Vol.2020-SE-204 No.8
2020/3/2

過去の実装情報を活用したプログラム自動生成

倉林 利行[1,a)]　吉村 優[1,b)]　切貫 弘之[1,c)]　丹野 治門[1,d)]　富田 裕也[2,e)]　松本 淳之介[2,f)]
まつ本 真佑[2,g)]　肥後 芳樹[2,h)]　楠本 真二[2,i)]

概要：本論文ではソフトウェア開発における実装の自動化に向けたファーストステップとして，プログラミングコンテスト AtCoder の正解プログラムを自動生成する技術の開発を目指す．自動プログラミングの既存研究としては，生成したいプログラムの入出力例からプログラムを合成する手法などが存在するが，入出力例は満たすが正しいプログラムではないというオーバーフィッティングしたプログラムが生成されてしまうという課題が存在した．本論文では過去の問題情報から解きたい問題と類似した問題を検索して取得し，その解答プログラムを雛形としてプログラムを合成することで，正解プログラムを生成する手法を提案した．提案手法は AtCoder の配点が 100 点の問題 5 問に対して評価を行い，4 問の正解プログラムを自動生成できることを確認した．

キーワード：自動プログラミング，プログラム合成，遺伝的プログラミング，プログラミングコンテスト

1. はじめに

経済産業省によると，2025 年には IT 人材が約 43 万人不足すると言われている（2025 年の崖）[*1]．そのためソフトウェア開発の自動化が強く求められている．特に決められたテスト項目に従って行うテストや，詳細設計後の実装にかかる工数を削減することができれば，人の工数をより創造的な活動へとシフトすることができる．テストの自動化を狙ったツールは数多く存在する．単体テストでは JUnit[*2]が広く使われており，E2E テストでは Selenium[*3]などが有名である．一方で実装の自動化を狙ったツールは少ない．最近では入出力例からそれを実現する関数を合成する FlashFill[1] があるが，ドメインが Excel に絞られている．他には GitHub[*4]のリポジトリを学習することでより高度なコード補完が可能となる IntelliCode[*5]も存在するが，コーディングの支援にとどまる．本研究ではあらゆる開発現場の実装の作業を自動化することで，ソフトウェア開発にかかる工数を大幅に削減することを最終的な目標とする．そのためのファーストステップとして，本論文ではプログラミングコンテストの問題の自動解答を目指す．プログラミングコンテストは自然言語で書かれた問題文やいくつかの入出力例から，プログラムを作成することでプログラミングの勉強をするコンテストである．もし問題文や入出力例からプログラムを自動生成することができれば，将来は要求やテストケースからプログラムを自動生成できる可能性がある．プログラミングコンテストは AtCoder[*6]を使用し，配点が 100 点の問題（以下 100 点問題と呼ぶ）を対象とした．本研究の貢献は以下の 2 点である．

- プログラミングコンテスト（AtCoder）の 100 点問題に対して，過去の問題情報を活用して自動で正解プログラムを生成する手法を提案した．
- 提案手法は AtCoder の 100 点問題 5 問に対して評価を行い，5 問中 4 問の正解プログラムを自動生成できることを確認した．

2. Motivating Example

2.1 AtCoder について

図 1 に用いられている問題は，AtCoder Beginner Con-

[1] NTT ソフトウェアイノベーションセンタ
〒 108-0023 東京都港区芝浦 3 丁目 4-1 グランパーク 33F
[2] 大阪大学 大学院情報科学研究科 コンピュータサイエンス専攻
〒 565-0871 大阪府吹田市山田丘 1-5
[a)] toshiyuki.kurabayashi.rb@hco.ntt.co.jp
[b)] yuu.yoshimura.zk@hco.ntt.co.jp
[c)] hiroyuki.kirinuki.ad@hco.ntt.co.jp
[d)] haruto.tanno.bz@hco.ntt.co.jp
[e)] y-tomida@osaka-u.ac.jp
[f)] j-matumt@ist.osaka-u.ac.jp
[g)] shinsuke@ist.osaka-u.ac.jp
[h)] higo@ist.osaka-u.ac.jp
[i)] kusumoto@ist.osaka-u.ac.jp

[*1] https://www.meti.go.jp/
[*2] https://junit.org/junit5/
[*3] https://selenium.dev/
[*4] https://github.com/
[*5] https://visualstudio.microsoft.com/services/intellicode/
[*6] https://atcoder.jp/

