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.

多目的遺伝的アルゴリズムを用いた自動プログラム生成手法の提案

—プログラミングコンテストを題材として—

渡辺　大登† 　柗本　真佑† 　肥後　芳樹† 　楠本　真二† 　倉林　利行††
吉村　　優†† 　切貫　弘之†† 　但馬　将貴†† 　丹野　治門††

† 大阪大学大学院情報科学研究科
†† 日本電信電話株式会社
E-mail: †{h-watanb,shinsuke,kusumoto}@ist.osaka-u.ac.jp

あらまし　人手を介さない自動プログラム生成の実現を目指して，生成と検証に基づく自動プログラム修正（APR）を転用した手法が提案されている．この手法では，初期状態でのソースコードを未実装，つまり複数のバグが含まれていると仮定し，ソースコードの改変・評価・選択を繰り返してソースコードを目的の状態に近づけていく．一般的なAPR では改変ソースコードの評価指標として，テストケースの通過数というスカラ値がよく用いられる．この指標では，単一バグの修正を目的とした場合には問題にならないが，複数バグの修正時にはコード評価の表現能力不足という問題に繋がる．よって，初期状態で複数のバグがあると仮定するプログラム生成に対しては，解決すべき重要な課題であるといえる．そこで，本研究では自動プログラム生成の効率向上を目的として，APR に対する多目的遺伝的アルゴリズムの適用を提案する．さらに，相補的なテスト結果を持つ 2 個体を合成する新たな交叉手法を提案する．評価実験として，プログラミングコンテストの問題 80 問を題材として，提案手法の効果を確かめる．実験結果より，80 題材中 39 題材で生成時間の短縮が確認できた．

キーワード　自動プログラム生成，自動プログラム修正，多目的遺伝的アルゴリズム，非優越ソート

1. はじめに

人手を介さない完全自動によるプログラムの生成を目指した研究が進められている [1] [2]．その実現手法の 1 つとして，生成と検証に基づく自動プログラム修正 [3]（Automated Program Repair，APR[1]）を転用した方法がある [4]．APR はバグを含むソースコードと対応するテストケースを入力とし，自動的なバグ箇所の特定 [5]，及びバグ箇所に対するソースコードの改変を繰り返すことで，全テストケースを通過する，すなわちバグのないソースコードを得る．自動プログラム生成に対する APR の転用 [4] においては，初期状態でのソースコードが完全に未実装であり，これを全テストが失敗する，つまり多数のバグを含んだ状態であると捉えることで，テストケースを 1 つずつ通過するよう探索的にソースコードを進化させる．

APR はこの 10 年間で数多くの研究が実施されている [6] が，実用と理論の両面に対して多くの課題が指摘されている．具体的な課題としては，探索空間が巨大であり修正に多くの時間を要する [7]，テストに過剰適合したオーバーフィットが発生する [8]，生成ソースコードの可読性が低く開発者に受け入れられない [9]，複数のバグを含むプログラムの修正が難しい [10]，などが挙げられる．先述の通り，自動プログラム生成では初期のソースコードが多数のバグを含んだ状態であり，APR の自動プログラム生成への転用という観点では，複数バグの修正という課題の解決が必須である．

本稿では複数バグという課題に対し，APR における選択時の評価値の改善について考える．APR では遺伝的アルゴリズム等の探索的手法に基づき，ソースコードの改変（個体の生成）とテスト実行による評価（個体の検証）を繰り返す．個体の検証では，生成した個体がバグ修正という目的に対して改善されたかを評価値という尺度として計測し，優秀な個体のみを次の探索に用いる．この選択における 1 つの問題は評価値の計算方法にある．多くの APR 技術では評価値をテスト通過数という単一のスカラ値で計測する [11]．この指標は単一バグを対象とする場合には問題とならないが，複数バグの場合には表現能力の不足という問題に繋がる．例えば，ある個体が 3 つのテストケースのうち 2 つを通過していた場合を考える．この個体を oox（テスト 2 つ通過を意味）と表記する．ここで，個体生成で得られた別の 2 つの個体 oxx と xxo におけるテスト通過数は 1 となり完全に同値である．しかし，oox を補完する改変情報を

(注1)：自動プログラム修正は，生成と検証ベース以外にも意味論ベースの手法も存在するが，本稿では生成と検証ベースのみを対象とし，簡略化のためにこれを単に APR と略す．

