Vol.2012-DPS-152 No.20
Vol.2012-GN-85 No.20
Vol.2012-EIP-57 No.20
2012/9/14

学生のプログラミング演習における
トークンベースのコードクローン検出手法

岩本 舞[1,a)]　小島 俊輔[2,b)]　中嶋 卓雄[3,c)]

概要：大学等におけるプログラミング系科目のレポート課題において，他人の記述したプログラムをそのまま流用して提出するする行為が問題となっている．このプログラムのコピーを自動で検出する技術に，コードクローン検出技術がある．本研究は，学生の提出する不正コピーの検出に注目した，コードクローン検出アルゴリズムの開発を目的としている．本研究で提案するアルゴリズムは，トークンごとの比較を基礎としており，学生が提出する不正コピーの特徴，たとえば，関数の場所やプログラムの行単位の入れ替えをすべて検出できる．提案するアルゴリズムを実装し，学生 119 名から提出された課題に対して評価実験を行った．目視による判定と比較した結果，学生の作成した非常に短いプログラムにおいて，36 件の不正コピーのうち 32 件を検出することができた．

キーワード：コードクローン，トークン，検出システム，短いソースコード，プログラミング演習

Token-based Code Clone Detection Technique in a Student's Programming Exercise

Mai Iwamoto[1,a)]　Shunsuke Oshima[2,b)]　Takuo Nakashima[3,c)]

Abstract: The acts to submit the copied programs of other person make problems in the subject of the programming exercise in university curriculum. The code clone detection technique is to automatically detect the copied programs. In this research, we developed the code clone detection algorithm focusing on the detection of illicit copied codes of submitted reports of students in a programming exercise. Our proposed algorithm is based on the comparison of tokens and can declare the illicit copied codes invalid. The features of illicit copied codes such as swapping the functions and program lines are detected. We implemented the proposed algorithm and experimented to evaluate our system for the submitted subjects of 119 students. Compared to the human detection for small size of source codes of students in a programming exercise, our system found 32 codes as the illicit copy in 36 illicit copied codes.

Keywords: Code Clone, Token, Detection System, Short Source Code, Programming Exercise

1. はじめに

大学等のプログラミング系の講義科目では，学生が他人の書いたプログラムをそのままレポートとして提出する行為が問題となっている．以降では，このようなコピープログラムを**不正コピー**と表記する．このような行為は，成績

[1] 熊本高等専門学校 情報電子工学科
866–8501 熊本県八代市平山新町 2,627
Dep. of Information and Electronic Engineering, Kumamoto National College of Technology, 2,627, Hirayama-Shinmachi, Yatsushiro, Kumamoto 866–8501, Japan
[2] 熊本高等専門学校 ICT 活用学習支援センター
ICT Center for Learning Support, Kumamoto National College of Technology
[3] 862–8652 熊本県熊本市東区渡鹿 9-1-1
東海大学 産業工学部 電子知能システム工学科
Dep. of Electronics and Intelligent Systems Engineering, Tokai University, 9-1-1, Toroku, Kumamoto 862–8652, Japan
[a)] ye8251iwam@y.kumamoto-nct.ac.jp
[b)] oshima@kumamoto-nct.ac.jp
[c)] taku@ktmail.tokai-u.jp

ⓒ 2012 Information Processing Society of Japan

が正当に評価できず，また，学生の到達レベルが把握できないといった問題を引き起こし，さらには，盗用した学生のスキル向上を阻害する要因となる．このプログラムの不正コピーを自動で検出する技術として，コードクローン検出手法がある．これまでに多くのコードクローン検出技報が発表されており，実装例も多数あるが，産業界で生産されるコードを主な対象としており，学生の不正コピーを検出するためには，いくつか解決しなければならない問題があった．

本研究は，学生の提出する C 言語で記述された不正コピーに特化した，盗用の有無を自動判別するアルゴリズムを提案する．学生の提出する不正コピーには，次にあげる特徴がある．

