修士学位論文

題目

自動プログラム修正を応用したプログラム生成の提案と評価

指導教員

楠本 真二 教授

報告者

富田 裕也

2021 年 2 月 2 日

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

令和 2 年度 修士学位論文

自動プログラム修正を応用したプログラム生成の提案と評価

富田 裕也

内容梗概

現在，自動運転や無人レジのように社会では自動化が進んでおり，ソフトウェア開発の自動化も求められている．単体テストやデプロイは自動化が進んでいるが，実装の自動化は進んでいない．本研究では実装の完全な自動化によりソフトウェア開発の工数の大幅削減を目標とする．しかし，オープンソースソフトウェアや業務で開発するプログラムのような大規模なプログラムの自動生成は難しい．そのため，実装の自動化の実現に向けた第一歩としてプログラミングコンテストの問題を対象としてプログラムの自動生成を目指す．本研究では，自動でバグの特定と修正を行う自動プログラム修正手法を応用したプログラムの自動生成手法を提案する．まず，プログラミングコンテストの問題文，テストケースおよび過去のプログラミングコンテストの解答を保存したデータベースを用いて，プログラミングコンテストの解答を生成する．問題文からその問題を解くために必要な制御構文のみを含む解答の雛形を推測する．次に，推測した雛形に対して過去に出題された問題の解答に含まれるプログラム文を再利用して解答を生成する．

AtCoder というプログラムコンテストで出題された問題を対象に実験をした．古いコンテストの問題は，傾向が固まっておらず現在の問題と難易度が異なる．そのため，現在の出題形式に近い問題を実験対象とした．現在の出題形式に近い 99 問の問題に対して雛形の推測を試みたところ 46 問で正しい雛形の推測に成功した．また，現在の出題形式に近く，すべてのテストケースが公開されている 35 問の問題に対して生成を試みたところ，10 問の問題で解答の生成に成功した．

主な用語

自動プログラム生成
自動プログラム修正
遺伝的アルゴリズム
機械学習

目次

図目次

表目次

1　はじめに

現在，社会の自動化が進んでいる．例えば，自動運転や無人レジが挙げられる．ソフトウェア開発の自動化も求められている．特に実装やテストを自動化し開発の工数を大幅に削減できれば，設計や要件定義といったより創造的な工程に割ける人の労力を増やせる．ソフトウェア開発において自動化が進んでいる工程は，単体テストとデプロイである．単体テストでは JUnit[*1]が広く使われており，デプロイでは Circle CI[*2]や Travis CI[*3]に代表される CI ツールが使われている．一方で，自動化が進んでいない工程は，実装と結合テストである．コードの補完を行うツールとして，機械学習を用いた TabNine[*4]や IntelliCode[*5]があるが，これらのツールは開発者の補助が目的であるため，実装の主体は開発者である．本研究では実装の完全な自動化を目標とする．しかし，オープンソースソフトウェアや業務で開発するプログラムのような大規模なプログラムの自動生成は難しい．そのため，実装の自動化への第一歩としてプログラミングコンテストの問題を対象としてプログラムの自動生成を目指す．問題を解くために必要なプログラム文は，過去の解答に含まれていると考えた．その仮説を検証するために，AtCoder というプログラミングコンテストで出題された 140 問の問題に対する解答を対象とし調査を行った．調査の結果，80% の解答は他の解答に含まれるプログラム文の再利用によって生成できることがわかった．また，プログラミングコンテストの問題にはさまざまな種類がある．プログラム文の組み合わせが非常に多いため，プログラム文の再利用だけでは現実的な時間内での解答の生成は難しい．提出された解答に存在する制御構文の構造に着目し，あらかじめプログラムの雛形を用意すれば解決の生成が容易になると考えた．プログラミングコンテストの問題を解くために必要な制御構文の構造を調べるために 140 問の問題を対象に調査を行った．調査の結果，90% の問題は 5 種類の構造を利用すれば解答できることがわかった．

調査の結果を踏まえ，本研究ではプログラムの再利用に基づく自動プログラム修正手法を応用した自動プログラム生成手法を提案する．提案手法の入力は，プログラミングコンテストの問題文・テストケース，過去の解答を集めたデータベース（以降，解答データベースという）である．出力として与えられたプログラミングコンテストの問題に対する解答が得られる．提案手法の流れは，最初に入力として与えられたプログラミングコンテストの問題文から解答に必要な制御構文を推測し，解答の雛形を作成する．次に，作成した雛形に対して解答データベースに保存されているプログラム文を再利用し与えられた全てのテストケースを通過する解答を生成する．

[*1] https://junit.org/juint5
[*2] https://circleci.com
[*3] https://travis-ci.org
[*4] https://www.tabnine.com
[*5] https://visualstudio.microsoft.com/services/intellicode

AtCoder で出題された問題を対象に実験をした．古いコンテストの問題は，傾向が固まっておらず現在の問題と難易度が異なる．そのため，現在の出題形式に近い問題を実験対象とした．現在の出題形式に近い 99 問の問題に対して雛形の推測を試みたところ 46 問で正しい雛形の推測に成功した．また，現在の出題形式に近く，すべてのテストケースが公開されている 35 問の問題に対して生成を試みたところ，10 問の問題で解答の生成に成功した．

以降，2 章では準備について述べる． 3 章では，自動プログラム修正を自動プログラム生成に応用する場合の課題およびその課題の解決策について述べる． 4 章では，予備調査について述べる． 5 章では，提案手法の概要について述べる． 6 章では，評価実験の方法や結果について述べる． 7 章では，妥当性への脅威について述べる． 8 章では，関連研究について述べる．最後に， 9 章では本研究のまとめと今後の課題について述べる．

2 準備

2.1 自動プログラム修正

ソフトウェア開発においてデバッグは多大な労力を必要とする作業であり，開発工数の半数以上を占めるといわれている [1, 2]．そのため，デバッグの支援はソフトウェア開発の効率化やソフトウェアの信頼性および安全性の向上に有益である．デバッグの労力を削減するために，デバッグの支援に関する研究が数多く行われている [3, 4, 5, 6]．デバッグの支援に関する研究の 1 つに自動プログラム修正 [7, 8, 9, 10] がある．

自動プログラム修正は，欠陥を含むプログラムおよびそのプログラムに対するテストケースを入力として，全てのテストケースに成功するプログラムを出力する手法である．自動プログラム修正の手法の 1 つに GenProg [7] がある．GenProg は遺伝的アルゴリズム [11] を用いた自動プログラム修正手法である．遺伝的アルゴリズムとは，解の候補を生物の個体に見立て，遺伝的操作を繰り返し行い解を探索するアルゴリズムである．

GenProg の流れを図 1 に示す．まず，GenProg は欠陥限局 [12] を行う．欠陥限局とはプログラム中の欠陥の位置を推測する手法である．欠陥限局後に，欠陥を含むプログラムに対して推測した欠陥箇所に変更を加え，プログラムを複数生成する．次に，生成したプログラムの評価を行う．プログラムの評価には適応度を用いる．適応度はプログラムの良さを表しており，適応度にはテストの通過率が用いられる．GenProg は全てのテストケースに成功するプログラムが得られるまでプログラムの生成および評価を繰り返す．プログラム生成処理で行われる操作を以下に示す．

選択 生成されたプログラムに対してテストを実行する．それらから適応度が高いプログラムを一定数取り出す．取り出されたプログラムは変異や交叉の対象となる．

変異 選択によって取り出されたプログラムの欠陥箇所に変更を加え，新しいプログラムを生成する．加える変更はプログラム文の挿入・削除・置換のいずれかである．挿入や置換で用いるプログラム文は入力プログラムに含まれるプログラム文からランダムに選ばれる．変異の際に，変更を加えた箇所と変更内容を記録する．

交叉 親となる 2 つのプログラムに記録されている変更箇所と変更内容をいくつか選び，新たなプログラムを生成する．生成されたプログラムは親に記録されている変更箇所と変更内容の一部を持っている．

