社団法人　電子情報通信学会
THE INSTITUTE OF ELECTRONICS,
INFORMATION AND COMMUNICATION ENGINEERS

信学技報
TECHNICAL REPORT OF IEICE.

自動プログラム進化における進化過程共有のための版管理技術の適用

出田　涼子† 　松本　真佑† 　肥後　芳樹† 　楠本　真二†

† 大阪大学大学院情報科学研究科
E-mail: †{r-izuta,shinsuke,higo,kusumoto}@ist.osaka-u.ac.jp

あらまし　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 の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，これをモノによる進化の記録に適用す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，我々が開発している自動プログラム修正ツール，kGenProg への適用を試みる．

キーワード　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

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，テストやメトリクス等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，遺伝的プログラミング [3] が取り入れられることも多い．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] はプログラム進化の一種のアプリケーションであり，バグのない，すなわち全てのテストが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 [6] や自動パフォーマンス改善 [7] 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）の記録と分析が欠かせない．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善手法を実験し，実装が期待通りの振る舞いをしているか，期待通りの効果を得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 [8]．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，計算能力的なコストの大幅な削減にも繋がる．

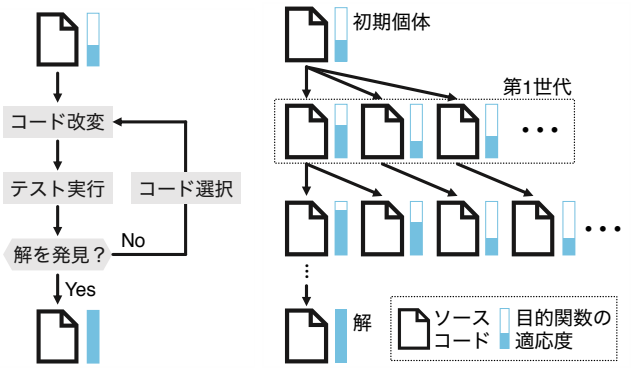
しかしながら，多くのプログラム進化ツールでは進化過程を保持する機能がサポートされていない．ツール実行により得られる情報は，発見した解の他には，生成個体の総数や到達世代数，起動時間といったプログラム進化の最終結果に限られている．一例として，GenProg[4] の Java 実装である jGenProg[9] では，過程を記録するオプションが実装されているものの，記録できる情報は個体ごとの改変ソースコードがファイルとして出力されるのみである．よって，その個体がどのような手法で生成されたか，その個体で目的関数の結果がどう変化したか，どの個体が親でどのような進化の系譜を辿ったか，といった進化過程の細部を把握することはできない．

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 の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，これをモノによる進化の記録に適用す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，我々が開発している自動プログラム修正ツール，kGenProg[10] への適用を試みる．

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一つとしては，テスト通過



(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

数の最大化を目的関数とする自動プログラム修正 [4,5] が広く知られている．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．プログラム進化は様々な目的に応用可能であり，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 [6] となる．また，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 [7] に用いることができる．

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等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，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．進化系譜の一例を図 1 (b) に示す．最上段は初期の個体，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，この個体生成を繰り返す．よって，解の発見に至るまでに膨大な数の個体が生成されることが一般的である．

2.2 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，進化の結果を出力するのみであり，その過程を把握することはできない．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等に出力するだけでは，進化過程の把握には繋がりにくい．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，及び乱択に依存する部分が多い(注1)点の 2 つの理由から，複数の乱数と複数の題材の組み合わせで評価実験が行われる．加えて，プログラム進化の一試行には多大な時間を要することが一般的であり，その再実験や追試には時間的，計算量的な困難さが伴う．

(注1)：生物進化のように意図的に外乱要素を加えることで局所解を避ける．

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，自動プログラム修正をはじめとするプログラム進化に対し，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．さらに，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．これにより，複数の生成リポジトリを比較することで，複数の実験を比較することが可能となる．

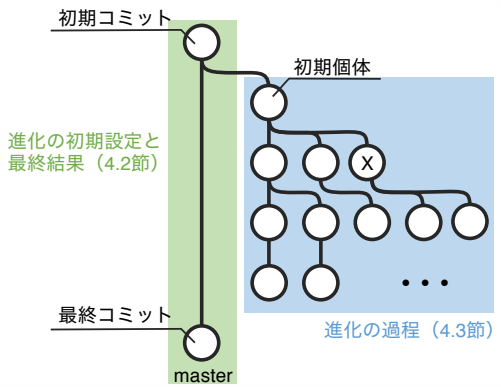
進化過程の記録に版管理を適用することで，以下の 4 つの効果が期待できる．なお以降では，版管理技術の一つとして広く利用されている Git を前提として説明する．

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く，さらに大多数のソースコードには改変が適用されない．よって Git 適用により，大量の生成個体に対しても少ないデータサイズでの記録が可能となる．