TypeA　変数名や関数名の ID，数値や文字列，文字定数の付け替え

TypeB　コメント行の付加，削除，変更

TypeC　インデント幅の変更，空行の追加，削除，およびソースコード整形ツールを使用した変形

TypeD　関数や関数の宣言場所の入れ換えや，ステートメント行単位の入れ替え

TypeE　行や変数を入れ換えたことでロジックがおかしくなったコピー

TypeF　教員によって提供された雛型プログラムを学生が変更する場合

提案するアルゴリズムは，記述言語を C 言語に限定し，トークンごとにプログラムを比較することで，不正コピーのプログラムが持つこれらの特徴をすべて検出することができる．

第 2 章では，これまでのコードクローンに関する研究についてまとめ，学生の不正コピーにおけるコードクローン判定における問題点を明らかにする．第 3 章で本研究で提案するアルゴリズムの詳細を述べ，第 4 章で，実験方法，第 5 章で実験結果を述べる．最後に，第 6 章において，結果に対する検討と今後の課題についてまとめる．

2. 関連研究

コードクローンの関連研究としては，主にテキストやトークンを基礎とする字句レベルの手法と，木やグラフを基礎とした構文レベルの手法に分類することができる．文献 [11] では，さまざまな手法によるコードクローン検出方法を比較・検証している．

字句レベルの手法として，文献 [6] は，判定に，21 種類の関数メトリクスを使用しており，それを 4 つに分類して比較・検討している．また，文献 [1][4] では，巨大なソフトウェアにおいて，コードクローンを高速に発見することを目的としたツールが開発されている．比較前にユーザ定義名を特殊文字に置き換えるため，定義名が異なる場合でもコードクローンとして検出が可能である．行単位での比較には接尾辞木検索アルゴリズムが用いられており，線形時間でコードクローンが検出可能である．文献 [9] では，動的計画法に基づいた手法により，以前のテキスト比較ツールより高精度に 2 つのプログラム間のコードクローンを検出することができる．文献 [10] では，ソフトウェア間の最大クローン長と部分類似度に着目し，どの程度の値であればソースコードの流用があると言えるかを実験的に導出している．また，文献 [12] では，本研究と同じく学生の演習課題における不正コピーの検出を行っている．この研究では，コーディングスタイルに着目しており，プログラムからインデントや演算子など 59 の特徴量を抽出し，盗用の発見に用いている．

構文レベルの手法として，例えば文献 [5] では，依存グラフによりコードクローンを同定する．この方法は再現率と適合率に背反関係がないため，良好な検出結果が期待できる．文献 [2] では，抽象構文木を用いたアルゴリズムによりコードクローンを発見する．コードクローンはプログラムのまとまりとして検出されるため，機械的な手法で調整が可能である．また，これ以外の研究として，文献 [8] では，Java におけるクラスのコードクローンを，バースマーク（Birthmark）と呼ばれる改編しにくい特徴を 3 種類用いて，コードクローンを検出している．この手法は，プログラムの変形に対して耐性があり，小さなクラスを除くほとんどの実用的なクラスでは，クラスごとに別々の特性を示している．さらに，文献 [3] では，文献 [1][2][4][5] など 6 つのコードクローン検出方法を，8 つの C や Java で記述されたプログラムを用いて検証している．その結果，トークンベースの技術 [1][4] はかなり高い再現率となることを示した．

本研究で提案するアルゴリズムは，文献 [4][6] で用いられたトークンベースのコードクローン検出手法に分類される．トークンベースの手法を選択した理由は，先に述べた不正コピーにおける TypeA や TypeB, TypeC の特徴を検出する機能を有しているためである．しかし，これまでのトークンベース手法は，企業などで開発された巨大なコードを対象としており，また，インデントやコメントが正しく記述されたプログラムを想定してメトリクスを設定している．一方，学生が課題演習で提出するプログラムは，これらがすべて正しく記述されてるとは限らず，また，コピーしたことを隠すための細工が施されていることもあり，正確なコードクローン判定ができない．さらに，制御ロジックがおかしくなるような行の入れ換えをしただけのプログラムを提出する学生もおり，接尾辞木や抽象構文木を利用した [2][4] においても，正しい判定が困難となる．

