生成 AI を利用したプログラミング学習の補完問題生成手法

宮崎 章太[1] 長島 和平[1] 島袋 舞子[2] 兼宗 進[1]

概要：本研究では教科書や授業で使用するサンプルプログラムや例題について、コードの静的解析により急激に難易度が高くなっている箇所を検出し、生成 AI によりそのギャップを埋めるようなサンプルプログラムを自動生成する仕組みを提案する。

1. はじめに

プログラミング教育において、学習者が基本的な制御構文から複雑なネスト構造まで段階的に理解することは重要である [1]。しかし、既存の教材やサンプルプログラム群では、隣接するプログラム間で難易度に大きなギャップが生じることがある。このようなギャップは学習者の理解を阻害し、プログラミング学習の挫折要因となりうる [2]。

近年、生成 AI 技術の発展により、プログラムコードの自動生成が実用的なレベルに達している [3]。本研究では、この生成 AI 技術を活用し、プログラム間の難易度ギャップを自動的に検出・補完するシステムを提案する。

具体的には、連続するサンプルプログラム間で制御構造の複雑さに大きな差がある場合、その間に適切な難易度の中間プログラムを生成 AI（Large Language Model: LLM）により自動生成する。これにより、学習者にとってより理解しやすい段階的な学習環境を提供することを目的とする。

2. 現状の課題

プログラミング教育における既存の課題として、以下が挙げられる：

(1) **教材間の難易度ギャップ**：教科書や授業で使用されるサンプルプログラム間で、制御構造の複雑さに急激な変化が生じることがある

(2) **手動による中間教材作成の困難性**：教員が全ての学習段階に対応した適切な中間難易度の問題を手動で作成することは時間的・労力的に困難

(3) **学習者個別のペースへの対応不足**：画一的な教材では、学習者の理解度や学習ペースの違いに対応しきれない

先行研究 [1] では、これらの課題に対してプログラム間の構文ギャップを検出する De-gapper ツールを提案した。しかし、検出されたギャップを補完するためのプログラムの生成は、授業を担当する教員の人手作業に依存していた。本研究では、この補完プログラム生成を生成 AI により自動化することで、より実用的なシステムの実現を目指す。

3. 実装

大きなギャップがあるプログラム間に中間難易度のプログラムを挿入することで、難易度を平滑化するシステムを開発した。中間難易度のプログラムとは、ギャップに含まれる新規要素のうち 1 つだけを導入し、残りは既習構文のみで構成されるプログラムを指す。ギャップサイズの閾値を 2 以上に設定した理由は、1 つの新規要素では学習者にとって負担が少ない一方、2 つ以上の新規要素が同時に導入される場合には学習の困難度が急激に上昇するためである。また、学習順序は、プログラムの制御構文の複雑さに基づいてソートされている。具体的には、制御構文の種類（if, for など）とそのネストの深さを基準に、小規模な構文から段階的に複雑な構文へと進む順序で配置されている。

システムの全体構成を図 1 に示す。

本システムは以下の処理フローで動作する：

(1) 連続したプログラム間のギャップを検出

(2) ギャップが閾値（本研究では 2）以上の場合、中間プログラム生成を実行

(3) 生成 AI による制約条件付きプログラム生成

(4) 生成されたプログラムの構文検証と条件適合性判定

(5) 適切な中間プログラムを学習順序に挿入

3.1 プログラム間のギャップ検出

本システムではプログラム間のギャップ検出に先行研究である De-Gapper で提案されている手法を参考に簡略化した手法を使用した。具体的には、Python における主要な制御構文（if, elif, else, for, break, continue）およびこれ

[1] 大阪電気通信大学
Osaka Electro-Communication University
[2] 沖縄国際大学
Okinawa International University