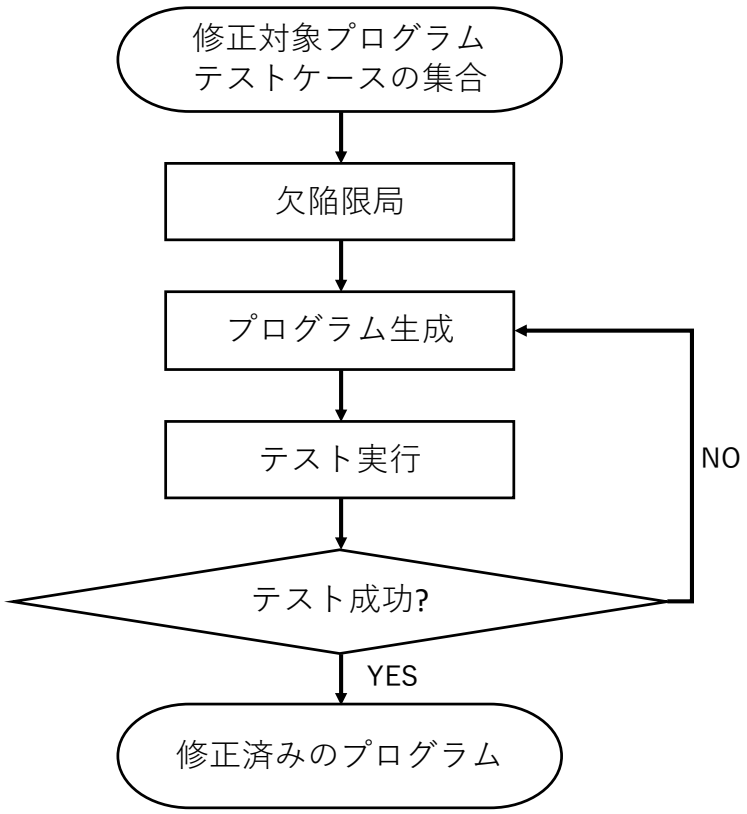


図 1　GenProg の流れ

2.2　AtCoder

AtCoder[*6]は，AtCoder 社が運営しているプログラミングコンテストである．AtCoder では，定期的にいくつかのコンテストが開催されている．それらで最も難易度が低いコンテストは AtCoder Beginner Contest（以降，ABC という）である．ABC はプログラミングコンテスト初心者向けのコンテストであり，最も難易度が低い A 問題から最も難易度が高い F 問題の 6 つの問題で構成されている．各問題は問題を説明する問題文，入力の制約，入力・出力の形式およびテストケースに当たる入出力例から構成されている．図 2(a)，(b) に A 問題の例および解答例を示す．

2.3　Long Short-Term Memory

Long Short-Term Memory [13]（以降，LSTM という）とは，音声，動画および文章のような時系列データを扱う機械学習モデルである．Recurrent Neural Network [14]（以降，RNN という）でも時

[*6] https://atcoder.jp
[*2] https://atcoder.jp/contests/abc130/tasks/abc130_a

問題文

X, A は0以上9以下の整数である.
X が A 未満の時 0, A 以上の時10を出力せよ.

制約

- 入力は全て整数
- $0 \leq A, X \leq 9$

入力

入力は以下の形式で標準入力から与えられる.

```
X A
```

出力

X が A 未満の時0, A 以上の時10を標準出力に出力せよ.

入出力例

入力	出力
$X = 3,\ A = 10$	0
$X = 7,\ A = 5$	10
$X = 6,\ A = 6$	10

(a) 問題

```java
import java.util.Scanner;

public class Main {

  public static void main(String[] args) {
    // 標準入力から読み込む
    Scanner scanner = new Scanner(System.in);
    int X = scanner.nextInt();
    int A = scanner.nextInt();

    if (X < A) {
      System.out.println(0);
    } else {
      System.out.println(10);
    }
  }
}
```

(b) 解答例

図 2　A 問題の例（Rounding）[*2]

系列データを処理できるが，長期の記憶保存ができない．LSTM は RNN の中間層を LSTM ブロックと呼ばれるユニットに置き換えることで，長期にわたる依存を持つ学習ができる．

2.4 分散表現

分散表現とは，実数のベクトルで表現された単語である．単語をベクトルに変換することで，概念の演算をそのままベクトルの演算で表現できる．例えば，“王”に対応するベクトルと“女”に対応するベクトルを加算すると，“女王”に対応するベクトルを導出できる．分散表現を作成する手法として，Word2Vec [15] や GloVe [16] がある．

3　自動プログラム修正を自動プログラム生成に応用する場合の課題と解決策

自動プログラム修正と異なり入力が空のプログラムであるため，再利用できるプログラム文が存在しない．また，自動プログラム修正では数行の変更でバグを修正できるが，自動プログラム生成では多くのプログラム文の組み合わせを考慮する必要がある．したがって，自動プログラム修正を自動プログラム生成に応用する場合の課題は以下の 2 つとなる．

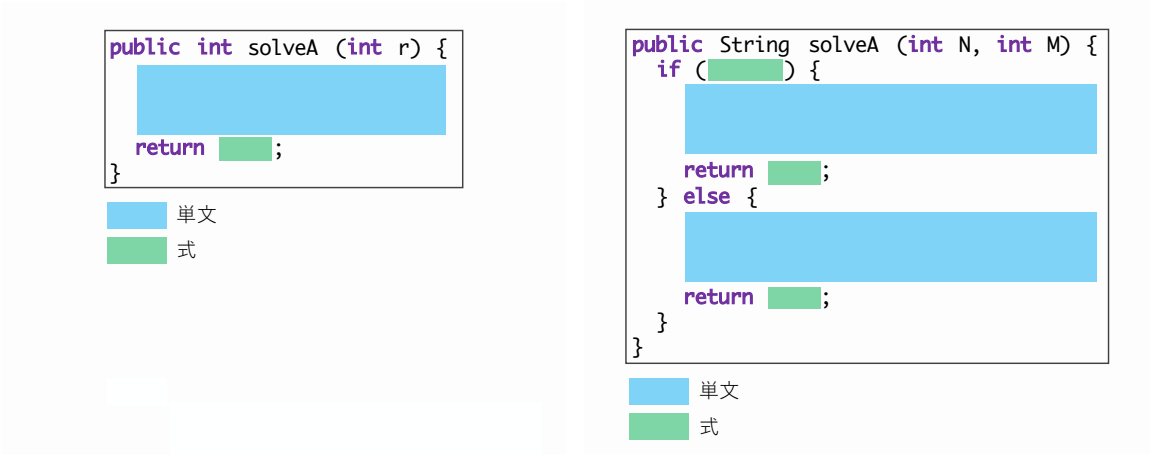
課題 1. 再利用候補になりうる文や式が存在しない
課題 2. プログラム文の組み合わせが膨大

3.1　解答データベース

“課題 1. 再利用候補になりうる文や式が存在しない” を解決するために解答データベースを用いる．解答データベースには，過去に出題された A 問題に対するプログラム文・式が保存されている．GenProg は，入力プログラムに含まれるプログラム文・式を用いるのではなく，あらかじめ用意した解答データベスに保存されたプログラム文・式を再利用する．

3.2　テンプレート

“課題 2. プログラム文の組み合わせが膨大” を解決するためにテンプレートを用いる．テンプレートとは，解答に出現する制御構文のうち if 文，for 文，switch 文，while 文および do-while 文の構造である．テンプレートの分類は排他的である．2 つのテンプレートの例を図 3 に示す．図 3(a) は分岐や繰り返しがない flat テンプレートである．図 3(b) は分岐が 1 つだけある if{}else{}テンプレートである．水色の部分に単文，緑色の部分に式が入る．



(a) flat テンプレート　　(b) if{}else{}テンプレート

図 3　テンプレート

4　調査

4.1　再利用率の調査

4.1.1　調査目的

本調査は，プログラムの再利用によって生成可能である解答が存在するか確認するために行った．

4.1.2　調査方法

調査対象は，ABC1〜140 の A 問題に対する Java で記述された 35,275 個の解答である．A 問題の解答に他の A 問題に対する解答に含まれるプログラム文が存在する割合（以降，再利用率という）を算出した．変数の正規化をした解答としていない 2 種類の解答に対して本調査を行った．

変数の正規化の流れを図 4 に示す．まず，Java のソースコード解析ツールである Java Development Tools [*7]（以降，JDT という）を用いて解答からプログラム文を抽出する．次に，抽出したプログラム文に対して変数を出現順に正規化する．

4.1.3　調査結果

再利用率を箱ひげ図にプロットした結果を図 5 に示す．図 5(a) は，対象にした全ての解答の再利用率をプロットした箱ひげ図である．図 5(b) は，対象にした各 A 問題に対して再利用率の最大値をプロットした箱ひげ図である．いずれの図も橙色が変数の正規化をしていない解答，青色が変数の正規化

