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.

再利用に基づく自動プログラム修正における更新順および類似度順の実装と評価

谷門　照斗† 　横山　晴樹† 　鷲見　創一† 　肥後　芳樹† 　楠本　真二†

† 大阪大学大学院情報科学研究科
〒 565-0871 大阪府吹田市山田丘 1-5
E-mail: †{a-tanikd,y-haruki,s-sumi,higo,kusumoto}@ist.osaka-u.ac.jp

あらまし　自動プログラム修正手法として，既存プログラム文の再利用に基づき修正を行う研究が行われている．自動プログラム修正は，プログラム文の変更（挿入，削除，置換）とテスト実行の 2 段階で構成され，すべてのテストを通過するまでプログラム文の変更からやり直す手法である．再利用に基づく手法におけるプログラム文の挿入では，挿入する文をソースコード中からランダムに選択する．しかし，大規模なソフトウェアでは膨大な数のプログラム文が存在しているためにプログラム文の変更とテスト実行の反復回数が増大し，プログラム修正が完了するまでに非常に長い時間を要する．より短い時間で欠陥を修正するため，プログラム文の変更において利用するプログラム文を絞り込むことが必要である．絞り込みの基準として，更新順と類似度順が先行研究で提案されている．しかし，先行研究では絞り込み基準の提案に留まっておりツールの実装と評価は行われていない．本研究では，これらの手法をツールとして実装し，有用性を評価した．

キーワード　デバッグ，自動プログラム修正，コード再利用

1. ま え が き

ソフトウェアの安全性や信頼性向上のためにデバッグは必要不可欠な作業である．デバッグとは，ソフトウェアに含まれる故障を検出し，ソフトウェアのソースコードの中から故障を引き起こす原因となった記述（以降，欠陥）を含む箇所を特定し，特定された欠陥を修正する作業のことである．

デバッグはソフトウェア開発において多大なコストを要する作業であり，プログラム開発工程の約 50%を占めるといわれている [1]．また，米国では 1 年間に約 3,000 億ドルをデバッグのために費やしているともいわれている [1]．このように多大なコストを要するデバッグ作業の負荷削減を目指して，デバッグの支援に関する研究が数多く行われている．

デバッグ支援の理想形はデバッグ作業の完全自動化である．ソフトウェアの故障の検出および欠陥箇所の限局の自動化に関しては，これまでに多数の研究が行われてきた．さらに近年では，欠陥修正の自動化を含めたプログラムの自動修正手法に関する研究も盛んに行われるようになってきており，デバッグ作業の完全自動化に向けて活発に研究が行われている．

自動プログラム修正手法では，GenProg [2] が有望視されている．GenProg は欠陥を含むプログラム（以降，修正対象プログラムと呼ぶ）とテストケースの集合であるテストスイートを入力とし，修正後のプログラムを出力する．GenProg は遺伝的プログラミングに基づき，修正対象プログラム中のコード片を用いて欠陥箇所に繰り返し変更を加えることによりプログラムの自動修正を行う．出力されるプログラムは，入力として与えられたテストケースを全て通過するプログラムである．テストケースはプログラムが満たすべき動作内容を記述したものであるため，GenProg ではテストケースを全て通過するプログラムを修正が完了したプログラムとしている．

GenProg は修正対象プログラムに対して繰り返し変更を行うことで欠陥修正を試みる．プログラムに加える変更には，挿入，削除，置換の 3 種類の操作がある．挿入操作は，修正対象プログラム中に存在するプログラム文をランダムに選択し，それを特定の箇所に挿入するという操作である．ランダムな選択を行うため，プログラムの規模が大きくなるに伴い修正に寄与しないプログラム文を選択する頻度が高くなり，欠陥修正に要する時間が増大してしまう．

そこで，欠陥の修正に寄与する可能性が高いプログラム文を選択する手法が提案されている．その手法として，類似度順 [3] と更新順 [4] が先行研究において提案されている．

しかし，これらの研究では手法の提案に留まり，ツールとしての実装はまだされていない．そこで，これらの手法を実装し，実際のソフトウェア開発の過程で生じた欠陥に対して動作させ，これらの手法が実際の欠陥修正においてどの程度有効に働くのかを評価した．