これらのことを踏まえ，提案するアルゴリズムは，TypeA, TypeB, TypeC に加え，単純に行を入れ換えた TypeD や TypeE の不正コピーについても，コードクローンを正しく検知するようにした．また，教員があらかじめ提示した雛

Vol.2012-DPS-152 No.20
Vol.2012-GN-85 No.20
Vol.2012-EIP-57 No.20
2012/9/14

型プログラムを改良する演習では，コードクローンと誤判定されてしまうことが多い．そこで，レーベンシュタイン距離を基に，提出してもらったプログラムに前処理を施すことで，TypeF の特徴にも対応する．

3. 提案手法

本章では，提案するアルゴリズムの詳細を述べる．このアルゴリズムは，次に示す手順によってコードクローンを検出する．

Step1: コメント，インデント，空行などの文字を削除
Step2: 雛型と提出プログラムとの差分トークンを検出
Step3: ID，数値，ブレースの正規化とトークンへの分解
Step4: トークンごとのコードクローン率を算出し，不正コピーか否かを判定

以下に，各ステップの詳細を述べる．

3.1 Step1

コメントおよびインデントを削除する．この処理は，字句解析や構文解析は必要ではなく，正規表現による単純な文字列のリプレースで実現できる．

3.2 Step2

レーベンシュタイン距離（編集距離）[7] とは，文字列の編集における文字の削除，および追加を距離 1 としたとき，文字列全体を目的の文字列に置き換えるために必要となる最小の距離のことである．レーベンシュタイン距離は，文字列の長さを n とすると，動的計画法により $O(n^2)$ で求めることができる．

提案するアルゴリズムは，雛型プログラムと学生が提出したプログラムにおいて，行を単位とするレーベンシュタイン距離を計算する．この操作は，雛型プログラムに対して学生が追加・削除した行のみを抽出するためであり，教員が与えた雛型プログラムによる影響を最小限に抑え，TypeF の変更箇所におけるコードクローン検出を図る．学生が追加した行は，最短パスの縦方向の移動であり，最短パスをトレースしながら，縦方向の移動の際に該当行を出力すると得られる．

3.3 Step3

Step3 では，Step1 や Step2 で得られたプログラムを字句解析し，ID，数値，文字列の付け変え処理や，`if` 文,`while` 文の 1 行に 1 個のトークン列に変換する．ところで，`for` や `while`, `if` というキーワードは，後に続くステートメントが 1 行の場合，ブレース記号（`{}`）を省略することができる．そこで，`for` や `while`, `if` では，さらに構文解析を実施することで，図 2 に示すような，`{}`の自動補完を実施する．最後に，得られたプログラムの各行を図 3 のようなトークン列に分解する．行頭の行番号は便宜上補ったものであり，実際の出力には行番号は付けない．この時点で，変数名や数値はすべて単一の文字に置き換えられる．この手法は，文献 [4] でも用いられており，学生にとって最も簡単に不正コピーを偽装するためのテクニックである TypeA, TypeB, TypeC の検出を図る．

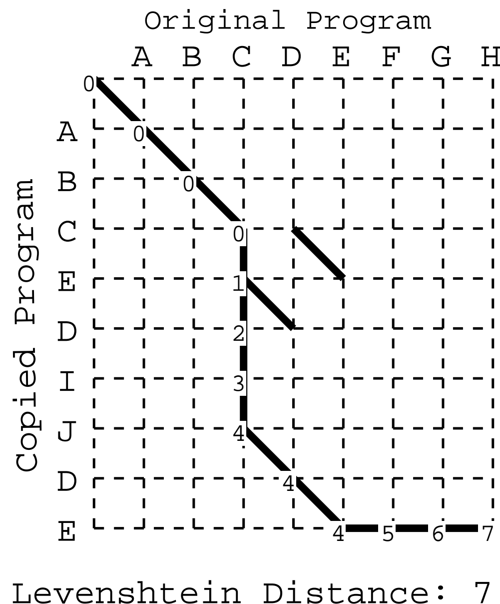


図 1 レーベンシュタイン距離の算出方法

正規化前

```
for(i=0; i<10; i++)
  for(j=i; j<10; j++)
    if(i==j)
      printf("*");
    else
      printf("#");
```