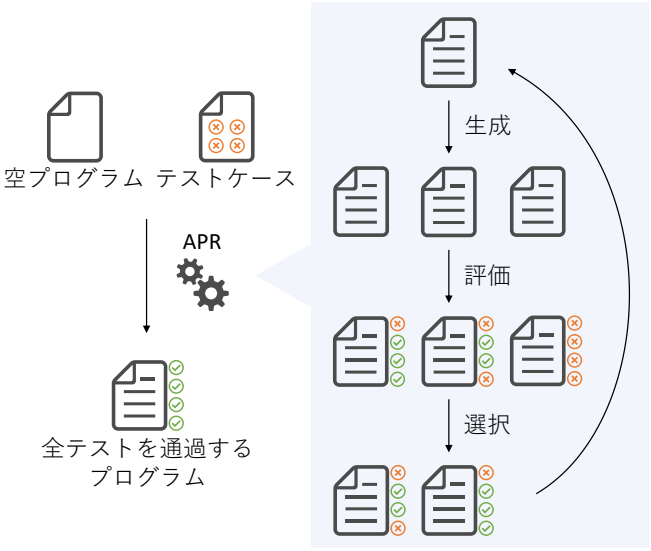

図 1: 生成と検証に基づく APR を転用した自動プログラム生成

持つという観点では，xxo を優先的に選択するべきである．

本研究では，APR を転用した自動プログラム生成の効率改善を目的として，APR への多目的遺伝的アルゴリズムの適用を提案する．多目的遺伝的アルゴリズムは，複数の評価関数を用いて生成個体を評価する遺伝的アルゴリズムの一種であり，単一のスカラ値と比べて高い表現能力を持つ個体評価が可能となる．また複数評価関数の採用により，局所解の回避も期待できる [12] ため，巨大な探索空間を持つ自動プログラム生成と高い親和性を持つと期待できる．上記提案に加え，相補的なテスト結果を持つ 2 個体を合成する新たな交叉手法も提案する．評価実験では，プログラミングコンテストの問題 80 問を題材として，提案手法と既存の APR ツールの生成効率を比較した．その結果，既存手法と比較して 80 題材中 39 題材で生成時間の短縮が確認できた．

2. 準　　備

本章では，生成と検証に基づく APR を転用した探索的な自動プログラム生成の流れと，既存手法の課題を具体例を用いて説明する．

2.1 自動プログラム生成の流れ

図 1 に生成と検証に基づく APR を転用した探索的な自動プログラム生成の流れを表す．入力は空プログラムとテストケース，出力は入力された全テストケースを通過するプログラムである．この手法は個体の生成，評価，選択の 3 つの処理（1 世代）の繰り返しからなる．以下では，APR 分野のブレイクスルーとなった GenProg [11] が用いる，遺伝的アルゴリズムに基づいた手法について上記 3 つの流れに従って説明する．

まず，入力プログラムから少しの改変が加えられた個体を生成する．この個体の生成方法には変異と交叉の 2 種類がある．変異はある 1 つの個体から新しい個体を生成する操作である．変異の操作として最も単純なものはプログラム文の挿入，削除，置換である．挿入と置換に利用するプログラム文は個体が持つプログラムもしくは再利用コードから取得する．交叉は 2 つ以