[*7] https://www.eclipse.org/jdt

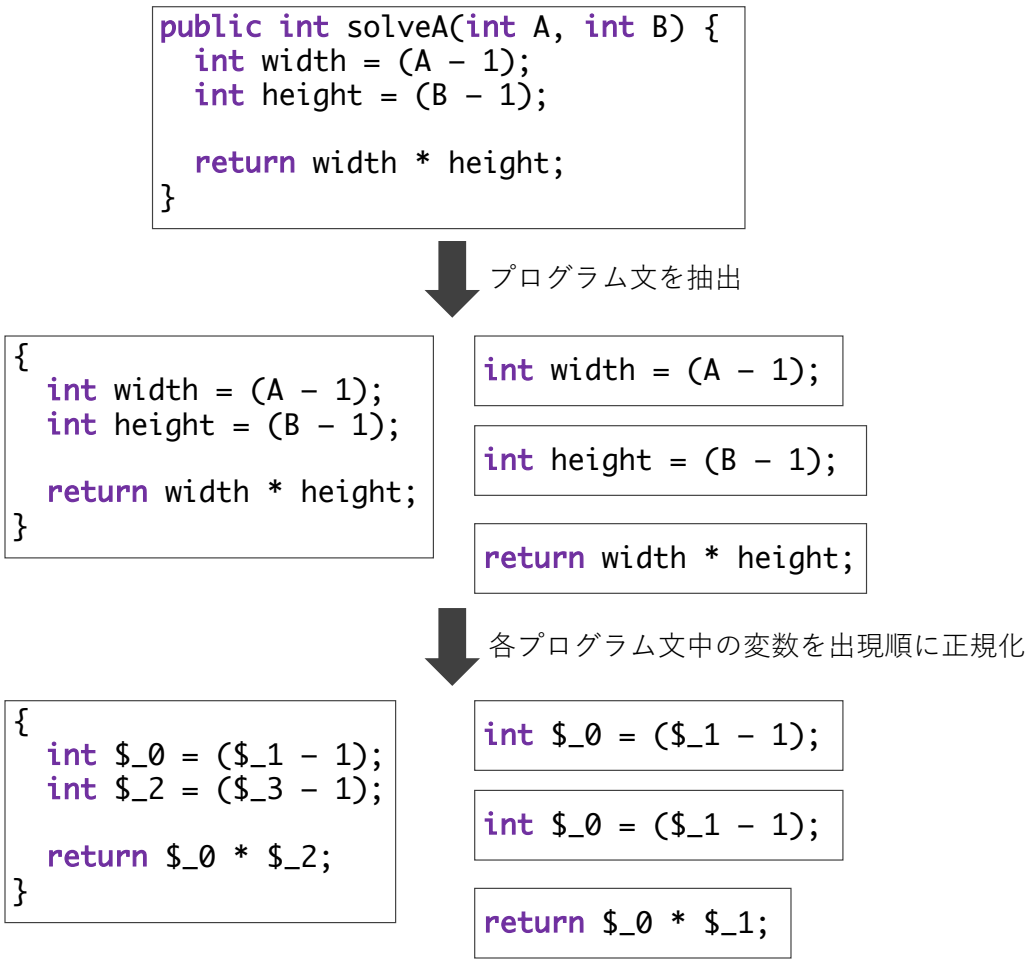


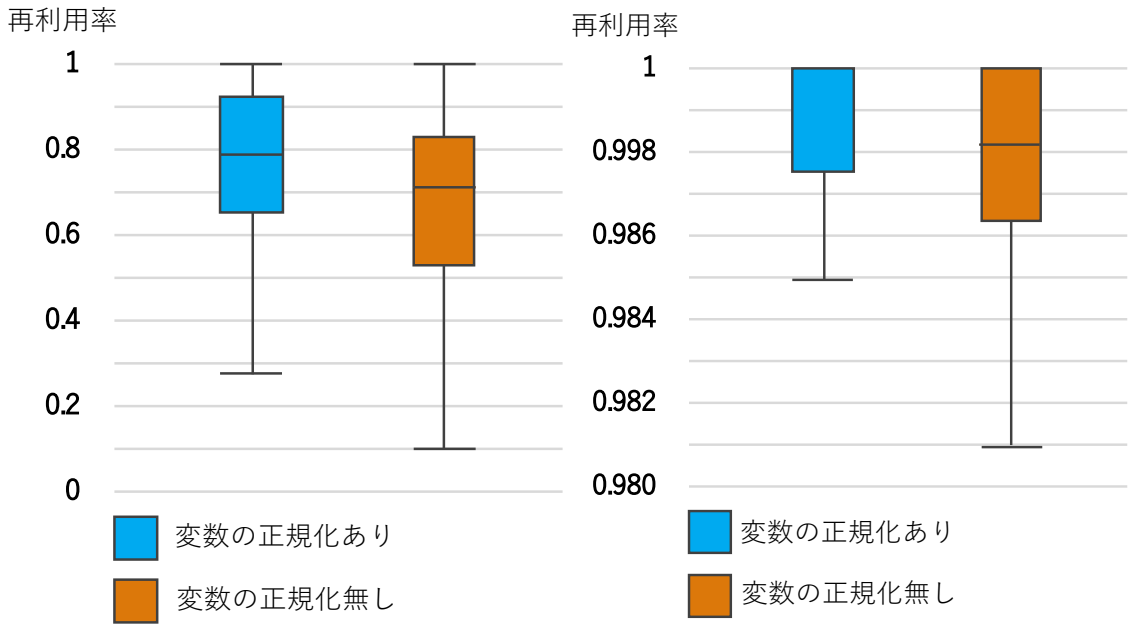
図 4　変数の正規化の流れ

をした解答である．図 5(a) より調査対象の半分の解答に含まれるプログラム文の 80% は，過去に出題された A 問題の解答に含まれるプログラム文であった．また，図 5(b) より調査対象にした 140 問の A 問題の半分に当たる 70 問の解答に含まれるプログラム文は，変数を正規化すれば過去に出題された A 問題に対する解答に含まれるプログラム文と一致した．

4.2　テンプレートの調査

4.2.1　調査目的

本調査は ABC の A 問題のテンプレートを分類するために行った．



(a) 各解答の再利用率

(b) 各問題の再利用率の最大値

図 5　再利用率の調査結果

4.2.2　調査方法

制御構文を使わずに分岐や繰り返しを実現している解答を除いた 27,151 個の解答からテンプレートの抽出を行った．以下の条件のいずれかを満たす解答を制御構文を使わずに分岐や繰り返しを実現している解答とした．

- ラムダ式を使用していない
- 三項演算子を含んでいない

各 A 問題に対する解答の中で最も多いテンプレートをその A 問題に対応するテンプレートとした．

4.2.3　調査結果

テンプレートの調査結果を表 1 に示す．A 問題は 13 種類のテンプレートに分類でき，9 割以上の A 問題は 5 種類のテンプレートのいずれかであった．さらに，43.6% の問題は分岐を 1 つだけ含む if{}else{}テンプレートであった．

5　提案手法

5.1　概要

本節では，プログラミングコンテストの解答を GenProg を応用し生成する手法について説明する．図 6 は手法の流れである．入力は以下の 3 つである．

- 解答を生成したいプログラミングコンテストの問題文
- テストケース
- 解答データベース

出力として与えられたテストケースを成功する解答が得られる．

提案手法は以下のステップで構成される．

Step 0 前準備　解答データベースを作成し，過去のプログラミングコンテストの問題文とテンプレートの関係性を学習する．

Step 1 テンプレート推測部　解きたいプログラミングコンテストの問題文からその問題を解くために必要なテンプレートを推測する．

表 1　テンプレートの調査結果

テンプレート	数	割合	割合の累積和
if{}else{}	61	43.6%	43.6%
flat	49	35.0%	78.6%
if{}else if{}else{}	10	7.14%	85.7%
for{}	5	3.57%	89.3%
for{if{}}	5	3.57%	92.9%
if{}if{}	2	1.43%	94.3%
for{if{}else{}}	2	1.43%	95.7%
if{}if{}if{}else if{}else{}	1	0.714%	96.4%
if{}for{if{}}	1	0.714%	97.1%
if{}else if{}else if{}else if{}else if{}else if{}	1	0.714%	97.9%
if{}else if{}else if{}else{}	1	0.714%	98.6%
if{}else{}if{}else{}if{}else{}	1	0.714%	99.3%
if{}else{}if{}else{}	1	0.714%	100.0%