正規化後

```
for(i=0; i<10; i++){
for(j=i; j<10; j++){
if(i==j){
printf("*");
} else {
printf("#");
}
}
}
```

図 2 ソースプログラム中のブレース記号の正規化

```
1: FOR    12: ++    23: NUM   34: )      45: (
2: (      13: )     24: ;     35: {      46: STR
3: ID     14: {     25: ID    36: FUNC   47: )
4: =      15: FOR   26: ++    37: (      48: ;
5: NUM    16: (     27: )     38: STR    49: }
6: ;      17: ID    28: {     39: )      50: }
7: ID     18: =     29: IF    40: ;      51: }
8: <      19: ID    30: (     41: }
9: NUM    20: ;     31: ID    42: ELSE
10: ;     21: ID    32: ==    43: {
11: ID    22: <     33: ID    44: FUNC
```

図 3 ソースコードをトークン列に分解した様子

Step4

コードクローンの有無を調べたい 2 つのトークン列において，共通に含まれるトークン列の連続数が，あらかじめ

ⓒ 2012 Information Processing Society of Japan

設定したしきい値を超えた場合，そのトークン列はコードクローンであると見なす．たとえば，不正コピーのファイルの先頭から現れるトークン $p_1, p_2, p_3, \ldots, p_n$ を要素とするトークン集合 $\boldsymbol{P}$ を考える．$\boldsymbol{P}$ に含まれる m 個の連続したコードクローンを要素とする集合 $\boldsymbol{C_i}$ は次のように記述できる．

$$\boldsymbol{C_i} = \{p_j, p_{j+1}, p_{j+2}, \ldots, p_{j+m-1}\} \quad (1)$$

ここで，j は $\boldsymbol{P}$ におけるコードクローンの開始位置とする．このようなコードクローン $\boldsymbol{C_i}$ はトークン列 $\boldsymbol{P}$ 中に複数存在する．そこで，$\boldsymbol{P}$ に含まれるコードクローン全体の集合 $\boldsymbol{C}$ を次の式で定義する．

$$\boldsymbol{C} = \bigcup_{|\boldsymbol{C_i}| \geq \lambda_1} \boldsymbol{C_i} \quad (2)$$

ここで，記号 $|\boldsymbol{C_i}|$ は $\boldsymbol{C_i}$ の要素数，λ_1 はトークン列 $\boldsymbol{C_i}$ をコードクローン列と見なすしきい値とする．$\boldsymbol{P}$ に含まれる $\boldsymbol{C}$ のコードクローン率 $CR(\boldsymbol{P}, \boldsymbol{C})$ を以下の式で定義する．

$$CR(\boldsymbol{P}, \boldsymbol{C}) = \frac{|\boldsymbol{C}|}{|\boldsymbol{P}|} \quad (3)$$

この式は，$0 \leq CR(\boldsymbol{P}, \boldsymbol{C}) \leq 1$ であり，以下の条件を満たすとき，$\boldsymbol{P}$ は不正コピーであると判定する．

$$CR(\boldsymbol{P}, \boldsymbol{C}) \geq \lambda_2 \quad (4)$$

ここで，記号 $\lambda_2 (0 \leq \lambda_2 \leq 1)$ は，トークン列 $\boldsymbol{P}$ が不正コピーか否かを判定するしきい値である．コードクローン率の計算のイメージを図 4 に示す．ここで，図中の $\bullet$ は，オリジナルとコピー側にあるトークンが同じだった箇所を意味し，コピートークンが続くほど，右下方向の長い列となる．

提案するアルゴリズムは，コードクローンのトークン集合をしきい値 λ_1 から求めており，そのため，λ_1 を調整することで，不正コピーにおける特徴である TypeD，すなわち関数や行の入れ換えを検出することができる．ところで，不正コピーには，コピー前とコピー後でロジックが変化する図 5 のようなコピーが存在している．本稿では，このようなコピーを以後**劣化コピー**と表記し，TypeE の特徴に分類する．この例では，処理 A と処理 B は正常に動作するが，処理 D は $5 <$ `ID1` ≤ 10 の範囲で正常動作し，処理 C はまったく動作しなくなる．このような劣化コピーは，抽象構文木や依存グラフでは別の構文と見なされる．一方，提案アルゴリズムは，この例の場合 λ_1 を 9 以下にすることで，コードクローンを検出することができる．

