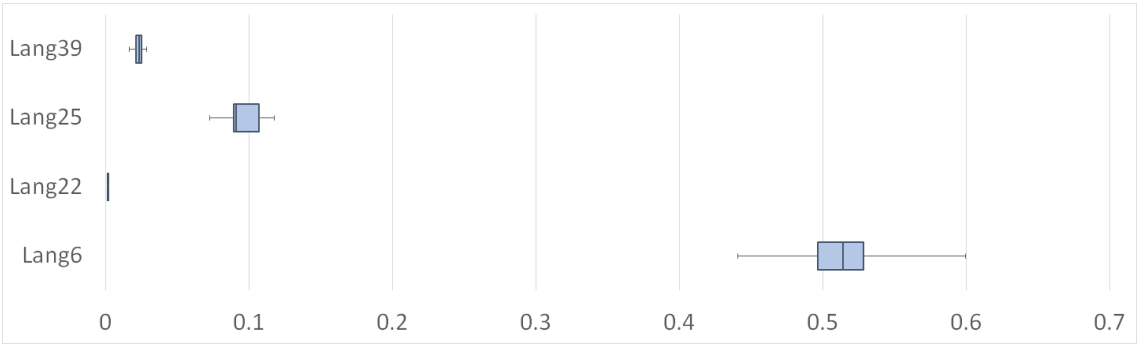

図 19　4 つのバグにおける STR の箱ひげ図

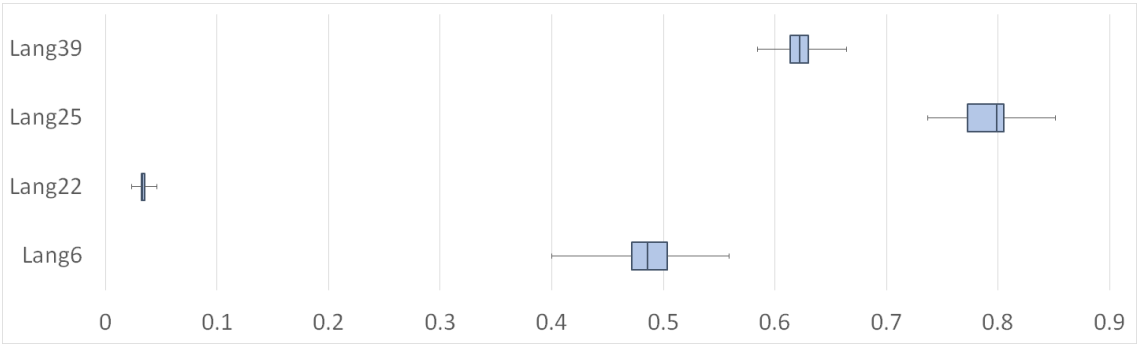

図 20　4 つのバグにおける FTR の箱ひげ図

6　考察

5 章では，1 世代あたりの変異・交叉個体数を表 1 のように設定したが，この値を変更することで STR および FTR の値に変化が見られるかを考察する．Lang39 バグに対して，1 世代あたりの個体数を変更した場合の STR と FTR の値の影響を調査し，その結果に対する考察を述べる．比較対象の設定として，表 7 に 5 章で行ったと比較対象となる実験の設定および生成時間に関係しない結果について相違点のある項目のみを示す．図 21 および図 22，表 8 および表 9 はそれぞれ比較実験における STR・FTR の値を表した箱ひげ図および有効数字 4 桁の値である．ここで，Lang39C は比較実験における素のデータを表すが，比較対象となった 30 回いずれの修正においても特定の 1 個体に費やした生成時間が他の生成時間に比べはるかに大きな値を記録した (いわゆる外れ値) ため，その個体の生成時間を除外したデータが Lang39C2 である．

なお，この個体は，ビルドには成功したが，解の個体の経路には含まれなかった．そのため，まず STR については，Lang39C と Lang39 が 0.02 前後の値をとったのに対し，Lang39C2 は 0.05 とほか 2 つより高い値を記録した．一方で FTR については Lang39C がほかの 2 つに比べ半分未満の値をとった．

このことから，個体の生成時間に制限を設けた場合に，制限時間をオーバーした個体の生成を中止するようにすれば，その個体が解となる個体の親たりえない場合は STR の値が向上するのではないかということが考えられる．

表 7　比較実験の設定・結果

項目	比較元	比較対象
1 世代ごとの変異個体数	70	5
1 世代ごとの交叉個体数	30	2
サンプル数	40	30
生成個体数	300	541
ビルド失敗個体数	274	461

表 8　Lang39 バグにおける比較実験の STR

評価指標	AVG	MIN	1Q	MED	3Q	MAX
Lang39C2	0.04953	0.04334	0.04671	0.04898	0.05187	0.06052
Lang39C	0.02045	0.01719	0.01958	0.02017	0.02167	0.02334
Lang39	0.02406	0.01645	0.02105	0.02326	0.02510	0.02873

