特別研究報告

題目

Git リポジトリを用いた自動プログラム進化の過程共有

指導教員

楠本 真二 教授

報告者

出田　涼子

2020 年 2 月 10 日

大阪大学 基礎工学部 情報科学科

2019 年度　特別研究報告

Git リポジトリを用いた自動プログラム進化の過程共有

出田　涼子

内容梗概

自動的なプログラミングの実現を目的として，探索ベースの自動プログラム進化に関する研究が盛んに行われている．自動プログラム進化では，ソースコードへの改変と評価を繰り返し，ソースコードを目的の状態に近づけていく．プログラム進化技術の改善のためには，生成された大量のソースコードを効率的に記録し，共有する仕組みが重要となる．

そこで本研究では，プログラム進化の全過程を Git リポジトリとして記録することを考える．一般的に Git リポジトリは開発者によるソースコードの進化過程の記録に用いられるが，この版管理利用の主体をモノである自動プログラム進化に置き換えるというアイデアである．Git リポジトリ化により，プログラム進化の過程を効率的かつ情報の漏れがない完全な形で記録可能となる．さらに，研究者間での進化過程の共有や再現実験が容易となるほか，既存の Git 支援ツールを適用して理解が容易になる可能性もある．

本アイデアの実現可能性を確かめるために，プロトタイプシステムの実装を行い，我が研究室で開発している自動プログラム修正ツール kGenProg への適用を試みた．また，ケーススタディを考え，この手法が有効な場面があることを確認した．

主な用語

自動プログラム進化，自動プログラム修正，進化過程，版管理，Git

目次

1 はじめに 1

2 準備 3

2.1 自動プログラム進化 3

2.2 自動プログラム進化の課題 4

3 版管理を用いたプログラム進化過程の記録 5

4 Git の適用の方針 7

4.1 概要 7

4.2 進化の実行概要 7

4.3 進化の過程 8

4.3.1 個体の生成方法 9

4.3.2 個体の生成結果 10

4.3.3 個体の評価結果 10

4.3.4 個体間の関係 10

5 kGenProg への適用 12

5.1 初期設定 12

5.2 個体情報 12

5.3 最終結果 13

6 速度評価実験 14

6.1 実験概要 14

6.2 実験結果 14

6.3 速度実験の考察 15

7 ケーススタディ 16

7.1 概要 16

7.2 プログラム修正の全体的な結果の把握 16

7.3 改変操作ごとの効果の確認 17

7.4 実験結果の共有 18

8　　おわりに　19

謝辞　20

参考文献　21

図目次

1 プログラム進化の概要 3
2 版管理利用指針の全体像 8
3 kGenProg への適用 12
4 速度評価実験の結果 15
5 プログラム修正全体の実行概要 16
6 解の確認 16
7 解の差分 17
8 解を得るまでの過程 17
9 各状態の個体の個数 17
10 改変操作ごとの重複個体の個数 18
11 GitHub への生成リポジトリの公開 18

表目次

1 生成方法の種類 9
2 実験対象の題材 14

1　はじめに

プログラミングの自動化を目的として，探索ベースの自動プログラム進化に関する研究が盛んに行われている [1, 2]．この手法では，元となるソースコードへの小規模な改変を繰り返し，テスト実行結果やメトリクスなどの情報に基づいてソースコードを目的の状態へ近づけていく．効率的な探索のために，生物進化を模倣したソースコード改変手法である遺伝的プログラミング [3] も頻繁に取り入れられている．近年，多くの研究者が取り組んでいる自動プログラム修正 [4, 5, 6] はプログラム進化の応用例の 1 つであり，バグのない，すなわち全てのテストを通過するソースコードを目的状態とする．他にも，プログラム進化は自動リファクタリング [7, 8] や自動パフォーマンス改善 [9] といった目的への応用が可能である．

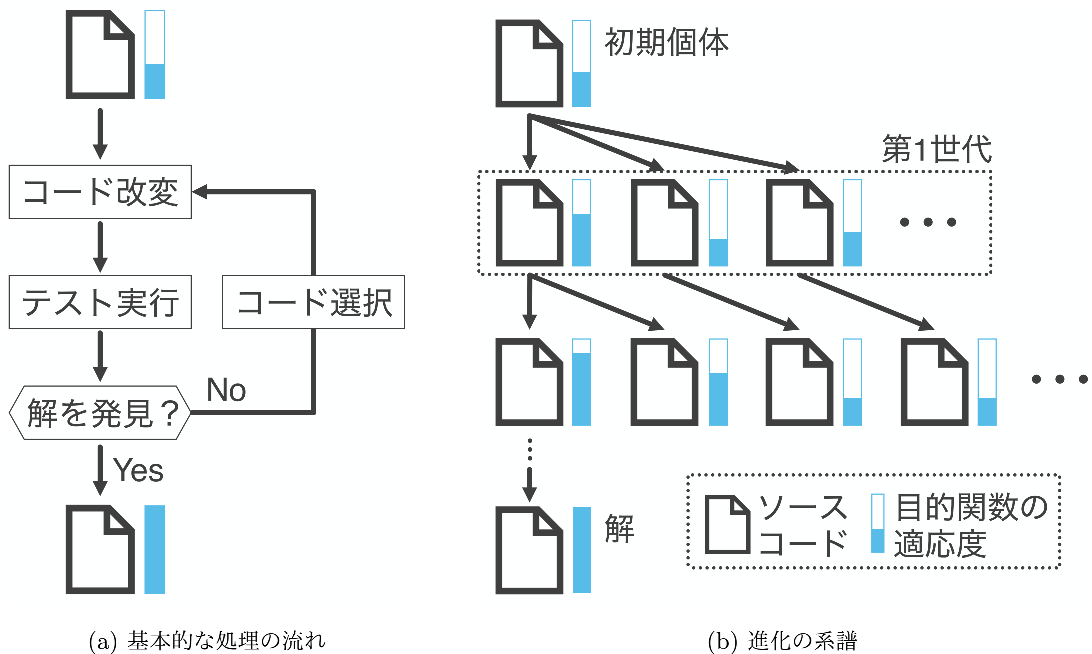
自動プログラム進化における手法改善や分野発展のためには，解となるソースコードの出力だけでなく，その進化の過程で生成された多数のソースコード（個体と呼ぶ）の記録と分析が不可欠である．例えば，新たなソースコード改変手法を提案し評価する場合を考える．その評価では適当な題材を用いて改変手法を実験し，生成された個体が想定通りのソースコードになっているか，実装がプログラム進化に対して良い効果を与えているかといった点を確認する必要がある．これには，進化の過程で得られた生成個体それぞれを記録し分析する必要がある．また，探索ベースのプログラム進化は一種の最適化問題であり，一般的に解の発見には多大な計算能力と時間を要する [10]．そのため，実験の一試行で得られた進化過程をデータとして保持できれば，実験実施後の詳細な分析が容易となる．さらに，この進化過程のデータを研究者間で共有できれば，再実験や追試に要する時間的及び計算能力的なコストを大幅に削減できる．