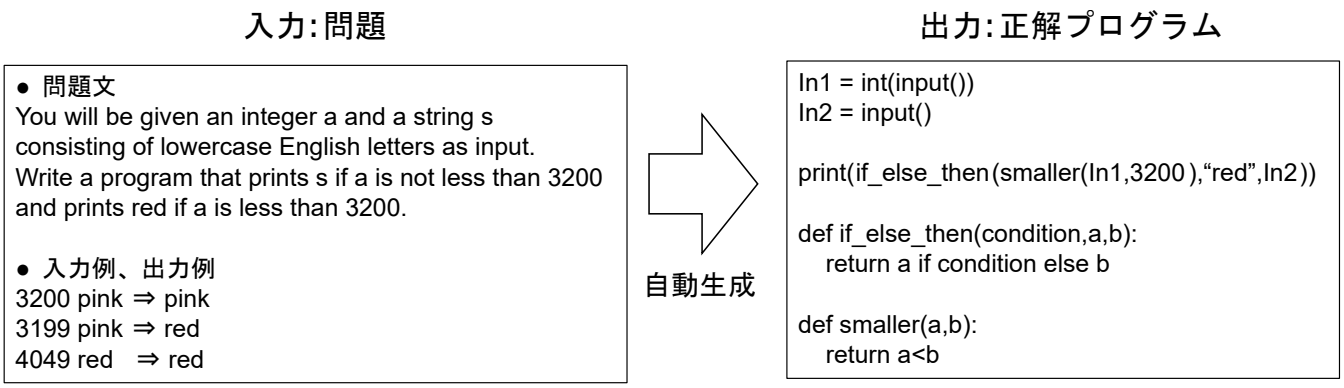

図 1 Motivating Example

test（以下 ABC） 138 の実際の 100 点問題の問題文と入出力例を抜粋したものである．問題には点数ごとに一定の出題傾向があり，100 点問題であれば四則演算を組み合わせた計算をするプログラム，指定の条件に沿って分岐するプログラム，入力された文字列に操作を加えて出力するプログラムを作る問題などが存在する．通常，ユーザは問題に書かれている情報を読み取って，その問題が要求するプログラムを作成し，AtCoder に提出する．提出後は AtCoder のサーバ側で自動的にテストが実行され，すべてのテストがパスすれば正解プログラムとして認定される．問題にも入出力例が用意されているが，それを満たすことが正解プログラムとなる条件ではない．また，サーバ側で実行されるテストの情報は，ユーザは確認することができない．

2.2 本研究の条件

本論文では図 1 のように，問題文と入出力例から正解プログラムを自動生成することを目指す．問題には他にも入力や出力について書かれた項目（どのような値を出力するかなど）や，入力制約について書かれた項目が存在するが，それらの情報は用いないものとする．理由としては本研究の最終的な目標はプログラミングコンテストの問題を解くことではなく，実際の開発現場の実装作業を自動化して工数削減をすることであり，入力とする情報は現場にすでに存在する，もしくは容易に作成できる情報を前提すべきだからである．問題文は要求，入出力例はテストケースと置き換えることができ，これらの情報は現場に存在するが，プログラムの入力，出力，入力制約に関する詳細な設計情報は必ず存在するものではない．したがって本研究では問題文と入出力例のみを用いるものとする．AtCoder では日本語版と英語版，両方の問題が用意されているが，本論文では英語版のみを用いるものとする．また AtCoder では過去の問題が公開されており，それらの情報を使用しても良いものとする．（実際の開発現場においても，GitHub 上のソースコードや過去のプロジェクトの実装情報は入手可能である．）出力するプログラムは Python 形式のものを対象とする．

3. 従来手法と課題

本章では，自動でプログラムを生成する手法の既存研究とその課題について記す．

3.1 自然言語を用いたプログラム自動生成

自然言語で記述された仕様からプログラムを自動生成する手法が存在する [2][3]．Juan Zhai らの研究 [2] では，ドキュメントは存在するが実装が不明なライブラリのメソッドを理解することを目的とし，ドキュメントの情報から実装が不明なライブラリのメソッドと同一の挙動をするプログラムの自動生成を実現している．はじめに Javadoc 内の自然言語情報から事前に設定したルールを用いてプログラムの部品を生成し，プログラムを合成する．続いてソースコードを入手できないライブラリのメソッドから Randoop[4] を用いてテストを自動生成し，合成されたプログラムに対して実行することで同一性を確認する．しかしこの手法は，実際に正しい挙動をするメソッドが存在することが前提となるため，本研究の対象のように新しくプログラムを生成するケースには適用することはできない．Desai ら [3] は，自然言語で記述された仕様からプログラムの部品を生成し，その部品を用いてプログラムを合成する手法を提案している．一般的に曖昧性のある自然言語からプログラムを的確に自動生成することは難しいが，この手法ではドメインを狭く絞り，各ドメインに特化した Domain Specific Language（以下 DSL）を設計することで精度を高めている．具体的なドメインとしては，テキスト編集において一文程度の自然言語で表されるプログラムの仕様（数値から始まる行の最初の文字を削除するなど）からそれを実現するためのプログラムを合成する等である．プログラムは複数のパターンが合成され，事前に学習をした自然言語とプログラムの対応関係を学習したモデルによってランキング付けして出力することができる．しかし，この手法では入力が自然言語の情報のみであるため，実現したい仕様を漏れなく的確に記述する必要がある．また，情報の漏れだけでなく実現したい仕様と関連性が低い情報が存在し

ても正しいプログラムが合成されないと考えられる．なぜなら情報の重要度を自動で判別することは難しく，本来不要な情報にもプログラム合成は影響を受けるためである．このように入力となる仕様情報を記した自然言語は，非常に制限の強いものとなる．AtCoder の場合，問題文は要求であり仕様ではない．問題文を自動で仕様に変換するためには，高度な意味解析技術が必要であり，現在の技術では困難である．例えば図 1 の問題の場合，問題文の要求を満したプログラムを作成する上で，一行目の「lowercase」という情報は必須ではない．しかし問題文だけからプログラムを生成しようとすると，「lowercase」の影響を受けたプログラムが生成されてしまう可能性がある．具体的には「lowercase」という単語に反応し，文字列を小文字へと変換するプログラムなどが生成されうる．そういった問題を防ぐには問題文から重要な箇所だけを抜き出す必要があるが，そのためには意味解析を高い精度で行う必要があり困難である．したがって AtCoder において自然言語（問題文）からプログラムを自動で生成する場合の課題は以下のようになる．