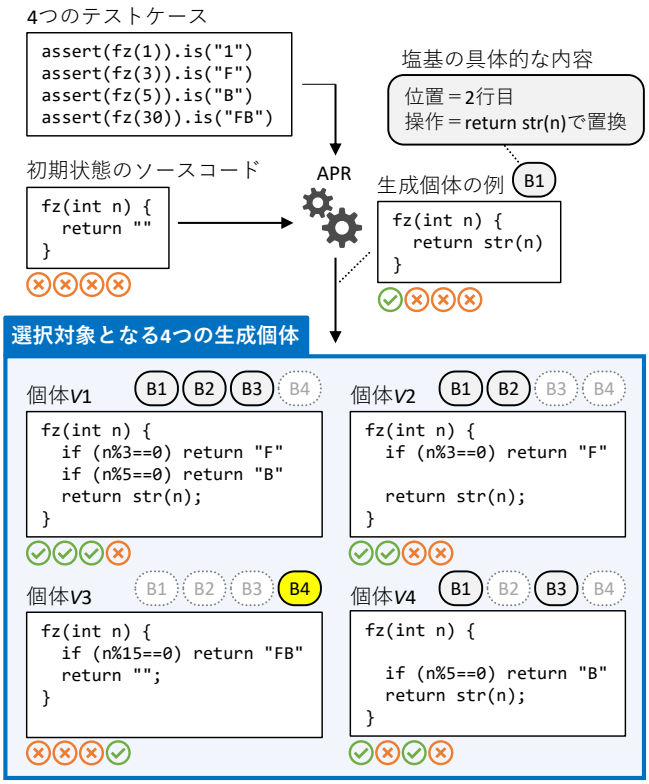

図 2: APR における個体選択の課題

上の個体から新しい個体を生成する操作である．生成元となる 2 つ以上の個体の改変履歴を再利用することによって，個体を生成する．既存の交叉法として，遺伝子上のある点を交叉点とし，交叉点より後ろの遺伝子を交換する一点交叉や，遺伝子の各部分を交換するか否かを部分ごとに定める一様交叉が存在する [13]．次に生成された個体それぞれについて，バグ修正という目的に対してどの程度近づいたかを評価する．個体評価における評価値は個体のソースコードの行数や遺伝子の長さ，生成した個体をコンパイルし，入力されたテストケースを実行した結果などから算出する．GenProg では，個体の評価値としてテスト通過数を用いている．このようにして算出された評価値に基づき次の世代に残すべき個体を選択する．選択する個体数を小さくすると，解への収束が速くなる一方で，個体の多様性を失い初期収束により局所解に陥りやすくなる．なお，入力された全テストケースを通過する個体を規定数以上生成したとき，遺伝的アルゴリズムを終了する．

2.2 既存手法の課題

既存手法の課題を説明するために，FizzBuzz を題材とした具体的なテスト，及び生成個体の例を図 2 に示す．図では 4 つのテストケース，及び最低限のコンパイルのみが成功する初期状態のソースコードを APR に入力している．APR では図右上に示すように，個々の生成個体が塩基の集合から構成される単一の遺伝子を持つ．塩基と遺伝子は様々な設計が可能であるが，ここでは塩基は位置と操作の 2 つの要素から構成されている．図の右上の個体は，単一の塩基 B1 からなる遺伝子を持っており，その塩基の位置は 2 行目 `return ""`の箇所，塩基の操作内容は `return str(n)` による置換である．この遺伝子を適用し

た結果，4 つのテスト全てが失敗する状態から，1 つ目のテストが通過する状態に進化している．

ここで，APR 処理の中で 4 つの個体（$V1$〜$V4$）が生成されたとする．左上の $V1$ は最も巨大な遺伝子（つまり多くの塩基）を含んでおり，通過テストも 3 個と最も多い．よって $V1$ は次のソースコードの改変ループに生かすべき，最も良い個体であると解釈できる．APR のような探索問題では，単一個体ではなく複数の個体を選択することが一般的である．これは遺伝子の多様性確保，及び局所解を回避するための戦略である．

従来よく用いられるテスト通過数というスカラ値を評価値として用いた場合，テスト通過数 2 である $V2$ と $V3$ が $V1$ に続いて選択される．しかしテスト通過の内容や個体の持つ塩基を確認すると，この 2 個体は塩基，テスト通過の両観点で $V1$ の完全なサブセットとなっており，効率的な選択とはいえない．他方，左下の個体 $V4$ はテスト通過数自体は 1 と最も少ないものの，他の個体では失敗する 4 つ目のテストを通過している．また，他の個体にはない特殊な遺伝子 B4 を持っており，FizzBuzz 問題における 15 除算時の処理を実現している．このように，APR の個体評価においてテスト通過数という評価値はその表現能力が不足しており，個々のテストの成否という情報を個体ごとに比較する必要がある．

3. 提 案 手 法

APR を転用した自動プログラム生成の効率向上のために，APR への多目的遺伝的アルゴリズムの適用，及び多目的遺伝的アルゴリズムによる高い個体評価の表現性を利用した，相補的なテスト結果の 2 個体を選択的に交叉する手法を提案する．

3.1 多目的遺伝的アルゴリズムの適用

多目的遺伝的アルゴリズムとは，複数の評価関数を用いて個体を評価する遺伝的アルゴリズムの一種である．以下で，多目的遺伝的アルゴリズムに必要な複数の評価関数の設計とそれによる個体の選択方法を説明する．また，提案する評価関数を用いた相補的なテスト結果の 2 個体を交叉する手法についても述べる．

3.1.1 評価関数と個体の評価ベクトル

生成した個体を評価する評価関数について説明する．2.2 節で説明した選択の課題を解決するために個々のテストの成否を独立した評価関数とする．つまり，テストが M 個入力されたとき，評価関数は M 個となる．i 個目の評価関数 $E_i(v)$ は個体 v が i 個目のテストを通過すれば 1 を，失敗すれば 0 を返す．個体 v の評価ベクトルは $(E_1(v), E_2(v), \cdots, E_M(v))$ となる．M 個の評価関数の値が全て 1 となる個体が生まれたとき，すなわち，評価ベクトルの全ての成分が 1 のとき，全てのテストケースを通過する個体の生成を意味する．

