プログラム部分点のための スライスを用いた類似度指標

宮前 和也[1,a)]　寺田 実[1,b)]

概要：プログラミング学習者が自習する際には部分点を用いた自動評価が望ましいと考える．しかし部分点評価を行う上で正解コードとの文字列比較や，AST の編集距離に基づく比較のみでは必ずしも妥当な点数が得られない場合がある．本研究ではコードから意味的な情報を抽出する方法のひとつであるプログラムスライスに着目する．学習者のコードからスライスを複数抽出してスライス群を作成し，正解のコードのスライス群との比較を行うことで類似度を求める．スライス群の比較により全体の違いだけではなく，コード中の意味のある一連のまとまりごとに部分点の付与が可能になると考える．スライス相互の比較にはスライスを構成する AST ノードの部分木どうしの編集距離を利用する．またスライス群の比較には二部グラフのマッチングを用いて最適な対応付けを行う．評価として，従来手法である AST の編集距離による部分点との比較，ユーザテストによる納得感の評価を行う計画である．

キーワード：プログラムスライス，部分点

1. はじめに

プログラミングを学習するうえで記述したコードを評価を受けることはとても大切である．コードを評価するためにテスト入力を用いた自動採点を利用する場合，正誤のみの結果では正解のコードとの近さがわからないため，学習者や教師がどの程度理解をしているかを知ることができないという欠点がある．

これを克服し，さらに学習のモチベーションを維持するために部分点評価を行う必要がある．しかし，部分点評価を行ううえで正解プログラムとの文字列比較や抽象構文木による比較のみでは必ずしも妥当な点数が得られない場合がある．

プログラムから意味的な情報を抽出する方法のひとつであるプログラムスライスはコードクローンの検出などにも利用されている [1]．

そこで，プログラムにより妥当な部分点を与える類似度指標としてプログラムスライスに着目する．

2. 目的

本研究は学習者がプログラムの正誤だけでなく，間違いの程度を理解することで学習やデバッグの支援になることを目標としている．

部分点をもとに学習やデバッグを行ってもらうためには部分点が学習者にとって妥当である必要がある．そこでプログラムスライスを利用した部分点付与の有効性を検証する．文字列や抽象構文木の比較による部分点より妥当な点数が出ることを期待する．

[1] 電気通信大学 大学院情報理工学研究科
[a)] m2231149@edu.cc.uec.ac.jp
[b)] terada.minoru@uec.ac.jp

2.1 部分点

2.1.1 部分点の意義

プログラムのコードに対して部分点を付ける理由は大きく分けて 2 つある．

- 学習者にとってコードが正解に近づきつつある指標になる．
- 教師などにとって採点をする際の点数の指標になる．

学習者側の利点

コードが正解に近づいていることがわかることで，自己効力感を得ることや問題に対するモチベーションの維持が可能となると考える．さらに，点数の変化により間接的に修正すべき箇所がわかるため学習者の試行錯誤のガイドにもなると思われる．

教師側の利点

コードに対する答えが「正しい」「正しくない」だけでは問題の学習者の理解度を測ることができない．そこで，プログラムコードを記述させる問題において，採点する基準の一つとして使える可能性がある．大量の問題に対するスクリーニングとして利用し，採点する手間の軽減ができる．

2.1.2 部分点の妥当性

部分点が妥当であるかどうかの判定を行うために，以下の 2 つを基準とする．これらは評価実験で検証を行う．

(1) 正解との近さを反映している

正解のコードに近づいていれば点数が上がっていくことが必要であるため，間違いの大きさや個数などで点数の調整ができている．

(2) 学習者が納得できる

多くの学習者が惜しいと思えるようなプログラムには，満点ではないが高得点を与えることで客観性を確保できている．

2.1.3 先行研究

初心者のプログラムを正誤と質で評価するシステム [2] では学習者が書いたコードに対して静的評価と動的評価を行うことにより部分点の付与を行っている．このうち静的評価では，予め用意してある正解のコードと学習者が書いたコードのそれぞれ抽象構文木を求め，zss アルゴリズム [3] による木の編集距離を使い比較を行っている．

zss アルゴリズムでは部分木の入れ替えを基本操作としていないため，意味に変化がない文の入れ替えによって編集距離が大きく変わってしまう場合がある．[2] ではその問題に対し，同レベルの文に対してソートを行っている．しかし，順序を変えてはいけない同レベルの文の場合，意味が変わってしまい誤って高得点を与えてしまう可能性がある．その点スライスを用いると変数ごとに関連する文を求められるため，スライスに含まれた文によって入れ替えが許されるかどうかの判断を行える．