課題 1：AtCoder の問題文からプログラムを自動生成する場合，問題文を適切な仕様へと変換する必要があり困難

3.2 入出力例を用いたプログラム自動生成

実現したいプログラムの入出力例からプログラムを自動生成する手法が存在する [1][5][6][7]．これらの手法は Programming by Example (以下 PbE) と呼ばれ，プログラム合成の中では最も盛んに研究が行われている分野である．SQLSynthesizer[5] や SCYTHE[6] では，表形式の入出力例からそれを実現する SQL クエリを自動生成する．これらの手法は求められる SQL クエリが複雑になるほど，合成に必要な計算量が爆発的に増えるといった欠点が存在する．そこでプログラム合成の効率化を目指した手法として，DeepCoder[7] が存在する．DeepCoder では，リスト型の入出力例と，その入出力例を実現するために使用されるプログラム内の関数の関係性を学習することで，与えられた入出力例に対して使用される関数を予測して合成の効率化を実現している．既存研究における PbE の特徴として，入出力例が表形式やリスト形式のプログラムが対象となることが挙げられる．これは入出力例が表形式やリスト形式の場合，情報量が多いことにより実現したい仕様を伝えやすいためである．入出力例が bool，int，float，string 型のみの場合，入出力例の情報が少ないため作成したいプログラムの仕様情報を十分に伝えることができず，入出力例にオーバーフィッティングしたプログラムが合成されてしまう．例えば図 1 の問題の場合，以下のようなプログラムは不正解であるにも関わらず入出力例を満たしてしまう．

```
In1 = int(input())
In2 = input()

if(In1!=3200):
    print("red")
else:
    print(In2)
```

このように入出力例の情報量が少ない場合，容易にオーバーフィッティングしたプログラムが生成されてしまう．AtCoder の 100 点問題の場合，入出力は int，float，string 型のいずれかとなる．その上，入出力例の数も多くの場合において 2 から 3 セットとなるため，上記のような問題が発生しやすい．したがって AtCoder の入出力例からプログラムを自動で生成する場合の課題は以下のようになる．

課題 2：AtCoder の入出力例からプログラムを自動生成する場合，オーバーフィッティングしたプログラムが生成されてしまう

4. 提案手法

本論文では過去の問題の情報を活用することで，正解プログラムを自動生成する手法を提案する．具体的には，以下のステップで問題の情報からプログラムを自動生成する．

Step 1：解きたい問題の問題文から過去の問題の類似問題を検索し，その解答プログラムを取得（プログラム検索部）

Step 2：取得した解答プログラムを雛形とし，入出力例をすべて満たすようにプログラムを合成（プログラム合成部）

提案の全体像を図 2 に示す．提案手法は 2 つの特徴がある．1 つ目の特徴は問題文から直接プログラムを生成するのではなく，類似の過去の問題を検索するために使用しているため，問題文を仕様に変換する処理が不要な点である．それにより高度な意味解析技術は不要となり，類似文章の検索技術という一定の成果が出ている技術 [8] を用いれば良いため，課題 1 を解決することが可能となる．2 つ目の特徴は類似の問題の解答プログラムを雛形としているため，正解プログラムに構造的に近いと想定されるプログラムから合成を行っている点である．それにより，入出力例にオーバーフィッティングしたプログラムが生成される可能性を減らすことができると考えられ，課題 2 を解決することが可能となる．以上の特徴により，提案手法では従来手法の課題を解決することが可能となり，正解プログラムが生成できると考えられる．なお，提案手法は AtCoder に一定の出題傾向がある点を利用しており，過去に類似の問題があることを前提としている．以下に提案手法の詳細なアルゴリズムを述べる．