2. 準　　備

2.1 テストケースを用いたデバッグ

一般的に，デバッグとはソフトウェア中に生じた欠陥の発見から始まり，欠陥を特定して修正し，最後に修正の成否を確認するまでを指す．デバッグを行うためにはまず欠陥が存在する箇所，すなわちソースコードの実装や機能的要求などの誤りを特定する必要がある．

簡単に欠陥を特定する手段としてテストケースの使用が挙げられる．テストケースとは入力値，期待値，実値の組を指し，ある入力値に対する出力値 (実値) が要求される値 (期待値) と等しいか否かを判定するものである．複数のテストケースを実行した上で，どの入力値を与えた場合に誤った動作となるかを把握することで欠陥を特定し易くなる．また，すべてのテストケースが正しく動作するならばそのソフトウェアは信頼性が高いと判断できる．理想的なテストケースはソフトウェアの全動作パターンを確認するものであるが，そのようなテストケースの作成や実行は膨大な時間や労力を必要とするため現実的ではない．したがって，実践上では到達可能な実行パスを網羅する程度に留める場合が大半である．しかし，その場合でも十分なテストケースを手動で用意するには多大な労力を要するため，テストケースを自動で生成する手法が提案されている．

以降，本研究ではデバッグ開始時のソフトウェアが通過するテストケースを通過済テストと呼び，通過しないテストケースを未通過テストと呼ぶ．通過済テストは正しい挙動を示し，未通過テストは誤った挙動すなわち欠陥を示す．

2.2 欠陥箇所の限局

デバッグを支援する主な手法に，欠陥箇所の限局が挙げられる．欠陥箇所の限局手法はソースコード解析を通して欠陥の候補を探す手法であり，最も欠陥の可能性が高い箇所を開発者に提示することでデバッグを補助する．欠陥箇所の限局手法には，テストケース毎の実行パスから算出した疑惑値に基づいて順序付けを行う手法が存在する [5]．このような手法では，通過済みテストのみが実行する文は疑惑値が低くなり，未通過テストのみが実行する文は疑惑値が高くなる．また，GenProg およびその関連手法のうちいくつかはプログラムの変更を行う際，テストケースの実行パスを調べて修正箇所を決定する．

2.3 GenProg

GenProg は遺伝的プログラミング [6] に基づいて自動的にプログラムの修正を行う手法である [2]．GenProg は欠陥を含むプログラムおよびテストケースの集合であるテストスイートを入力として受け取り，再利用に基づく自動プログラム修正を行う．出力はテストスイートに含まれるすべてのテストケースを通過するプログラムである．ここで，入力として与える欠陥を含むプログラムのことを修正対象プログラム，出力として得られる修正が完了したプログラムのことを修正済みプログラムと呼ぶ．また，GenProg は自動プログラム修正を行う前に，入力されたテストケースを用いて修正対象プログラムの欠陥箇所の限局を行う．

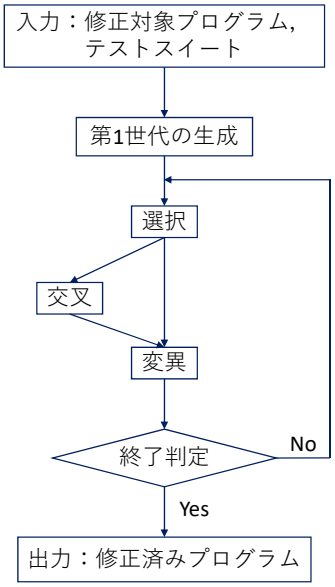

図 1　GenProg の動作の流れ

GenProg の動作の流れを図 1 に示す．GenProg は欠陥箇所の限局を行った後，欠陥箇所に変異操作を行ったプログラム（以降，変異プログラムと呼ぶ）を複数生成する．これらの変異プログラム群のことを第 1 世代と呼ぶ．変異操作では，次の 3 処理のうちいずれか 1 つを行う．

