深層学習と遺伝的アルゴリズムを用い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部品を合成する手法などが存在するが，プログラム部品の組み合わせ爆発により入出力例を満たすプログラムが生成できない，また生成できたとしても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．本論文では深層学習を用いて過去の問題情報から問題文と正解プログラムの関係性を学習することで上記の問題を解決する．具体的には学習済みモデルを用いて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，その正解プログラムを雛形としてプログラムを複数個合成し，再び学習済みモデルを用いて生成されたプログラムから最も問題文との関係性が近いプログラムを判定して出力する手法を提案した．提案手法は AtCoder の配点が 100 点の問題 92 問に対して評価を行い，24 問の正解プログラムの自動生成を確認した．

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できれば，人の工数をより創造的な活動へとシフトできる．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，実装の支援にとどまる．本研究では開発現場の実装作業の自動化によって，ソフトウェア開発にかかる工数の大幅な削減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では，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．もし問題文や入出力例からプログラムの自動生成ができれば，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，深層学習を用い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 92 問に対して評価を行い，24 問の正解プログラムの自動生成を確認した．

2. Motivating Example

2.1 AtCoder について

図 1 は，AtCoder で提供されている 100 点問題 AtCoder Beginner Contest 138（以下 ABC138）の問題文と入出力

[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/

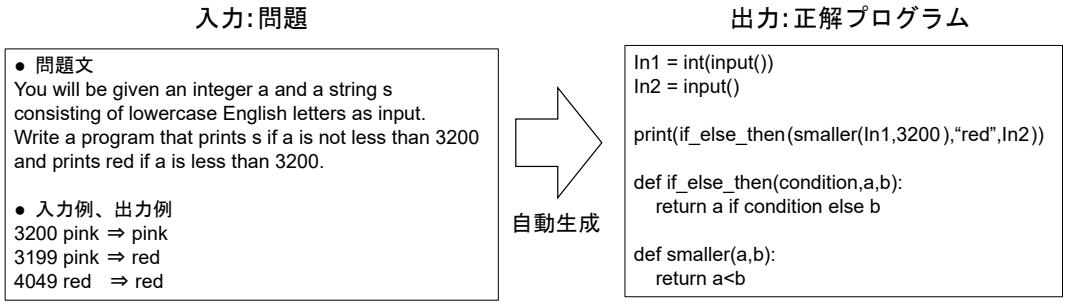

図 1 Motivating Example

例を抜粋した問題である．問題文には数行で構成され，使用する入力変数と，得たい出力について記述されている．基本的には問題文だけを見ても正解プログラムが作成できるようになっている．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．問題文だけでなく入出力例も例示されているが，それを満たすことは正解プログラムとなる必要条件ではあるが十分条件ではない．また，サーバ側で実行されるテストの情報は，ユーザは確認できない．

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．ABC042 から 150 までの問題のうち，本論文で評価の対象とした 92 問においては，数値計算問題が 41 問，分岐問題が 44 問，文字列操作問題が 7 問となっている．ただし if 文を含む問題は分岐，if 文を含まず数値の計算を行う問題は計算，if 文を含まず文字列の結合や削除等を行う問題は文字列操作として分類している．すべての問題は表 2 の 25 個のメソッドと任意の定数の組み合わせで解くことができる．著者らが正解プログラムを作成したところ，87%の問題で 20 個以下のメソッドと任意の定数の組み合わせで作成できることがわかった．残りの 13%の問題を外れ値として除外した上で，各問題の正解プログラムで使用するメソッド及び定数の数を分析した結果を表 1 に示す．表 1 の標準偏差の値を見る限り，プログラムのサイズは問題によって大きく異ならないことがわかる．