本研究ではこの静的評価に新たな指標を使うことでより良い部分点を導き出す．

3. 準備

本研究で利用する技術について説明する．

3.1 プログラムスライス

プログラムスライスとは，ある地点のある変数（これをスライス基準と呼ぶ）に着目し，命令間の依存関係を利用して元のプログラムから当該の変数に影響を与えている文だけを抜き出す技術である [4]．プログラムから意味構造を抽出できるため，デバッグ，テスト，保守など広範囲で使われている．

スライスにはスタティックスライスとダイナミックスライスの 2 種類がある．スタティックスライスはある変数に関して，入力に関係なく同じ動作を行う部分プログラムであり，ダイナミックスライスは同じ入力であれば同じ動作を行う部分プログラムである．本研究でのプログラムスライスはスタティックスライスのことを指す．

図 1 はプログラムスライスを行った簡単な例である．このように必要な変数に関連する文のみを抽出することができ，不要な文を減らすことが可能である．

本研究でのスライスは [5] で述べられている手順により実装した．スライスを行うコードの各行に対し，定義されている変数名や参照されている変

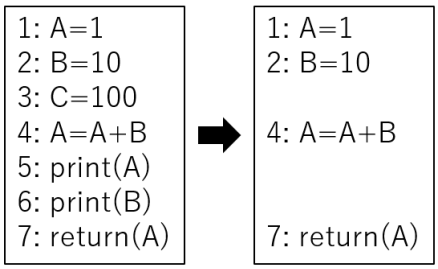

図 1 スライス例
7 行目の変数 A をスライス基準としている．3,6 行目は変数 A に無関係，5 行目は変数の参照のみであるため除外されている．

数名，次に実行する行，下位に文を持つ行などの情報を取得し，そのデータをもとに直接的関連している変数や間接的に関連している変数を求める．それらの変数の結果に変化が生まれなくなるまで繰り返し関連変数を求めることにより最終的に実行に影響のある行であるスライスを求めることができる．

3.2 抽象構文木

抽象構文木はプログラム中の意味に関係のある情報のみを取り出した木構造のことであり，本研究ではスライスの作成とスライスどうしを比較する際に利用する．

図 1 のスライス後のコードに対する抽象構文木は図 2 のようになる．根ノードはコード全体を指す Module ノードであり，一つ深い層のノードは同じレベルの文の種類名（Assign 文や Return 文など），さらに深い層は各文に対する処理内容が含まれている．

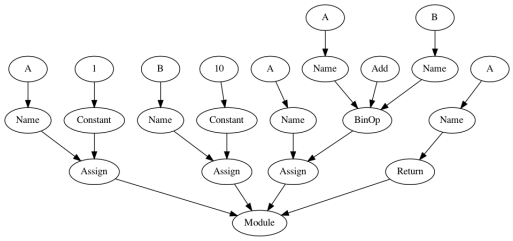

図 2 図 1 のスライス後のコードに対する抽象構文木

3.3 木の編集距離

本研究では，スライスを比較するために木の編集距離を求めることができる zss アルゴリズムを利用している．木の編集距離の求め方は片方の木に対し以下の 3 つの操作を繰り返し，もう一方の木と等しくなる最小の回数としている．3 つの操作はそれぞれに対しコストを自由に決めることができるため特定の処理のみコストを重くすることも可能である．