4.1 Step 1：プログラム検索部

複数の文書の中から，与えられた文書に最も類似した文

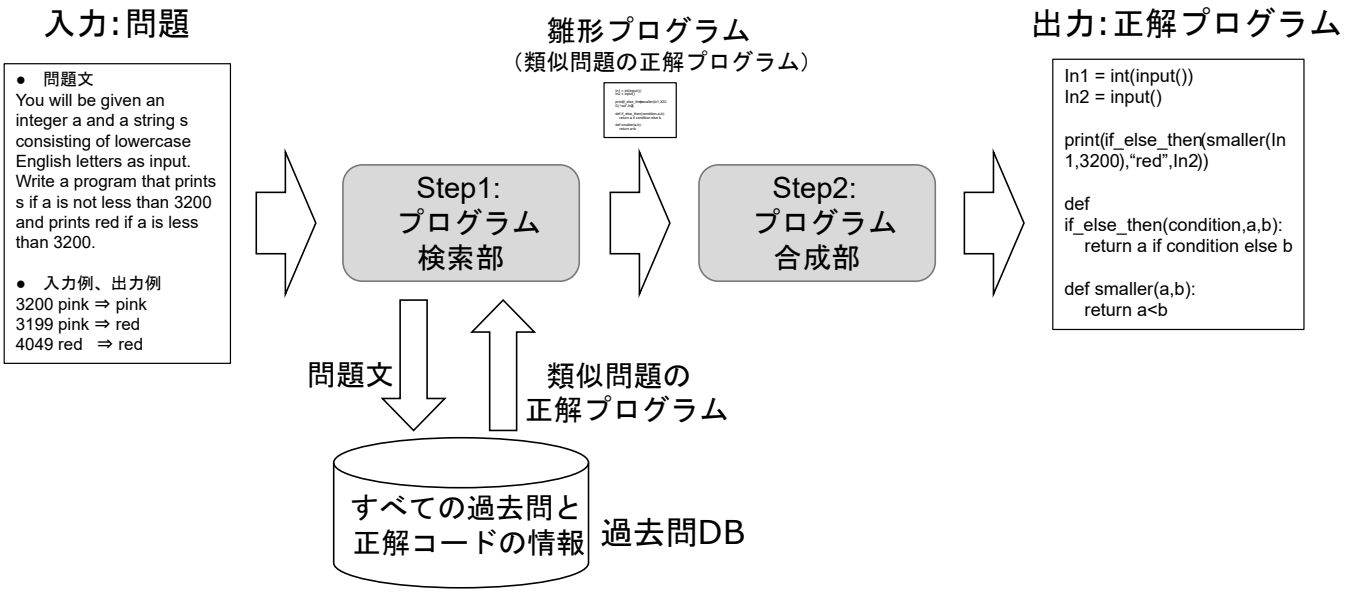

図 2 提案手法

書を検索する手法として，TF-IDF[8] を用いた手法が存在する．TF-IDF とは，ある文書内の単語の出現頻度と逆文書頻度をかけ合わせた値であり，単語の重要度を評価することができる．この値を各文書において，全文書内に存在する全単語で計算することでその文書を特徴づけるベクトルを得ることができる．２つの文書同士においてその特徴ベクトルのコサイン類似度を計算することで，文書同士の類似性を定量的に測定することができる．提案手法ではこの TF-IDF に基づく文書同士の類似度測定方法を問題文へ適用することで，過去の問題から類似した問題を取得する．具体的には過去の問題とその正解プログラムの情報をすべて保有したデータベース（過去問 DB）を事前に用意する．続いて解きたい問題の文書と過去問 DB 内のすべての問題の類似度をそれぞれ測定し，もっとも類似度が大きい過去の問題を取得して，その正解プログラムを得る．

4.2 Step ２：プログラム合成部

プログラム合成における合成プログラムの探索方法は，設計した文脈自由文法に基づいて合成することで探索空間を絞る方法や，ヒューリスティックに基づいて探索する方法などが存在する [9] が，本論文ではより多くの種類の問題を解ける可能性を高めるために，ヒューリスティックに基づいて探索する方法を採用する．ヒューリスティックに基づいて探索する方法では，遺伝的プログラミング [10][11] と呼ばれる手法が存在する．遺伝的プログラミングは，木構造で表されるデータを対象とし，設計した評価関数を満たすような個体を遺伝的アルゴリズムによって探索する手法である．プログラムを木構造のデータとして扱うことで，遺伝的プログラミングを適用することができる．例えば図 1 の正解プログラムは，木構造で表すと図 3 のように

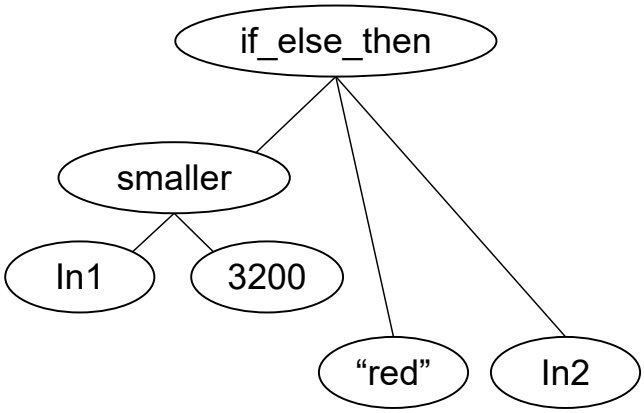

図 3 木構造で表されたプログラムの例

なる．遺伝的プログラミングでは，個体は各プログラム，遺伝子は個体の中の各メソッド，変数，定数となる．遺伝的プログラミングは，対象に合わせて評価関数の設計方法や選択，交叉，突然変異の方法を決める必要がある．以下に本論文で用いた遺伝的プログラミングの設定や，アルゴリズムを述べる．

4.2.1 用意する遺伝子の種類

合成で用いる遺伝子（メソッド，変数，定数）は種類が多いほど表現できるプログラムの幅が広がるが，一方で探索空間が指数関数的に増大するため，必要最小限に絞る必要がある．本研究では 100 点問題でよく用いられる，表 1 に示される 25 個のメソッドを用いた．n から始まる引数は int，float を問わない数値型，s から始まる引数は string 型，b から始まる引数は bool 型，v から始まる引数はあらゆる型の入力を想定している．この想定された型の情報は後の合成の時や評価の時に用いる．定数は 0 から 2 の int 型の整数と，問題文中に存在する数値を用いている．例え

