変数の値の変化を利用したソースコード分析手法の提案

鈴木 凌斗 *1　　矢谷 浩司 *1

A Code Analysis Method Utilizing Variable Behavior

Ryoto Suzuki*1, Koji Yatani*1

Abstract – **Code similarity analysis enables important software engineering applications including automated code clustering and plagiarism detection. Existing methods are based on textual similarity or abstract syntax trees, but such approaches are often vulnerable to simple textual modifications. We present a code similarity analysis approach utilizing the information of how variables in each code change. By looking at how similarly or differently variables behave, our approach can identify code that demonstrates the same data processing procedure despite their textual differences. This paper discusses our current implementation and preliminary results of our method.**

Keywords : **Code similarity analysis, programming, code clustering**

1.　はじめに

ソースコードの類似性分析は，コードクローン・盗用の検出やソースコードの分類といった応用のために重要である．コードクローンは，主にソースコードの一部をコピーして流用することによって発生し，ソフトウェアのメンテナンスコストを高める原因となる [1] ため，クローンを検出することによりソフトウェア開発を支援することができる．ソースコードの盗用はプログラミングの授業において大きな問題となっていて，学生に公平な評価を与えるためにソースコードの盗用を自動的に検出する方法が研究されている [2]．近年オープンソースでのプログラミング環境が普及していることにより，ユーザは既存のソースコードアーカイブにアクセスすることが容易になり，恩恵を受けている．このようなアーカイブ内のソースコードを手作業で分類するのは手間がかかるため，自動的に分類を行う技術が研究されている [3]．

既存研究では，ソースコードを木構造で表現したものの類似性や文字列としての類似性を評価するなどプログラムの構造に注目した分析が多く，ソースコードを 1 行変更しただけでプログラムの動作が大きく変わる場合であっても構造的な変化は小さいため類似度を過大評価してしまうといったように，処理の類似性を検出することが難しい．本研究では，変数の値の変化を利用した類似度を定義し，その類似度に基づいてソースコードをクラスタリングするプログラムを実装した．さらに予備的な実験としてオンラインジャッジシステム (プログラム問題を自動採点するシステム) に提出されたソースコードに対してクラスタリングを行い，処理の類似性を検出できる可能性がわかった．

*1: 東京大学 IIS Lab
*1: Interactive Intelligent Systems Lab., The University of Tokyo

2.　関連研究

本章では，まずソースコード間の類似度を計算する手法についての先行研究を紹介し，続いてソースコードの分類に関する既存の技術に触れる．

2.1　ソースコードの類似度に関する研究

Ira らは，抽象構文木 (Abstract syntax tree) を用いて，ソースコードのクローンを検出する方法について研究を行った [4]．抽象構文木とは，プログラムの構造を木構造で表し，そこから空行やコメントといったプログラムの意味に関係ない情報を取り除いたものである．この研究では，抽象構文木の部分木のペアについて，部分木 1 と部分木 2 の類似度を次のように定義する方法を紹介している．

$$\frac{2S}{2S+L+R} \tag{1}$$

ここで，S は 2 つの部分木に共通するノードの数，L は部分木 1 のみに含まれるノードの数,R は部分木 2 のみに含まれるノードの数を表す．

Chaiyong らは，ソースコードの類似度を計算する手法を比較する研究を行った [5]．そこで取り上げられた手法は，上で述べたような抽象構文木を用いた手法に加えて，テキストベース，トークンベース，グラフベースの方法であり，これらを比較した．以下にそれぞれの手法について説明する．まず，テキストベースの手法では，ソースコード同士を文字列として比較し，類似度を計算する．特に良く用いられている方法は，2 つのソースコードの最長共通部分列を計算する方法であり，最長共通部分列の長さが大きいほどよく

類似しているとみなす．トークンベースの手法では，ソースコードをトークンの集合に変換することによってソースコードのテキストを抽象化している．例えば，ソースコード中のすべての単語を W というトークンに変換した場合，“int x = 0;” という文と “String s = ”Hi”;” という文は共に変換によって，“W W = W;” と表されるため同一視されることになる．このようにして生成されたトークンの集合に対して，接尾辞木や Jaccard 係数などといった様々な手法が適用され，類似度が計算される．グラフベースの手法では，ソースコードをグラフで表現し，グラフにおける類似度計算の手法を適用する．一般的にグラフの類似度計算は計算コストが大きく，グラフ同士を比較するほとんどのアルゴリズムは NP 完全である [5]．