- 1 つのノードの挿入
- 1 つのノードの削除
- 1 つのノードのタグの変更

図 3 のような木の比較例の場合は全ての操作のコストが 1 の場合，木の編集距離は 2 となる．

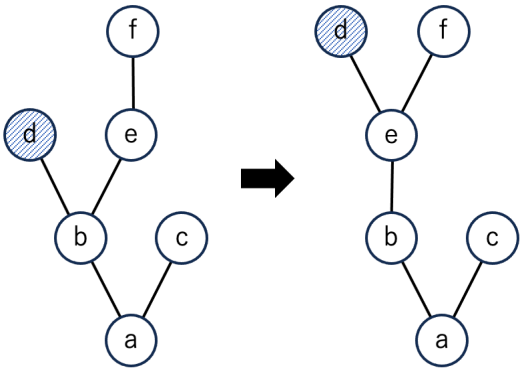

図 3 木の編集距離例
ノード d の削除とノード e の子への d の追加によって同じ木になる．

4. 関連研究

4.1 プログラムの正しさの評価

プログラムを評価するためには，モデル検査，定理証明，ソフトウェアテストなど様々な手法が存在する．

モデル検査
システムをアルゴリズム的に検証する方法であり，取り得る状態や遷移などを網羅的に検査することで不具合を検出する．

定理証明
プログラムに不具合がないことを数学的に証

明するものであり複雑なシステムであっても検証が可能である．

ソフトウェアテスト

テストセットの実行によって，プログラムが意図していない動作をしている場合を見つけることが可能である．

本研究ではプログラムが「正しい」「正しくない」を出すだけでなくどちらに偏っているかを導き出す必要がある．そこで解答とのコードレベルでの比較を行うことによって部分点の付与を行う．

4.2 プログラムスライスの利用

プログラムスライスを用いたデバッグの支援を行えるツール [6] では，デバッグにかかる時間を短縮させるためにスライシングを用いている．スライシングによって，プログラムの命令間の依存関係を明らかにし，作業者が読み取るコード量を少なくすることでデバッグする際の手間を減らすことができ，プログラム支援が行えたとしている．

コードクローンの脆弱性をプログラムスライスを基準として求めるシステム [1] では，検査対象から抽出したスライスが脆弱性を持つコードを含んでいるかにより脆弱性判定を行っている．スライス単位で解析を行うことによる脆弱性判定に無関係な情報を除外できることをメリットとしている．

5. スライス群の比較による部分点

5.1 スライス群を使う意義

スライスは，あらかじめ決めておいたスライス基準に基づいて抽出される．スライス基準に関しては，コード中の return 文などの最終的に答えが出力される変数を対象としている．これにより，関係のない変数や文（たとえばデバッグ用の出力文など）を削除することでコード評価におけるノイズを減らす事が可能になる．

しかし，コードからスライス基準をもとにスライスを 1 つ抽出しただけでは基本的に元のコードとの違いがほとんどなく，コード全体で抽象構文木を比較することとあまり変わりがない．そこで，スライスを 1 つだけでなく複数取得することでコード内で細かく比較を行えると考えた．

スライスを複数取得する際にどのような基準でスライスを取るか決める必要があるため，本研究ではプログラム内に存在する全ての変数の「そのプログラム内で最後に参照または定義された場所」をスライス基準とした．プログラム内の変数それぞれについて，その最後の出現をスライス基準としてスライスを取ることによって，変数ごとの挙動をまとめることができ，全体の比較ではできなかった同レベルのコードの入れ替えに対応することができる．

5.2 スライス群の抽出

前節で述べた方法により変数それぞれについてスライスを抽出しスライス群を作る．この処理は解答コード，正解コードそれぞれに行う．ここで言う解答コードは学習者が記述するコードであり，正解コードは事前に設定しておいたコードである．

ソースコード 1 は `n` の `m` 乗を求める関数 `power` であり，7 行目の変数 `ans` をスライス基準としたコードでもある．ソースコード 2 は 6 行目の変数 `i` のスライスであり，`i` の定義や値の更新の処理が含まれている．ソースコード 3，4 も同様に定義とスライス基準がスライスに含まれている．

ソースコード 1 の 5 行目と 6 行目の文には依存関係がなく交換可能である．そのことはソースコード 2，3 のスライスに片方ずつ出現していることから確認できる．しかし，`ans` のスライス（ソースコード 1）では双方を含むため，交換可能かどうかはわからない．スライス中での文の順序は必ずしも依存関係を表さないということである．

このようにそれぞれの変数に対してスライスを取ることにより，プログラム内の変数の依存関係や挙動をまとまりごとに分けることができる．

5.3 スライス群の比較

スライス群の比較を行うために以下の 3 つの階層ごとの比較を行う必要がある．

- 文どうしの比較
- スライスどうしの比較
- スライス群どうしの比較

ソースコード 1　対象コード
（変数 ans のスライスとも一致）

```
1 def power(n, m):
2     ans = 1
3     i = 0
4     while m > i:
5         ans = ans * n
6         i = i + 1
7     return ans
```

ソースコード 2　変数 i のスライス

```
1 def power(n, m):
2
3     i = 0
4     while m > i:
5
6         i = i + 1
7
```

ソースコード 3　変数 n のスライス

```
1 def power(n, m):
2
3
4
5         ans = ans * n
6
7
```