以上より AtCoder の 100 点問題において，各問題の正解プログラムのサイズと使用するメソッドの種類は，問題によって大きく異ならないと言える．

表 1 各問題で使用するメソッド及び定数の数

	数値計算	分岐	文字列操作
平均	6.7	11.5	7.3
標準偏差	2.6	3.7	4.5
最小値	3	6	3
最大値	13	20	14

表 2 100 点問題で使用されるメソッド一覧

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

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の自動生成を目指す．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，それらの情報は本研究では使用しない．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の自動回答ではなく，実際の開発現場の実装作業の自動化による工数削減であり，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しも存在しない．したがって本研究では問題文と入出力例のみを使用する．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使用する．また AtCoder では過去の問題が公開されており，それらの情報は使用可能とする．（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]．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の理解を目的とし，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の存在が前提となるため，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の自動生成は難しいが，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）の設計により精度を高めている．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．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の自動判別は難しく，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，非常に制限が強くなる．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．しかしこれらの手法は求められるプログラムが複雑になるほど探索空間が広がり，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．例えば最も単純には N 種類のプログラム部品を M 個組み合わせるパターン数は N^M となり，指数関数的に増加する．この課題は AtCoder の問題を解く際も同様に発生すると考えられる（課題 2）．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の学習によって，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，情報量の多さ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（課題 3）．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．以上より AtCoder の入出力例からプログラムを自動で生成する場合，以下の 2 つの課題が存在する．
課題２：プログラム部品の合成パターンは無数に存在するため，現実的な時間内でプログラムの合成は困難
課題３：入出力例を満たすプログラムが合成できた場合でも，入出力例にオーバーフィッティングしている可能性がある

4. 提案手法

本論文では深層学習を用いて，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 ０：過去の問題情報を使用して，問題と正解プログラムの関係性を深層学習を用いて学習（プログラム学習部）
Step １：学習済みモデルを用いて解きたい問題の問題情報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複数個合成（プログラム合成部）
Step ３：入出力例を満たす複数個のプログラムに対し，学習済みモデルを用いて問題文と最も関連性が高いプログラムを判定して出力（プログラム判定部）
Step0 は学習フェーズ，Step1 から 3 は自動解答フェーズとなる．学習フェーズでは問題情報とプログラムの関係性を学習し，その学習モデルを Step1 のプログラム検索と Step3 のプログラム判定に用いる．提案の学習の流れを図 2 に，自動解答の流れを図 3 に示す．提案手法は 3 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るため，課題 1 の解決が可能となる．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，探索空間を狭めることが可能となり，課題 2 の解決ができる．3 つ目の特徴はプログラム合成を複数回試し，その中で問題情報と最も関連性の高いプログラムを推定して出力している点である．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を取り除くことができ，課題 3 の解決が可能となる．以上の特徴により，提案手法では従来手法の課題の解決が可能となり，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，過去の類似問題の存在を前提としている．以下に提案手法の詳細なアルゴリズムを述べる．

4.1 Step ０：プログラム学習部

プログラム学習部では，Step 1 と Step 3 に向けて問題情報とプログラムの関係性を学習する．問題情報とプログラムのような異なる種類の情報の類似性を学習する方法として，Joint Embedding[8] と呼ばれる手法が存在する．Joint Embedding では，各入力情報をニューラルネットワークを介してベクトル化し，関連性のある情報同士ならお互いのベクトルのコサイン類似度が 1 に，そうでないなら 0 に近づくように学習する．それにより異なる種類の情報が与えられた時，その情報同士の関連性を 0 から 1 の範囲内で推測できる．本論文では，図 2 に示すように問題情報として問題文と入出力の型を用いてプログラムとの関係性を学習する．任意の 2 つの問題における問題情報と正解プログラムの関係性の学習を，すべての組み合わせにおいて実施している．つまり問題が N 問ある場合，N^2 の学習データが得られる．