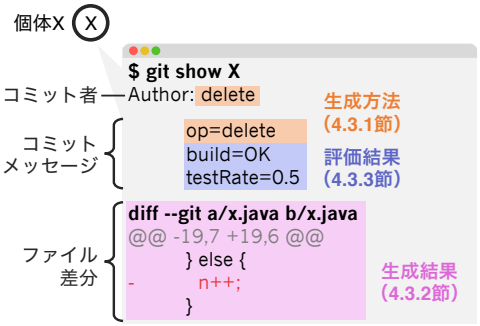
実験試行の完全な記録：プログラム進化はソースコードファイルを中心として改変と評価を繰り返すため，Git 適用によりファイル改変の系列，すなわち進化の過程を完全な形で保持できる．さらに，このファイル改変の系列に加え，記録対象となる試行の初期情報（例えばツール名やツールのバージョン，最大世代数などの初期パラメタ等），及び最終結果（到達世代数や実行時間，diff パッチ等）を記録すれば，どのようにプログラム進化ツールを実行し，どのような結果が得られたかという情報も保持可能となる．プログラム進化の試行には数時間以上を要することが多いが，その試行に関するあらゆる情報を記録し共有できれば，他の開発者による改善や，他研究者による追試の手助けとなる．

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(注2)や GitUp(注3)がある．また，Git リポジトリの分析ツールとしては Gitinspector(注4)や PyDriller[11] が知られている．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ることができる．プログラム進化に Git を適用することで，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把握に利用でき，また Git 分析

(注2)：https://www.gitkraken.com/git-client
(注3)：https://gitup.co
(注4)：https://github.com/ejwa/gitinspector



(a) Git のブランチトポロジ



(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

ツールを用いれば，どの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．

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おり [12, 13]，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る．よって Git の学習コストは低く，導入も容易である．GitHub のようなホスティングサイトも数多く存在しており，生成したリポジトリの共有も容易である．

4. Git 適用の方針

4.1 概　　要

本節では，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，一方が進化全体の要約となる初期設定と最終結果（4.2 節）を，他方が進化の過程（4.3 節）を保持する．さらに進化過程については，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（4.3.1 節），生成結果（4.3.2 節），評価結果（4.3.3 節）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（4.3.4 節）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 進化の初期設定と最終結果

記録対象となる試行の全体像として，進化の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パッチ（diff）情報だけでなく，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，進化過程の細部を無視したプログラム進化の入出力と捉えることが可能であり，試行の全体像把握に活用できる．

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 ブランチにこの概要情報を記録する理由は，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概要を保持させることは直感的かつ自然であるといえる．

ブランチを概要（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．概要は対象試行の一種のメタ情報である一方で，進化過程は試行自体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．

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（以降，初期個体）は，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，個体それ自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すことが可能となる．さらには，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コード改変を加えられたかを意味する．具体的な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 [14] や，他のステートメントからの再利用 [4]，過去の開発者による変更を参考にする方法 [15]，条件分岐文の条件を変更する方法 [16] など多数存在する．jGenProg[9] や kGenProg[10] では，上記の処理をランダムで選択して適用する．

この個体生成方法は，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させる．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る．進化過程の分析目的に応じて，これらの使い分けが可能となる．

4.3.2 個体の生成結果

個体の生成結果とは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．パッケージ構造等のフォルダ構成も初期個体と同様に維持させておくことで，どのファイルが書き換わったかを一般的な 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当てはまり具合を意味する．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情報も評価結果の一種と見なせる．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ことが多い．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り，リポジトリ内に記録することにする．

この評価結果は，4.3.1 節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．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こともある．この重複個体は探索済み空間の再探索に繋がるため，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，コード改変の結果の一つであり，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 節から 4.3.3 節で個体単体に関する情報の記録方針を整理した．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，コミットオブジェクトの 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が生物進化における家系図のように表すことが可能となる．

次に，個体を一意に特定する識別子を考える．Git ではコミット自体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う．プログラム進化の理解という観点では，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"g2.9"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．タグはコミットを識別する一意な情報として利用されるため，個体の識別子と親和性が高い．これらの方針により，例えば"git show --tags=g2.*"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することが可能となる．

4.4 kGenProg への適用

本アイデアの実現可能性を確かめるために，我々が開発している自動プログラム修正ツール kGenProg[10] へ適用しプロトタイプを実装した．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．例えば，個体が生成される度に，その個体を 1 つのリビジョンとしてコミットする．Git リポジトリの生成，操作には JGit(注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．

5. 速度評価実験

5.1 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の度合いを調査することである．評価の題材には自動プログラム修正ツール kGenProg を用いる．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に繋がる．提案手法の導入により，どの程度の速度低下の影響が発生するかを確認する．

実験では，4.4 節で実装したプロトタイプを用いて 8〜30 LOC 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 1 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．