挿入　欠陥を含む行の直後に，修正対象プログラムに含まれるプログラム文の挿入を行う操作

削除　欠陥を含む行を削除する操作

置換　修正対象プログラムからランダムに選択された行によって欠陥を含む行を上書きする操作（挿入操作＋削除操作）

次に，評価関数に基づいて各変異プログラムの評価値を計測し，評価値の高いものを一定数残し，それ以外を削除する．ここで残った変異プログラム群に対して交叉操作および変異操作を行うことで新たな変異プログラム群を得る．交叉操作は 2 つの変異プログラムを組み合わせることで新たな変異プログラムを生成する操作である．交叉操作および変異操作によって得られた変異プログラム群のことを次世代と呼ぶ．

次世代の変異プログラム群に対してテストを行い，すべてのテストケースを通過する変異プログラムがあれば，それを修正済みプログラムとして出力する．そのような変異プログラムが存在しなければ，選択操作からやり直す．この処理をすべてのテストケースを通過する変異プログラムが生成されるか，あらかじめ定められた世代数に到達するまで繰り返す．

GenProg では，ソースコード中に存在する行を用いて欠陥を修正できると仮定している．この仮定の正しさを検証するために，Barr らは実際に行われた変更を基に調査を行った [7]．調査の結果，ソースコード中の行を用いることで，10%の変更において追加された行のすべての行を記述することができ，42%の変更において追加された半数以上の行を記述できることが分かった．

2.4 再利用に基づく自動プログラム修正手法の課題点

挿入・置換操作で挿入するプログラム文をランダムに選択

するため，プログラムの規模が大きくなれば，修正に寄与するプログラム文を選択するまでに行う変異操作の数が増大し，欠陥修正に膨大な時間を必要とする．また，修正候補の探索は一定の時間もしくは世代数で打ち切るため，探索回数や修正時間の増大は欠陥修正の可否にも影響を及ぼす．

したがって，修正時間の増大を抑え効率的に修正に寄与するプログラム文を選択できるような手法が求められる．

3. 研 究 目 的

本章では，研究の目的について述べる．本研究に至った背景として，本研究グループの横山らと山本らが行った研究がある．これらの研究について説明する．

3.1 横山らの研究

本節では，既存の再利用に基づく自動プログラム修正手法が抱える課題の解決に向けて調査した横山らの研究 [3] について説明する．はじめに横山らの研究で用いられた用語を説明する．これらの用語は本論文中でも用いる．

挿入候補　変異処理において挿入の対象となりうる行の集合

挿入行　挿入候補のうち，実際に挿入された行

周辺領域　ある行を中心とした前後数行

横山らは，先述した再利用に基づく自動プログラム修正手法の課題を改善するために，ソースコードの行に優先度付けを行い，優先度順に挿入することを提案した．優先度付けの基準として，横山らは類似度順を提案した．

類似度順ではまず，すべての挿入候補に対して，その周辺領域と欠陥が存在する行の周辺領域との類似度を計測する．そして，この類似度が高いものから順に挿入候補を選択する．

周辺領域間の類似度はレーベンシュタイン距離を用いて算出する．レーベンシュタイン距離とは，2 つの文字列に対して片方の文字列を他方の文字列へ編集するのにかかる最小手数を表す．文字列の編集には文字の挿入・削除・置換があり，各操作は 1 回につき 1 手として手数をカウントする．横山らの研究では，文字列ではなくトークン列のレーベンシュタイン距離を考え，トークン単位の編集距離を求める．レーベンシュタイン距離は対象とする文字列（トークン列）が長くなるにつれて大きくなる．そこで，様々な文字列（トークン列）における類似度を公平に比較するために，値の正規化を行う．また，レーベンシュタイン距離が小さいほど類似度は高くなるべきなので，レーベンシュタイン距離を正規化した値の大小を逆転させる必要がある．これらのことを考慮して 2 つのトークン列 t_1, t_2 に対する類似度順 $s(t_1, t_2)$ を以下のように定める．

$$s(t_1, t_2) = 1 - \frac{dist(t_1, t_2)}{max\{len(t_1), len(t_2)\}}$$

ここで，$dist(t_1, t_2)$ は t_1 ，t_2 間のレーベンシュタイン距離，$len(t_1), len(t_2)$ はそれぞれ t_1, t_2 のトークン列の長さを表す．$dist(t_1, t_2)$ の範囲は，$0 \leqq dist(t_1, t_2) \leqq max\{len(t_1), len(t_2)\}$ であるため，$s(t_1, t_2)$ の範囲は $0 \leqq s(t_1, t_2) \leqq 1$ となる．2 つのトークン列のレーベンシュタイン距離が小さくなるほど，つまり 2 つのトークン列が類似しているほど $s(t_1, t_2)$ は 1 に近づくことになる．