2.2 ソースコードの分類に関する研究

Nguyen らは，抽象構文木の編集距離に基づいてソースコードを分類した [6]. この手法では，ソースコードを木構造で表現したときの構造的な類似性はとらえられるが，変数の値の変化は無視しているため，意味的な類似性はとらえられない．

Glassman らは，MOOCs (Massive Open Online Courses) というオンラインコースにおいて，学生が提出した大量のソースコードを指導者が読むのを補助するために，類似したソースコードを分類して可視化するシステムを作成した [7]．ここでは次のような手順で分類が行われている．

1. コメントや空行を削除してインデントやスペースを統一する．
2. 同じテストケースに対して各プログラムを一度だけ実行する．
3. 各行の実行時における変数の名前と値，関数からの戻り値が記録される．この記録をプログラムトレースと呼ぶ．プログラムトレース内の各変数について，変数の値の変化の列を抽出する．
4. すべてのプログラムのトレースを分析し，値の変化の列が同じであるような変数の組を特定する．2 つ以上のプログラムトレースで同一の値の変化の列を持つ変数を common variable と呼ぶ．
5. common variable の変数名を，最も多く出現した名前に統一する．
6. 手順 1 から 5 まででフォーマットや変数名の違いが吸収されたソースコードのペアについて，行の順番を任意に入れ替えたときにすべての文が一致するようなソースコードを同じクラスタに割り振る．

この方法では，同じクラスタに分類されるための条件が厳しいため，単一の簡単な関数を実装するようなプログラムに対しては有効に機能するものの，そうでない複雑なプログラムに適用することができない．

本研究では，変数の値の変化の列に対して類似度を定義することでソースコード間の類似度を計算する手法を提案する．この手法では，抽象構文木やテキストベースのアプローチといったプログラムの構造に基づいて分類する手法とは異なり，変数の値の変化に注目することでプログラムの意味的な類似性をとらえることが期待できる．

3. 提案手法

本研究では，プログラムを実行した時の変数の値の変化を記録し，その値の変化同士の類似度を定義することでソースコードの間の類似度を計算する手法を提案する．さらに，ここで計算した類似度に基づいてソースコードのクラスタリングを行う．具体的な手法は次の手順からなる．

1. プログラムを実行し，変数の値の変化を記録する．
2. 変数の値の変化同士の類似度を計算する．
3. 2 で計算した類似度に基づいて 2 つのソースコード間の変数をマッチングする．
4. マッチングされた変数間の類似度の平均値としてソースコード間の類似度を定める．
5. 4 で計算した類似度に基づいて階層型クラスタリングを行う．

3.1 変数の値の変化の記録

変数の値の変化を記録するために，Guo によって開発された pg_logger [8] を用いた．これは Python プログラムの実行ログを記録するためのソフトウェアで，プログラムが 1 行実行されるごとにその時点での各変数の値を記録する．変数ごとに個別の配列を用意し，pg_logger によって記録された値を末尾に追加していく．したがって，最終的に変数の値の変化は各時点における値の列として記録される．

3.2 変数間の類似度計算

2 つの変数 x, y の間の類似度を次のように定める．

$$VarSim(x, y) = \frac{1 - d}{1 + d} \tag{2}$$

ここで d は，x, y の値の変化の間の編集距離 (一方の列をもう一方の列に一致させるために必要な編集（挿入・削除・置換）の最小回数) を長い方の列の長さで割り，正規化したものであり，$[0, 1]$ の値を取る．このため，$VarSim(x, y)$ も $[0, 1]$ の値を取る．

例として，整数型変数 x, y の値の変化がそれぞれ $x_t = \{1, 2, 3, 4, 5\}, y_t = \{1, 2, 4, 6\}$ となっているときの $VarSim(x, y)$ の計算手順を説明する．y_t の 6 を 5 に置換し，3 を挿入することにより，2 回の編集で y_t を x_t に一致させることができる (これより少ない回数で一致させることはできない．)．したがって，x_t, y_t の