しかしながら，多くのプログラム進化ツールでは進化過程を保持する機能がサポートされていない．ツール実行により得られる情報は，発見した解や生成個体の総数，到達世代数，起動時間といったプログラム進化の最終結果に限られている．一例として，GenProg [4] の Java 実装である jGenProg [11] では，過程を記録するオプションが実装されているものの，記録できる情報は個体ごとの改変ソースコードがファイルとして出力されるのみである．従って，その個体がどのような手法で生成されたか，その個体で目的関数の値がどの程度改善もしくは悪化したか，どの個体が親でどのような進化の系譜を辿ったか，といった進化過程の細部を把握できない．

本研究では，プログラム進化における進化過程の保持と共有を目的として，プログラム進化の全過程を Git リポジトリとして記録することを考える．一般的に Git リポジトリは，開発者によるソースコードの進化過程の記録に用いられるが，この版管理利用の主体をモノである自動プログラム進化に置き換えるというアイデアである．Git の適用により，プログラム進化の過程を効率的かつ情報の漏れがない完全な形で記録可能となる．さらに，研究者間での進化過程の共有や再現実験が容易となるほか，既存

の Git 支援ツールをプログラム進化の分析に適用できる可能性もある．本アイデアの実現可能性を確かめるために，我が研究室で開発している自動プログラム修正ツール，kGenProg [12] への適用を試みた．

以降，2 章では準備として，本研究の提案手法に必要な事柄について説明する．3 章では提案手法の概要について説明し，4 章ではその提案手法をどのように適用するかを具体的に述べる．5 章では実装したプロトタイプシステムの内容を説明する．6 章ではこのプロトタイプシステムを用いた簡単な速度実験の結果を，7 章ではそのプロトタイプシステムを例にケーススタディを示す．最後に，8 章で本研究のまとめと今後の課題について述べる．



(a) 基本的な処理の流れ

(b) 進化の系譜

図 1: プログラム進化の概要

2　準備

2.1　自動プログラム進化

自動プログラム進化とは，与えられたソースコードへの改変と評価を繰り返し，自動的に目的の状態のソースコードを得る技術である．プログラム進化の応用の 1 つとして，テスト通過数の最大化を目的関数とする自動プログラム修正 [4, 5, 6] が広く知られている．この自動プログラム修正はソフトウェア開発において多大な労力を要するデバッグを支援する技術である．プログラム進化は他にも様々な目的に応用可能である．例えば，全テストの通過を前提としてソースコードの内的品質の最大化を目的関数とすれば，自動リファクタリング [7, 8] となり，プログラムの実行速度の最小化を目的関数とすれば，自動パフォーマンス改善 [9] となる．

プログラム進化の概要を図 1 に示す．図 1 (a) はその基本的な処理の流れを表す．まず，プログラム進化では入力されたソースコードに対して小規模な改変を加える．以降ではこの改変を加えられたソースコードを，生物進化のメタファになぞらえて個体と呼ぶ．次に，生成個体に対してテストを実行する．その後，テスト実行結果やメトリクスなどに基づいて目的関数を当てはめ，生成個体が目的の状態を満たしていれば，それを解とする．満たしていなければさらに個体選別を行い，ソースコード改変と評価を繰り返す．

上記の流れを繰り返した結果，プログラム進化ではソースコードの進化の系譜が得られる．進化の系譜の一例を図 1 (b) に示す．最上段は初期個体，すなわち入力として与えられたソースコード群を意味する．この初期個体を直接の親とする個体は，第 1 世代に相当する．図では 3 つの個体が第 1 世代として生成されている．プログラム進化は目的の状態に達するまで，この個体の生成を繰り返す．よって，一般的に解の発見に至るまでには膨大な数の個体が生成される [13].

2.2 自動プログラム進化の課題

プログラム進化技術の発展のためには，進化の過程で得られる様々な情報を効率的に保持し，共有する必要がある．しかし，現在公開されているプログラム進化ツールの多くは，進化の最終結果を出力するのみであり，その過程を把握できない．一般にプログラム進化の過程では膨大な数の個体が生成される．よって，その過程を単にコンソールやファイルなどに出力するだけでは，進化過程の把握は困難である．また，プログラム進化手法の効果が題材に強く依存する点，及び乱択[*1]に依存する部分が多い点の 2 つの理由から，複数の乱数と題材の組み合わせで評価実験が行われる．加えて，一般的にプログラム進化の一試行には多大な時間を要し，その再実験や追試には時間的，計算量的な困難さが伴う．

[*1] 生物進化のように意図的に外乱要素を加えて局所解を避けること．

3　版管理を用いたプログラム進化過程の記録

本研究の目的は，プログラム進化に対してその進化の過程を効率的に記録し共有することである．この目的を達成するために，プログラム進化過程の保持に版管理を適用することを提案する．一般に，版管理はヒト（開発者）によるプログラム開発に用いられるが，提案手法では版管理をモノによるプログラム進化に適用する．言い換えれば，提案手法では版管理上でのソースコード改変の主体を，ヒトからモノに置き換える．ある題材に対する実験の単一試行を単一の版管理リポジトリとして記録するため，複数の生成リポジトリを比較することで，複数の実験が比較可能となる．