3.2 山本らの研究

山本らは新たな基準として更新順を提案した [4]．更新順とは，挿入候補をその最終更新日時の新しい順で選択するという評価基準である．以降，更新順による挿入候補の選択手順について説明する．

更新順は入力として，修正対象プログラムとテストスイートに加えて，修正対象プログラムの開発履歴を受け取る．開発履歴はバージョン管理システムにより取得する．入力された開発履歴を基に，修正対象プログラム中のすべての行の最終更新日時を取得する．そして，この最終更新日時がより最近のものから順に挿入候補を選択する．

3.3 研 究 動 機

3.1 節，3.2 節において，類似度順および更新順について述べた．これらの手法はどちらもツールとして実装されて評価されているわけではないため，実際の欠陥修正においてどれほど効率よく動作するのかということは分かっていない．そこで本研究ではまず，類似度順と更新順をツールとして実装しする．そして，そのツールを実際の開発現場で生じた欠陥に対して動作させることで，これらの手法が実社会で生じた欠陥をどの程度効率的に修正できるのかを評価する．

3.4 研 究 課 題

本研究では，実社会で生じた欠陥の自動修正に対して類似度順と更新順がどの程度有効なのかを明らかにすることを目的とする．そのため，以下の 2 つの研究課題を設定した．

RQ1　類似度順はランダムに挿入候補を選択する手法よりも多くの欠陥を修正できるか．

RQ2　更新順はランダムに挿入候補を選択する手法よりも多くの欠陥を修正できるか．

RQ3　類似度順はランダムに挿入候補を選択する手法よりも高速に欠陥を修正できるか．

RQ4　更新順はランダムに挿入候補を選択する手法よりも高速に欠陥を修正できるか．

4. 類似度順および更新順による選択を行うシステムの開発

4.1 Astor

類似度順と更新順の実装は， Astor [8], [9] という既存の再利用に基づく自動プログラム修正ツールに対して両手法を組み込むことで行った．本節では，この Astor の概要について説明する．

Astor はオープンソースで開発が進められ，誰もが使用できるよう公開されている．さらに，Astor は自動プログラム修正手法のプラットフォームとなることも目的としてる．Astor は拡張が行いやすいように設計されており，Astor に対して独自の拡張を行うことにより新たな自動プログラム修正手法の実装を容易に行えるようになっている．

Astor には GenProg [2], Kali [10], MutRepair [8], [11] の 3 つの自動プログラム修正手法が実装されており，どれか 1 つを選んで実行することができる．これらの自動プログラム修正手

法を実現するツールはすでに存在していたが，これらのツールが修正対象とするプログラムの記述言語は C 言語であった．一方，Astor は Java を対象としている．Astor における GenProg, Kali および MutRepair の Java 向け実装はそれぞれ jGenProg2, jKali および jMutRepair と呼ばれている．本研究では jGenProg2 に対して拡張を行うことで類似度順と更新順を実装した．

jGenProg2 には独自の手法の実装を容易に行えるように IngredientSearchStrategy という抽象クラスが用意されている．これは変更操作に用いるプログラム文の選択アルゴリズムを表現するための抽象クラスである．この抽象クラスには抽象メソッドが 1 つ定義されており，このメソッドの実装によって，変更操作に用いるプログラム文をどのようなアルゴリズムに基づいて選択するかを規定することができる．このメソッドは，どこにどんな変更操作を行うのかという情報を受け取り，その変更操作に用いるプログラム文を返すメソッドである．各変更操作は，変更を行う箇所（欠陥限局箇所）と変更操作の種類の組によって一意に識別できる．以降，この組のことを変更クエリと呼ぶ．jGenProg2 では，どのような変更クエリに対しても，修正対象プログラム中からランダムに選択してきたプログラム文を返している．ただし，同一のクエリに対して一度適用したプログラム文は再度返さないようになっている．