編集距離は 2 であり，これを x_t, y_t のうちより長い方の長さである 5 で割って $d = 2/5$ となる．したがって

$$VarSim(x, y) = \frac{1 - \frac{2}{5}}{1 + \frac{2}{5}} = 0.428... \quad (3)$$

と求められる．

本提案手法では全ての変数の組み合わせに対して比較を行う．違う型の変数同士の類似度は，基本的には 0 となるが，現在の実装では Python の暗黙的な型変換を通した比較に依存するため，型が違う変数でも類似度が 0 にならないことがあり得る（例えば，0 と False）．一般的には，型の違う変数間の類似度は非常に小さくなり，これらが以下に説明するマッチングではマッチする可能性は低いと期待できる．

3.3 変数のマッチング

2 つのソースコード中に出現する変数同士のマッチングを行う．このとき，マッチングされた変数間の類似度の総和が最大になるようにする．これは二部グラフの最大マッチング問題であるので，最小費用流問題に帰着して厳密解を求めることができる．この処理により 2 つのソースコード C_1, C_2 に含まれる変数の総数を N_1, N_2 としたとき，$min(N_1, N_2)$ 個のペアが生成される．

3.4 ソースコード間の距離

ソースコード間の距離は，マッチングされた変数間の類似度の平均値を正規化したものを用いて定める．2 つのソースコード C_1, C_2 における，マッチングされた変数の集合を S として，次式で表される値である．

$$CodeDis(C_1, C_2) = 1 - \frac{\sum_{\{x,y\} \in S} VarSim(x, y)}{max(N_1, N_2)} \quad (4)$$

これにより，2 つのソースコード間の距離は，$[0, 1]$ の範囲で決定される．分母に $max(N_1, N_2)$ を用いているのは，マッチングされずに余った変数を考慮するためである（マッチングされなかった変数が多い場合，$CodeDis(C_1, C_2)$ は大きくなる）．また，これはマッチングされなかった変数の $VarSim$ が 0 となっている，ともみなせる．

3.5 クラスタリング

完全連結法 (Complete linkage method) により階層型クラスタリングを行う．まず，2 つのクラスタ D_1, D_2 間の距離を，それぞれのクラスタに含まれるソースコードのすべてのペアにおける距離の最大値として定める．形式的には次式で表される値である．

$$ClusterDis(D_1, D_2) = \max_{C_1 \in D_1, C_2 \in D_2} CodeDis(C_1, C_2) \quad (5)$$

そして，次の手順でクラスタリングを行う．

1. 初期状態では各ソースコードに 1 つのクラスタを割り当てる．
2. すべてのクラスタのペアについて，最も $ClusterDis$ が小さいペアを結合する．
3. すべてのクラスタが結合されるまで 2 を繰り返す．

4. ケーススタディ

提案手法との比較のために抽象構文木を利用した類似度判定手法を比較対象として用い，クラスタリングを行うことによって手法の性能を比較することとした．このケーススタディとして，オンラインジャッジシステムに提出されたソースコードを収集し，それぞれの手法でソースコード間の距離を計算した後，クラスタリングを行った．なお，我々の手法において距離の計算を行う際には，プログラムに与える入力データとして一様乱数により生成したものを用いた．

ソースコードのクラスタリングは，完全連結法を用いた階層型クラスタリングによって行った．我々の手法においてはソースコード間の距離は $CodeDis(C_1, C_2)$ を用いた．一方，抽象構文木を利用した手法では，まず，ソースコードを抽象構文木に変換するとともに．コメントやドキュメンテーションといったプログラムの動作に無関係な行は削除した．続いて．変数名や関数名の違いを吸収するために，変数名や関数名を番号を振ったものに置き換えた後，抽象構文木をソースコードに再変換し，再変換後のソースコード間の編集距離に基づいてクラスタリングを行った．コード・抽象構文木間の変換には，pyastsim [1] を用いた．

4.1 バブルソートの実装方法による分類

Aizu Online Judge [2] より，バブルソートを行い，ソートの結果とスワップの回数を答える問題に対する Python の提出を 20 件集め，クラスタリングを行った．収集したソースコードにおいて，バブルソートの実装方法は大きく以下の 2 通りに分かれた．