4.1.1 学習データ（入力）

問題文は自然言語であり時系列データであるため，token に分けた単語を Word Embedding 層を用いて埋め込みを行った後に，LSTM 層でベクトル化する．ただし今回対象としている AtCoder の 100 点問題は 100 問程度しか存在せず，単語の種類が少ない．そのため新しい問題を解こうとした際に，未知の単語を扱わなくてはならないケースが多くなると推測されるため，Embedding 層には英語版の wikipedia を用いて学習した学習済みの埋め込みベクトル[*7]を用いた．入出力の型については入出力例から型情報を推測し，それぞれ 3 次元のベクトルに変換してから Dence 層でベクトル化をした．入力の場合，3 次元のベクトルは [数値型の引数の数，文字列型の引数の数，それ以外の引数の数] とした．ある引数が数値型と文字列型ともにとり得る場合は，それ以外の引数としてカウントした．出力の場合も同様である．プログラムも時系列データとして捉え，関数名とその関数の引数で分けて one-hot 化してから LSTM 層でベクトル化した．ただし引数は入力変数の場合は variale，定数は数値型なら number，文字列型なら string という名前に変換をしている．例えば $sum(x, 1)$ というプログラムの場合，$[sum, variable, number]$ というベクトルに変換してから one-hot 化し，LSTM 層に渡している．なおプログラムのベクトル化については，プログラムを木構造データとして捉え， Tree-LSTM でベクトル化す

[*7] http://vectors.nlpl.eu/explore/embeddings/en/models/

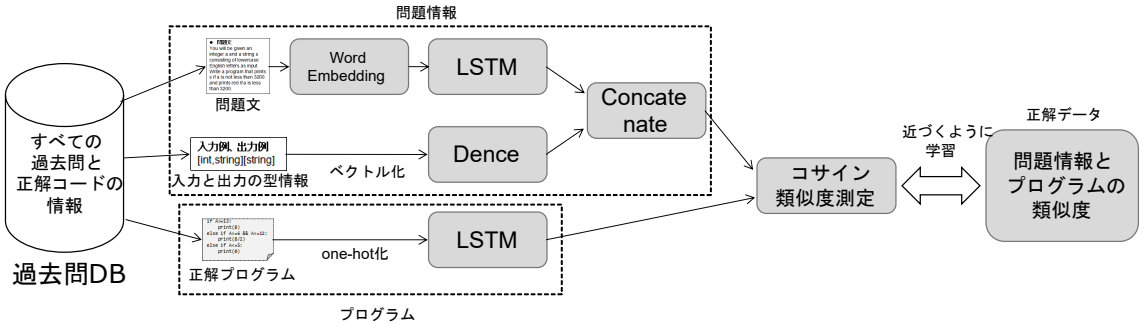

図 2　提案手法（学習フェーズ）

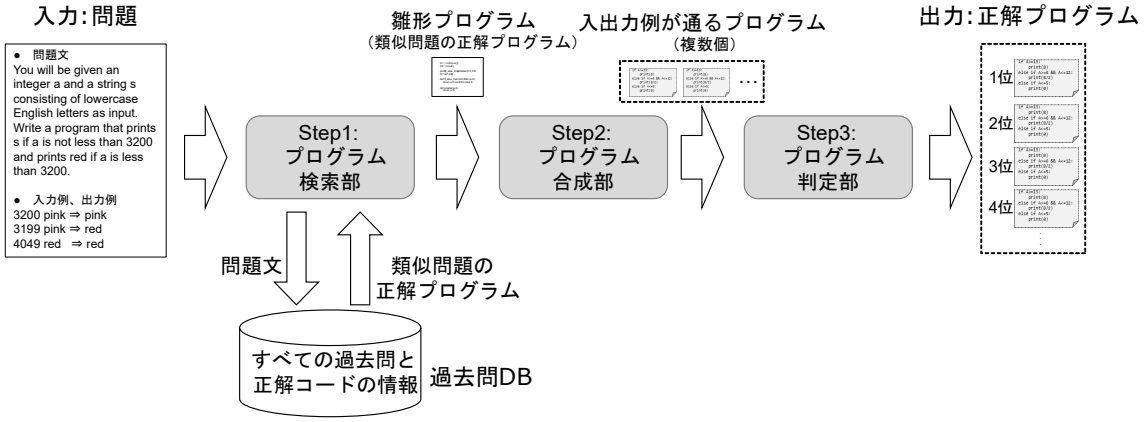

図 3　提案手法（自動解答フェーズ）

る手法 [9] や，プログラムの token の適切なベクトル表現を取得する Code2vec と呼ばれる手法 [10] も存在する．最適な学習モデルの見極めは今後の課題としたい．