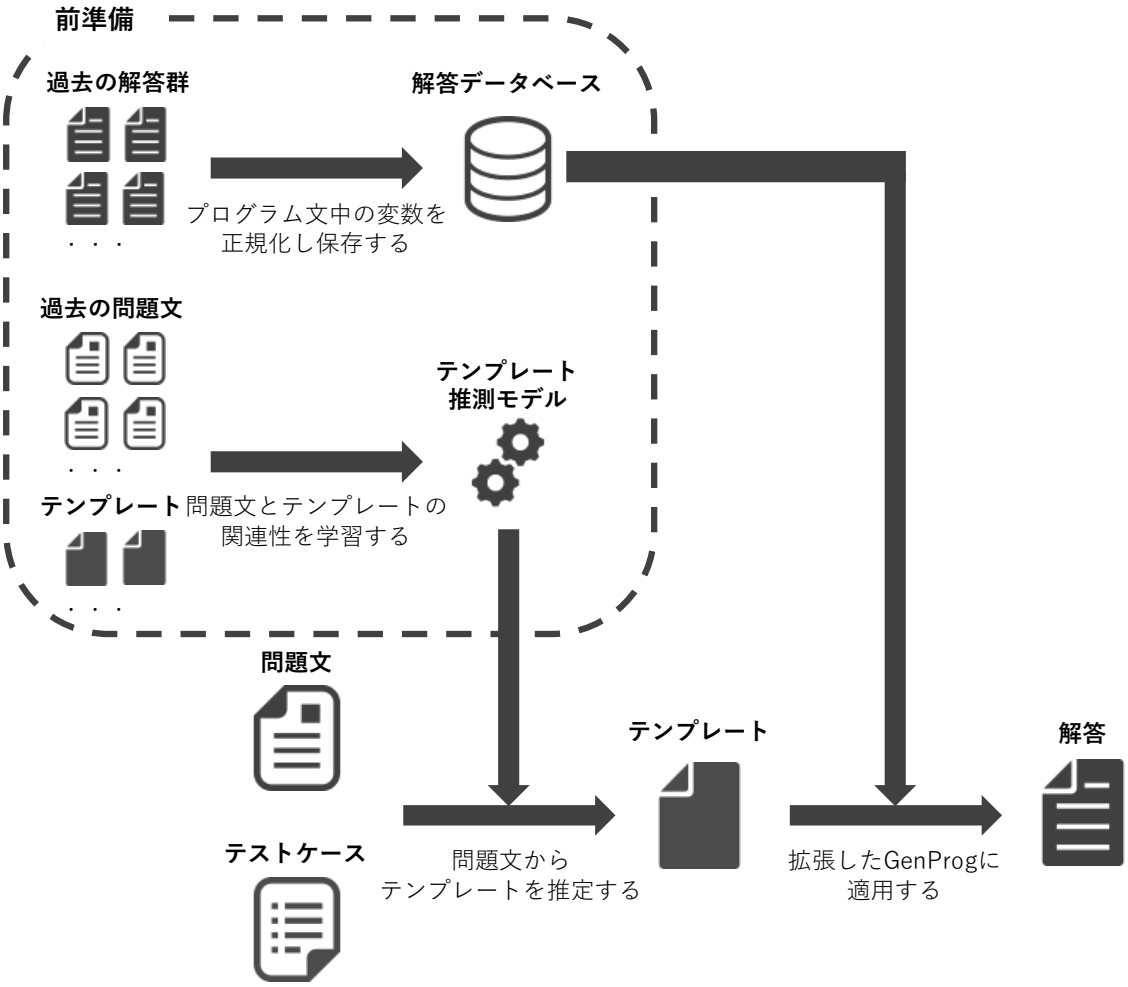


図 6　手法の流れ

Step 2 プログラム生成部　GenProg，推測したテンプレートおよび解答データベースを用いてプログラムを生成する．

Step 0 では，前準備としてプログラミングコンテストの問題に対する Java で記述された解答からプログラム文・式を抽出し，解答データベースに保存する．以降，解答データベースの作成処理をデータベース作成部という．さらに，過去の問題文とテンプレートの関係性を学習し，テンプレート推測モデルを作成する．以降，テンプレート推測モデルの作成処理をテンプレート学習部という．この Step は一度だけ実行すればよい．

Step 1 では，テンプレート推測モデルを用いて，プログラミングコンテストの問題文からテンプレートを推測し，解答の雛形を作成する．

Step 2 では，Step 1 で作成した雛形と入出力例をテストケースとして拡張した GenProg に与え，プ

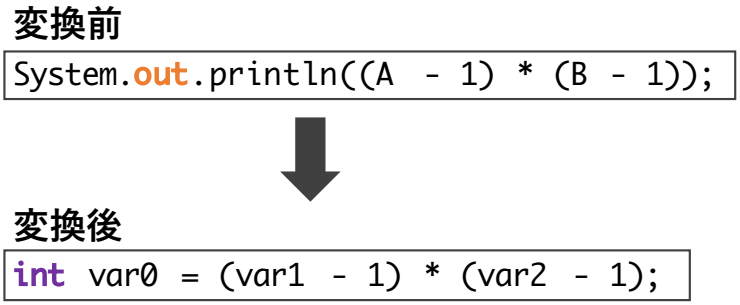


図 7　出力処理の変換の例

ログラムの生成を行う．GenProg は解答データベースから再利用するプログラム文・式を検索する．

5.2　データベース作成部

プログラミングコンテストの問題に対する Java で記述された解答からプログラム文・式を抽出し，解答データベースを作成する．解答データベースのスキーマを以下に示す．

- コンテスト名
- プログラム文・式
- プログラム文の種類
- プログラム文・式に含まれる変数の情報（型，final 修飾子の有無）

解答データベースのレコード数を削減するために，プログラム文・式に含まれる変数を正規化し，解答データベースに保存する．変数の正規化は，4.1 節と同じ方法を用いる．解答データベースには全く同じプログラム文は存在しない．さらに，再利用しても解答の生成に影響しないと考えられるプログラム文・式は解答データベースに保存しない．例えば，int j; のような初期化をしていない変数宣言文を再利用してもその変数を利用した式から解答を得られるとは考えにくい．また，再利用できない標準出力への出力処理は図 7 のように出力を行うメソッドへの引数を抽出し変数を正規化した上で変数宣言文に変換する．

5.3　テンプレート学習部

テンプレート学習部では，問題文とテンプレートの関係性を学習する．問題文とテンプレートのような異なる種類の情報の関係性を学習する手法として Joint Embedding [17] と呼ばれる手法がある．Joint Embedding では，入力をニューラルネットワークを用いてベクトル化し，コサイン類似度を入力の関係性が高ければ 1，関係性が低ければ 0 になるように学習する．コサイン類似度とは，2 つのベクトルの向きの近さを表す値である．

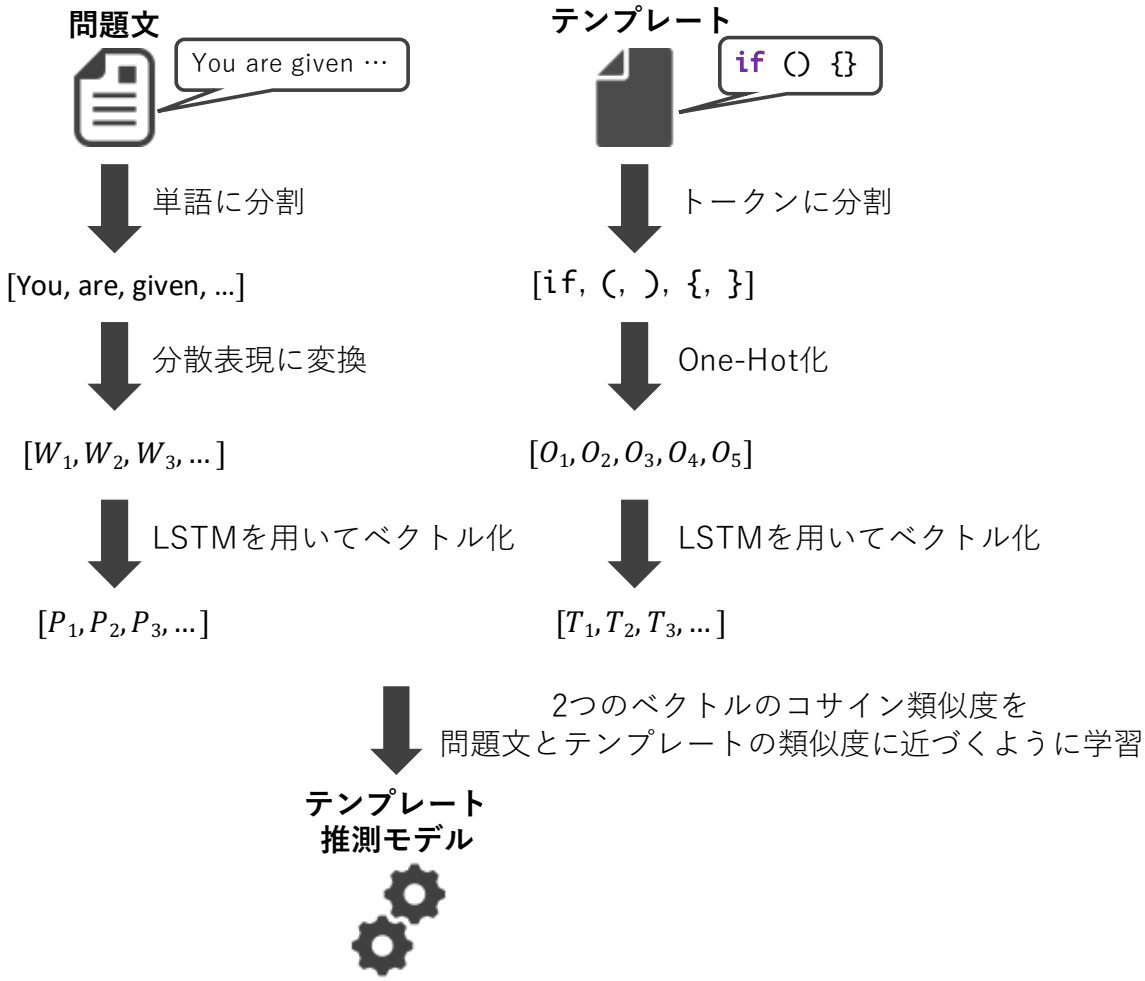


図 8　テンプレート学習部の流れ

提案手法では，図 8 のように問題文を単語に分割してから分散表現に変換し，LSTM を用いてベクトルに変換する．しかし，AtCoder の A 問題は，2021 年 2 月 2 日現在で 190 問しかないため，出現する単語の種類が少ない．単語数の不足を補うために，英語版 Wikipedia の記事から作成したモデル[*8]を利用して，単語を分散表現に変換した．一方で，テンプレートは，図 8 のようにトークンに分割してから One-Hot 化を行い，LSTM を用いてベクトルに変換する．One-Hot 化とは，スカラー値を $[0, 0, 1, 0]$ のような 1 成分だけが 1 であるベクトルに変換する処理である．2 つのベクトルのコサイン類似度が問題文とテンプレートの類似度に近づくように学習する．本研究では，このモデルをテンプレート推測モデルという．