進化過程の記録への版管理の適用により，以下の 4 つの効果が期待できる．なお以降では，版管理技術の 1 つとして広く利用されている Git を前提として説明する．

実験試行の完全な記録： プログラム進化はソースコードファイルを中心として改変と評価を繰り返すため，Git の適用によりファイル改変の系列，すなわち進化の過程を完全な形で保持できる．言い換えると，ソースコードや目的関数の値などのプログラム進化の途中で生成される全ての情報が漏れなく記録できる．さらに，このファイル改変の系列に加え，記録対象となる試行の初期情報（ツール名やツールのバージョン，最大世代数などの実行パラメタ），及び最終結果（到達世代数や実行時間，diff パッチなど）を記録すれば，どのようにプログラム進化ツールを実行し，どのような結果が得られたかという情報も保持可能となる．プログラム進化の試行に関するあらゆる情報を記録し共有できれば，異なる実行パラメタや複数のプログラム進化ツールによる結果を比較できる．1 つの試行には数時間以上を要する場合が多いため [13]，記録と共有を可能にすることで他の開発者による改善や，他研究者による追試を容易にする．

効率的なデータサイズ： Git では改変ファイルのみを保持する仕組みが導入されており，改変部分の効率的な記録が可能である．2.1 節でも述べた通り，一般的なプログラム進化では大量の個体を生成する．一方で，各個体の変更差分は 1 行から数行程度と小さい．さらに，大部分のソースコードには改変が適用されない．よって，大量の個体を生成しても改変部分はごく一部であるため，Git の適用により，全てをファイルとして出力するよりも少ないデータサイズでの記録が可能となる．

ユニバーサルなデータ書式： Git は多くのソフトウェア開発プロジェクトで採用されている [14, 15]．また，Defects4J [16] を始めとする自動プログラム修正のデータセットも Git リポジトリとして提供されている．つまり，Git はソフトウェア開発に関わる開発者，及び研究者での一種の共通言語であるといえ，導入が容易である．また，生成したリポジトリの共有も容易にできる．なぜなら，Git リポジトリを直接やりとりするだけでなく，数多く存在している GitHub のようなホスティングサイトも利用できるからである．

既存の Git ツール適用の可能性： 様々な Git の支援ツールが公開されている．例えば，Git ブランチトポロジを可視化するツールとして GitKraken[*2]や GitUp[*3]がある．また，Git リポジトリの分析ツールとしては Gitinspector[*4]や PyDriller [17] が知られている．プログラム進化への Git の適用により，これらの Git 支援ツールをプログラム進化の分析に適用できる可能性がある．可視化ツールは進化過程の全体像の把握に利用でき，Git 分析ツールを用いれば，どのソースコード改変手法がどの程度進化に寄与しているかの把握に活用できると考えられる．例えば，Gitinspector では作者ごとのコミットの数，挿入行数，変更の割合といった様々な統計情報を得られるため，作者とソースコード改変手法を対応づければ，この統計情報を手法ごとに得られる．

[*2] `https://www.gitkraken.com/git-client`
[*3] `https://gitup.co`
[*4] `https://github.com/ejwa/gitinspector`

4　Git の適用の方針

4.1　概要

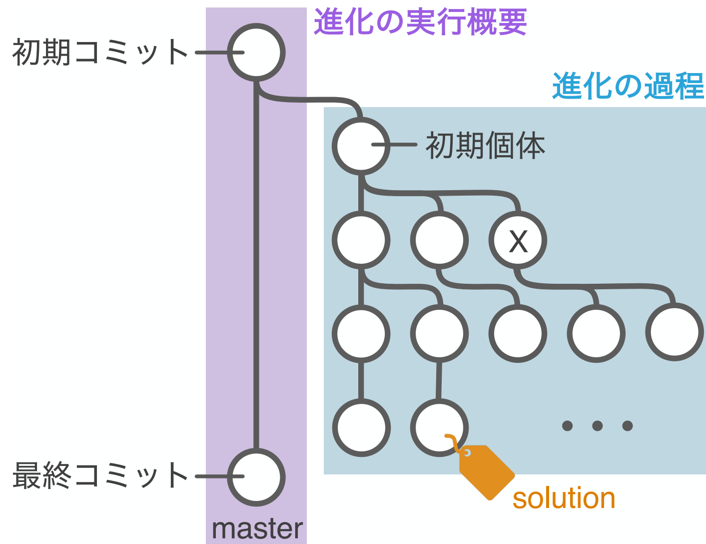
本章では，提案手法におけるプログラム進化への版管理適用の具体的な方針について述べる．図 2 に方針の全体像を示す．図 2 (a) が生成リポジトリのブランチトポロジを，図 2 (b) が単一コミットの詳細を表す．まず図 2 (a) について説明する．ブランチトポロジは 2 種類のブランチ系列から構成されており，一方が進化の実行概要を，他方が進化の過程を保持する．さらに，進化の過程については各個体が持つ詳細な進化情報と Git 情報の対応付けを設計する必要がある．図 2 (b) がその対応付けを表す．Git の単一コミットには，プログラム進化における生成方法，評価結果，生成結果を保持させる．また図中では記載されていないが，個体間の親子関係はコミットの持つ `Parent` 属性に対応付けさせる．以降では，各方針の詳細について説明する．

4.2　進化の実行概要

記録対象となる試行の全体像として進化の実行概要，すなわち初期設定と最終結果の記録を考える．初期設定は，プログラム進化のツール名やバージョン，設定した実行パラメタなど，対象試行の実行方法を唯一に特定できる情報から構成される．また最終結果とは，生成されたソースコードの diff パッチだけでなく，最終的に生成した個体総数や実行時間といった試行に対するメタ情報を含む．これらの情報は，進化過程の大筋に着目したプログラム進化の入出力と見なせ，試行の全体像の把握に活用できる．

この全体像の記録には master ブランチを用いる．master ブランチには進化過程に関する情報は記録せず，プログラム進化の実行概要のみを記録する．具体的には，図 2 (a) における左上の初期コミットに初期設定情報のみを記録し，左下の最終コミットにて最終結果を追記する．ブランチの命名が自由に行える中で master ブランチにこの実行概要を記録する理由は，master が最新かつ主体のブランチという Git の文化に従うためである．例えば，GitHub では master ブランチが初期ページとして表示されるため，ここにプログラム進化の実行概要が保持されることは直感的かつ自然であるといえる．