- while ループの内部で for ループを回す方法．次にその擬似コードを示す．ここで A は配列，N は要素数とする．

```
1: procedure BUBBLE-SORT(A, N)
2:     flag = True
3:     while flag == True do
4:         flag = False
5:         for j = N − 1 ... 1 do
6:             if A[j] < A[j − 1] then
7:                 swap(A[j], A[j − 1])
8:                 flag = True
```

1: https://github.com/jncraton/pyastsim
2: http://judge.u-aizu.ac.jp/onlinejudge/

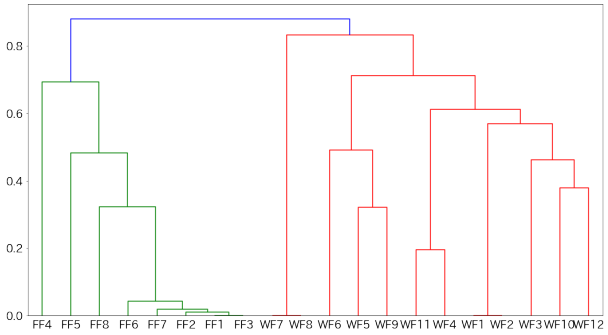

図 1: 提案手法による，バブルソートのソースコードの分類結果．横軸はソースコードの番号，縦軸はクラスタ間の距離を表す．WF と FF の分類に成功している．

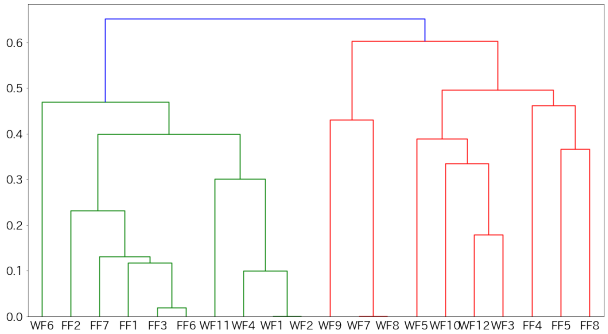

図 2: 抽象構文木を用いた手法による，バブルソートのソースコードの分類結果．WF と FF の分類が明確でない．

- for ループを 2 重に回す方法．次にその擬似コードを示す．ここで A は配列，N は要素数とする．

```
1: procedure BUBBLE-SORT(A, N)
2:     for i = 0 ... N − 1 do
3:         for j = N − 1 ... i + 1 do
4:             if A[j] < A[j − 1] then
5:                 swap(A[j], A[j − 1])
```

これら 20 件のソースコードを，12 件の while ループと for ループを組み合わせるもの（WF1〜12）と，8 件の 2 重 for ループのもの（FF1〜8）に，著者らの手によって分類した．

次に，提案手法によりクラスタリングを行った結果の樹形図を図 1に示す．{FF1, FF2,..., FF8} と {WF1, WF2,..., WF12} の 2 つのクラスタに分かれており，while ループと for ループを組み合わせるものと 2 重 for ループのものを分類できている．

続いて，抽象構文木ベースの方法でクラスタリングを行った結果の樹形図を図 2に示す．{FF1, FF2, FF3, FF6, FF7, WF1, WF2, WF4, WF6, WF11} と {FF4, FF5, FF8, WF3, WF5, WF7, WF8, WF9, WF10, WF12} の 2 つのクラスタに分かれていて，while ループと for ループを組み合わせるものと 2 重 for ループのものを分類できていない結果となった．

4.2 深さ優先探索・幅優先探索・素集合データ構造の分類

AtCoder [3] より，無向グラフの連結成分の個数を数える問題に対する Python の提出を 30 件集め，提案手法と抽象構文木ベースの手法でそれぞれクラスタリングを行った．この問題に対しては，主に次の 3 通りのアプローチが存在する．

- 深さ優先探索 (Depth-first search) による方法．
- 幅優先探索 (Breadth-first search) による方法．
- 素集合データ構造 (Disjoint-set data structure) を用いる方法．

これら 30 件のソースコードを，4 件の深さ優先探索のもの (DFS1〜4)，2 件の幅優先探索のもの (BFS1, BFS2)，24 件の素集合データ構造のもの (Dis1〜24) に，著者らの手によって分類した．