表 1　合成で用いたメソッド一覧

メソッド	処理
sum(n_1,n_2)	return n_1+n_2
minus(n_1,n_2)	return n_1-n_2
multi(n_1,n_2)	return n_1*n_2
div(n_1,n_2)	return n_1/n_2
quo(n_1,n_2)	return n_1//n_2
mod(n_1,n_2)	return n_1%n_2
larger(n_1,n_2)	return n_1>n_2
larger_equal(n_1,n_2)	return n_1>=n_2
smaller(n_1,n_2)	return n_1<n_2
smaller_equal(n_1,n_2)	return n_1<=n_2
equal(v_1,v_2)	return v_1==v_2
not_equal(v_1,v_2)	return v_1!=v_2
max(n_1,n_2)	return max(n_1,n_2)
min(n_1,n_2)	return min(n_1,n_2)
abs(n_1)	return abs(n_1)
splite(s_1,n_1)	return s_1[n_1]
concat(s_1,s_2)	return s_1+s_2
count(s_1,s_2)	return s_1.count(s_2)
replace(s_1,s_2,s_3)	return s_1.replace(s_1,s_2,s_3)
length(s_1)	return len(s_1)
upper(s_1)	return s_1.upper()
lower(s_1)	return s_1.lower()
and(b_1,b_2)	return b_1 and b_2
or(b_1,b_2)	return b_1 or b_2
if_then_else(b_1,v_1,v_2)	return v_1 if b_1 else v_2

ば図 1 の問題の場合，3200 という int 型の定数が合成の時に用いられる．その他，出力例の string 型の定数と bool 型の True も合成の時に使用する．入力変数の数は入力例から推測し，In1，In2…という名前で使用している．

4.2.2　評価関数

合成されたプログラムを評価する時に用いられる評価関数について説明する．評価関数の設計は，個体の進化を効率的に進める上で重要である．遺伝的アルゴリズムによるバグ自動修正技術 GenProg[12] では，評価関数を入出力例（テストケース）を満たした数と設計している．しかしこの設計方法では，各プログラムがどの入出力例を満たしているのかという情報が消えてしまうため，本論文では評価関数を入出力例ごとに設計している．（評価関数を入出力例ごとに設計することによる利点は，4.2.4 項で述べるものとする．）評価関数 $E(x)$ は，出力が数値型の時は期待結果（出力例）の値と実際に得られた値の差の二乗，それ以外の型の場合は一致していれば 0，していなければ 1 を返す．つまり入出力例を満たせば 0，それ以外なら 0 より大きい値を返す．ただしプログラムが以下の条件を最低 1 つでも満たす場合は，戻り値に関わらず評価関数 $E(x)$ の値を最悪値（非常に大きい値の任意の定数）とする．

条件 1　最大メソッド使用数以上のメソッドが使用されている

条件 2　メソッドの引数の型が表 1 で想定されている型ではない

条件 3　実行時にエラーが発生する

これらの条件はすべて探索空間を狭めることが目的である．条件 1 はメソッドを必要以上に組み合わせた方向への探索を防ぐために導入した．条件 2，3 は正解プログラムから離れた方向への探索を防ぐために導入した．$E(x)$ の値は，入出力例ごとに保持する．入出力例が n セットある場合は $E(x_1), E(x_2)...E(x_n)$ のようになる．これらの値がすべて 0 になれば，全部の入出力例を満たすプログラムが合成されたことになり，そのプログラムを出力して遺伝的プログラミングを終了する．

4.2.3　初期生成個体

遺伝的プログラミングを開始する時にはじめに用いる個体は，4.1 節で取得した過去問の解答プログラムを用いる．ただし場合によっては引数の数が異なるため，すべての引数を組み合わせたパターンのプログラムを生成し，それらをすべて初期生成個体とする．例えば元のプログラムが $sum(In1, In2)$ で解きたい問題の入力の数が 3 個の場合 sum(In1,In1)，sum(In1,In2)，sum(In1,In3)，sum(In2,In1)，・・・sum(In3,In3) の合計 9 パターンが初期生成個体となる．初期生成個体を実行して評価関数の値を得た後は，選択，交叉，突然変異を経て次の世代へと移る．

4.2.4　選択

選択では，評価関数の値を元に次の世代へと残す個体を決定する．選択はトーナメント方式 [11] を用いている．トーナメント方式では全個体の中から無作為に一定数（トーナメント選択数）の個体を選択し，その中で最も評価が良かったものを次世代に残す．1 回目のトーナメントは，$E(x_1)$ の値が最も優れている（0 に近い）個体を選択する．$E(x_1)$ の値が同一の場合は，他の評価関数の値が 0 となっている数が多い個体を選択する．その数も同一の場合は，無作為に選択する．2 回目のトーナメントでは，$E(x_2)$ の値を参照にする．n 回目のトーナメントでは $E(x_n)$ の値，n+1 回目のトーナメントでは，$E(x_1)$ の値に戻る．これを n の値があらかじめ設定した総個体数の数になるまで繰り返す．このように各入出力例ごとに設定された評価関数の値によって個体を選択することで，個体にバリエーションが生まれるという利点がある．

4.2.5　交叉

交叉では，2 つの個体の部分木を組み合わせて新しい個体を作り出す．交叉は One-Point Crossover[11] を用いている．ただし，Fault-Localization[13] の技術を用いて交叉する箇所を絞っている．Fault-Localization はテストが失敗した時に実行されたプログラムの箇所にバグがあると推定する技術である．この考え方を本手法にも適用している．具体例を図 4 に示す．図 4 のような分岐のあるプログラムにおいて，入出力例を満たした時に通過したパス（正）と，