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

5.3.1 学習データ

学習に用いるデータは，以下の 3 つの情報から構成される．

- 問題文
- テンプレート
- 問題文とテンプレートの類似度

類似度とは，問題文とテンプレートの関連性を表す値であり，0 から 1 の値をとる．問題文とテンプレートの関連性が高いほど，類似度は 1 に近づく．問題文とテンプレートの類似度は，その問題に対応するテンプレート（以降，正解テンプレートという）とテンプレートの類似度とした．テンプレート t とテンプレート s の類似度 $S(t,s)$ は以下の式から求める．

$$S(t,s) = Max\left(\frac{C - TED(t,s)}{C}, 0\right)$$

TED とは，t と s の Tree Edit Distance である．Tree Edit Distance とは，木構造で表したデータの類似度である．C は閾値であり，TED が閾値以上であれば類似度は 0 になる．

5.4 テンプレート推測部

まず，テンプレート学習部では，テンプレート推測モデルを用いて解きたい問題文とテンプレートの類似度を求め，最も類似度が高いテンプレートをプログラム生成部に入力として渡す．

5.5 プログラム生成部

GenProg の Java 実装である kGenProg[18] を再利用候補を解答データベースからランダムに取得するように拡張した．

解答データベースから取得したプログラム文の変数はすべて正規化されているため，プログラム文の挿入あるいは置換時に変数名の書き換えを行うように拡張した．

挿入するプログラム文内で変数が宣言済みか否かによって変数名の書き換え方法が異なる．挿入するプログラム文内で宣言されている変数は，挿入先よりも前の行で宣言されている変数と変数名が衝突しないように変数名を書き換える．挿入するプログラム文内で宣言されていない変数は以下の条件を満たす変数のいずれか 1 つと同じ変数名に書き換える．また，条件を満たす変数が存在しない場合は，挿入したとしてもコンパイルができないためプログラム文の挿入をしない．

- 挿入先よりも前の行で宣言されている
- 型が一致する

挿入前のプログラム

```
...
10  public int solveA(int n, String m) {
11    String k = n + m;
12    int i = n * n;
13
...
```

挿入先

挿入後のプログラム

```
...
10  public int solveA(int n, String m) {
11    String k = n + m;
12    int i = n * n;
13    int var0 = i + n;
...
```

挿入するプログラム文

```
int var0 = var1 + var2;
```

変数名	型	宣言済みか
var0	int	T
var1	int	F
var2	int	F

図 9　変数の書き換え例

変数の書き換えの例を図 9 に示す．13 行目（緑色の行）がプログラム文の挿入先である．挿入するプログラム文に含まれる宣言されている変数は $val0$ である．13 行目以前で宣言されている int 型である変数は n および i であり，そのままの変数名でも変数名は衝突しないため，$val0$ の変数名は書き換えない．挿入するプログラム文に含まれる宣言されていない変数は $val1$, $val2$ である．これらの変数を n または i のいずれかに書き換える．

5.5.1　適応度関数

提案手法では，期待値と出力値の近さを表現するために，テストの通過率に加えて期待値と出力値のレーベンシュタイン距離を適応度の計算に用いる．プログラム x のテストの通過率を $S(x)$，テストケースの総数を n，テストケース y の期待値とプログラム x の出力値をレーベンシュタイン距離を $L(x, y)$ とすると，適応度関数 $F(x)$ は以下の式で表される．

$$F(x) = (1 - S(x)) + \sum_{i=0}^{n} L(x, y_i)$$

この適応度関数を適用すると，より多くのテストに通過しより期待値に近い値を出力するプログラムほど適応度は 0 に近づく．

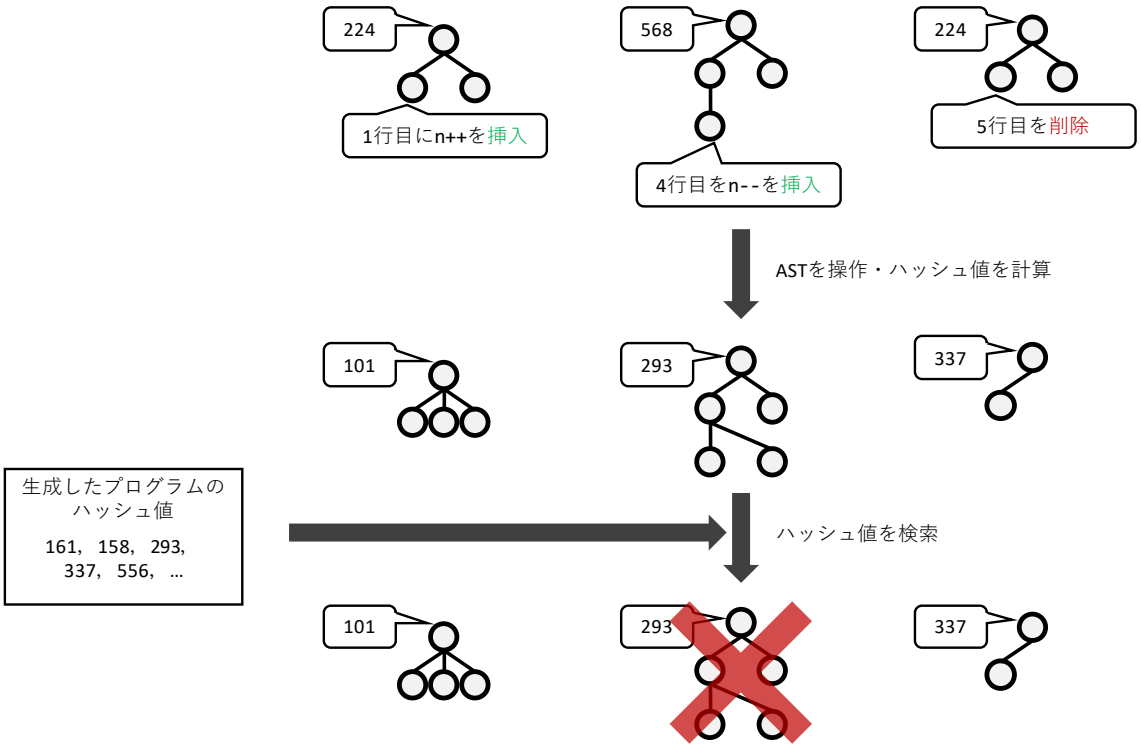


図 10　kGenProg の重複排除処理

5.5.2　重複排除処理

GenProg では，バグを修正するために非常に多くのプログラムを生成する．そのため，ビルドや抽象構文木（以降，AST という）の操作が実行時間の多くを占めている．図 10 のように，kGenProg は生成したプログラムのハッシュ値を保存し，ビルドの前に同じハッシュ値を持つプログラムの有無を調べる．同じハッシュ値を持つプログラムが存在した場合は，そのプログラムに対してビルドを実行しない．この処理により，kGenProg は無駄なビルドの回数を減らしている．提案手法の重複排除処理を図 11 に示す．まず，AST の操作の前にハッシュ値と行う操作を調べ，過去に同じソースコードに同じ AST の操作を行ったのであれば AST の操作を行わない．次に，kGenProg と同様にハッシュ値を用いた重複排除も行う．この処理によって，無駄な AST の操作を減らす．以降，AST 操作の前に重複を排除する処理を事前重複排除処理という．