3.1.2 個体の優越と順序関係の定義

個体 v_a，v_b に関して，$E_i(v_a) < E_i(v_b)\ (\forall i = 1, \ldots, M)$ のとき，v_b は v_a を優越するという．また，このとき，$v_a \prec v_b$ として個体の順序関係 $\prec$ を定義する．このとき，$\prec$ は半順序関係であり，個体の集合は束[2]となる．

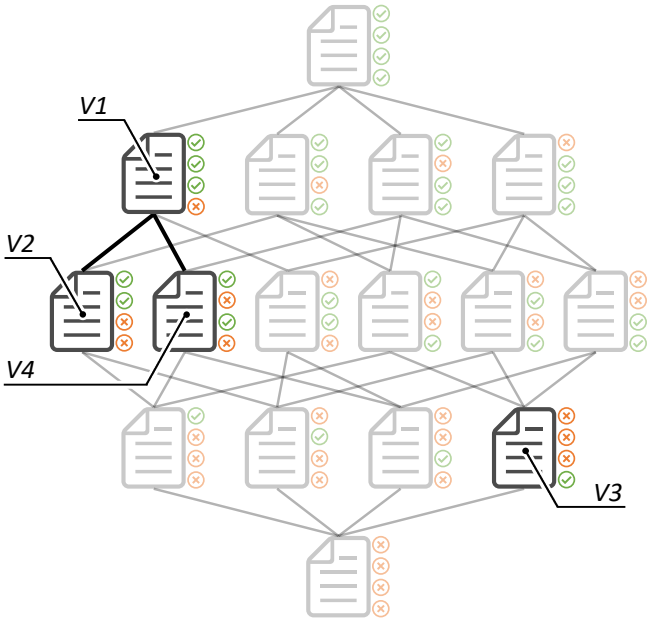

図 3: テスト数 4 の時の全個体のハッセ図と $V1$ から $V4$ の関係

3.1.3 選　　択

提案する選択では，各個体の評価ベクトルを元に次の世代に残す個体を決定する．具体的には，パレートランキング法 [14] により個体にランク付けを行い，個体数が上限となるまで，ランクに応じて個体を選択する．ここで，個体 v のランクは v が属する世代中の $v \prec v'$ となる個体 v' の個数 +1 とする．

図 3 に $M = 4$ での存在しうる全個体の関係をハッセ図で示す．本図では $v_a \prec v_b$ かつ $v_a \prec v_c \prec v_b$ なる v_c が存在しない場合のみ v_a から v_b に上向きに線を引いた．図中の最下段の個体は初期個体を表し，最上段の個体は全てのテストを通過する生成目標の個体である．本図の個体が全て同一世代中に存在する場合，それぞれの個体のランクは，最上段の個体が 1，2 段目の 4 つの個体が 2，3 段目の 6 つの個体が 4，4 段目の 4 つの個体が 8，最下段の個体が 16 となる．

図 2 に示した個体のランクは $V1$ と $V3$ が 1，$V2$ と $V4$ が 2 となる．$V1$ と $V3$ は図 3 において，上部に個体が存在しないが，$V2$ と $V4$ の上部には $V1$ が存在するからである．このランクに従うことで，$V3$ を $V2$，$V4$ に優先して選択できる．

個体の選択に必要なランク付けは個体が半順序集合であるので，全順序集合のようにソートができず簡単でない．最も素朴な方法として全探索が挙げられるが必要な計算量が大きい．そこで，高速化のため Deb らの高速非優劣ソート[3] [15] を用いる．

3.2 交叉の改善

本研究では，前節で述べた多目的遺伝的アルゴリズムの適用に加え，ソースコードを改変する操作の 1 つである交叉の改善も行う．本節では提案する新しい交叉法である直列交叉について説明する．直列交叉では前節で述べた評価ベクトルを用いて相補的なテスト結果を持つ 2 個体を選択的に交叉する．

(注2)：Lattice

(注3)：fast non dominated sort

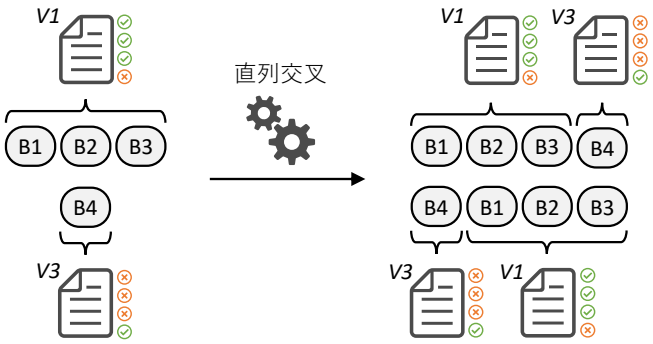

図 4: $V1$ と $V3$ を直列交叉して生成される遺伝子

3.2.1 直列交叉

直列交叉は，交叉する 2 個体の遺伝子を全て繋ぎ合わせることで個体を生成する交叉法である．交叉する 2 個体を v_a，v_b とすると，直列交叉は以下の処理からなる．