ブランチを実行概要（初期設定と最終結果）と進化過程の 2 つに分割する理由は，情報の性質が大きく異なるためである．実行概要は対象試行の一種のメタ情報である一方で，進化過程は試行の細部に該当する情報である．この 2 種類の情報の混同を避けるために，対象試行の中で唯一の情報である初期設定と最終結果を，それぞれ初期コミットと最終 master コミットへ対応付け，過程は別ブランチとして記録する．



(a) Git のブランチトポロジ

個体 X

コミット者

コミットメッセージ

ファイル差分

生成方法

評価結果

生成結果

```
$ git show X
Author: replace
    build=OK
    fitness=0.86
    failed-test=jp.kusulab...
diff --git a/x.java b/x.java
@@ -19,7 +19,6 @@

- for(i=0; i<length; i++) {
+ for(i=0; i<length-1; i++) {
```

(b) Git の単一コミットの詳細

図 2: 版管理利用指針の全体像

4.3 進化の過程

次に，進化過程と Git の対応付けを検討する．基本的な方針としては，単一の個体を単一のコミットで表現する．図 2 (a) の右側に示すように，まず，プログラム進化への入力となるソースコード群を，初期コミットの直後に master とは別のブランチに記録する．以降の生成個体はこの初期個体を親とし，個体に関する情報をコミットメッセージやファイルに記録する．これにより，個体の進化の系列を版管

理におけるコミットの系列として表せる．さらに，個体間の差分を確認することで，その個体がどのように生成され，どのような結果となり，どのような効果を得たかといった詳細な情報の把握ができる．以降では，個体に関する生成方法，生成結果，評価結果の 3 つをどのようにコミットに対応付けするかを説明する．

4.3.1 個体の生成方法

個体の生成方法とは，各個体がどのような手法に基づいてソースコード改変を加えられたかである．具体的なソースコード改変手法としては，ステートメント単位での単純な削除 [18] や，他のステートメントからの再利用 [19]，過去の開発者による変更を参考にする方法 [20]，条件分岐文の条件を変更する方法 [21] などがあり，様々な手法が存在する．jGenProg [11] や kGenProg [12] では，上記の処理をランダムで選択して適用する．また，複数の親から単一の個体を生成する交叉という手法 [22] もある．本研究で登場する生成方法とその説明を表 1 に示す．

この個体の生成方法は，コミットのメタ情報である `Author` 属性に書き込む．すなわち誰がソースコードを改変したか，という情報に個体の生成方法を対応付ける．Git では `Author` 属性によるコミットのフィルタリングも可能であり，各生成方法で生み出された個体の選別も可能となる．Git の可視化ツールによっては `Author` 属性に応じてコミットを色分けするケースも多く，個体の生成方法とその評価結果を効果的に可視化できる可能性がある．

さらに，上記の生成方法に関する情報を，コミットメッセージ，及びリポジトリ内に設置する自動生成ファイル（例えば，`commit-summary.txt`）の両方に書き加える．同一情報を複数のリソースに記録することになるが，この方針により，コミットメッセージを用いたコミット検索，及びファイル間 diff による個体間差分の確認が可能となり，進化過程の分析目的に応じて使い分けが可能となる．

表 1: 生成方法の種類

操作名（日本語）	操作名（英語）	説明
削除	delete	ステートメント単位で単純に削除する
挿入	insert	他のステートメントを注目箇所に挿入する
置き換え	replace	他のステートメントで注目箇所を置き換える
コピー	copy	前の世代の個体を改変せずにコピーする
交叉	crossover	2 つの個体を混ぜ合わせる

4.3.2　個体の生成結果

個体の生成結果とは，ソースコード改変により生成されたソースコードの集合である．これは一般的な版管理と同様にリポジトリ内のファイル（blob ファイル）として記録する．パッケージ構造などのフォルダ構成も初期個体と同様に維持することで，どのファイルが書き換わったかの確認を一般的な Git 利用と同様の操作で実現できる．例えば，`"git diff 個体 X 初期個体 -- src"`というコマンドを用いれば，`個体 X` のソースコードが初期個体からどのように変化したかを把握できる．

4.3.3　個体の評価結果

個体の評価結果とは，生成された個体に対する目的関数の計算結果である．自動プログラム修正の場合は，テストの通過率が評価結果に相当する．また，具体的にどのテストが失敗していたか，失敗したテストアサーションが何だったか，といったテストの詳細な情報も評価結果の一種である．一般的な版管理では，このテスト実行結果のような自動生成可能なデータはリポジトリ内に保持しない場合が多い．一方プログラム進化では，個体ごとの評価結果は進化理解という目的に対して重要な情報源であるため，リポジトリ内に記録する．この評価結果は，4.3.1 項の生成方法と同様に，コミットメッセージとリポジトリ内の生成ファイル（`commit-summary.txt`）の両方に記載する．

なお，生成された個体は必ずしもビルド可能とは限らない．ソースコード改変の結果，ソースコードの構文的正しさが常に保証されるとは限らないためである．さらに，進化の中では既に生成された個体と同一の個体を生成してしまう場合もある．この重複個体は探索済み空間の再探索を引き起こすため，進化の系列からは除外されることが一般的である．これらビルド成否や重複といった情報も，ソースコード改変の結果の 1 つであり，個体の評価結果として記録する．

4.3.4　個体間の関係

4.3.1 項から 4.3.3 項で個体単体に関する情報の記録方針を述べた．次に，個体間の関係を考える．ここでの個体間の関係とは，ある個体がどの個体から生成されたかを表す親子関係や，ある個体が生成された世代数，さらに，多数の個体の中からある個体を特定する一意な識別子といった情報を含む．

まず，個体の親子関係は，コミットの `Parent` 属性を用いて管理する．つまり，ある個体の `Parent` 属性の示す個体が，プログラム進化における生成元を意味する．これにより，図 2 (a) の右側に示すように，Git のブランチトポロジを生物進化における家系図のように表せる．