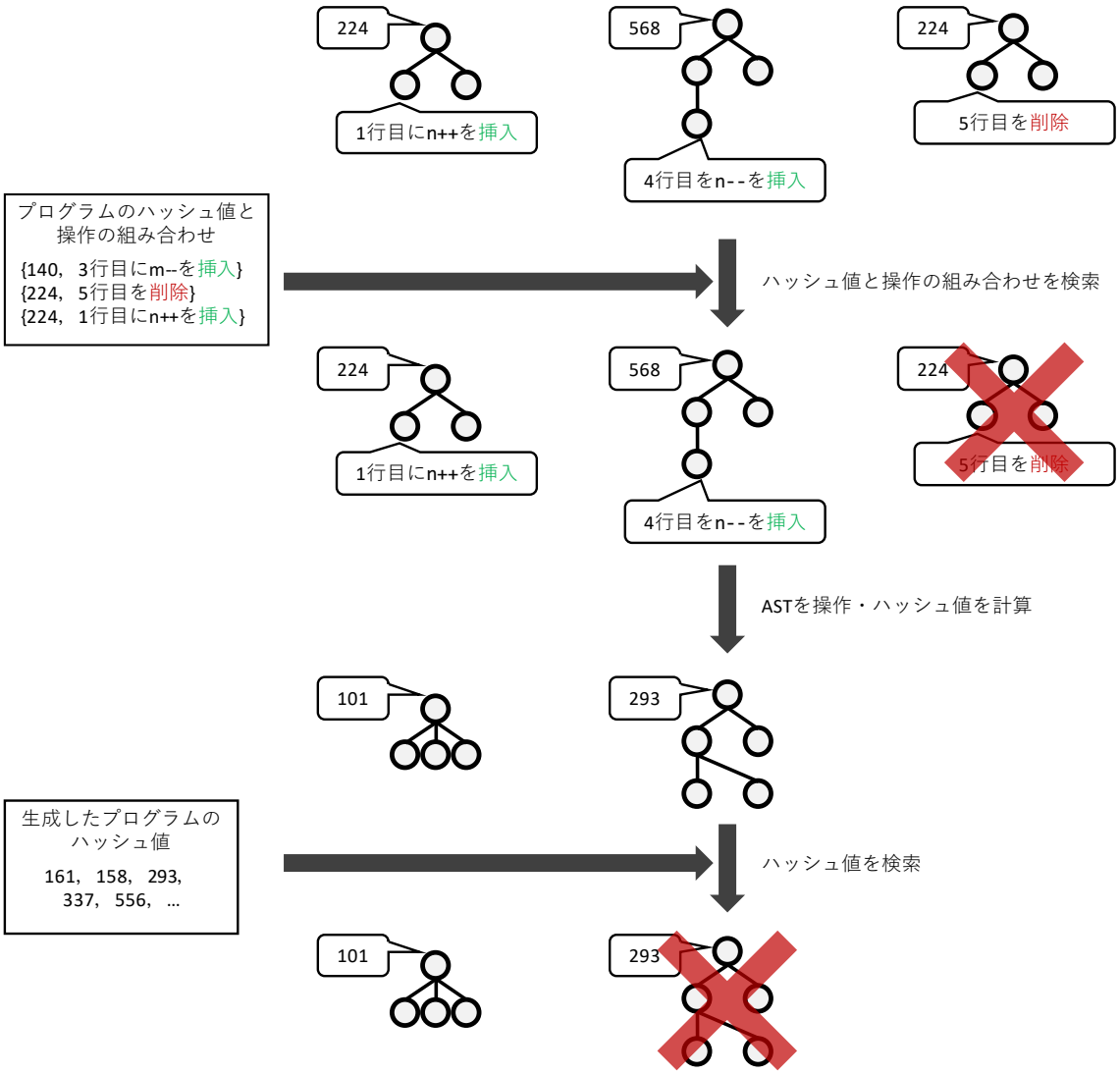


図 11　提案手法の重複排除処理

6　評価実験

本研究では，以下の 5 個の評価実験を行った．

1. テンプレート推測部の評価
2. プログラム生成部の評価
3. テンプレートの有効性の評価
4. 解答データベースの評価

5. 重複排除処理の評価

6.1 テンプレート推測部の評価

6.1.1 実験対象

ABC42 以前の A 問題は，現在の A 問題と傾向が異なる．そのため，ABC42 以降の A 問題から 2019 年 9 月 9 日以前に出題された 99 問の A 問題を対象とした．

6.1.2 実験方法

本実験では，対象とした問題の正解テンプレートをその問題に対する解答の中で最も多いテンプレートとした．学習に用いたテンプレートは以下の 10 種類である．

- if{}else{}
- flat
- if{}else if{}else{}
- for{if{}}
- if{}if{}
- for{}
- for{if{}else{}}
- if{}if{}if{}else if{}else{}
- if{}else{}if{}else{}if{}else{}
- if{}else{}if{}else{}

Leave-One-Out を用いて学習を行った．Leave-One-Out とは，データを N 個に分割し，1 つのデータを評価，残りのデータを学習に用いる手法である．例えば，ABC51 を評価する場合は，ABC51 以外の問題文を用いて学習する．

6.1.3 実験結果

実験の結果を図 12 に示す．図の横軸が順位を表しており，縦軸が横軸の順位以内に正解テンプレートが含まれていた問題の割合を表している．図 12 より，問題文と最も類似度が高いテンプレートがその問題の正解テンプレートと一致した問題は 46 問であった．テンプレートの推測に成功した．また，約 7 割の問題では上位 4 つのテンプレートのいずれかが正解テンプレートであった．

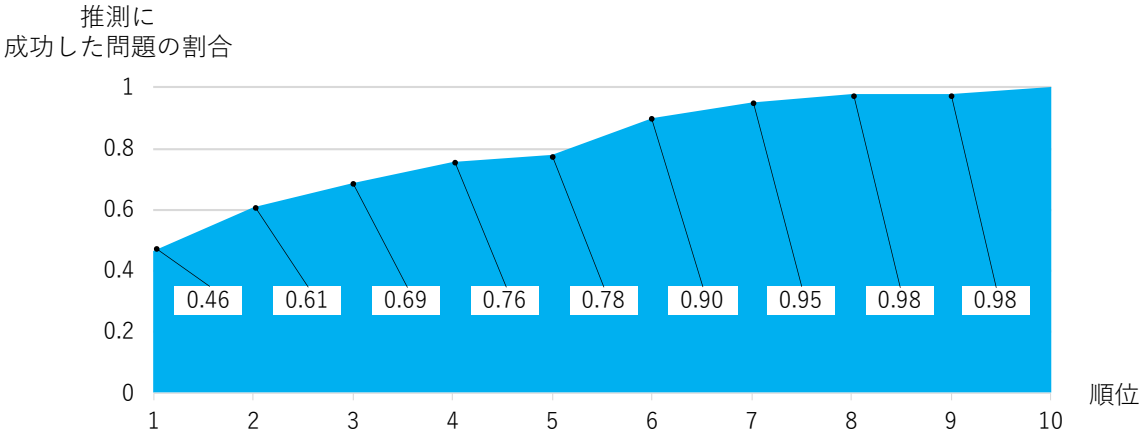


図 12　テンプレート推測モデルの実験結果

6.2　プログラム生成部の評価

6.2.1　実験対象

以下の条件を満たす 35 問の A 問題を対象とした．

- 全てのテストケースが公開されている
- 正解テンプレートに繰り返しを含まない
- 2019 年 9 月 9 以降に出題された

6.2.2　実験環境

本実験は 2.53GHz で駆動する CPU を 2 コア，16GB のメモリを割り当てた仮想マシン上で行った．

6.2.3　実験方法

各 A 問題に対して，Seed 値を変えて 10 回実行した．本実験で設定した kGenProg のパラメータを表 2 に示す．全てのテストケースに通過するプログラムが生成された時点で実行を打ち切った．

表 2　kGenProg のパラメータ

変異で生成するプログラムの数	10
交叉で生成するプログラムの数	0
1 世代あたりに選択するプログラムの数	10
制限時間	15 分

```
...
11   int solveA(int ARG_0, int ARG_1) {
       if ((ARG_0 % ARG_1 > 0) || (ARG_0 % CONST_3 > 0)) {
12       ARG_1 += Math.max(Math.max(ARG_0, CONST_2) - 1,
                                    Math.min(ARG_0, CONST_2));
13       int $$_11 = ARG_1 + (CONST_2 / 2);
14       int $$_15 = Math.min(ARG_0 + ARG_0,
                              Math.min(CONST_3 + ARG_0, ARG_0 + ARG_1));
15       ARG_0 = (CONST_2 + ARG_0) - 24;
16     } else {
17       return CONST_2;
18     }
19
20     int $_0 = Math.max(0, ARG_1 - ARG_0 + CONST_2);
21     $_0 = $_0 % $_0;
22     $_0 *= $_0;
23     return CONST_3;
24   }
```

意味のないプログラム文

図 13　意味のないプログラム文が含まれるプログラム文

6.2.4　実験結果

実験の結果，10 問の A 問題に対して解答の生成に成功した．解答の生成に失敗した理由として以下の 2 つが挙げられる．

- プログラムの生成に失敗しやすい
- プログラムの生成を繰り返すほど if 文の条件式の書き換えが発生しにくくなる

現在，再利用するプログラム文はランダムに選択している．そのため，変数の書き換えができずにプログラムの生成に失敗しやすいと考えられる．図 14 に変数の書き換えに失敗する例を示す．12 行目（緑色の行）がプログラム文の挿入先である．挿入するプログラム文に含まれる宣言されている変数は $val0$，書き換えが必要な変数は $val1$ および $val2$ である．しかし，12 行目以前に書き換えが必要な変数と同じ String 型である変数は存在しない．その結果，変数の書き換えに失敗する．

提案手法では，解答データベースに保存されている単文や式を再利用してプログラムを生成する．プログラムの生成を繰り返すと図 13 のような意味のないプログラム文が多く含まれるプログラムが生成される．そのため，それらのプログラム文に対する操作が増え，解答の生成が難しくなると考えられる．

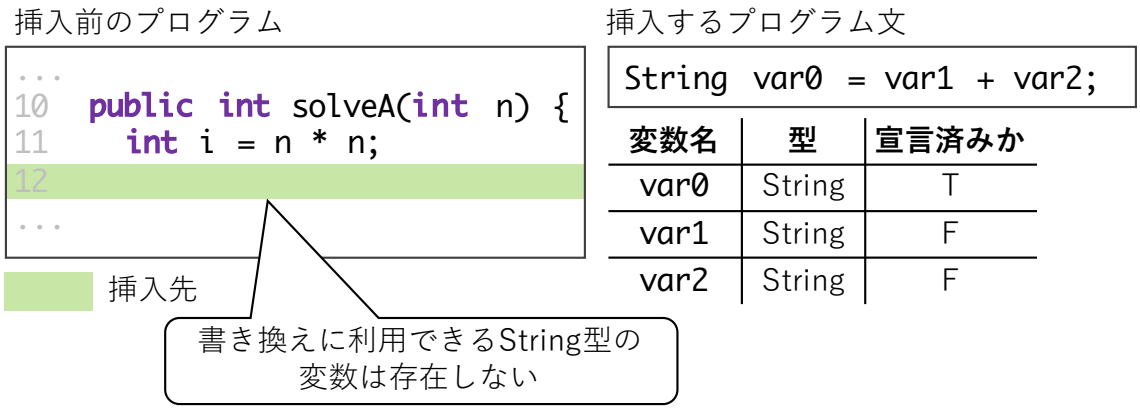


図 14　変数の書き換えの失敗例

6.3　テンプレートの評価

6.3.1　実験方法

6.2 節と同じ問題を対象とし，同じパラメータを設定した．正解テンプレートを用いる kGenProg（以降，kGenProg-A という）および常に flat テンプレートを用いる kGenProg（以降，kGenProg-B という）に対して Seed 値を変えて 10 回実行した．常に flat テンプレートを用いると分岐が必要な問題が解けないため，kGenProg-B は，単文に加えて if 文も再利用できるようにした．