4.1.2　学習データ（正解）

正解データは，入力された問題 n の正解プログラム p_n と，入力されたプログラム p_m の類似度 $S(n,m)$ とする．$S(n,m)$ は以下の式で評価される．

$$S(n,m) = Max((T_{const} - TED(p_n, p_m))/T_{const}, 0)$$

ただし，TED は p_n と p_m の Tree Edit Distance[11] を表す．Tree Edit Distance は木構造で表現されるデータ同士の類似度を表す指標である．本論文ではプログラムを木構造で表現することでプログラム同士の類似度を測定する．例えば図 1 の正解プログラムは，木構造で表すと図 4 のようになる．T_{const} はしきい値であり，Tree Edit Distance がしきい値以上の場合，$S(n,m)$ は 0 になる．p_n と p_m が同一の場合，つまり p_m が問題 n の正解プログラムの場合は，$S(n,m)$ は 1 になる．これにより任意の問題情報とプログラムが与えられた時，そのプログラムが問題の正解プログラムに構造的に近い場合は 1 に近い値を，そうでない場合は 0 に近い値を推測する AI モデルの構築が可能となる．以下，本論文ではこの学習済みモデルを類似度推測モデルと呼ぶ．

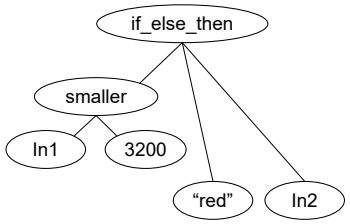

図 4　木構造で表されたプログラムの例

4.2　Step 1：プログラム検索部

AI 技術の進歩により，AI を活用したプログラム検索技術が成果を挙げている．例えば Deep Code Search[12] では，検索文とプログラム情報の関連性を Joint Embedding で学習し，ある検索文が与えられた時に関係の深いプログラムを推測する新しいプログラム検索手法を提案している．本論文においても，類似度推測モデルを用いて，同様の方法でプログラム検索を行う．具体的には解きたい問題の問題情報と，過去問 DB 内のすべての問題の正解プログラムそれぞれに対し，類似度推測モデルで類似度を推測し，一番類似度の高い値のプログラムを取得する．

4.3　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 [13] 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 [14][15] 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できる．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．本論文では，初期 Seed 値の異なるプロセスで複数回合成を行い，プログラムを複数個生成する．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．

4.3.1 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 2 に示される 25 個のメソッドを用いた．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．

4.3.2 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[16] 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 $E(x)$ 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 2 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．

4.3.3 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，4.2 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．

4.3.4 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 [15] 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．

4.3.5 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[15] を用いている．ただし，Fault-Localization[17] 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 5 に示す．図 5 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，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

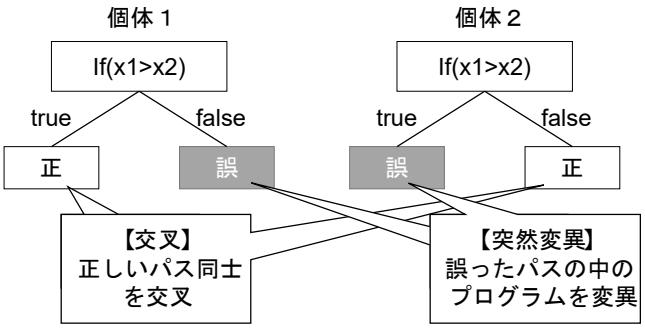

図 5　Fault-Localization を用いた交叉と突然変異の例

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．

4.3.6　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[17]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 5 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 2 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．

4.4　Step 3：プログラム判定部

プログラム判定部では Step 2 で生成した入出力例を満たす複数のプログラムに対し，類似度推測モデルを用いて，最も解きたい問題に合ったプログラムを取得し，出力する．具体的には，生成されたすべてのプログラムに対して，解きたい問題の問題情報に対する類似度測定を測定し，最も類似度が高いプログラムを出力する．