4. 実験方法

4.1 評価基準

コードクローン検知手法を評価する際，誤検知となる False-Negative（以下 FN と表記）や False-Positive（以下 FP と表記）を評価の基準とすることが多い [3]．本研究においても，FP, FN により精度を評価する．R は再現率（Recall），P は適合率（Precision）と呼ばれており，以下の式 (5), 式 (6) で定義される．

$$R = \frac{tp}{tp + fn} \quad (5)$$

$$P = \frac{tp}{tp + fp} \quad (6)$$

コードクローンの判定の結果，tp はコードクローンを正しく検知した True-Positive（TP）の数，また fp, fn は誤検知となった FP や FN の数である．再現率と適合率は，どちらも 0 から 1 の間の値をとり，1 に近いほどコードクローンの有無の判定が正確であることを意味する．本研究は，不正コピーの常習者がいるか否かを判定することを目標としている．そのため，コードクローンの有無の調査は 1 つの演習ではなく，数多くの課題演習を通して実施する．もし，ある学生が多くの演習課題でコードクローン有りと判定されれば，その学生は不正コピーの常習者である．逆に，大部分の演習課題でコードクローンが無いと判定されれば，その学生は不正コピーの常習者ではない．これらのことから，提案する手法は，再現率（R）を最も優先順位の高い評価基準とし，適合率（P）の値は参考程度とする．

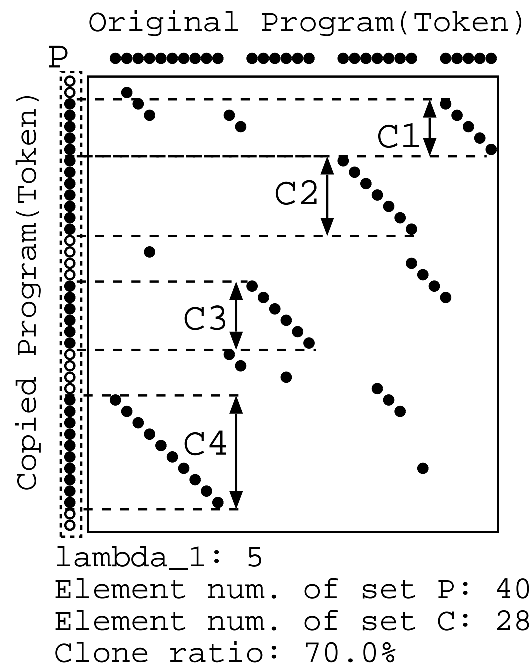


図 4　2 つのコード間のコードクローンとコードクローン率の計算

オリジナル	劣化コピー
`if(ID1<-10) A;`	`if(ID1<-10) A;`
`else if(ID1<-5) B;`	`else if(ID1>5) D;`
`else if(ID1>10) C;`	`else if(ID1>10) C;`
`else if(ID1>5) D;`	`else if(ID1<-5) B;`

図 5　劣化コピーにおけるコードクローンの発見

ⓒ 2012 Information Processing Society of Japan

4.2　実験データ

2012 年 6 月に熊本高等専門学校八代キャンパスの 3 年次のクラスにて実施したプログラム演習において，学生から提出された 119 件のプログラムを実験データとした．演習では，教員側から簡単な数字あてゲームの雛型プログラムを与えた．演習内容は，入力履歴，得点表示，当たるまでの平均回数，ヒントの出し方など，さまざまな拡張を自由に追加する演習とした．そのため，学生の作成したプログラムには，雛型プログラムで示した行が多く含まれているものの，拡張部分のプログラムは千差万別であった．なお，学生に提示した雛型プログラムは 35 行であり，学生が追加または変更した平均行数は 38.2 行，平均のプログラム総行数は 54.9 行であった．今回は，可能なすべての組合せとなる $n \times (n-1)$ 通り（$n = 119$ では，$14,042$ 通り）について調査した．最初に，目視検査により学生の提出したプログラム相互間における不正コピーの有無をチェックした．その結果，不正コピーが強く疑われる組合せが 36 通り見つかった．そこで，この 36 件を正解データとして，Perl にて実装した提案アルゴリズムにより不正コピーの判定をおこない，FP, FN から R, P を求めた．