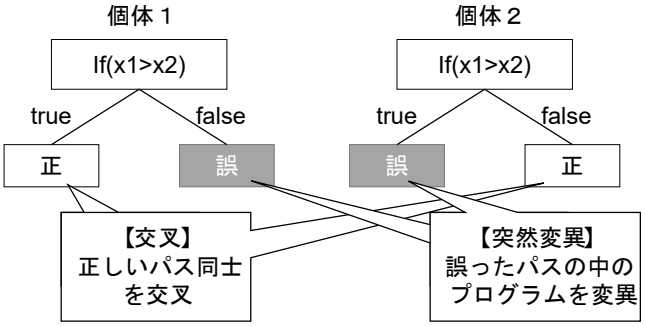

図 4　Fault-Localization を用いた交叉と突然変異の例

満たさなかった時に通過したパス（誤）の情報を記録する．交叉の時は，ベースとなる片方の個体のパス（誤）の中から置き換える部分木をランダムに選択する．その後，もう片方の個体のパス（正）の中から部分木をランダムに選択し，ベースとなる個体の中で選択した部分木に対する置き換えを実行する．これにより，誤っていると推測される部分木を正しいと推測されるプログラムの部分木に置き換えることができるため，正しいプログラムが生成される確率が上がると考えられる．交叉は交叉率×総個体数の数だけ行われる．

4.2.6　突然変異

突然変異では個体の部分木の一部を，ランダムに生成した部分木に置き換える．交叉と同様に，Fault-Localization[13]の技術を用いて突然変異する箇所を絞っている．具体例を図 4 に示す．交叉の時と同様，パス（誤）の中で置き換えられる部分木をランダムに選択し，新しく生成した部分木に置き換える．部分木は表 1 のメソッドや用意された定数をランダムに組み合わせて生成する．これにより，誤っていると推測される部分木を新しく生成した部分木に置き換えることができるため，正しいプログラムが生成される確率が上がると考えられる．突然変異は突然変異率×総個体数の数だけ行われる．

5.　評価

5.1　評価方法

AtCoder における ABC134 から 138 までの実際の 100 点問題 5 問を用いて，提案手法の評価を行った．提案手法の特徴は，解きたい問題において過去の類似問題を検索し，その解答プログラムを雛形としてプログラムの合成を行うことで，正解プログラムが生成される確率を上げることである．したがって提案手法の有効性を確認するために，以下の 2 つの評価を行った．

- プログラム検索部でどれくらい正解プログラムに類似した雛形プログラムが得られたか（プログラム検索性能の評価）
- AtCoder の実際の 100 点問題において，提案手法が正解プログラムをどれくらい生成できたか（過去問を用いたプログラム合成性能の評価）

表 2　実験で使用したパラメータ

最大世代数	5000
総個体数	300
トーナメント選択数	3
交叉率	0.5
突然変異率	0.5
最悪値	999999999
最大メソッド使用数	8

以下に各評価方法について説明する．

5.1.1　プログラム検索性能の評価

プログラム検索部によって得られた雛形プログラムが，解きたい問題の正解プログラムに対してどれだけ類似しているかを評価する．類似性の指標は，木構造のデータ同士の編集距離を表す，Tree Edit Distance[14]（以下 TED）を用いた．TED は 2 つの木構造のデータを同一にするまでに必要な，葉の追加・削除・置換の 3 操作の最小数を表す．TED によって木構造のデータの構造的な類似度を測定することができる．遺伝的プログラミングでは，部分木を追加・削除・置換するため，雛形プログラムと正解プログラム間の TED の値が小さいほど雛形として優秀だと評価することができる．したがって，解きたい問題の正解プログラムと過去の問題の解答プログラムすべてに対して TED の値を測定して値が小さい順番にランク付けし（以下類似度順位と呼ぶ），プログラム検索部がもっとも類似したと判定した過去の問題の解答プログラム（雛形プログラム）のそのランキング内での順位の評価を行った．TED の値が同一の場合は，同一順位とした．例えば過去問 DB に過去問が 3 つ存在し，それぞれ TED の値が 1,1,2 の時，順位はそれぞれ 1 位，1 位，2 位となる．プログラムの構造的な類似度を測りたいため，各プログラムの定数，入力変数はすべて正規化してから TED の測定を行った．具体的には数値型の定数は const-num，文字列型の定数は const-string，入力変数は In という名前に変換した．過去問 DB には ABC042 から ABC133 の問題と正解プログラムを用意した．過去問 DB 内のすべての正解プログラムは，表 1 に存在するメソッドと任意の定数を組み合わせて作成した．なお ABC042 以降の問題から配点という仕組みが導入されたため，それより以前の問題は過去問 DB に含んでいない．

5.1.2　過去問を用いたプログラム合成性能の評価

提案手法によって正解プログラムが自動生成できた問題の数を，過去問（雛形プログラム）を用いずにプログラム合成を行った場合（以下従来手法）と比較して評価する．従来手法の初期生成個体は，表 1 のメソッドや用意された定数をランダムに組み合わせて生成したプログラムとする．各手法において，入出力例をすべて満たすまでプログラム合成を行い，出力されたプログラムを目視で確認して正解