5.　評価

5.1　評価方法

AtCoder における ABC042 から 150 までの実際の 100 点問題のうち，92 問を対象に実験を行った．文字列中に「+」等の文字が含まれる問題はプログラム合成で使用したライブラリ DEAP[*8]が対応していないため除外した．問題中に画像の情報が含まれる問題も除外した．また ABC041 以前の問題は点数の概念が存在しないため，対象外としている．問題とプログラムの学習は 5 問ずつ（ABC042 から ABC045 までは 4 問）行った．具体的にはまずは ABC042 から 045 の問題に対して，それら以外の問題を用いて学習を行い評価する．続いて ABC046 から 050 の問題を同様の方法で評価する．この繰り返しを残りの ABC051 から ABC150 までに対しても行った．Epoch 数は 100 回，7569 個のデータセットを対象に学習を行った．データセットは 4.1 節で示した方法で作成した．提案手法の特徴は，4 章で述べたとおり，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ができ，正解プログラムの生成確率が高いということと，プログラム合成を複数回試し，その中で作りたいプログラムの要求がすべて書かれている問題情報と最も関連性の高いプログラムを推定して出力しているため，入出力例にオーバーフィットしにくいプログラムが出力できるということである．したがって提案手法の有効性を確認するために，以下の 3 つの評価を行った．

- プログラム検索部の有無による正解プログラム生成数の比較（プログラム検索部＋合成部の評価）
- プログラム判定部の有無による正解プログラム生成数の比較（プログラム判定部の評価）
- 従来手法と提案手法の正解プログラム生成率の比較（提案手法のアプローチ全体の評価）

またこれらの評価はプログラムの性質における性能を見るため，問題を数値計算，分岐，文字列操作の 3 種類に分けて評価を行った．if 文を含む問題は分岐，if 文を含まず数値の計算を行う問題は計算，if 文を含まず文字列の結合や削除等を行う問題は文字列操作として分類した．以下に各評価方法について説明する．実験で用いたパラメータは表 3 に示す．

5.1.1　プログラム検索部＋合成部の評価

プログラム検索部で得た雛形を用いた場合と用いなかった場合のプログラム合成の精度について比較した．各問題に対して初期 Seed をランダムに変更した 32 回のプロセスで合成を行い，一度でも正解プログラムが生成できた問題数を比較した．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 に追加実装をしたプログラムを用いた．また生成されたプログラムが正解か不正解かの判断は，目視で行った．

5.1.2　プログラム判定部の評価

プログラム判定部の目的は，プログラム合成部で出力した複数のプログラムに対して，入出力例にオーバーフィットしたプログラムを除き，正解プログラムを出力することである．そのため正解プログラムとオーバーフィットしたプログラムの両方が合成された問題において，プログラム判定部がある場合とない場合の正解プログラム出力問題数を評価した．プログラム判定部がない場合は 32 個のプログラム合成プロセスにおいて一番最初に合成された入出力例をすべて満たすプログラムを出力とした．一番最初に各合成プロセスは互いに独立しており，合成されたプログラ

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

表 3　実験で使用したパラメータ（プログラム合成）

合成プロセス数	32
最大世代数	1000
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

表 4　プログラム検索部の有無による正解プログラム生成数の比較

	問題種別			全問題
	数値計算（41 問）	分岐（44 問）	文字列操作（7 問）	（92 問）
検索部なし	23 (56%)	7 (16%)	5 (71%)	35 (38%)
検索部あり	24 (58%)	10 (23%)	6 (86%)	40 (43%)

ムを出力することは，32 回の合成プロセスの中で生成されたすべての入出力例を満たすプログラムの中から無作為に 1 つ選択することと同値である．したがってプログラム判定部ある場合の方がない場合よりも正解プログラムの出力問題数が上回れば，プログラム判定部がオーバーフィットしていないプログラムを，無作為に選ぶ場合より適切に判定できていると評価できる．