（1） v_a，v_b の順に遺伝子を繋ぎ合わせ，重複塩基を除く．
（2） v_b，v_a の順に遺伝子を繋ぎ合わせ，重複塩基を除く．
（3） これらを遺伝子とする個体を 2 体生成する．

直列交叉の特徴として，乱択によらないことや，常に 2 個体生成されること，交叉する 2 個体が重複する塩基を持たない場合に生成される個体の遺伝子長は交叉する 2 個体の遺伝子長の和となることが挙げられる．

図 2 の $V1$ と $V3$ を直列交叉することで，生成される 2 個体の遺伝子を図 4 に示す．$V1$ と $V3$ の持つ重複した塩基 B2 が交叉後には重複しないように取り除かれる．

重複する塩基を取り除く理由は，直列交叉において重複する塩基は個体に悪影響を与える可能性が高いからである．例として，変数宣言を挿入する塩基を考える．この塩基が重複した場合，同じ識別子の変数が 2 つ宣言され，個体のコンパイルに失敗する．

3.2.2 交叉する個体の選択

ランクが 1 の個体群のうち，評価ベクトルが一致しない 2 組の個体に対して直列交叉を行う．このような交叉により，2 個体の通過テスト全てを通過する個体の生成が期待できる． 図 2 に示した 4 個体の場合，ランク 1 の個体は $V1$ と $V3$ であるので，これらに対してのみ直列交叉を行う．

3.2.3 交叉によって生成された個体の検証

個体 v_a，v_b から直列交叉により個体 v_c が生成されたとき，$v_a \prec v_c$ かつ $v_b \prec v_c$ を満たす場合のみ，生成した個体 v_c を次の世代への選択対象とする．この理由は前節で述べたように，直列交叉は交叉する 2 個体の通過テスト全てを通過する個体の生成を目標とするからである．$V1$ と $V3$ を直列交叉した場合，生成された個体は 4 つ全てのテストを通過する場合のみ，次の世代への選択対象となる．

4. 実　　験

4.1 概　　要

本章では，提案手法を既存の APR ツールである kGenProg [16] を拡張して実装し，その効果を確認する．比較対象として拡張前の kGenProg を従来手法とする．本実験の目的は，提案手法によって自動プログラム生成の効率がどのように変化するかを確認することである．評価指標として，自動プログラム生成の成功数と所要時間を用いる．

4.2 実 験 設 定

実験設定の一覧を表 1 に示す．実験の題材として，プログラミングコンテスト AtCoder[4]で過去に開催された AtCoder Biginner Contest（ABC）のうち ABC101 から ABC180 までの 100 点問題 80 問を用いる．従来手法と提案手法はともに遺伝的アルゴリズムに基づきプログラムを生成するため，個体の生成や選択などで乱択を行う．よって，乱数シードに 1 から 10 までの 10 個を設定してプログラムの生成を試みる．

従来手法，及び提案手法の拡張元である kGenProg は再利用に基づく APR ツールであり，個体の生成時にはプログラム改変のために改変元となるソースコードが必要である．この再利用ソースコード片としては実験題材 80 問の全ての正解コードを取得し，そのネスト構造を全て平坦化して用いる．平坦化処理では `if` や `for` のブロック内の個々のステートメントをブロック外に移動する．これにより独立した小さなコードスニペットの再利用のみでプログラム生成が可能かを確かめる．なお，再利用コードには正解コードを構成する全ステートメントが含まれるため，十分な時間をかければ，理論的には正解コードを生成できる．両ツールに入力するテストケースは AtCoder 社の公開するテストケースを全て利用した．その他の遺伝的アルゴリズムの動作に必要なパラメータは kGenProg のデフォルト値を用いた．

4.3 実 験 結 果

4.3.1 生成に成功した題材数の比較

従来手法と提案手法それぞれに対する生成成功題材の個数を図 5 に示す．ここで，生成に成功した題材とは，入力した全テストケースを通過するプログラムを全 10 回の試行のうち 1 回でも生成できた題材とする．図中の左側の円は従来手法で生成に成功した題材の集合を表し，右側の円は提案手法で生成に成功した題材の集合を表す．従来手法と提案手法の両方で生成に成功した題材は 55 個，従来手法でのみ生成に成功した題材は 4 個，提案手法でのみ生成に成功した題材は 6 個，両者ともに生成できなかった題材は 19 個であった．