ソースコード 4　変数 m のスライス

```
1 def power(n, m):
2
3
4     while m > i:
5
6
7
```

文どうしを比較した結果をスライスどうしの比較に用い，その結果をさらにスライス群の比較に利用することでスライス群の違いを求めることができる．

抽出したスライス群の比較手順を図 4 に示す．コードの抽象構文木を比較した際の編集距離の総和がコードどうしの類似度指標となる．

5.3.1　文どうしの距離

コードを文単位に分け，その部分木どうしの編集距離を利用し比較を行う．基本的には文ごとに

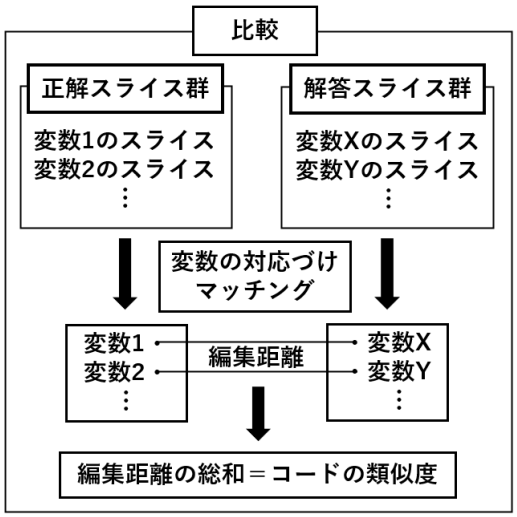

図 4　スライス群の比較手順

部分木が取得できるが if 文や while 文など下位に文を持つ文に関しては，図 5 の左部のように下位の文も部分木に含まれてしまう．そこで下位に文を持つ文に関しては図のように条件部のみの部分木を取るようにした．図 5 の右部は「if A: B=10」の if 文の部分木のみを取った結果である．

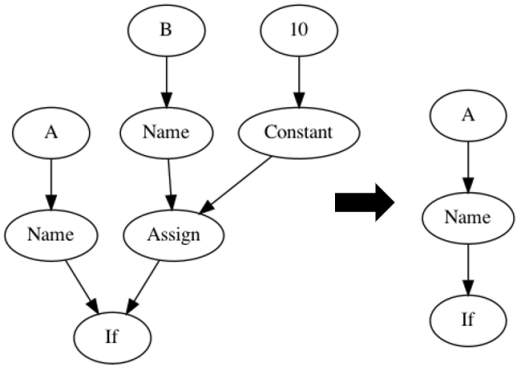

図 5　下位の文を持つ場合の部分木

このようにして求めた部分木を比較するために木の編集距離を利用する．木の編集距離のコストに関しては以下のようにする．

- 比較している部分木の文の種類を表す根ノードが同じ場合，ノードの「挿入」「削除」「タグの変更」のコストは基本的には 1 とする．
- 部分木の根ノードが異なる場合，文の種類を

変更するペナルティとしてそれぞれのコストを大きく増加させる．

図 6 は 2 つの文「A=1」「A=N+1」の部分木の編集距離を求める例である．

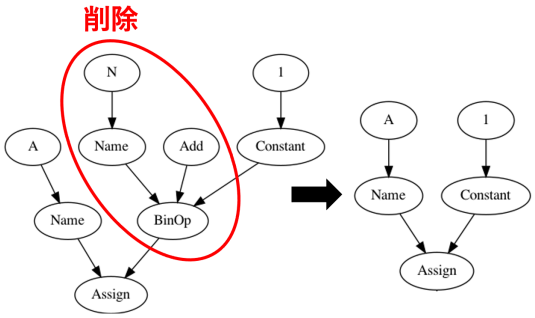

図 6 「A=1」「A=N+1」の実際の編集距離
囲まれた部分の削除を 4 回行うため編集距離は 4 となる．

構文木の比較の際に変数名を統一することで変数名の違いによる減点を防ぐ必要がある．統一の方法に関しては 5.3.3 で説明を行う．

5.3.2 スライスどうしの距離

スライスどうしの距離を求めるために，スライス内の文どうしの対応付けを行う必要がある．

文をノード，前節で求めた文どうしの距離を重みとした二部グラフを作成し，重みが最小になるようにマッチングを行う．文どうしのペアを作るため，片方のスライスにはない文を余分な文として認識でき，その文の抽象構文木の大きさによって余計なコストが出ないようにする事ができる．コスト最小マッチングによりペアになった文の編集距離を全て足したものをスライスどうしの距離とする．