5.1.3　提案手法のアプローチ全体の評価

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，プログラム検索部とプログラム判定部を用いずに，プログラム合成のみを行った場合（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 2 に存在するメソッドや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．また従来手法では最大で 32 回，表 3 の条件で合成を行い，入出力例をすべて満たすプログラムが 1 つ合成された時点でそのプログラムを出力し，合成を終了させた．32 回の合成プロセスで一度も入出力例を満たすプログラムが合成できなかった場合，失敗とした．

5.2　評価結果と考察

5.2.1　プログラム検索部＋合成部の評価

正解プログラムの合成に成功した問題数を表 4 に示す．提案手法では従来手法より 5 問多い，40 問の正解プログラムの合成に成功した．またすべての問題種別で従来手法よりも優れた結果が得られた．検索によって得られたプログラムを雛形として用いることが具体的にどのように有効に働いたかについて，ABC138 の結果を用いて確認する．ABC138 は提案手法でのみ合成ができた問題である．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).
```

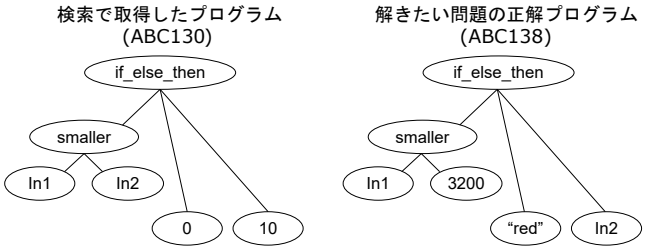

図 6　検索で取得したプログラムと解きたい問題のプログラム (ABC138)

表 5　プログラム判定部の有無による正解プログラム生成数の比較

	問題種別			全問題
	数値計算（17 問）	分岐（9 問）	文字列操作（2 問）	（28 問）
判定部なし	7 (41%)	1 (11%)	1 (50%)	9 (32%)
判定部あり	10 (59%)	4 (44%)	1 (50%)	15 (54%)

```
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．またそれぞれの問題の正解プログラムを図 6 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．

5.2.2　プログラム判定部の評価

提案手法で正解プログラムが生成できた問題 40 問のうち 28 問において，正解プログラムとオーバーフィットしたプログラムの両方を合成した．この 28 問を対象に評価を行った結果を表 5 に示す．プログラム判定部によって，6 問においてオーバーフィットしたプログラムを排除できた．プログラム判定部の精度は 54%で，約半数の問題で正しく判定できた．分岐のプログラムは条件と 2 つの遷移先を持つためプログラムが複雑になりやすく，解にたどり着くまでにオーバーフィットしたプログラムが得られやすい．そのため判定部なしでは 1 問しか正解プログラムを出力できなかったが，判定部の導入によって様々な入出力を満たすプログラムの中から適切なプログラムを選定でき，結果として 4 問の正解プログラムを出力できた．

5.2.3　提案手法のアプローチ全体の評価

全問題における提案手法と従来手法の結果を表 6 に示す．提案手法では従来手法より 12 問多い，24 問の問題に対して正解プログラムを自動生成できた．これはプログラム検索部と判定部が，上記の評価結果の通り有効に機能したためだと考えられる．今後の課題としては，分岐系のプログラムに対する精度向上が挙げられる．そのためには，問題文からプログラムの傾向を推定して合成に使用するメ

表 6　提案手法と従来手法の正解プログラム生成数の比較

	問題種別			全問題
	数値計算（41 問）	分岐（44 問）	文字列操作（7 問）	（92 問）
従来手法（合成部のみ）	8 (20%)	1 (2%)	3 (43%)	12 (13%)
提案手法	16 (39%)	4 (9%)	4 (57%)	24 (26%)