表 1: 実験設定

項目	値
実験題材	ABC101~ABC180 100 点問題
題材数	80
乱数シード	1~10 （= 10 試行）
制限時間	1 試行あたり 1 時間
再利用コード	全題材の正解コード片
最大世代数	無制限
終了条件	正解個体の発見・時間切れ
実験環境	Xeon E5-2630 2CPUs 16GB mem

（注4）：`https://atcoder.jp/`

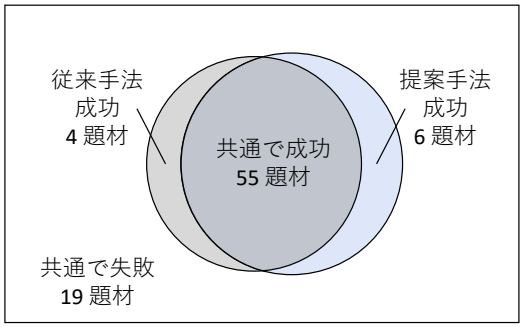

図 5: 生成に成功した題材数のベン図

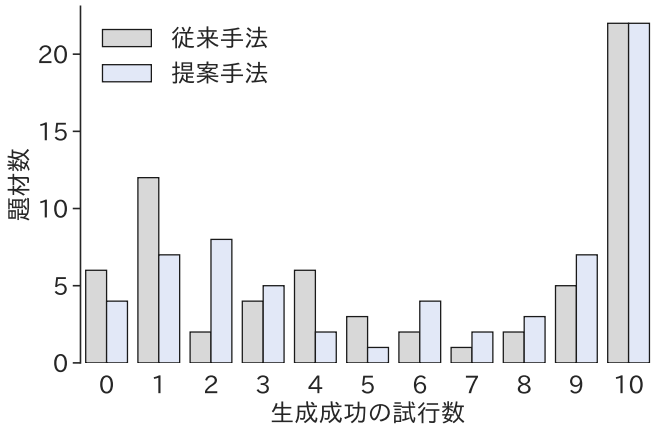

図 6: 生成に成功した試行数の比較

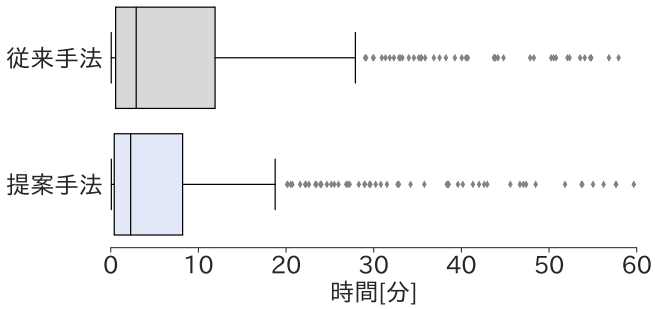

図 7: 生成時間の比較

全 80 題材に対する 10 試行中の成功試行数の頻度を図 6 に示す．左端の生成成功の試行数 0 の題材は生成に失敗した題材を，右端の生成成功の試行数 10 の題材は全ての試行で生成に成功した題材を意味する．生成に成功した試行数を比較すると，提案手法が従来手法を上回った題材は 28 題材，下回った題材は 15 題材であった．また，プログラムの生成に成功した試行の和は全 800 試行中で，従来手法は 367 試行，提案手法は 396 試行であり，提案手法が 29 試行だけ多い．共通で失敗した 19 題材を除く 61 題材において，各題材の代表値を生成に成功した試行数として Wilcoxon の符号順位検定により有意差を検定したところ，$p = 5.5 \times 10^{-2}$ であり，有意な差は認められなかった．

4.3.2 プログラム生成時間の比較

生成成功時に要した生成時間を図 7 に示す．本図では，全 800 試行のうち生成に成功した試行のみを抽出している．横軸は所要時間を表し，短いほどプログラム生成の効率が高いといえる．図より，生成時間の第一四分位数と中央値は従来手法と提案手法とで大きな差はないが，第三四分位数では従来手法は 11.9 分のところ提案手法は 8.2 分と，提案手法は従来手法に比べ約 1.45 倍速くなった．

共通で成功した 55 題材の題材ごとの生成時間の平均値を図 8 に示す．図の横軸は題材名であり，横軸は生成に成功したときの所要時間である．提案手法がより短い時間で生成に成功した題材は 42 題材，従来手法がより短い時間で生成に成功した題材は 13 題材であった．また，所要時間に 2 倍以上の差が見られた題材数について着目すると，提案手法が短い題材は 16 題材，従来手法が短い題材は 4 題材であった．共通で成功した 55 題材において，代表値を平均値として Wilcoxon の符号順位検定により有意差を検定したところ，$p = 1.60 \times 10^{-5}$ であり，有意な差が確認できた．