6.3.2　実験結果

kGenProg-A は 10 問の A 問題，kGenProg-B は 12 問の A 問題に対して解答の生成に成功した．A 問題の種類別に解答の生成に成功した A 問題の数を表 3 に示す．A 問題の入力値と出力値の型を用いて，A 問題を以下の 3 種類に分類した．

計算問題　四則演算や数値の比較を用いて解く問題である．例えば，半径 r の円に内接している正十二角形の面積を求める問題[*9]は計算問題である．

文字列操作問題　文字列の結合や置換などの文字列に対する操作を用いて解く問題である．例えば，大文字のアルファベットのみを含む文字列の一部を小文字に変換する問題[*10]は文字列操作問題である．

分類問題　文字列の結合や置換などの文字列に対する操作を用いて解く問題である．例えば，ある整数が 7，5，3 のいずれかであれば Yes，そうでなければ No と出力する問題[*11]は文類問題である．

[*9] https://atcoder.jp/contests/abc134/tasks/abc134_a
[*10] https://atcoder.jp/contests/abc126/tasks/abc126_a
[*11] https://atcoder.jp/contests/abc114/tasks/abc114_a

```
...
57      if (2 == CONST_3 && 2 == CONST_4) {
58        ARG_0 = $_0 * 2;
59      } else if (CONST_4 / 10 % 111 == 0 || CONST_4 % 1000 % 111 == 0) {
60        ARG_0 = $_0 + ($_0 - 1);
61      } else {
62        if (CONST_4 == 23) {
63          if (CONST_3 + CONST_2 <= 23) {
64            if ($_0 > '0') {
65            }
66          } else if (CONST_2 - 2 >= 0) {
67          } else {
68            int $_5 = ($_0 + ARG_0 + 10 * $_0);
69          }
70        } else {
71        }
72      }
...
```

図 15　テンプレートを用いない場合に生成された解答の一部

表 3 より，テンプレートを用いない kGenProg-B はより多くの分類問題に対して解答の生成に成功し，テンプレートを用いた kGenProg-A はより多くの計算問題に対して解答の生成に成功した．

kGenProg-B が kGenProg-A よりも多くの分類問題に対して解答の生成に成功した理由として，分岐条件の書き換えが行われる可能性が高いことが挙げられる．分類問題を解くためには分岐条件の書き換えが必要である．テンプレートを用いた場合は，単文のみが再利用される．プログラムの生成を繰り返すと，多くのプログラム文が挿入されるため，分岐条件の書き換えが行われにくい．一方で，テンプレートを用いない場合は単文に加えて if 文も再利用される．プログラムの生成を繰り返して多くの文が挿入されてもプログラムに複数の if 文が含まれている．そのため，テンプレートを用いた場合よりも分岐条件の書き換えが行われやすい．

kGenProg-A が kGenProg-B よりも多くの計算問題に対して解答の生成に成功した理由として，無駄な分岐が発生しないことが挙げられる．テンプレートを用いない場合は if 文も再利用するため，図 15 のような if 文が入れ子になっているプログラムが生成される．不要な分岐によってある入力値に対して

表 3　解答の生成に成功した問題の数（問題種類別）

	計算問題	文字列操作問題	分類問題
kGenProg-A のみ	1	0	4
kGenProg-B のみ	3	0	0
kGenProg-A・kGenProg-B の両方	4	1	2

正しい値を返すがほかの入力値に対しては誤った値を返してしまうプログラムが生成されると考えられる．一方で，テンプレートを用いた場合は単文のみが再利用されるため，無駄な分岐が発生しない．

6.4 変数の正規化の評価

6.4.1 実験方法

6.2 節と同じ問題を対象とし，同じパラメータを設定した．変数の正規化をした解答データベースを用いる kGenProg（以降，kGenProg-C という）と正規化をしていない解答データベースを用いる kGenProg（以降，kGenProg-D という）に対して Seed 値を変えて 10 回実行した．解答データベースのレコード数を表 4 に示す．

6.4.2 実験結果

kGenProg-C は 1 問も解答を生成できなかったが，kGenProg-D は 10 問の A 問題に対して解答を生成できた．したがって，変数の正規化と変数の書き換えは非常に有効といえる．変数の正規化によって探索空間を大幅に削減できた結果だと考えられる．

6.5 重複排除処理の評価

6.5.1 実験方法

6.2 節と同じ問題を対象とし，同じパラメータを設定した．事前重複排除を実装した kGenProg（以降，kGenProg-E という）と実装していない kGenProg（以降，kGenProg-F という）に対して Seed 値を変えて 10 回実行した．

6.5.2 実験結果

kGenProg-E と kGenProg-F は，10 問の A 問題に対して解答の生成に成功した．事前重複排除によって新たに解けた問題は存在しなかった．図 16 は，解答の生成に成功した試行の実行時間をプロットした箱ひげ図であり，青色は kGenProg-E，kGenProg-F の実行時間である．図 16 より，事前重複排除の有無によって実行時間は変化しなかった．

表 4　解答データベースのレコード数

	プログラム文	プログラム式
変数の正規化を行った解答データベース	79,428	6,363
変数の正規化を行っていない解答データベース	627,626	16,878