ソッドを絞り込むなど，合成の効率化を検討していく必要がある．

6. 議論

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の自動回答ではなく，実際の開発現場の実装作業の自動化による工数削減である．そのため本章では提案手法の実用化について考察する．提案手法を実用化する第一歩として，ソフトウェア構成部品の最小単位であるメソッドの自動生成について考えている．メソッドの仕様を表す情報は大きく分けて 2 つ存在する．1 つ目はメソッドの説明文，2 つ目はメソッドに対する単体テストである．提案手法は，問題文と入出力例を入力としており，これはそれぞれ説明文とテストケースに該当する．したがって説明文とテストケースを先に作成することで，提案手法はメソッドを自動生成する技術へ応用できると考えられる．懸念点としては，学習データをどのように用意するかという点と，テストケースを実装より先に用意することの実現性などが挙げられる．学習データについては GitHub 上のプログラムを学習すればよいと考えられる．ただし，特定のドメインに特化した処理が必要となるメソッドについては，学習データが十分に得られない可能性がある．そのため Utility 等の汎用的なメソッドがまずは自動生成の対象になると思われる．テストケースを実装より先に用意することの実現性については，実装よりも先にテストケースを準備することが前提となるテスト駆動開発を採用している開発現場と相性が良いと思われる．具体的にはテスト駆動開発において，テストケースだけでなくメソッドの説明文も先に作成することで，メソッドを自動生成できる可能性がある．また本論文の評価では提案手法は最も問題文に対して関連性が高いと推定される 1 つのプログラムを出力したが，実用化においては複数個プログラムの候補を提示し，ユーザが選択するといった使い方も考えられる．提案手法は 24 問において正解プログラムを出力できることを確認したが，プログラム判定部が上位 5 位のプログラムを出力するとした場合，36 問において正解プログラムが含まることを確認している．したがって複数の候補からユーザが選択するというユースケースの場合，より多くの場合で提案が有効に働く可能性がある．今後はこのように様々なユースケースを考慮しつつ，提案手法の実用化を検討していきたい．

7. 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を目標とし，深層学習を用いて過去の問題情報から問題文と正解プログラムの関係性を学習して，プログラムを自動生成に用いる手法を提案した．具体的には学習済みモデルを用いて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，その正解プログラムを雛形としてプログラムを複数個合成し，再び学習済みモデルを用いて生成されたプログラムから最も問題文との関係性が近いプログラムを判定して出力する．提案手法は AtCoder の配点が 100 点の問題 92 問に対して評価を行い，24 問の正解プログラムの自動生成を確認した．今後の課題としては，今回正解プログラム生成率が低かった分岐のプログラムに対する精度向上などが挙げられる．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の自動解答ではなく，ソフトウェア開発における実装の自動化による工数削減であるため，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. In *ACM SIGPLAN Notices*, 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. In *PLDI 2017*, 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] Ran Xu, Caiming Xiong, Wei Chen, and Jason J. Corso. Jointly modeling deep video and compositional text to bridge vision and language in a unified framework. In

Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence, AAAI'15, p. 2346–2352. AAAI Press, 2015.

[9] Yao Wan, Jingdong Shu, Yulei Sui, Guandong Xu, Zhou Zhao, Jian Wu, and Philip S. Yu. Multi-modal attention network learning for semantic source code retrieval. In *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering*, ASE '19, p. 13–25. IEEE Press, 2019.

[10] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. Code2vec: Learning distributed representations of code. *Proc. ACM Program. Lang.*, Vol. 3, No. POPL, January 2019.

[11] Philip Bille. A survey on tree edit distance and related problems. *Theor. Comput. Sci.*, Vol. 337, No. 1–3, p. 217–239, June 2005.

[12] X. Gu, H. Zhang, and S. Kim. Deep code search. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, pp. 933–944, 2018.

[13] Sumit Gulwani, Oleksandr Polozov, and Rishabh Singh. Program synthesis. *Foundations and Trends in Programming Languages*, Vol. 4, No. 1-2, pp. 1–119, 2017.

[14] J. R. Koza. Survey of genetic algorithms and genetic programming. In *Proceedings of WESCON'95*, pp. 589–, Nov 1995.

[15] 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.

[16] 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.

[17] 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.