図 7 は片方に余分な文があった場合の例であり，太字が編集距離を示している．余計な文とみなされた場合は固定値が編集距離として加算される．(図の場合は 5 としている)

5.3.3 スライス群どうしの距離

前節でスライスどうしの比較を行う事ができるようになった．しかし，スライス群を比較するには変数に対するスライスの対応関係も決める必要がある．各プログラムで使われた変数を正しく対応付けすることによって機能が似ている部分を比較できるようになる．

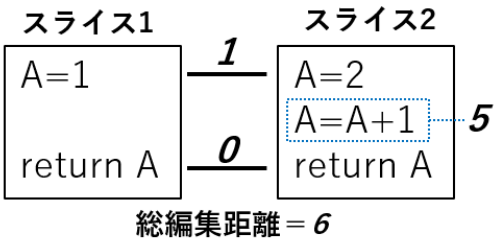

図 7 スライス比較による編集距離

比較方法は図 8 のように正解コードと解答コードに使われている変数名をそれぞれノードとし，スライスどうしの編集距離を重みとした二部グラフを用い，重みの総量が最小になるようにマッチングを行った．このスライスのマッチングが結果として変数名のマッチングになる．

変数名が同じものを対応させる方法もあったが，学習者の名前の付け方が違う可能性があるため行わなかった．そのことから構造上似ているという指標である編集距離を比較し，構文木の変数名を統一することで結果に影響を及ぼさないようにした．

5.3.1 にて触れた変数名の統一の方法について述べる．変数名を統一する際はスライス基準となっている変数を `var1`，それ以外の変数名を `var2` と改称した．スライス基準の変数とそれ以外の変数で名前を変えることによって，全ての変数名を統一する場合より少し評価を厳しくした．

図内の実線は最小マッチングによって行われた対応付けの例である．

5.3.4 正解と異なるコード例と編集距離

ソースコード 1 と以下のソースコード群を比較する．ソースコード 5 は変数名を別のものと変更し，入れ替えてもいい部分（2 と 3 行目，5 と 6 行目）を入れ替えたコードであり，ソースコード 6 は変数 `i` の初期化を記述していなかった場合である．ソースコード 7 は考え方が少し違う同じ機能のコードである．コード内に文が足りなかったり，多かったりする場合の編集距離の固定値は 5 としている．

比較した結果は表 1 に示す．解答 1 は変数名の

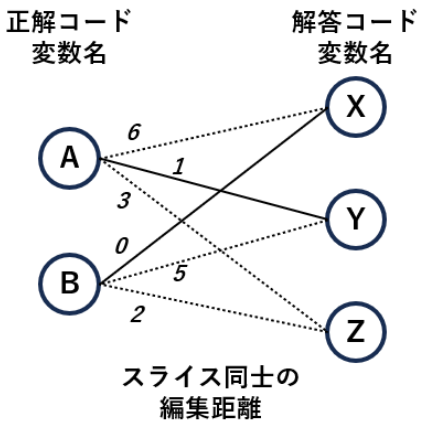

図 8　変数名のマッチング

ソースコード 5　解答コード 1
（変数名の変更と文の入れ替え）

```
1 def power(x, y):
2     i = 0
3     ans = 1
4     while y > i:
5         i = i + 1
6         ans = ans * x
7     return ans
```

ソースコード 6　解答コード 2
（i の定義忘れ）

```
1 def power(n, m):
2     ans = 1
3     while m > i:
4         ans = ans * n
5         i = i + 1
6     return ans
```

ソースコード 7　解答コード 3
（別解のコード）

```
1 def power(n, m):
2     ans = 1
3     while m > 0:
4         ans = ans * n
5         m = m - 1
6     return ans
```

違いによる差はなく入れ替えも問題なく機能している．解答 2 は i の挙動のみが間違っているため i とそれを含む ans の編集距離が増えている．解答 3 は i がなく m を利用しているため i の対応変数が無くその分のペナルティが発生している．

表 1　編集距離の結果

正解コード	解答 1	解答 2	解答 3
変数名	対応変数：距離		
i	i:0	i:5	無し:20
m	y:0	m:0	m:7
n	x:0	n:0	n:0
ans	ans:0	ans:5	ans:8
総編集距離	0	10	35