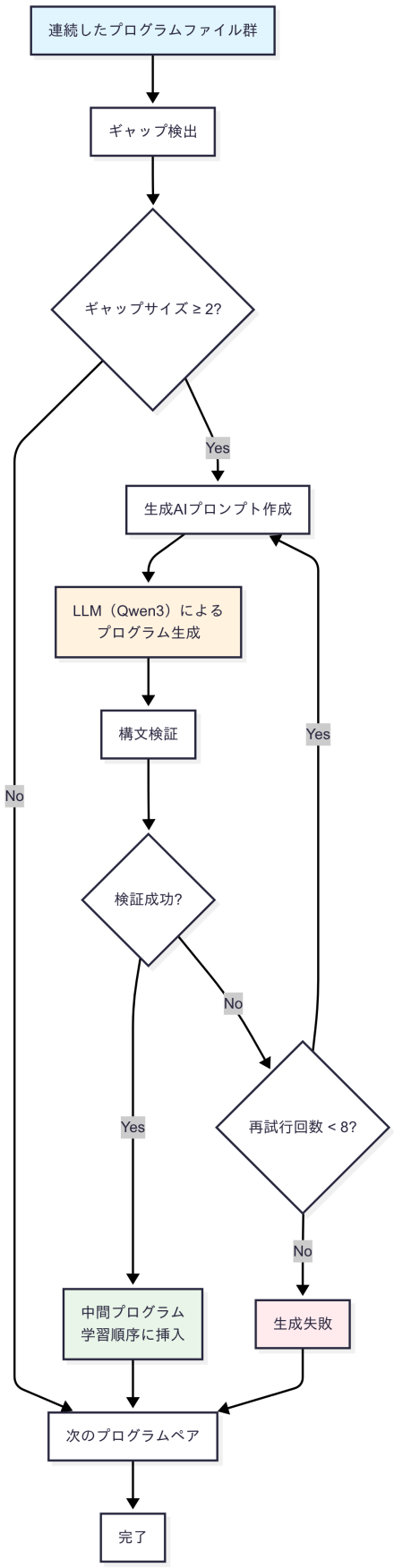

図 1　システム全体構成

らのネスト構造のみに焦点を絞った。この簡略化により、解析負荷を軽減し、教育で頻出する構文のみに限定することで実用性を高めた。

本システムは、入力された Python プログラムを抽象構文木（AST）を用いて静的に解析し、制御構造を抽出する。プログラムファイルを順番に構文解析し、各プログラムで新しく出てきた構文を抽出する。これがプログラム間の「ギャップ」である。先行研究ではプログラミング言語の全構文要素に対して検出を行っていたが、本研究では Python における一部の制御構文（if, elif, else, for, break, continue）とそれらのネスト構造を対象に抽出を行った。

制御構造は、そのネスト関係をスラッシュ区切りで表現する階層パス形式で表現される。例えば、「for 文のブロック内に if 文がある」場合は「for/if」、「if 文の中に for 文がありさらにその中に break 文がある」場合は「if/for/break」と表現する。

図 3 は元のサンプルプログラムの例であり、図 4 はより複雑な構文を含むプログラムの例を示している。この 2 つのプログラム間では、図 5 に示すように「elif」「else」「if/if」の 3 つの新規要素が検出され、ギャップサイズ[*1]が 3 であると判定される。具体的には、図 3 のプログラムで学習済みの制御構造は if のみである。これに対し図 4 のプログラムでは、if に加えて新たに elif と else が同じ階層で出現し、さらに if 文のブロック内部に別の if 文が記述されている（ネスト構造 if/if）。したがって、これら 3 つが新規学習要素として検出される。

3.2　生成 AI による中間プログラム生成

ギャップサイズが閾値以上の場合、生成 AI を用いて中間難易度のプログラムを生成する。生成時には以下の制約条件を設定する：

- **必須要素**：新たに学習すべき制御構造（1 つずつ段階的に導入）
- **許可要素**：既に学習済みで使用可能な制御構造
- **禁止要素**：まだ学習していない制御構造

例えば、「if」のみ学習済みの状態で「elif」を新たに導入する場合、「elif」を必須要素、「if」を許可要素、「else」「for」等を禁止要素として設定し、プロンプト生成を行う。

図 2 に実際の生成 AI へのプロンプト例を示す。このプロンプトでは、学習段階に応じた制約条件を明確に指定し、教育的に適切なプログラムが生成されるよう工夫している。