なお，交叉によって生成された個体は親を 2 つ持つ．そのため，`Parent` 属性と親子関係を一致させるためにブランチのマージを行う．親 A が最新コミットであるブランチに親 B が最新コミットであるブランチをマージし，そのマージコミットに生成された個体の情報を記載する．

次に，個体を一意に特定する識別子を考える．Git ではコミットのハッシュをコミットの識別子として用いるが，これは Git リポジトリの内部識別子としてのみ扱い，Git 操作者は利用しない．ハッシュはランダムな文字列であり，どのコミットと対応しているか把握するのが困難だからである．プログラム進化の理解を助けるために，ハッシュとは別に，自然数の連番による識別子を付与する．さらに，この識別子には世代数を併記する．例えば，識別子`"#2.9"`は第 2 世代目の 9 番目に生成された個体であることを意味する．この識別子はコミットメッセージとファイルだけでなく，Git のタグにも併記する．タグは Git においてコミットを識別する一意な情報として利用されるため，プログラム進化における個体の識別子と親和性が高い．一般的な開発では，Git のタグは公開したバージョンなどの一部のコミットにだけ付与するが，本研究では全ての個体に識別子を記したタグを付与する．これらの方針により，例えば`"git show --tags=#2.*"`というコマンドを用いれば，第 2 世代目の全個体を列挙し分析できる．

プログラム進化において，解と初期個体は特に頻繁に参照する．そのため，Git のタグを用いて特定しやすくする．解には `solution` タグを，初期個体には `initial` タグを付与する．Git のタグは 1 つのコミットに対して複数付与することができるため，この方針は識別子のタグと共存する．これによりそれぞれを命名した状態と見なせ，`git-diff` コマンドなどを用いる時に個体の識別子を調べることなく参照できる．

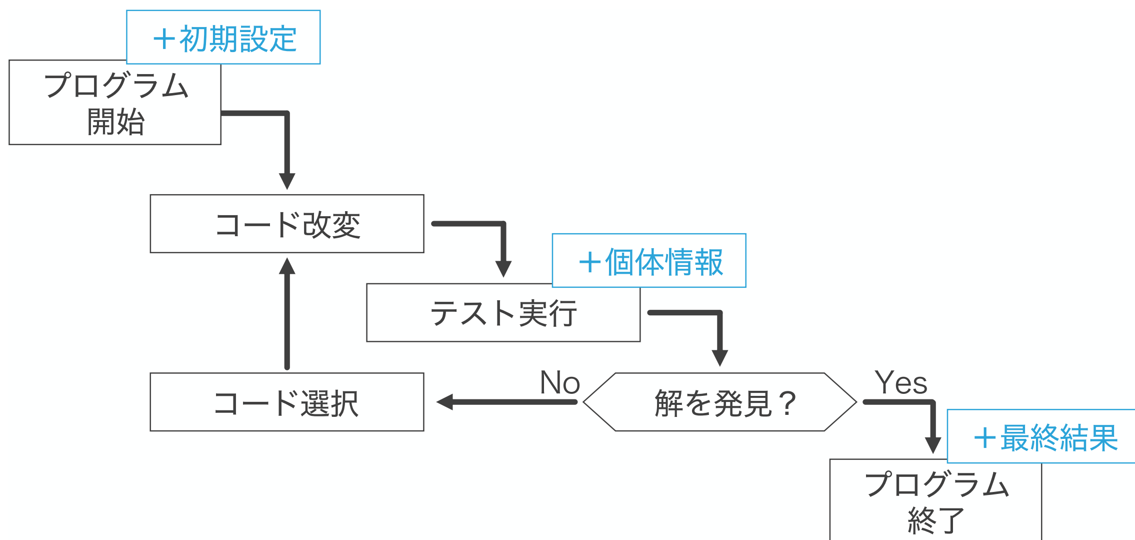


図 3: kGenProg への適用

5 kGenProg への適用

本アイデアの実現可能性を確かめるために，プロトタイプシステムを実装し，我が研究室で開発している自動プログラム修正ツール kGenProg [12] に適用した．具体的には，図 1 (a) に示すようなプログラム修正の全体的な処理フローを管理するクラスに対し，適宜 Git 操作を行う処理を加えた．追加した Git 操作とその場所を図 3 の青字で示す．Git リポジトリの生成，操作には JGit*5を用いた．改変した kGenProg を実行すると，従来のプログラム修正処理に加えてプログラムの進化過程を記録した Git リポジトリを生成する．以降では，Git 操作ごとにその実装内容を簡単に説明する．

5.1 初期設定

この操作では，プログラムの実行を開始した時に試行の初期設定をコミットする．これは図 2 (a) における左上の初期コミットに対応する．具体的には，プログラム進化のツール名やバージョン，設定した実行パラメタなどの情報を全て `README.md` に出力し，master ブランチにこのファイルの追加をコミットする．

5.2 個体情報

この操作では，個体が生成およびテスト実行される度に，その個体を 1 つのリビジョンとしてコミットする．これは図 2 (a) における右側の各コミットに対応する．

*5 `https://www.eclipse.org/jgit/`

各操作では次の処理を行う．

1. 親のブランチにチェックアウト
2. 生成ファイル（`commit-summary.txt`）の更新
3. 個体のソースコードを出力
4. 上記の変更ファイルをコミット
5. 識別子を示すタグを付与

なお，個体の状態によっては次の差異がある．まず，4.3.4 項でも述べた通り，交叉操作によって生成された個体は 2 つの親を持つ．そのため，1. でもう 1 つの親のブランチを現在のブランチにマージする処理を追加で行う．なお，マージをした場合でもこの時点ではまだコミットをしない．次に，ビルド失敗個体は 2. で追加でファイルを生成する．具体的には，ビルドできなかった原因（return 文がないなど）を記したファイル（`build-error-log.txt`）を生成する．さらに，個体が解である場合は 5. で `solution` タグを追加で付与する．