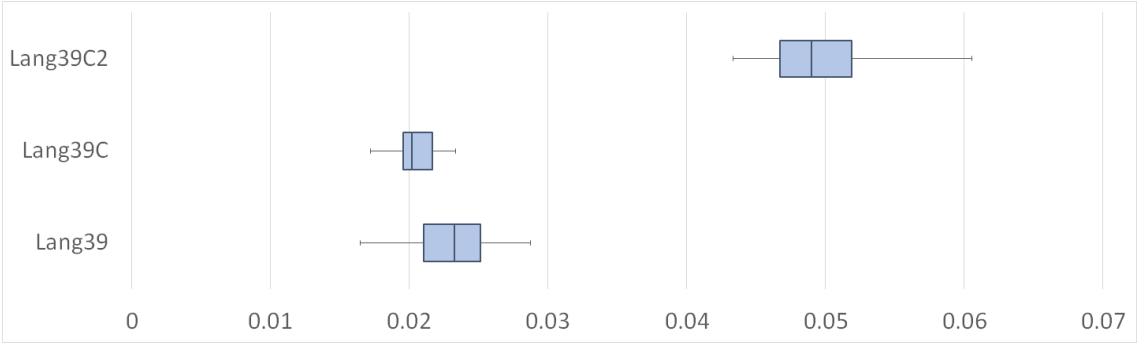

図 21　Lang39 バグにおける比較実験の STR 箱ひげ図

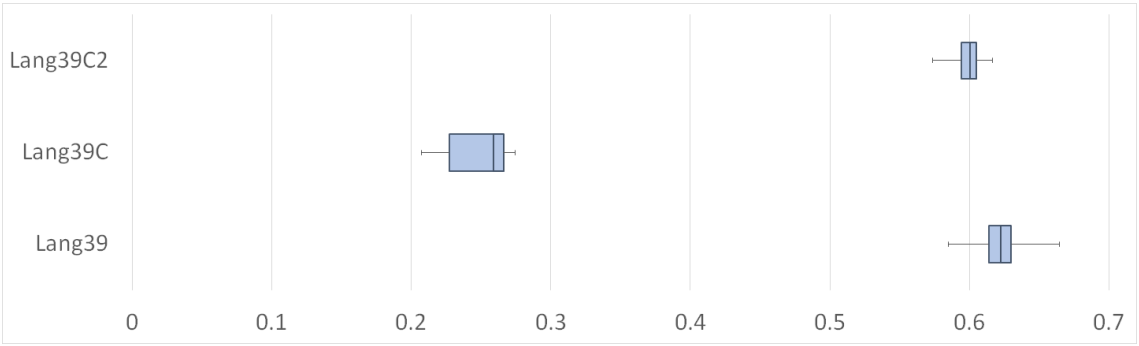

図 22　Lang39 バグにおける比較実験の FTR 箱ひげ図

表 9　Lang39 バグにおける比較実験の FTR

評価指標	AVG	MIN	1Q	MED	3Q	MAX
Lang39C2	0.5987	0.5735	0.5940	0.6006	0.6049	0.6163
Lang39C	0.2480	0.2070	0.2272	0.2587	0.2662	0.2746
Lang39	0.6218	0.5847	0.6137	0.6224	0.6299	0.6644

7　妥当性の脅威

5 章における実験において考えうる妥当性の脅威について論ずる．この実験では内的要因と外的要因に大別される．まず内的要因として，今回実行した環境とは異なる環境において実行した際に算出される STR と FTR の値が異なる可能性がある点があげられる．また，実行時に他のタスクによりメモリが占領されている場合，普段と異なる値が算出されうる点があげられる．

次に外的要因として，今回対象としたバグを含むプログラムは生成個体数が多いもので 600 程度と比較的少なかったが，大規模なプログラムにおいて何万といった個体が生成されたときに STR と FTR の値が大きく異なる可能性がある点があげられる．

8　今後の課題

本研究においては，探索ベースで遺伝的アルゴリズムを採用した kGenProg を対象に個体の生成時間計測を行ったが，将来的な研究においては以下のような課題について調査する．

8.1　APR ツールによる生成時間の違いの検証

本研究で用いた APR ツールは探索ベースで遺伝的アルゴリズムを採用した kGenProg を対象に個体の生成時間計測を行ったが，遺伝的アルゴリズムでないほかの探索ベースの APR ツールや意味論ベースの APR ツールにおいて時間計測を行う．

8.2　時間計測を行う対象プロジェクトの拡張

本研究では，Defects4J の Lang バグのうち，4 つのバグを対象として STR と FTR の計算を行った．今後は，Defects4J のほかのバグやその他の対象について時間研究を行う．

9　おわりに

本稿では，APR ツールに対して個体の生成時間を計測する機能を追加し，実際のバグに対して STR と FTR という 2 つの指標を定義した．さらに実際のバグに対して指標を計算する実験を行うことで時間的コストの評価を行った．結果として，多くのバグにおいて解の個体生成にかかる時間は，ビルドに失敗した個体の生成にかかる時間に比べて はるかに小さいことが示された．しかし，Lang6 バグのように，例外的な値をとるプログラムも存在する．そこで，将来的な研究ではビルドに失敗した個体に対する解の個体の時間的生成効率の改善を行う，より多くのバグに対する提案指標の検証を行う．