5.　実験結果

実験は，次に述べる 3 つの条件を変更して実施する．

(1) 判定に使用するソースコードの範囲は，全ソースプログラムか，変更部分のみか
(2) 変数/数値/文字列の置換の有無
(3) `if`, `for`, `while` の後の括弧の補完の有無

まず，全ソースプログラムから得たトークン列を用いてコードクローンを判定した．結果を図 6 に示す．また，表 1 は，図 6 において，再現率 P として $R \geq 0.8$ を確保した際の λ_1, λ_2 の値および適合率の一覧である．図 6(a) では，変数や数値の置換をせず，図 6(b) では，変数や数値を置換した．どちらの図も，括弧の補完は実施していない．図 6(a) において，再現率として $R \geq 0.8$ を確保した場合，トークン列のしきい値が 9 から 23 の範囲であるのに対し，図 6(b) では，範囲が 13 から 32 と高くなる．この原因について，実験データを詳細に分析したところ，名前や数値の付け替えにより，変数名や数値だけを細工した不正コピーにおいて，不正コピーと正しく判定していることがわかった．再現率が全体的に上昇したことにともない，トークン列も長くなり，間違ったトークン列をコードクローンとする判定も少なくなることが期待できる．すなわちこの結果は，提案するアルゴリズムは TypeA の特徴を持つプログラムに対して，不正コピー判定が可能であったことを示している．次に，再現率を 0.8 まで確保した場合，適合率の範囲が 0.040 から 0.063 と低くなった．これは，不正コピーと判定されたもののうち，93.7% から 96.0%が FP，すなわち不正コピーではなかったことを意味する．この原因について実験データを詳細に分析したところ，学生が雛型プログラムに対して追加や改良をした行数が少ないほど，コードクローン率が高くなることがわかった．このような場合に，FP を減少させる方法として，コードクローン率のしきい値 λ_2 を大きくすることが考えられる．しかし，雛型プログラムをどの程度利用するかは学生に任されており，λ_2 の設定は一般に困難である．また，λ_2 を大きくすると，再現率が低下することが知られている．以上のことから，雛型プログラムに改良を加えた場合，全トークン列を基にしたコードクローン判定が難しいといえる．

次に，提案手法の Step2 においてレーベンシュタイン距離を求め，学生の追加したトークンのみを取り出した後，同様の実験を行った．結果を図 7，および表 2 に示す．こ

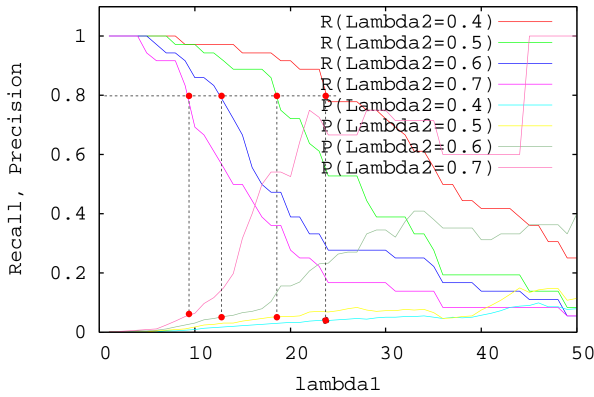
表 1　全トークン列より求めた最大 λ_1 と最大適合率 $P(R \geq 0.8)$