5.3 最終結果

この操作では，プログラムの実行を終了する時に試行の最終結果をコミットする．これは図 2 (a) における左下の最終コミットに対応する．具体的には，最終的に生成した個体総数及び実行時間といった試行に対するメタ情報や生成されたソースコードの diff パッチを `README.md` に追記し，master ブランチにこのファイルの変更をコミットする．

6 速度評価実験

6.1 実験概要

本実験の目的は，提案手法の導入による速度低下度合いの調査である．プログラム進化は最適化問題の一種であり，実行速度の低下は探索効率の低下を引き起こす．提案手法の導入により，速度低下の影響がどの程度発生するかを確認する．

評価の題材には自動プログラム修正ツール kGenProg を用いる．実験では，5 章で実装したプロトタイプシステムを用いて 8〜30 LOC[*6]程度のサンプル題材の実行時間を測定した．用意した 3 つの題材の概要を表 2 に示す．各題材に対して既存手法（改変前の kGenProg）と提案手法（Git 生成機能付きの改変 kGenProg）をそれぞれ適用し，実行時間を測定した．実験結果の誤差を回避するために，題材ごとに 10 個の異なる乱数の組を与えて実験を行った．与える乱数の組は既存手法と提案手法で共通である．

6.2 実験結果

実験結果を図 4 に示す．図における縦軸は実行時間（秒）であり，横軸は適用対象の題材を表している．ただし，グラフによってスケールが異なる．また，自動プログラム修正を行った結果には，解が得られたものの他に，解を得る前に指定した最大世代数に到達したもの，指定した時間を過ぎたために途中で実行が中断されたものも含まれる．

中央値で比較すると，いずれの題材でも提案手法の方が値が大きい．つまり，提案手法の導入によって実行性能は低下している．ただし，低下の度合いは題材によって異なる．CloseToZero は低下が激しいが，残りの 2 つは低下が緩やかである．具体的には，CloseToZero は実行時間が 19.0 秒程度増加しており，これは改変前の 8.3 倍である．一方で GCD は 2.6 秒程度増加して 1.06 倍に，QuickSort は 2.5 秒程度増加して 1.07 倍になっている．

表 2: 実験対象の題材

名前	概要	総テスト数	LOC
CloseToZero	0 に 1 近い値を返す	4	8
GCD	最大公約数を求める	3	13
QuickSort	クイックソートを行う	7	30

[*6] プログラミング言語で書かれたソースコードの内，コメントや空行を除いた行数．

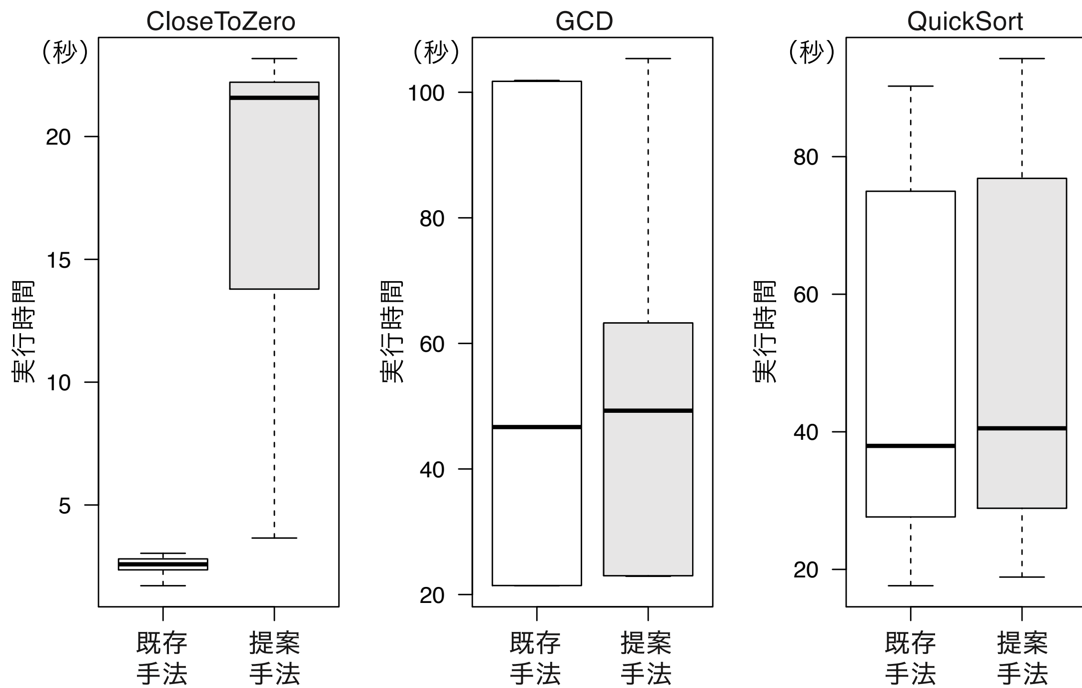


図 4: 速度評価実験の結果

6.3 速度実験の考察

実験結果から，テストに時間がかかる題材ほど性能低下の割合が低いと考えられる．提案手法ではソースコードの書き出しやブランチの切り替え，コミット操作などによって個体の生成に要する時間が増加する．しかし，テストの実行時間は変わらない．そのため，テストの実行時間が長い場合は，提案手法による性能低下は抑えられる．よって，提案手法では性能が低下するが，実際のプログラム修正題材ではテストに時間がかかるため，その割合は大きな問題にならないと考えられる．

```
1  $ cd generated-repo; git checkout master
2  $ cat README.md
3  This is an automatically generated content...
4  # Final results
5  Status: solution found
6  Generated variants: 133
7  Total execution time: 3m19s
8  ...
9  # Initial settings
10 randomSeed = 0
11 timeLimit = 60m
12 maxGenerations = 100
13 ...
```

図 5: プログラム修正全体の実行概要

```
1  $ git show --tags="solution*" --oneline
2  tag solution
3  ccd2c55 (tag: #7.18, tag: solution) op=replace, build=OK, fit=1.0, solution=yes
```

図 6: 解の確認

7 ケーススタディ

7.1 概要

本章では，提案手法の有効性を確認するためのケーススタディを述べる．本ケーススタディでは前章の速度評価実験と同様，5 章で実装したプロトタイプシステムを題材として，生成した Git リポジトリの活用シナリオについて述べる．