表 3　プログラム検索部の検索性能の結果

	検索で取得したプログラムの問題番号	検索で取得したプログラムの類似度順位	過去問 DB の全プログラムの平均類似度順位
ABC134	ABC116	1 位	7.425 位
ABC135	ABC109	2 位	5.5125 位
ABC136	ABC072	2 位	7.5125 位
ABC137	ABC098	1 位	8.4375 位
ABC138	ABC130	3 位	7.4125 位

表 4　AtCoder の問題に対する解答プログラム自動生成の結果

	従来手法	提案手法
ABC134	○ (7)	○ (2)
ABC135	× (-)	× (-)
ABC136	× (1007)	○ (10)
ABC137	× (40)	○ (1)
ABC138	× (37)	○ (225)

かどうか判定した．なお 5000 世代に到達しても合成が終わらない場合は，合成失敗とした．プログラム合成は遺伝的プログラミングのライブラリ DEAP[*7]に追加実装をしたものを用いた．実験で使用した遺伝的プログラミングの各種パラメータは表 2 に示す．

5.2　評価結果

5.2.1　プログラム検索性能の評価

プログラム検索性能の評価結果を表 3 に示す．検索で取得したプログラムの問題番号は，プログラム検索部が過去問 DB の中から解きたい問題に最も類似していると判定した問題の番号である．検索で取得したプログラムの類似度順位は，プログラム検索部が過去問 DB の中から解きたい問題に最も類似していると判定した問題の解答プログラムの類似度順位を表す．この順位が高いほど解きたい問題に対する正解プログラムに類似したプログラムが取得できたことになる．過去問 DB には問題が全部で 92 問あるが同一 TED 値を持つ問題が多かったため，過去問 DB の全プログラムの平均類似度順位はすべて一桁台となった．表 3 に示す通り，すべての問題においてプログラム検索部が取得したプログラムは平均類似度順位を上回り，3 位以内に収まるなど高性能な検索結果を得ることができた．

5.2.2　過去問を用いたプログラム合成性能の評価

過去問を用いたプログラム合成性能の評価結果を表 4 に示す．○は正解プログラムの生成に成功，×は失敗を表す．括弧内の数値は入出力例をすべて満たすプログラムが生成された時の世代数（”-”は最大世代数を超えたもの）を表す．従来手法では 5 問中 1 問しか正解プログラムが生成できなかったのに対し，提案手法では 4 問の正解プログラムを生成することができた．

[*7] https://github.com/deap/deap

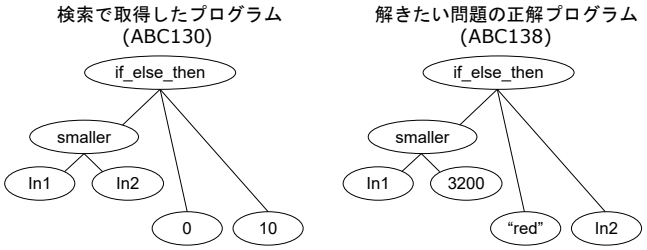

図 5　検索で取得したプログラムと解きたい問題のプログラム (ABC138)

5.3　考察

5.3.1　プログラム検索性能の評価

検索によって，具体的にどのような問題とプログラムが取得できたか，ABC138 の結果を用いて確認する．ABC138 の問題文は図 1 に示すとおりだが，最も類似していると判定された ABC130 の問題文は以下のようになる．

```
X and A are integers between 0 and 9(inclusive).
If X is less than A, print 0;
if X is not less than A, print 10.
```

どちらも指定の条件において，指定の値を出力するという問題であり，類似した問題が取得できていることがわかる．またそれぞれの問題の正解プログラムを図 5 に示す．類似した問題同士であるため解答プログラムの構造も類似しており，雛形として使用するために適したプログラムが取得できていることがわかる．

5.3.2　過去問を用いたプログラム合成性能の評価

提案手法によって，従来手法より多くの正解プログラムが生成できるようになった理由を考察する．従来手法の特徴として，4 問は入出力例をすべて満たすプログラムが生成されたにもかかわらず，3 問が正解プログラムではなかったことが挙げられる．これは課題 2 で示した，オーバーフィッティングの問題が発生したと考えられる．例えば ABC138 で従来手法が生成したプログラムを図 6 に示す．Python の仕様として True==1 は True を返すため，図 1 に示す入出力例をすべて満たすプログラムとなる．このように雛形を用いずに合成を行うと，問題文に合わないプログラムが生成されてしまう．一方提案手法では図 5 に示すような正解プログラムに類似したプログラムを雛形として合成を開始したため，正解プログラムが生成できたと考えられる．続いて提案手法では正解プログラムを生成することができなかった，ABC135 について考察する．ABC135 において検索で取得したプログラム（ABC109）と正解プログラムを図 7 に示す．図 7 どちらのプログラム