類似度順と更新順のそれぞれの絞り込み基準に従ってプログラム文を選択するような IngredientSearchStrategy の子クラスを作成することで，類似度順と更新順を実現できる．以降，類似度順と更新順を実現する IngredientSearchStragety の子クラスをそれぞれ SimilaritySearch, ChronologicalSearch と呼ぶ．

4.2 類似度順の実装

SimilaritySearch は類似度順位表を持つ．類似度順位表とは，修正対象プログラムの各行を，その行の周辺領域と欠陥限局箇所の周辺領域との類似度が高い順に並べたものである．欠陥限局箇所は複数存在する可能性があるので，欠陥限局箇所ごとに類似度順位表を作成する．SimilaritySearch は変更クエリを受け取るごとに変更箇所に対応する類似度順位表の上から順にプログラム文を返す．

4.3 更新順の実装

ChronologicalSearch は更新順位表をもっている．更新順位表とは，修正対象プログラムの各行を，その行の最終更新日時が最近のものから順に並べたものである．ChronologicalSearch は変更クエリごとに更新順位表の上から順にプログラム文を返す．

5. 評価実験

4.2 節，4.3 節において，類似度順と更新順をどのように実装したのかについて述べた．本研究ではこの実装したツールを用いて， 類似度順および更新順が実際の欠陥修正においてどの程度有効に動作するかを評価するために実験を行った．本章では，この評価実験について述べる．

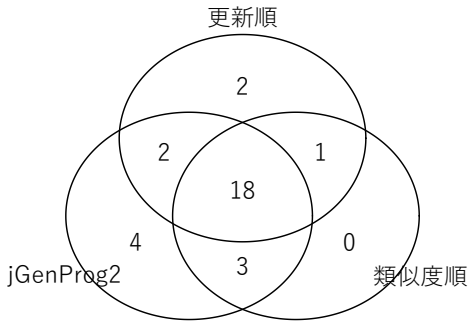

図 2　各種法によって修正できた欠陥の個数

5.1 実験設定

類似度順と更新順の有用性を評価するために，類似度順，更新順および jGenProg2 を実際の欠陥に対して動作させ，その結果を比較した．評価項目は以下の 2 つである．

修正欠陥数　修正することのできた欠陥の個数

修正時間　修正済みプログラムが得られるまでに要した時間

今回の実験の対象は Apache Commons Math というオープンソースで開発されているソフトウェアの開発過程で生じた 106 個の欠陥である．これらの欠陥は Defects4j [12] という欠陥のデータセットより取得した．

5.2 実験結果

5.2.1 修正欠陥数

各手法によって修正できた欠陥の個数を図 2 に示す．図中の Similarity と Chronological はそれぞれ類似度順と更新順を意味する．図から分かる通り，jGenProg2 には修正できなかったが類所度順または更新順では修正することができた欠陥が存在する．つまり，jGenProg2 に加えて類似度順や更新順を用いることで修正できる欠陥の数を増やすことができる．

ただし，種法ごとに修正できた欠陥の個数を比較すると，jGenProg2 が 27 個，類似度順が 22 個，更新順が 23 個となり，類似度順と更新順ともに jGenProg2 よりは多くの欠陥を修正することができないことが分かる．このことから，RQ1, RQ2 への回答はともに No である．

5.2.2 修正時間

3 手法すべてが修正できた欠陥に関して，各手法の修正時間の分布を図 3 に示す．図中の Similarity と Chronological はそれぞれ類似度順と更新順を意味する．データの正規性を検証するためコルモゴロフ・スミノフ検定を行った．その結果，jGenProg2, 類似度順，更新順に対する p-値はそれぞれ 1.8×10^{-2} 程度，3.0×10^{-3} 程度，6.0×10^{-2} 程度となり，jGenProg2 と類似度順には正規性がないことが分かった．そこで，正規性のない対応のある 2 群の有意差を検証するため，ウィルコクサン符号順位検定を jGenProg2 と類似度順，jGenProg2 と更新順の 2 組に対してそれぞれ行った．その結果，jGenProg2 と類似度順の組に対する p-値は 8.2×10^{-1} 程度，jGenProg2 と更新順の組に対する p-値は 3.3×10^{-1} 程度となり，有意差がないことが分かった．ただし，ウィルコクサン符号順位検定で有意差なしと判定されても t 検定で有意差が検出されることがある．そこで，この 2 組に対してそれぞ