7.2 プログラム修正の全体的な結果の把握

自動プログラム修正の研究者が単一のバグを含む題材に自動プログラム修正を適用し，その結果を確認する例を考える．まず，プログラム修正全体の実行概要を把握するために，master ブランチで README ファイルを `cat` コマンドで確認する．結果を図 5 に示す．5 行目から解が得られたこと，6 行目から生成個体数が 133 であること，7 行目から解発見には 3 分 19 秒かかったことが分かる．また，10 行目以降を確認すると，乱数が 0 であったなどの実行パラメタも把握できる．

次に，解となる個体を特定するために，解に付与された `solution` タグを `git-show` コマンドにより絞り込む．得られる結果を図 6 に示す．3 行目に示されるコミットメッセージから解は置き換え操作によって生成されたことなどが分かる．このタグの検索は，重複個体の重複元の表示にも利用できる．さらに，解の詳細を把握するために `git-diff` コマンドを用いて差分を確認する．その結果を図 7 に示す．ここでは `solution` タグが付いている個体（解）を `initial` タグが付いている個体（初期個体）と比較している．6 行目から 7 行目のように `if` 文の追加と `return` 文の変更を初期個体に適用すれば解が得られることが分かる．

```
1  $ git diff initial solution
2  diff --git a/GCD.java b/GCD.java
3  @@ -14,2 +14,3 @@ public class GCD {
4     public int gcd(int a, int b) {
5  -       return 0;
6  +     if (a == 0)
7  +       return b;
```

図 7: 解の差分

```
1  $ git checkout solution; git log --oneline
2  ccd2c55 (tag: #7.18, tag: solution) op=replace, build=OK, fit=1.0, solution=yes
3  f277eb2 (tag: #6.5) op=copy, parent=#5.1
4  0662773 (tag: #5.1) op=copy, parent=#4.2
5  08ddc7b (tag: #4.2) op=copy, parent=#3.10
6  2b1fada (tag: #3.10) op=replace, build=OK, fit=0.75, solution=no
7  3e79d79 (tag: #2.2) op=copy, parent=#1.4
8  e8b0c3e (tag: #1.4) op=insert, build=OK, fit=0.75, solution=no
9  3cd5c43 (tag: initial) op=initial, build=OK, fit=0.75, solution=no
10 33c40aa Initial Settings
```

図 8: 解を得るまでの過程

```
1  $ git log --oneline --all --grep="build=OK"|wc -l
2        31
3  $ git log --oneline --all --grep="build=NG"|wc -l
4        21
5  $ git log --oneline --all --grep="duplicated"|wc -l
6        59
```

図 9: 各状態の個体の個数

最後に，解を得るまでの過程を確認する例を考える．これは図 8 に示すように，解のコミットでの git-log コマンドの実行によって得られる．この結果から，解を得るまでに 4 回もコピー操作を行なっていることが分かる．コピー操作を省いた進化でも同じ解が得られるため，無駄なく進化をした場合，この解は 3 世代目で得られたということである．3 世代目で得られたはずの解が 7 世代目でようやく得られているため，自動プログラム修正の改善の余地がある．

7.3 改変操作ごとの効果の確認

前節で全体的な結果を把握した研究者が，プログラム修正を改善するために，改変操作ごとの効果を確認する例を考える．特定の改変操作に注目する前に，まずは試行全体の傾向を把握する．そのため，図 9 のように各状態の個体の個数を調べる．ここでは git-log コマンドと grep オプションを用いて生成個体の特定の状態（ビルドの成否，あるいは重複）を絞り込む．grep オプションはコミットメッセージに対する検索が可能であり，コミットメッセージには上記の個体の状態が記述されている．さらに wc コマンドで絞り込んだ個体の個数を数え上げる．図 9 の 2 行目からビルド成功個体が 31 個，4 行目からビルド失敗個体が 21 個，6 行目から重複個体が 59 個あることが分かる．割合としては，重複個体がビルド成功個体の倍近く存在しており，無駄な進化が繰り返されている．次に，改善するべき操作を特定するため，改変操作ごとに生成している重複個体数を調べる．その手順を図 10 に示す．先ほど

```
1  $ git log --oneline --all --grep="duplicated" --author="replace" | wc -l
2        13
3  $ git log --oneline --all --grep="duplicated" --author="insert" | wc -l
4        15
5  $ git log --oneline --all --grep="duplicated" --author="delete" | wc -l
6        31
```

図 10: 改変操作ごとの重複個体の個数

```
1  $ git remote add origin https://github.com/x/x.git
2  $ git push --all
3  $ git push --tags
```

図 11: GitHub への生成リポジトリの公開

と同じく `git-log` コマンドと `wc` コマンドを組み合わせる．ただし，`grep` オプションによる重複個体の絞り込みに加えて，`author` オプションによる改変操作の絞り込みを適用している．6 行目から削除操作による個体が他の操作によるものより 2 倍ほど多く存在することが分かる．よって，削除操作の改善により自動プログラム修正全体の効率を改善できる可能性が高い．

7.4 実験結果の共有

最後に，生成した Git リポジトリを他の研究者と共有する例を考える．この共有には，GitHub のような Git ホスティングサイトの利用も可能である．GitHub へのリポジトリ公開のコマンド系列を図 11 に示す．`git-remote` コマンドによりリモートリポジトリを追加したのち，まず `git-push` コマンドと `all` オプションを用いて master 以外の全ブランチをアップロードする．次に `git-push` コマンドと `tags` オプションを用いて全タグをアップロードする．GitHub 側でリポジトリを用意しておけば，このように 3 つのコマンドでアップロード可能である．GitHub の利用により，他の研究者は `git-clone` コマンド 1 つで手元に Git リポジトリをダウンロードできる．また，GitHub のトップページには master ブランチが表示されるため，自動プログラム修正の実行概要が把握できる．

8　おわりに

本研究では，自動プログラム進化の過程を効率的に記録するために版管理を導入する手法を提案した．提案手法の有用性を確認するために，kGenProg を拡張したプロトタイプシステムを実装した．また，ケーススタディを考え，この手法が有効な場面があることを確認した．