ID/数値置換	括弧の正規化	λ_2	λ_1	再現率 R	適合率 P
No	No	0.4	23	0.889	0.040
		0.5	18	0.861	0.051
		0.6	12	0.833	0.046
		0.7	9	0.833	0.053
No	Yes	0.4	29	0.806	0.046
		0.5	18	0.861	0.057
		0.6	14	0.806	0.063
		0.7	10	0.833	0.046
Yes	No	0.4	32	0.861	0.051
		0.5	26	0.806	0.073
		0.6	15	0.806	0.040
		0.7	13	0.861	0.055
Yes	Yes	0.4	23	0.889	0.040
		0.5	18	0.861	0.051
		0.6	12	0.833	0.046
		0.7	9	0.833	0.053

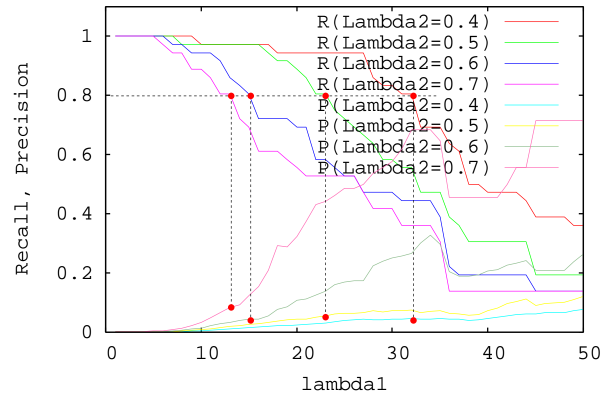
表 2　差分トークン列より求めた最大 λ_1 と最大適合率 $P(R \geq 0.8)$

ID/数値置換	括弧の正規化	λ_2	λ_1	再現率 R	適合率 P
No	No	0.4	14	0.833	0.149
		0.5	13	0.806	0.269
		0.6	9	0.806	0.209
		0.7	7	0.889	0.308
No	Yes	0.4	15	0.833	0.124
		0.5	12	0.833	0.150
		0.6	10	0.833	0.233
		0.7	7	0.889	0.185
Yes	No	0.4	21	0.806	0.131
		0.5	18	0.833	0.204
		0.6	14	0.806	0.230
		0.7	10	0.861	0.258
Yes	Yes	0.4	20	0.806	0.124
		0.5	16	0.833	0.229
		0.6	13	0.861	0.158
		0.7	9	0.806	0.184