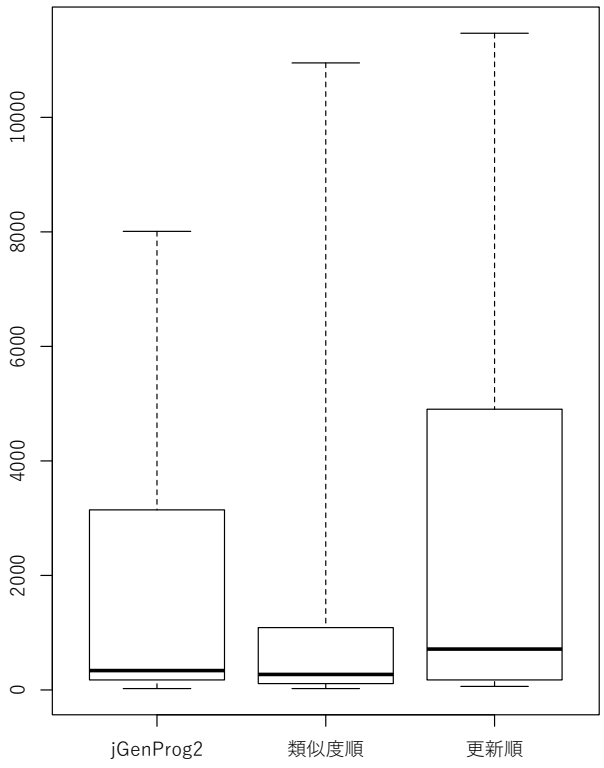

図 3　3 手法が修正した欠陥の修正時間の箱ひげ図

れ対応のある t 検定も行った．その結果，jGenProg2 と類似度順の組に対する p-値は 7.4×10^{-1} 程度，jGenProg2 と更新順に対する p-値は 1.9×10^{-1} 程度となり，2 組ともほぼ確実に有意差がないことが分かった．

以上の結果より，RQ3, RQ4 への回答はともに No である．

6. 考　　察

実験結果より，類似度順および更新順は修正欠陥数に関しては jGenProg2 に勝っているとは言えず，修正時間に関しては有意な差が見られなかった．つまり，類似度順および更新順は jGenProg2 よりも有用性が高いとは言い切れないということである．しかし，類似度順と更新順がまったく有用でないわけでもない．修正欠陥数に関しては，図 2 に示した通り，jGenProg2 には修正できなかったが類似度順もしくは更新順では修正できる欠陥も存在する．また修正時間に関しても，jGenProg2 では修正するのに比較的時間がかかるが類似度順もしくは更新順では比較的短時間で修正できる欠陥も存在する．このように，類似度順や更新順で効率よく修正できる欠陥に対してはランダムに挿入候補を選択する手法よりも有用である．

修正時間の評価実験では，対応のある 2 群の検定を行った．これは，同一の欠陥に対して jGenProg2 と類似度順（もしくは更新順）を動作させるという試行を多数行った際に，これらの手法間で修正時間に差があるかを確かめるものである．この結果，有意な差は見られなかった．そこで，3 手法のうち少なくとも 1 つの手法によって修正できた欠陥 30 個について，欠陥ごとにそれぞれの手法がどの程度の時間でその欠陥を修正できたのかを比べてみた．その結果を図 4 に示す．ただし，グラフ中の網掛けが施されているものは，その欠陥をその手法によって修正できなかったことを表している．修正できなかった欠陥に関しては，タイムアウト時間に到達して終了したものと，修正途中で例外が発生して終了したものがある．また，横軸の各欠陥 ID には色が着けてあるが，この色はその欠陥を最も早く修正した手法の色に対応している．

欠陥ごとに 3 手法の修正時間を比べてみると，3 手法の修正時間の傾向によってこれらの欠陥を大きく 2 つに分けることができる．1 つは，3 手法ともに数分から数十分程度と比較的短時間で修正できているもの．もう 1 つは，比較的長時間かかっている（もしくは，修正できていない）手法がある一方，比較的短時間で修正を終えている手法もあるものである．