AI によって生成されたコードに対し、本システムは再度 AST 解析を実行する。これにより、まず Python のプログラムとして構文的に正しいか（syntax error がないか）を検証する。次に、抽出された制御構造が、プロンプトで与えた制約条件（必須要素、許可要素、禁止要素）を完全

[*1] ギャップサイズは新規要素数のことで、「$gap\_size = |current\_elements - learned\_elements|$」で計算される。

```
Generate a Python program that demonstrates the use
    of elif.

Requirements:
– MUST use: elif
– MAY use: if
– MUST NOT use: else, for, break, continue

Please create a simple, educational Python program
that introduces elif statement in a clear way.
The program should be easy to understand for
programming beginners.

Example format:
– Use simple variable assignments
– Include clear print statements
– Focus on demonstrating elif usage
```

図 2 生成 AI へのプロンプト例（elif を導入する際の制約条件付きプロンプト）

```
n = int(input("number:␣"))
if n % 2 == 0:
    print("even")
```

図 3 プログラム例 1

```
n = int(input("number:␣"))
if n % 2 == 0:
    print("2の倍数")
    if n % 4 == 0:
        print("4の倍数")
elif n % 3 == 0:
    print("3の倍数")
else:
    print("その他")
```

図 4 プログラム例 2

```
elif, else, if/if
```

図 5 抽出される 3 つの新規構文要素

に満たしているかを判定する。条件を満たさない場合、同じプロンプトで最大 8 回まで生成を自動的に再試行する。この検証と再試行のサイクルにより、学習段階に適した品質の高い中間プログラムの自動生成を実現している。

4. 評価実験

提案手法の有効性を検証するため、中間プログラムの生成能力について評価実験を行った。

4.1 実験方法

単純な制御構造から複雑なネスト構造までを含む 18 個の Python プログラムで構成された評価用データセット (sample_test/) を入力として使用した。このデータセットに対して本システムを実行し、ギャップが 2 以上と検出された箇所で中間プログラムが正しく生成されるかを確認した。具体的には、(1) 各制御構造要素に対する生成成功率、(2) 制約条件（必須・許可・禁止要素）を遵守したコードが生成されるまでの平均試行回数を計測した。実験には AI モデルとして、Ollama 上の Qwen3 32B Q4_K_M モデルを使用した。

4.2 実験結果と考察

実験の結果、if, else のような単一で基本的な制御構造を必須要素とする場合、ほぼ 100 この結果から、本手法は基本的な制御構造のギャップ補完には非常に有効である一方、複雑性が増すと AI が制約を完全に満たすコードを生成することが困難になる傾向が示唆された。これは、プロンプトの複雑化が LLM の性能限界に達するためと考えられる。

5. まとめ

本研究では、プログラミング学習における教材間の難易度ギャップを自動的に検出し、生成 AI により中間難易度のプログラムを生成するシステムを提案・実装した。

本システムは、教師が授業中に実施する演習問題の補完に活用することができる。例えば、if 文を学んだ直後の学習者には、elif を段階的に導入する中間プログラムを提示することで、スムーズな理解を促すことが可能になる。

今後の課題として、複雑なネスト構造における生成精度の向上、より多様なプログラミング言語への対応、実際の教育現場での有効性検証などが挙げられる。また、生成されたプログラムの教育的価値をより詳細に評価する指標の開発も重要な課題である。

本システムにより、教員の負担軽減と学習者個々のペースに応じた段階的なプログラミング学習環境の提供が期待される。

参考文献

[1] 長慎也, 保福やよい, 西田知博, 兼宗進: De-gapper - プログラミング初学者の段階的な理解を支援するツール. 情報処理学会論文誌, Vol.55, No.1, pp.1–12 (2014)

[2] 漆原宏丞, 岸本有生, 本多佑希, 兼宗 進: 抽象構文木を利用したサンプルプログラムの難易度判定手法. 情報処理学会全国大会, vol.85, No.4, pp.385-386 (2023)

[3] 前田悠翔, 田中英武, 井垣宏, 福安直樹: 多様な問題パターンおよび難易度を考慮した言語系生成 AI によるプログラミング演習問題自動生成手法の検討. 第 10 回実践的 IT 教育シンポジウム rePiT 論文集, Vol.10, pp.88-99 (2024)