謝辞

本研究を進めるにあたり，多くの方々からご支援およびご助言を賜りました．

楠本 真二 教授には，本研究を快く快諾し，暖かく見守ってくださりました．心より感謝申し上げます．

肥後 芳樹 教授には，議論を重ねに重ね，本研究の完成のご支援及び的確なご助言を賜りました．心より感謝申し上げます．

柗本 真佑 助教には，テーマが決まらず途方に暮れていた際，鋭くも的確なご助言を賜りました．深く感謝いたします．

古藤 寛大先輩には，困難に直面した際いつも迅速にご助言を賜り感謝してもしきれません．

本研究に至るまでに，講義，演習等でお世話になりました大阪大学基礎工学部情報科学科の諸先生方に，御礼申し上げます．

最後に，これまでお世話になりました家族，小中高校の教員方，その他すべての方に感謝申し上げます．

参考文献

[1] B. Hailpern and P. Santhanam. Software debugging, testing, and verification. *IBM Systems Journal*, Vol. 41, No. 1, pp. 4–12, 2002.

[2] Tom Britton, Lisa Jeng, Graham Carver, and Paul Cheak. Quantify the time and cost saved using reversible debuggers. Technical report, Cambridge Judge Business School, 2012.

[3] Luca Gazzola, Daniela Micucci, and Leonardo Mariani. Automatic software repair: A survey. In *Proceedings of the 40th International Conference on Software Engineering*, pp. 1219–1219, 2018.

[4] Claire Le Goues, Michael Pradel, and Abhik Roychoudhury. Automated program repair. *Communications of the ACM*, Vol. 62, No. 12, pp. 56–65, 2019.

[5] Xuan-Bach D Le, Ferdian Thung, David Lo, and Claire Le Goues. Overfitting in semantics-based automated program repair. In *Proceedings of the 40th International Conference on Software Engineering*, pp. 163–163, 2018.

[6] Matias Martinez and Martin Monperrus. Astor: Exploring the design space of generate-and-validate program repair beyond genprog. *Journal of Systems and Software*, Vol. 151, pp. 65–80, 2019.

[7] Derrick Lin, James Koppel, Angela Chen, and Armando Solar-Lezama. Quixbugs: A multi-lingual program repair benchmark set based on the quixey challenge. In *Proceedings Companion of the 2017 ACM SIGPLAN international conference on systems, programming, languages, and applications: software for humanity*, pp. 55–56, 2017.

[8] 古藤寛大, 肥後芳樹, 柗本真佑, 楠本真二. 変更コード片の動的切替による自動プログラム修正のビルド時間削減の試み. 電子情報通信学会技術研究報告, Vol. 120, No. 407, pp. 19–24, 2 2021.

[9] Yuhua Qi, Xiaoguang Mao, and Yan Lei. Efficient automated program repair through fault-recorded testing prioritization. In *2013 IEEE International Conference on Software Maintenance*, pp. 180–189. IEEE, 2013.

[10] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. GenProg: A Generic Method for Automatic Software Repair. *IEEE Transactions on Software Engineering*, Vol. 38, No. 1, pp. 54–72, 2012.

[11] Claire Le Goues, Stephanie Forrest, and Westley Weimer. Current challenges in automatic software repair. *Software quality journal*, Vol. 21, pp. 421–443, 2013.

[12] Edward K Smith, Earl T Barr, Claire Le Goues, and Yuriy Brun. Is the cure worse than

the disease? overfitting in automated program repair. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pp. 532–543, 2015.

[13] Seemanta Saha, Ripon K. Saha, and Mukal R. Prasad. Harnessing Evolution for Multi-Hunk Program Repair. In *Proc. International Conference on Software Engineering*, pp. 13–24, 2019.

[14] Liushan Chen, Yu Pei, and Carlo A Furia. Contract-based program repair without the contracts. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 637–647. IEEE, 2017.

[15] Ali Ghanbari, Samuel Benton, and Lingming Zhang. Practical program repair via bytecode mutation. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pp. 19–30, 2019.

[16] Y. Higo, S. Matsumoto, R. Arima, A. Tanikado, K. Naitou, J. Matsumoto, Y. Tomida, and S. Kusumoto. kGenProg: A High-Performance, High-Extensibility and High-Portability APR System. In *Proc. Asia-Pacific Software Engineering Conference*, pp. 697–698, 2018.

[17] Yuya Tomida, Yoshiki Higo, Shinsuke Matsumoto, and Shinji Kusumoto. Visualizing code genealogy: How code is evolutionarily fixed in program repair? In *2019 working conference on software visualization (VISSOFT)*, pp. 23–27. IEEE, 2019.

[18] René Just, Darioush Jalali, and Michael D Ernst. Defects4j: A database of existing faults to enable controlled testing studies for java programs. In *Proceedings of the 2014 international symposium on software testing and analysis*, pp. 437–440, 2014.