提案手法によりクラスタリングを行った結果の樹形図を図 3に示す．{DFS1, DFS2, DFS3, DFS4, BFS1, BFS2} とそれ以外の 2 つのクラスタに分かれている．つまり，深さ優先探索と幅優先探索の区別は明確にはできていないが，素集合データ構造を用いたアプローチかそうでないかを判別することに成功している．

続いて，抽象構文木ベースの方法でクラスタリングを行った結果の樹形図を図 4に示す．{BFS1, BFS2, DFS1, DFS2, DFS3, DFS4, Dis1, Dis2, Dis6, Dis10, Dis19, Dis20, Dis21, Dis22, Dis23} と {Dis3, Dis4, Dis5, Dis7, Dis8, Dis9, Dis11, Dis12, Dis13, Dis14, Dis15, Dis16, Dis17, Dis18, Dis24} の 2 つのクラスタに分かれている．1 つ目のクラスタに深さ優先探索・幅優先探索・素集合データ構造による 3 種類のソースコードがすべて含まれていることから，どのアプローチ同士も区別できていないことが分かる．

5. 考察

ケーススタディで扱った 2 つの例では，既存手法である抽象構文木ベースの手法と比べ，提案手法の方が処理の類似性をより正確に検出することができた．しかし，4.2節では幅優先探索と深さ優先探索を区別することができなかった．以下に考えられる原因と解決策を述べる．

5.1 変数の型の違いによる影響

4.2節において，LIFO (Last-In First-Out) のデータ構造を実現するために list 型の変数を用いるソース

[3] https://atcoder.jp/

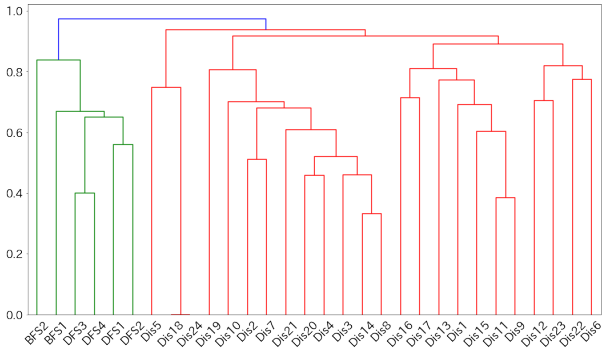

図 3: 提案手法による，無向グラフの連結成分の個数を数えるソースコードの分類結果．DFS と BFS が区別できていないものの，Dis との分類に成功している．

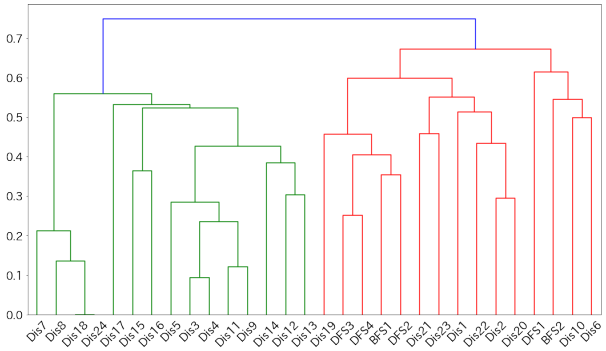

図 4: 抽象構文木を用いた手法による，無向グラフの連結成分の個数を数えるソースコードの分類結果．3 種類のソースコードが混在し，明確なクラスタが生成されていない．

コードと deque 型の変数を用いるソースコードが見られた．3.2 節で述べたように，これら 2 種類の変数は類似の処理を行うにもかかわらず VarSim が 0 と計算され，その結果，同じ深さ優先探索のソースコードであってもソースコード間距離が大きく評価される．このように，類似した処理を行うために異なる型の変数が使用されている場合に VarSim を小さく評価してしまうことが，類似性の検出に失敗する原因になっていると考えられる．この問題は，異なる型の変数の間に比較関数を用意することで解決できる．例えば tuple 型変数と list 型変数を比較する場合には，1 要素ずつ比較してすべて一致するかどうかを比較結果の真偽値とすればよい．