5.2 結　　果

実験結果を図 3 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．自動プログラム修正

表 1: 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

(注5)：https://www.eclipse.org/jgit/

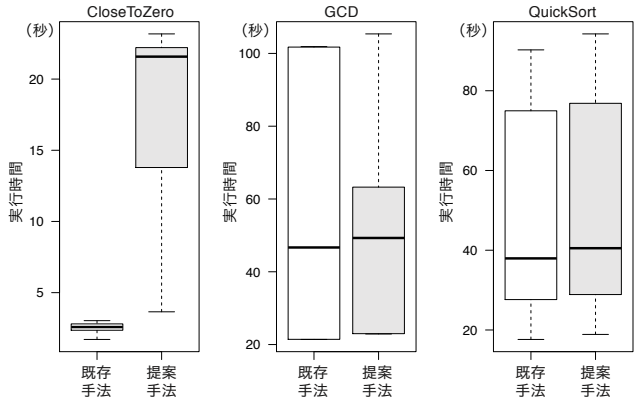


図 3: 実験結果

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，提案手法を導入すること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 は大幅に低下しているが，残りの 2 つは少量の低下である．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 8.3 倍程度の倍率である．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．

5.3 考　　察

性能低下の度合いが題材によって異なる．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の割合は下がる．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はならないと考えられる．

6. ケーススタディ

6.1 概　　要

本節では，提案手法の有効性を確認するためのケーススタディを行う．本ケーススタディでは前節の速度評価実験と同様，自動プログラム修正ツール kGenProg を題材として，提案手法により生成した Git リポジトリの活用シナリオについて述べる．

6.2 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ことを考える．まず，プログラム修正全体の概要を把握するために，master ブランチで `git-log` コマンドを実行する．結果を図 4 に示す．6 行目から十分な解が得られたこと，7 行目から到達世代数が 7 であること，8 行目から解発見には 3 分 19 秒かかったことがわかる．また，12 行目以降を確認することで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

```
1  $ cd generated-repo
2  $ git checkout master
3  $ git log -p
4  diff --git a/result.txt b/result.txt
5  @@ -0,0 +1,3 @@
6  +enough solutions have been found.
7  +GA stopped at the era of 7th generation.
8  +execution time: 3 minutes 19 seconds
9
10 diff --git a/config.txt b/config.txt
11 @@ -0,0 +1,23 @@
12 +mutationGeneratingCount = 10
13 +crossoverGeneratingCount = 10
14 +maxGeneration = 100
15 +randomSeed = 4
16 ...
```

図 4: プログラム修正の総合結果

```
1  $ git show --tags="solution*" --oneline --no-patch
2  tag solution1
3  5a72bcd (tag: v52, tag: g3.13, tag: solution1)
4    op=replace, build=OK, testRate=1.0, solution=yes
5
6  tag solution2
7  0733182 (tag: v138, tag: g7.15, tag: solution2)
8    op=replace, build=OK, testRate=1.0, solution=yes
```

図 5: 解の一覧

```
1  $ git diff v0 solution1
2  diff --git a/x.java b/x.java
3  @@ -14,9 +14,8 @@ public class GCD {
4     public int gcd(int a, int b) {
5
6        if (a == 0)
7  -       return 0;
8  +       return b;
```

図 6: 解の差分

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

図 7: 条件を満たす個体の個数

`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 5 に示す．タグが `solution1` と `solution2` の 2 個存在していることから，計 2 個の解が得られたことが分かる．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 6 に示す．ここでは `solution1` タグが付いている個体（解）を `v0` タグが付いている個体（初期個体）と比較している．8 行目のような `return` 文の変更を初期個体に適用することによって解が得られることが分かる．

6.3 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ことを考える．図 7 に全体傾向を把握する手順を示す．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 7 の 2 行目から重複個体が 59 個，4 行目からビルド成功個体が 31 個，6 行目からビルド失

```
$ git log --oneline --all --grep="duplicated"
  --author="replace" | wc -l
      13
$ git log --oneline --all --grep="duplicated"
  --author="insert" | wc -l
      15
$ git log --oneline --all --grep="duplicated"
  --author="delete" | wc -l
      31
```

図 8: 改変操作ごとの重複個体の個数

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

図 9: GitHub への生成リポジトリの公開

敗個体が 21 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の倍近く存在しており，無駄な進化が繰り返されていることが把握できる．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 8 に示す．先ほどと同じく `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と，`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．9 行目から削除操作が他より 2 倍ほど多く生成していることが分かる．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ことが分かる．

6.4 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ことを考える．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 9 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することができる．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．

7. 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．

今後の課題としてプロトタイプの改変と他のプログラム進化ツールへの導入が考えられる．本研究で実装したプロトタイプ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，プロトタイプを改変し，提案手法の拡張性を持たせたい．

また，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すことができる．

謝辞 本研究の一部は，日本学術振興会科学研究費補助金基盤研究（B）（課題番号：18H03222）の助成を得て行われた．

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, AnikóEkárt, 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] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] D. Spinellis, “Git,” IEEE Software, vol.29, no.3, pp.100–101, 2012.

[14] 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.

[15] 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.

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