後者の欠陥群について見てみると，jGenProg2 の数分の 1 程度の時間で修正できているものが類似度順，更新順の両方に対して存在する．例えば，math60 については，類似度順は jGenProg2 の 10 分の 1 程度の時間で修正できており，math84 については，更新順は jGenProg2 の 30 分の 1 程度の時間で修正できている．このように欠陥によっては，1 つの手法では修正に時間がかかっても他の手法では比較的高速に修正できることがある．また，ある手法では修正できないくても，別な手法では修正できる欠陥もある．つまり，jGenProg2，類似度順および更新順はある程度相補的な関係にあるといえる．したがって，修正したい欠陥に応じて，その欠陥を効率よく修正できる手法を選択して実行できれば，修正欠陥数の増加および修正時間の短縮につながる．しかし，欠陥を効率よく修正できる手法を事前に判断するのは困難だと考えられる．そこで，jGenProg2, 類似度順および更新順を一定時間ごとに切り替えながら実行することで修正時間を平均的に短縮することができるのではないかと考えられる．

7. 妥当性への脅威

本研究の実験対象プロジェクトは Apache Commons Math のみである．プロジェクトが変われば発生する欠陥の傾向も変わる可能性が考えられるため，その他のプロジェクトで生じた欠陥に対しても実験を行ってみるべきである．

本研究では，既存の再利用に基づく自動プログラム修正ツールである Astor に対して拡張を行うことで類似度順と更新順の実装を行った．Astor は挿入候補の選択以外にも内部で乱数を多用するため，実行時引数として乱数のシード値を指定する．今回はこのシード値を 0 で固定して実験を行ったが，シード値を変えることで実験結果が大きく変わる可能性もある．

8. あ と が き

本研究では，類似度順と更新順の実装を行い，これらの手法の有用性を評価するために実験を行った．欠陥修正数に関しては，類似度順と更新順ともにランダムに修正候補を選択する手法よりも多くの欠陥を修正することはできないことが分かった．修正時間に関しては，更新順と類似度順ともにランダムに挿入候補を選択する手法との有意な差は見られなかった．