5.2 複数の要素を持つ変数同士の比較方法

$VarSim(x, y)$ は，変数の値の変化の列に対して編集距離を計算することで類似度を評価するが，編集距離の性質上，2 つの列の間に等しい要素が多いほど必要な編集回数が小さくなりやすい．しかし，list 型や tuple 型のように変数が複数の要素を持つ場合はそのうちのすべての要素が一致しなければ等しいと判定されないため，必要な編集回数が大きくなりやすい．例えば，list 型変数 x, y の値の変化がそれぞれ $x_t = \{\{1\}, \{1,2\}, \{1,2,3\}, \{1,2,3,4\}\}, y_t = \{\{\}, \{2\}, \{2,3\}, \{2,3,4\}\}$ となっているとき，x と y の違いは初期状態のみで，その後の処理は 2, 3, 4 を追加するという点で一致しているにもかかわらず，必要な編集回数は x_t, y_t の列の長さと同じ 4 となる．このように，複数の要素を持つ変数同士の比較では，処理が類似していても類似度を小さく評価してしまう場合がある．そのため，4.2節において，同じ深さ優先探索であっても頂点に訪問する順序が 1 つ異なるだけで変数間の類似度を小さく評価してしまい，深さ優先探索と幅優先探索を区別することが難しくなっているという可能性が考えられる．変数 x, y が複数の要素を持つ場合，単純な編集距離から類似度を計算するのではなく，一致する要素の個数を考慮することでこのような問題を解決できると考えられる．

6. 結論と今後の課題

変数の値の変化を利用してソースコード間の類似度を評価する手法を提案し，ソースコードのクラスタリングを行うケーススタディを通して既存の手法と比較した結果，プログラムの処理の類似性を既存手法よりも正確に検出できる可能性があることが分かった．ケーススタディで扱った問題は 1 つのアルゴリズムで解くことができる比較的簡単なものであったので，今後は，上記で挙げた問題を解決するとともに，複数のアルゴリズムを組み合わせる複雑なプログラムに対する提案手法の適用可能性を明らかにする予定である．

謝辞

本研究にご協力いただいた研究室のメンバーに感謝いたします．

参考文献

[1] Ira D Baxter, Andrew Yahin, Leonardo Moura, Marcelo Sant'Anna, and Lorraine Bier. Clone detection using abstract syntax trees. In *Proceedings. International Conference on Software Maintenance (Cat. No. 98CB36272)*, pp. 368–377. IEEE, 1998.

[2] O. Karnalim. Detecting source code plagiarism on introductory programming course assignments using a bytecode approach. In *2016 International Conference on Information Communication Technology and Systems (ICTS)*, pp. 63–68, 2016.

[3] Elife Ozturk Kiyak, Ayse Betul Cengiz, Kokten Ulas Birant, and Derya Birant. Comparison of image-based and text-based source code classification using deep learning. *SN Computer Science*, Vol. 1, No. 5, pp. 1–13, 2020.

[4] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier. Clone detection using abstract syntax trees. In *Proceedings. International Conference on Software Maintenance (Cat. No. 98CB36272)*, pp. 368–377, 1998.

[5] Chaiyong Ragkhitwetsagul, Jens Krinke, and David Clark. A comparison of code similarity analysers. *Empirical Softw. Engg.*, Vol. 23, No. 4, p. 2464–2519, August 2018.

[6] Andy Nguyen, Christopher Piech, Jonathan Huang, and Leonidas Guibas. Codewebs: Scalable homework search for massive open online programming courses. In *Proceedings of the 23rd International Conference on World Wide Web*, WWW '14, p. 491–502, New York, NY, USA, 2014. Association for Computing Machinery.

[7] Elena L. Glassman, Jeremy Scott, Rishabh Singh, Philip J. Guo, and Robert C. Miller. Overcode: Visualizing variation in student solutions to programming problems at scale. *ACM Trans. Comput.-Hum. Interact.*, Vol. 22, No. 2, March 2015.

[8] Philip J. Guo. Online python tutor: Embeddable web-based program visualization for cs education. In *Proceeding of the 44th ACM Technical Symposium on Computer Science Education*, SIGCSE '13, p. 579–584, New York, NY, USA, 2013. Association for Computing Machinery.