5. 考　　察

提案手法は従来手法に比べ短い時間で正解プログラムの生成に成功しており，提案手法は従来手法に比べて効率的にプログラムの生成を行えると結論付けられる．

生成に成功した題材数について，Wilcoxon の符号順位検定では有意な差が認められなかった．その理由として，全ての試行において生成に成功した題材が従来手法と提案手法ともに 55 題材中 22 題材 (40%) と大きな割合を占めていたからと考えられる．つまり，必要なソースコード片が少なくプログラム生成の難易度が低い題材の割合が大きいために，成功数について従来手法と提案手法の差を明らかにできなかった可能性がある．

プログラム生成時間については，Wilcoxon の符号順位検定により有意な差が認められた．これより，提案手法は従来手法と比較して高速にプログラムを生成したといえる．よって，提案手法によりプログラム生成の効率化が図られたと考えられる．

生成時間について，より詳細な考察を行う．図 8 より，従来手法において生成時間が短い題材ほど提案手法による効率の向上が得られにくい傾向が読み取れる．この理由を考察する．従来手法において生成時間が短い題材の具体的な題材として，ABC134 を挙げる．ABC134 は入力を a として $3a^2$ を出力する題材である．この題材では，必要なコード片は 1 つであり，その 1 つのコード片を挿入できれば全てのテストに通過する個体を生成できる．このような題材では生成個体は全てのテストを通過するか，全てのテストに失敗するかのどちらかとなるので，評価値に高い表現性は不要であり，従来手法と提案手法の差がつかなかったと考えられる．その一方，生成時間の差が最も大きい ABC120 では正解コードに分岐が必要であることから，2.2 節で説明した課題が従来手法において発生しており，提案手法により効率的な個体選択を行うことで，生成時間を短縮できたと考えられる．

6. おわりに

本稿では，個々のテスト結果を独立した評価関数とする多目的遺伝的アルゴリズムを適用した自動プログラム生成手法を提案した．また，プログラミングコンテストの問題 80 問を題材