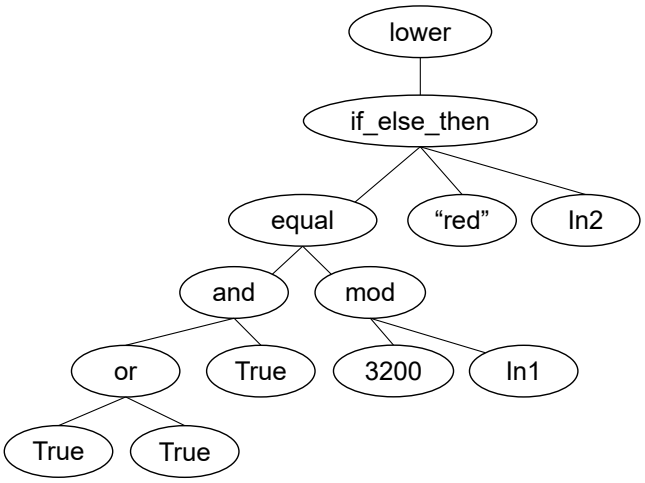

図 6　従来手法が生成した ABC138 のプログラム

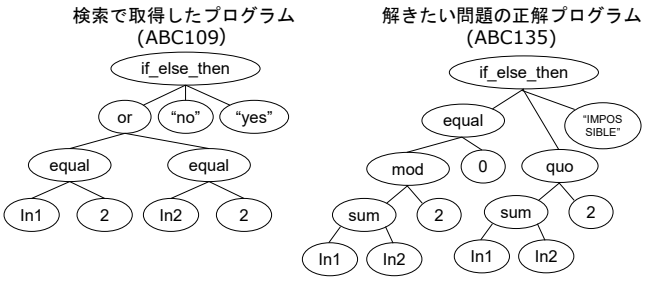

図 7　検索で取得したプログラムと解きたい問題のプログラム (ABC135)

も if 文を用いた分岐構造になっているが，条件文や True の時の戻り値の構造が大幅に異なる．そのため，正解プログラムにたどり着くまでの探索空間が広がり，生成することができなかったと考えられる．今後の課題としては，プログラム合成を効率化することが考えられる．具体的には現在はランダムに組み合わせているプログラムのメソッドを，問題文の情報を用いて使用されるメソッドを推測することなどが考えられる．

6. おわりに

本論文ではソフトウェア開発における実装の自動化に向けたファーストステップとして，プログラミングコンテスト AtCoder の正解プログラムの自動生成する技術の開発を行った．自動プログラミングの既存研究としては，生成したいプログラムの入出力例からプログラムを合成する手法などが存在するが，入出力例は満たすが正しいプログラムではないというオーバーフィッティングしたプログラムが生成されてしまうという課題が存在した．本論文では過去の問題情報から解きたい問題と類似した問題を検索して取得し，その解答プログラムを雛形としてプログラムを合成することで，正解プログラムを生成する手法を提案した．提案手法は AtCoder の配点が 100 点の問題 5 問に対して評価を行い，4 問の正解プログラムを自動生成できることを確認した．提案手法の今後の課題としては，より多くの問題で評価を行うことや，プログラム合成の効率化の検討などが挙げられる．また本研究の最終的な目標はプログラミングコンテストの問題をすべて自動で解くことではなく，ソフトウェア開発における実装を自動化し工数を削減することであるため，実用化に向けた検討も行っていきたい．

参考文献

[1] Sumit Gulwani. Automating string processing in spreadsheets using input-output examples. Vol. 46, pp. 317–330, 01 2011.
[2] J. Zhai, J. Huang, S. Ma, X. Zhang, L. Tan, J. Zhao, and F. Qin. Automatic model generation from documentation for java api functions. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, pp. 380–391, May 2016.
[3] Aditya Desai, Sumit Gulwani, Vineet Hingorani, Nidhi Jain, Amey Karkare, Mark Marron, Sailesh R, and Subhajit Roy. Program synthesis using natural language. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE ’16, p. 345–356, New York, NY, USA, 2016. Association for Computing Machinery.
[4] Carlos Pacheco and Michael D. Ernst. Randoop: Feedback-directed random testing for java. In *Companion to the 22nd ACM SIGPLAN Conference on Object-Oriented Programming Systems and Applications Companion*, OOPSLA ’07, p. 815–816, New York, NY, USA, 2007. Association for Computing Machinery.
[5] Sai Zhang and Yuyin Sun. Automatically synthesizing sql queries from input-output examples. In *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering*, ASE’13, p. 224–234. IEEE Press, 2013.
[6] Chenglong Wang, Alvin Cheung, and Rastislav Bodik. Synthesizing highly expressive sql queries from input-output examples. pp. 452–466, 06 2017.
[7] Matej Balog, Alexander Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. Deepcoder: Learning to write programs. In *Proceedings of ICLR’17*, March 2017.
[8] Juan Ramos. Using tf-idf to determine word relevance in document queries. 01 2003.
[9] Sumit Gulwani, Alex Polozov, and Rishabh Singh. *Program Synthesis*, Vol. 4. NOW, August 2017.
[10] J. R. Koza. Survey of genetic algorithms and genetic programming. In *Proceedings of WESCON’95*, pp. 589–, Nov 1995.
[11] Milad Taleby Ahvanooey, Qianmu Li, Ming Wu, and Shuo Wang. A survey of genetic programming and its applications. *KSII Transactions on Internet and Information Systems*, Vol. Vol.13, pp. 1765–1793, 04 2019.
[12] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. Genprog: A generic method for automatic software repair. *IEEE Trans. Software Eng.*, Vol. 38, No. 1, pp. 54–72, 2012.
[13] 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, Aug 2016.
[14] Philip Bille. A survey on tree edit distance and related problems. *Theor. Comput. Sci.*, Vol. 337, No. 1–3, p. 217–239, June 2005.