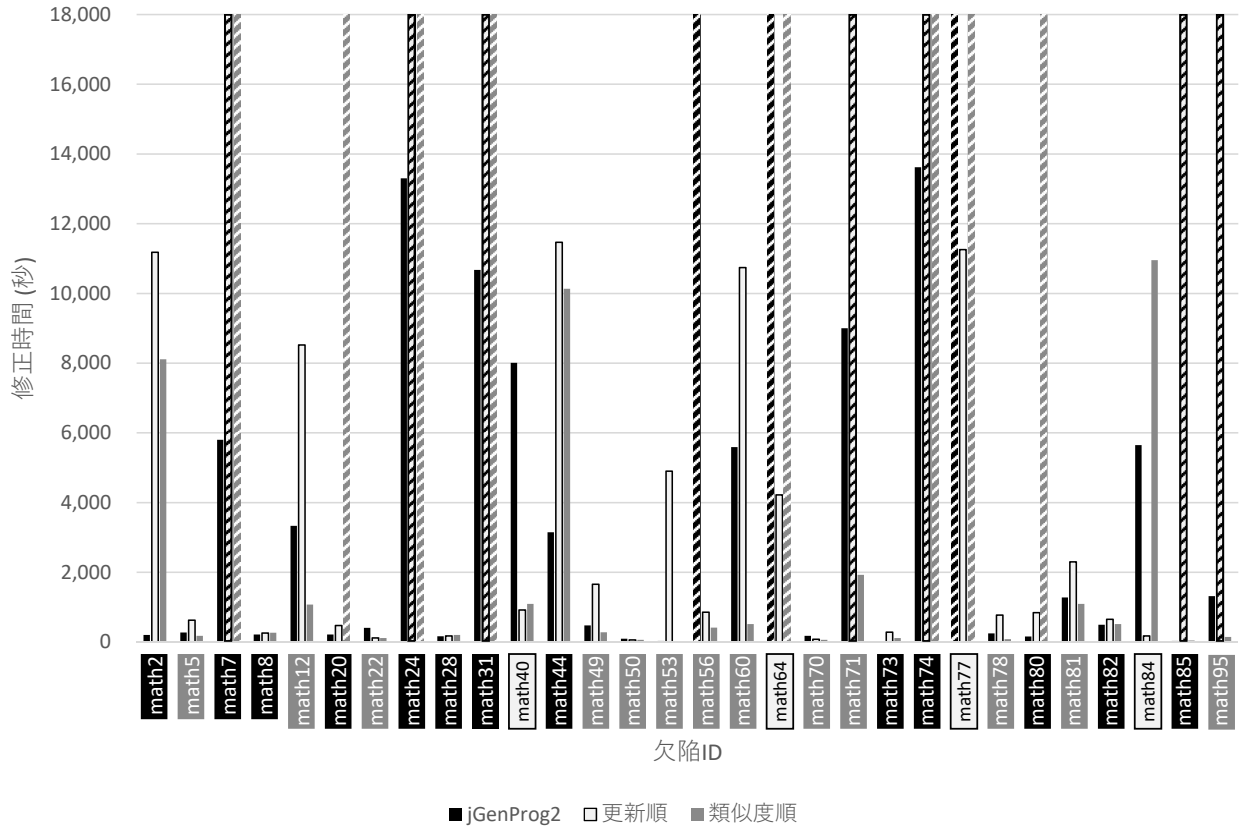

図 4　修 正 時 間

今後の課題は，類似度順，更新順およびランダムに挿入候補を選択する手法をローテーションさせながら実行する手法を実装し，欠陥修正に要する時間を短縮できるかどうか調査することである．

謝辞　本研究は，日本学術振興会科学研究費補助金基盤研究（S）（課題番号：JP25220003）の助成を得て行われた．

文　　献

[1] T. Britton, L. Jeng, G. Carver, P. Cheak, and T. Katzenellenbogen, “Reversible debugging software,” Technical report, University of Cambridge, Judge Business School, 2013.

[2] C. Le 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,” Proceedings of International Conference on Software Engineering (ICSE), pp.3–13, 2012.

[3] 横山晴樹，大田崇史，堀田圭佑，“再利用に基づく自動バグ修正における再利用候補の絞込に向けた調査 (ソフトウェアサイエンス),” 電子情報通信学会技術研究報告，vol.115，no.20，pp.47–52，2015.

[4] 山本将弘，横山晴樹，肥後芳樹，楠本真二，“再利用に基づく自動プログラム修正における更新順を用いた挿入候補の絞込の提案 (ソフトウェアサイエンス),” 電子情報通信学会技術研究報告，vol.115，no.508，pp.79–84，2016.

[5] R. Abreu, P. Zoeteweij, R. Golsteijn, and A.J. Van Gemund, “A practical evaluation of spectrum-based fault localization,” Journal of Systems and Software, vol.82, no.11, pp.1780–1792, 2009.

[6] J.R. Koza, Genetic programming: on the programming of computers by means of natural selection, vol.1, MIT press, 1992.

[7] E.T. Barr, Y. Brun, P. Devanbu, M. Harman, and F. Sarro, “The plastic surgery hypothesis,” Proceedings of the International Symposium on Foundations of Software Engineering, pp.306–317, 2014.

[8] M. Martinez and M. Monperrus, “ASTOR: Evolutionary Automatic Software Repair for Java,” Technical report, Inria, 2014.

[9] M. Martinez and M. Monperrus, “Astor: a program repair library for java,” Proceedings of the International Symposium on Software Testing and Analysis (ISSTA), pp.441–444, 2016.

[10] Z. Qi, F. Long, S. Achour, and M. Rinard, “An analysis of patch plausibility and correctness for generate-and-validate patch generation systems,” Proceedings of the International Symposium on Software Testing and Analysis (ISSTA), pp.24–36, 2015.

[11] V. Debroy and W.E. Wong, “Using mutation to automatically suggest fixes for faulty programs,” Proceedings of International Conference on Software Testing, Verification and Validation (ICST), pp.65–74, 2010.

[12] R. Just, D. Jalali, and M.D. Ernst, “Defects4j: A database of existing faults to enable controlled testing studies for java programs,” Proceedings of the International Symposium on Software Testing and Analysis (ISSTA), pp.437–440, 2014.