今後の課題としては次のようなものが考えられる．

プロトタイプシステムの改変： 本研究で実装したプロトタイプシステムでは記録できる情報が限定されている．例えば，個体を生成した時に適用した操作が何行目を改変したのかを記録できていない．そこで，プロトタイプシステムを改変し，提案手法に拡張性を持たせる．

他のプログラム進化ツールへの導入： 現在の実装では kGenProg の進化過程しか Git リポジトリ化できない．他の探索ベースの自動プログラム修正ツールの進化過程も Git リポジトリ化できるようにすることで，提案手法の有用性を示せる．

定量的な比較実験： 本研究での実験は，サンプル題材の実行時間の比較にとどまっている．実際のプログラムの修正に適用し，実行時間だけでなく生成される Git リポジトリのサイズを比較することで定量的な実験結果が得られる．また，プログラム進化の試行における実行パラメタを変更し，結果がどのように変化するかを調べる．変更する実行パラメタは，最大世代数や 1 世代あたりの個体数などが考えられる．この実験を行うことで，Git リポジトリ化する際に負担となる要素は何かを知れる．

謝辞

本研究を行うにあたり，理解あるご指導を賜り，励ましていただきました，楠本真二 教授に心より感謝申し上げます.

本研究に関して，貴重で有益な助言をしていただきました，肥後芳樹 准教授に深く感謝申し上げます.

本研究の全過程を通し，研究に対する方針や実現方法など，終始丁寧かつ熱心なご指導を賜りました，柗本真佑 助教に心より感謝申し上げます.

本研究を進めるにあたり，様々な形で励まし，協力を頂きました，楠本研究室の皆様に心より感謝申し上げます.

本研究に至るまでに，講義，演習，実験など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この場を借りて心から御礼申し上げます.

最後に，大学生活を支え，励ましてくれた家族にも心より感謝致します.

参考文献

[1] L. Gazzola, D. Micucci, and L. Mariani. Automatic software repair: A survey. *IEEE Transactions on Software Engineering*, Vol. 45, No. 1, pp. 34–67, 2019.

[2] K. Becker and J. Gottschlich. AI programmer: Autonomously creating software programs using genetic algorithms. Technical report, Arxiv, Tech. Rep., 2017.

[3] S. Gustafson, A.Ekárt, E. Burke, G. Kendall. Problem difficulty and code growth in genetic programming. *Genetic Programming and Evolvable Machines*, Vol. 5, No. 3, pp. 271–290, 2004.

[4] C. L. Goues, T. Nguyen, S. Forrest, and W. Weimer. GenProg: A generic method for automatic software repair. *IEEE Transactions on Software Engineering*, Vol. 38, No. 1, pp. 54–72, 2012.

[5] C. L. Goues, M. Dewey-Vogt, S. Forrest, and W. Weimer. A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. In *Proc. International Conference on Software Engineering*, pp. 3–13, 2012.

[6] E. K. Smith, E. T. Barr, C. Le Goues, and Y. Brun. Is the cure worse than the disease? Overfitting in automated program repair. In *Proc. Joint Meeting on Foundations of Software Engineering*, pp. 532–543, 2015.

[7] A. C. Jensen and B. H. Cheng. On the use of genetic programming for automated refactoring and the introduction of design patterns. In *Proc. Annual Conference on Genetic and Evolutionary Computation*, pp. 1341–1348, 2010.

[8] I. H. Moghadam and M. Ó Cinnéide. Code-Imp: A tool for automated search-based refactoring. In *Proc. Workshop on Refactoring Tools*, pp. 41–44, 2011.

[9] Q. Luo, D. Poshyvanyk, and M. Grechanik. Mining performance regression inducing code changes in evolving software. In *Proc. International Conference on Mining Software Repositories*, pp. 25–36, 2016.

[10] Y. Qi, X. Mao, Y. Lei, Z. Dai, and C. Wang. The strength of random search on automated program repair. In *Proc. International Conference on Software Engineering*, pp. 254–265, 2014.

[11] M. Martinez and M. Monperrus. ASTOR: A program repair library for java. In *Proc. International Symposium on Software Testing and Analysis*, pp. 441–444, 2016.

[12] 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.

[13] Xuan-Bach D. Le. Towards efficient and effective automatic program repair. In *Proc. International Conference on Automated Software Engineering*, pp. 876–879, 2016.

[14] J. D. Blischak, E. R. Davenport, and G. Wilson. A quick introduction to version control with Git and GitHub. *PLOS Computational Biology*, Vol. 12, No. 1, pp. 1–18, 2016.

[15] D. Spinellis. Git. *IEEE Software*, Vol. 29, No. 3, pp. 100–101, 2012.

[16] R. Just, D. Jalali, and M. D. Ernst. Defects4J: A database of existing faults to enable controlled testing studies for java programs. In *Proc. International Symposium on Software Testing and Analysis*, pp. 437–440, 2014.

[17] D. Spadini, M. Aniche, and A. Bacchelli. PyDriller: Python framework for mining software repositories. In *Proc. Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 908–911, 2018.

[18] Z. Qi, F. Long, S. Achour, and M. Rinard. An analysis of patch plausibility and correctness for generate-and-validate patch generation systems. In *Proc. International Symposium on Software Testing and Analysis*, pp. 24–36, 2015.

[19] W. Weimer, T. Nguyen, C. Le Goues, and S. Forrest. Automatically finding patches using genetic programming. In *Proc. International Conference on Software Engineering*, pp. 364–374, 2009.

[20] D. Kim, J. Nam, J. Song, and S. Kim. Automatic patch generation learned from human-written patches. In *Proc. International Conference on Software Engineering*, pp. 802–811, 2013.

[21] F. Long and M. Rinard. Staged program repair with condition synthesis. In *Proc. Joint Meeting on Foundations of Software Engineering*, pp. 166–178, 2015.

[22] R. Kou, Y. Higo, and S. Kusumoto. A capable crossover technique on automatic program repair. In *Proc. International Workshop on Empirical Software Engineering in Practice*, pp. 45–50, 2016.