5.4　スライス群の距離からの部分点算出

正解コードと解答コードの編集距離を用いた，現状考えている部分点付与の方法について以下に示す．

編集距離が計算されるのは文単位での構文木比較の際であるため，文を書き換えるための労力とみなすことができる．そこで，文の比較で求められた編集距離を正しい文のノード数で割ることによって文単位での間違っている割合を求める．あっている割合を文単位での部分点とし，それらの平均を取ることによって総合的な点数を求めることができると考えている．しかし，この部分点についても多くのプログラムで試す必要があるため，実際にプログラムに対し部分点を付与し評価しつつ妥当な部分点付与の方法を決めていく．

5.5　部分点算出の流れ

以下の手順で解答コードと正解コードから必要な情報を抽出，比較を行い部分点を付与する．

(1) 解答，正解コードからスライス群の抽出
(2) 解答，正解コードのスライス群の比較
(3) 部分点の付与

システムの流れを図 9 に示す．事前に用意していた問題に対する解答を学習者に記述してもらい，そのプログラムコードと正解のコードからスライス群を抽出し，それらを元に部分点を与える．その部分点をもとに学習者はプログラムコードの修正を行い，再度点数評価を確認する．

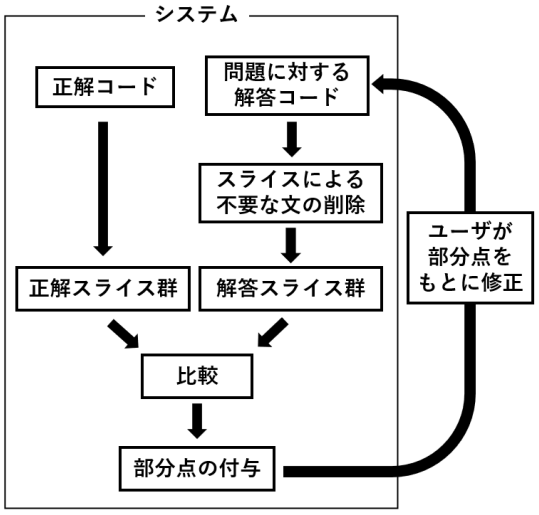

図 9 システムの流れ

5.6 実装

評価対象とする言語は Python3 とし，プログラムスライスや部分点の処理も Python3 を利用した．以下に使用したモジュールとその用途を示す．

- ast: 抽象構文木の作成
- zss: 木の編集距離
- networkx: 二部グラフのマッチング

6. 評価方法

部分点の妥当性について述べた二つの観点のそれぞれについて検証し評価する．

正解との近さを反映できているか

複数の解答パターン（スペルミス，解答が異なるなど）で，それぞれ部分点を付与しあらかじめ設定していた点数に近い値が出るかどうかを検証し，比較する．

学習者が点数を納得するか

複数の例題に対して，学習者にコードを書いてもらい部分点を付与する．その後，自身が書いたコードの部分点が妥当であるかどうかの評価を行ってもらう．その上で抽象構文木のみを使った部分点を付与するシステムとの比較を行ってもらいどちらの部分点が妥当であったかについても回答してもらう．

7. まとめと課題

プログラムに部分点を付与するために，正解のコードと解答のコードの比較を行なった．比較の指標としてコード内で使われている全ての変数を基準としスライスを取得した．スライスをもとに文単位での抽象構文木を取得し比較，木の編集距離を利用し類似度を求めることができた．

課題として，解答コードと正解コードが異なるアルゴリズムだった場合は当然スライスも全く異なるため，妥当な部分点を与えられない可能性がある．本研究ではそのような状態を考慮していない．事前に解法ごとに別の正解を用意することや，問題自体に解き方の制限を与えることによって対策を行う．

参考文献

[1] Song Xiaonan et al., “Program Slice based Vulnerable Code Clone Detection”, TrustCom, 2020.

[2] 鄭 佳健, 寺田 実, “初心者のプログラムを正誤と質で評価するシステム”, 情報処理学会 第 62 回プログラミング・シンポジウム報告集, 75-82, 2021.

[3] D. E. Shasha, Kaizhong Zhang, “Simple fast algorithms for the editing distance between trees and related problems”, Society for Industrial and Applied Mathematics, 10, 707-710, 1989.

[4] M. Weiser, “Program Slicing”, IEEE Transactions on Software Engineering, 10, 4, 352-357, 1984.

[5] Frank Tip, “A Survey of Program Slicing Techniques”, Centre for Mathematics and Computer Science, 1994.

[6] 工藤 英男, 中井 伸郎, 内田 眞司,“スライシングを用いたデバッグ支援ツールに関する一考察”, 奈良工業高等専門学校 研究紀要, 第 37 号, 2001.