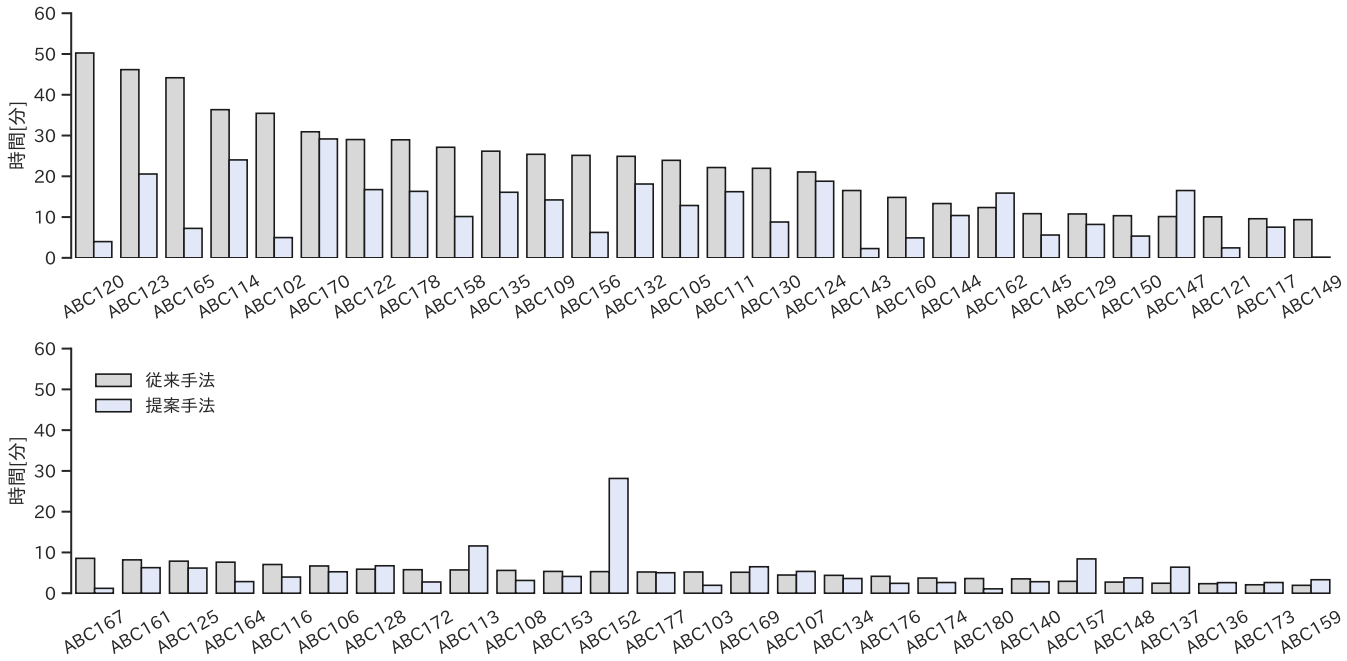

図 8: 題材ごとの生成時間の平均値

とした評価実験を行い，本手法の有効性を確認した．

今後の課題として，実験結果の詳細な分析が挙げられる．提案手法の生成時間が著しく悪化した ABC152 など各題材の性質が提案手法に与えた影響の分析を行う必要がある．また，本稿で提案した多目的遺伝的アルゴリズムによる選択法や直列交叉がどの程度効率向上に寄与したのか，その内訳は明らかになっていない．多目的遺伝的アルゴリズムでは局所解の回避という効果も得られるが，この効果が本手法で得られているかは確認できておらず，詳細な分析が必要である．手法の改善という観点では，テスト通過の可否に加え，コード行数やテスト時間などの指標の組み込みも検討している．また，直列交叉の改善として，全ての遺伝子を利用するのではなく，動的解析などにより，テストケースを通過させる有用な塩基のみを選別して交叉を行うことも考えられる．実験においては AtCoder の最も簡単な 100 点問題を題材としたが，より複雑な題材への適用実験は 1 つの課題である．

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] A. Desai, S. Gulwani, V. Hingorani, N. Jain, A. Karkare, M. Marron, and S. Roy, “Program Synthesis using Natural Language,” Proc. International Conference on Software Engineering, pp.345–356, 2016.

[2] S. Zhang and Y. Sun, “Automatically Synthesizing SQL Queries from Input-Output Examples,” Proc. International Conference on Automated Software Engineering, pp.224–234, 2013.

[3] M. Wen, J. Chen, R. Wu, D. Hao, and S. Cheung, “Context-Aware Patch Generation for Better Automated Program Repair,” Proc. International Conference on Software Engineering, pp.1–11, 2018.

[4] 富田裕也，松本淳之介，柗本真佑，肥後芳樹，楠本真二，倉林利行，切貫弘之，丹野治門，“遺伝的アルゴリズムを用いた自動プログラム修正手法を応用したプログラミングコンテストの回答の自動生成に向けて，” Proc. 情報処理学会研究報告，第 2020-SE-204 巻，pp.1–8，March 2020.

[5] W.E. Wong, R. Gao, Y. Li, R. Abreu, and F. Wotawa, “A Survey On Software Fault Localization,” IEEE Transactions on Software Engineering, vol.42, no.8, pp.707–740, 2016.

[6] L. Gazzola, D. Micucci, and L. Mariani, “Automatic Software Repair: A Survey,” IEEE Transactions on Software Engineering, vol.45, no.1, pp.34–67, 2017.

[7] F. Long and M. Rinard, “An Analysis of the Search Spaces for Generate and Validate Patch Generation Systems,” Proc. International Conference on Software Engineering, pp.702–713, 2016.

[8] E.K. Smith, E.T. Barr, C.L. Goues, and Y. Brun, “Is the Cure Worse Than the Disease? Overfitting in Automated Program Repair,” Proc. Joint Meeting on Foundations of Software Engineering, pp.532–543, 2015.

[9] Z. Qi, F. Long, S. Achour, and M. Rinard, “An Analysis of Patch Plausibility and Correctness for Generate-and-Validate Patch Generation Systems,” Proc. International Symposium on Software Testing and Analysis, pp.24–36, 2015.

[10] S. Saha, R.K. Saha, and M.R. Prasad, “Harnessing Evolution for Multi-Hunk Program Repair,” Proc. International Conference on Software Engineering, pp.13–24, 2019.

[11] 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.

[12] J.D. Knowles, R.A. Watson, and D.W. Corne, “Reducing Local Optima in Single-Objective Problems by Multi-objectivization,” Proc. International conference on evolutionary multi-criterion optimization, pp.269–283, 2001.

[13] M.T. Ahvanooey, Q. Li, M. Wu, and S. Wang, “A Survey of Genetic Programming and Its Applications,” KSII Transactions on Internet and Information Systems, vol.13, no.4, pp.1765–1794, 2019.

[14] C.M. Fonseca and P.J. Fleming, “Genetic Algorithms for Multiobjective Optimization: FormulationDiscussion and Generalization,” Proc. International Conference on Genetic Algorithms, pp.416–423, San Francisco, CA, USA, 1993.

[15] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan, “A fast and elitist multiobjective genetic algorithm: NSGA-II,” IEEE Transactions on Evolutionary Computation, vol.6, no.2, pp.182–197, 2002.

[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,” Proc. Asia-Pacific Software Engineering Conference, pp.697–698, 2018.