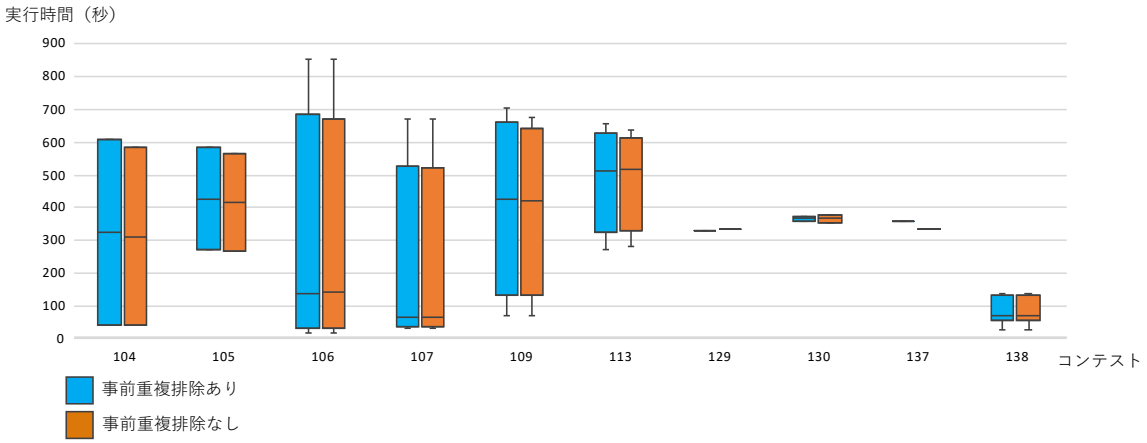


図 16　実行時間

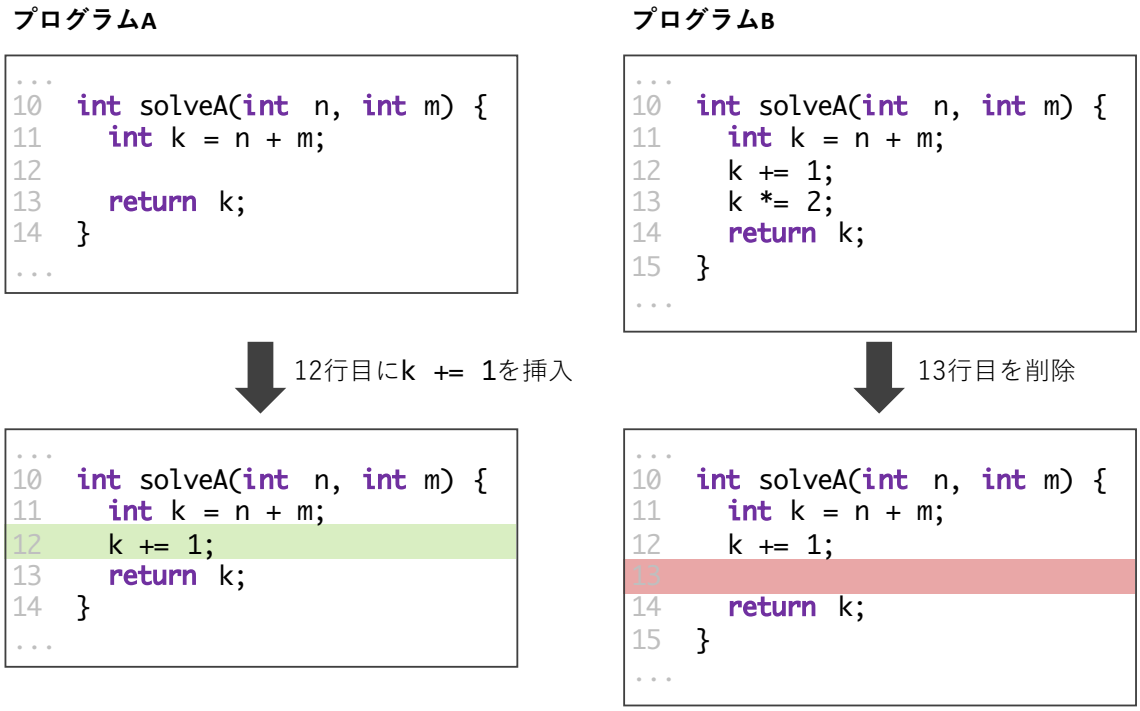


図 17　異なるプログラムと異なる操作の組み合わせで同じプログラムが生成される例

実行時間に変化がなかった理由として，異なるプログラムと異なる操作の組み合わせでも同じプログラムが生成できることが挙げられる．図 17 に例を示す．プログラム A の 12 行目に k += 1; を挿入したプログラムとプログラム B の 13 行目を削除したプログラムは全く同じである．事前重複排除処理は同じプログラムに対して同じ AST の操作を行う場合のみ有効であるため，図 17 のような場合では AST の操作を行う前に重複を排除できない．

7　妥当性の脅威

異なる 10 個の Seed 値を用いて実験を行った．そのため，本研究と異なる Seed 値を用いるあるいは Seed 値の数を増やすと実験の結果が大きく変わる可能性がある．

8　関連研究

本章では，プログラムの自動生成に関する研究を紹介する．入出力仕様からプログラムを自動生成する手法の 1 つに Microsoft 社が開発した DeepCoder [19] がある．DeepCoder は与えられた入出力仕様からその入出力仕様を満たす 34 種類ある命令の組み合わせを深さ優先探索を行い探索する．DeepCoder は深層学習を用いて制御構文，演算子や定数などの出現確率を予測したモデルを利用して，効率的に深さ優先探索を行う． Becker らが提案した AI Programmer [20] は，遺伝的アルゴリズムを用いてプログラムを生成する手法である．AI Programmer は，与えられた入出力仕様からポインタの操作やメモリの内容の書き換えなどのプリミティブな命令を組み合わせてその入出力仕様を満たすプログラムを生成する．DeepCoder や AI Programmer はいずれもその手法に特化した言語で書かれたプログラムを出力するため，生成できるプログラムには制約がある．

入出力仕様ではなく自然言語からプログラムを生成する手法もある．それらの手法の 1 つに Gvero らが提案した AnyCode [21] がある．AnyCode は関数の機能を自然言語で与えるとその機能を持つ関数の候補を出力する．

9　おわりに

本研究では，自動プログラム修正を応用した自動プログラム生成手法を提案した．AtCoder というプログラムコンテストで出題された問題を対象に実験をした．古いコンテストの問題は，傾向が固まっておらず現在の問題と難易度が異なる．そのため，現在の出題形式に近い問題を実験対象とした．現在の出題形式に近い 99 問の問題に対して雛形の推測を試みたところ 46 問で正しい制御構文の構造の推測に成功した．また，現在の出題形式に近く，すべてのテストケースが公開されている 35 問の問題に対して生成を試みたところ，10 問の問題で解答の生成に成功した．

今後の課題として，以下の 2 つが挙げられる．

- 再利用するプログラム文の検索方法の改善
- 定期的なリファクタリングの導入

現在，提案手法では再利用するプログラム文はランダムに選択している．そのため，変数の書き換えに失敗しやすい．この問題は，再利用するプログラム文をランダムではなく，挿入先以前の行で宣言されている変数の型のみを含んでいるプログラム文を解答データベースから検索すれば解決できる．

定期的にリファクタリングを行い無意味なプログラム文を削除すれば，問題を解くために必要でないプログラム文に対する変更を行う回数を減らせる．

謝辞

本研究を行うにあたり，わかりやすい指導をしていただいた楠本真二教授に，感謝申し上げます．本研究の計画や方向性に関して熱心な指導を賜りました肥後芳樹准教授および柗本真佑助教に，感謝申し上げます．本研究を進めるにあたり，多くのご助言をいただいた NTT ソフトウェアイノベーションセンタの倉林利行様に感謝申し上げます．

参考文献

[1] Tom Britton, Lisa Jeng, Graham Carver, Paul Cheak, and Tomer Katzenellenbogen. Reversible Debugging Software: Quantify the time and cost saved using reversible debuggers , 2013.

[2] Brent Hailpern and Padmanabhan Santhanam. Software debugging, testing, and verification . *IBM Systems Journal*, Vol. 41, pp. 4–12, 2002.

[3] L. A. Clarke. A System to Generate Test Data and Symbolically Execute Programs . *IEEE Transactions on Software Engineering*, Vol. SE-2, No. 3, pp. 215–222, Sep. 1976.

[4] Carlos Pacheco, Shuvendu K. Lahiri, Michael D. Ernst, and Thomas Ball. Feedback-Directed Random Test Generation . In *Proceedings of the 29th International Conference on Software Engineering*, pp. 75–84, 2007.

[5] James A. Jones and Mary Jean Harrold. Empirical Evaluation of the Tarantula Automatic Fault-localization Technique . In *Proceedings of the 20th ACM/IEEE International Conference on Automated Software Engineering*, pp. 273–282, 2005.

[6] Daniele Zuddas, Wei Jin, Fabrizio Pastore, Leonardo Mariani, and Alessandro Orso. MIMIC: Locating and understanding bugs by analyzing mimicked executions . *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, pp. 815–825, Sep. 2014.

[7] Claire Le Goues, Michael Dewey-Vogt, Stephanie Forrest, and Westley Weimer. A Systematic Study of Automated Program Repair: Fixing 55 out of 105 Bugs for $8 Each . In *Proceedings of the 34th International Conference on Software Engineering*, pp. 3–13, 2012.

[8] Yuhua Qi, Xiaoguang Mao, Yan Lei, Ziying Dai, and Chengsong Wang. The Strength of Random Search on Automated Program Repair . In *Proceedings of the 36th International Conference on Software Engineering*, pp. 254–265, 2014.

[9] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. SemFix: Program Repair via Semantic Analysis . In *Proceedings of the 2013 International Conference on Software Engineering*, pp. 772–781, 2013.

[10] J. Xuan, M. Martinez, F. DeMarco, M. Clément, S. L. Marcote, T. Durieux, D. Le Berre, and M. Monperrus. Nopol: Automatic Repair of Conditional Statement Bugs in Java Programs . *IEEE Transactions on Software Engineering*, Vol. 43, No. 1, pp. 34–55, Jan. 2017.

[11] K. F. Man, K. S. Tang, and S. Kwong. Genetic algorithms: concepts and applications [in

engineering design] . *IEEE Transactions on Industrial Electronics*, Vol. 43, No. 5, pp. 519–534, Oct. 1996.

[12] Rui Abreu, Peter Zoeteweij, Rob Golsteijn, and Arjan J. C. van Gemund. A Practical Evaluation of Spectrum-based Fault Localization . *J. Syst. Softw.*, Vol. 82, No. 11, pp. 1780–1792, Nov. 2009.

[13] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, Vol. 9, pp. 1735–80, 12 1997.

[14] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Learning Representations by Back-propagating Errors . *Nature*, Vol. 323, No. 6088, pp. 533–536, 1986.

[15] Yoav Goldberg and Omer Levy. word2vec explained: deriving mikolov et al.'s negative-sampling word-embedding method, 2014.

[16] Jeffrey Pennington, Richard Socher, and Christopher Manning. GloVe: Global vectors for word representation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing*, pp. 1532–1543. Association for Computational Linguistics, October 2014.

[17] 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*, p. 2346–2352. AAAI Press, 2015.

[18] Yoshiki Higo, Shinsuke Matsumoto, Ryo Arima, Akito Tanikado, Keigo Naitou, Junnosuke Matsumoto, Yuya Tomida, and Shinji Kusumoto. kGenProg: A High-performance, High-extensibility and High-portability APR System . In *the 25th Asia-Pacific Software Engineering Conference*, pp. 697–698, Dec. 2018.

[19] M. Balog, A. L. Gaunt, M. Brockschmidt, S. Nowozin, and D. Tarlow. Deepcoder: Learning to write programs. In *Proceedings International Conference on Learning Representations 2017*, Apr. 2017.

[20] Kory Becker and Justin Gottschlich. AI programmer: Autonomously creating software programs using genetic algorithms. *CoRR*, Vol. abs/1709.05703, , 2017.

[21] Tihomir Gvero and Viktor Kuncak. Synthesizing java expressions from free-form queries. *Acm Sigplan Notices*, Vol. 50, No. 10, pp. 416–432, 2015.