ⓒ 2012 Information Processing Society of Japan

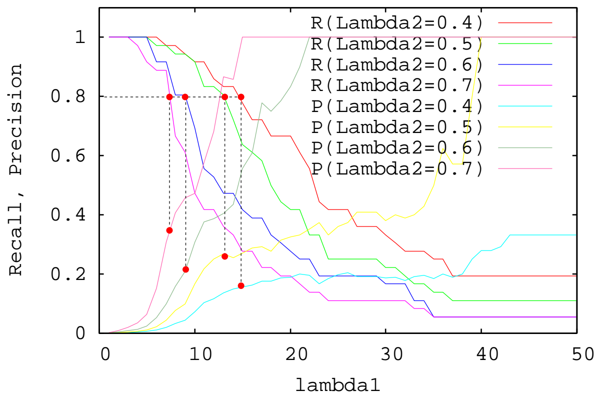


(a) 変数/数値/文字列の置換なし，括弧の補完なし

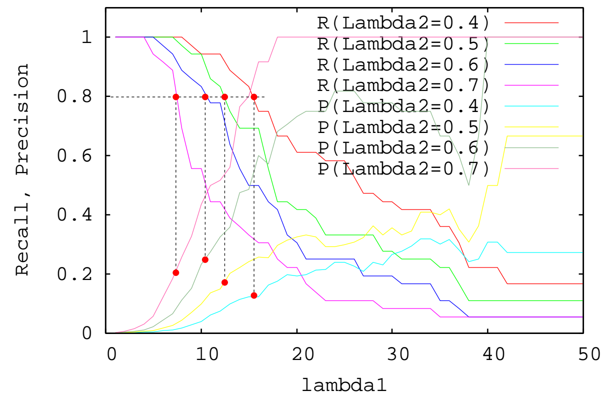


(b) 変数/数値/文字列の置換あり，括弧の補完なし

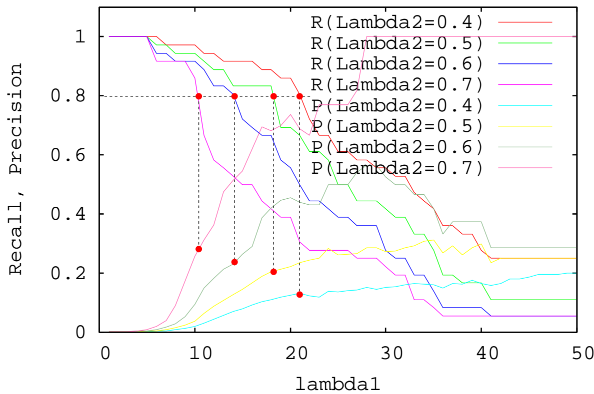
図 6 全トークン列より求めた再現率 R, 適合率 P



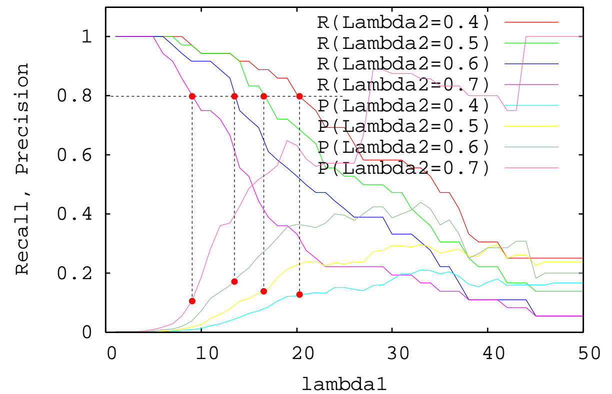
(a) 変数/数値/文字列の置換なし，括弧の補完なし



(b) 変数/数値/文字列の置換なし，括弧の補完あり



(c) 変数/数値/文字列の置換あり，括弧の補完なし



(d) 変数/数値/文字列の置換あり，括弧の補完あり

図 7 差分トークン列より求めた再現率 R, 適合率 P

の結果から，差分トークンのみをコードクローン判定に利用した場合，FP の数は明らかに少なくなり，全トークンを用いた場合と比べて，TypeF の特徴による影響が抑えられ，適合率が上昇することがわかった．次に，`if`, `while`, `for` の括弧の補完の有無によるコードクローンの判定結果の比較をする．まず，実験データから，括弧が補完されたことでコードクローンが新たに 2 件発見されている．これにより，再現率を上昇させることができ，TypeD の特徴を持つコードクローンの検出に対して有向に機能することを確認した．

一方，適合率については，図 7(a) と図 7(b)，あるいは図 7(c) と図 7(d) を比較すると，括弧の補完をしない手法が，若干高くなることがわかった．FN が高くなった原因を調べたところ，今回の実験データでは，TypeD のコードクローンが 2 件であったのに対し，括弧を補完したことでまったく別の用途に使用された `if` 文などの制御構造となってしまったケースがあることがわかった．ブレース記号{} も 1 つのトークンと数えられており，補完したブレースにより，コードクローンのしきい値 λ_1 を超えてしまう例が多く見られ，これにより，結果的に括弧を補わない場

ⓒ 2012 Information Processing Society of Japan

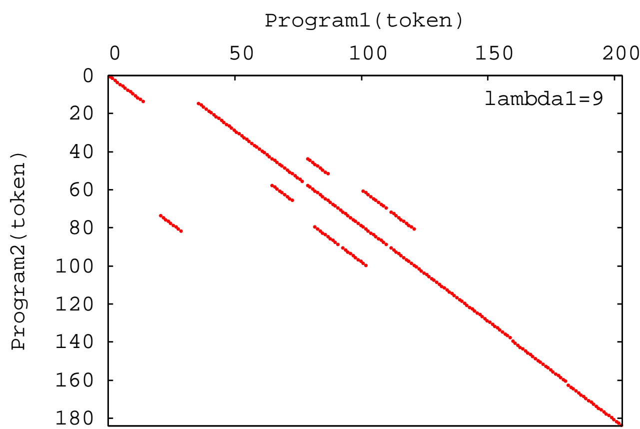


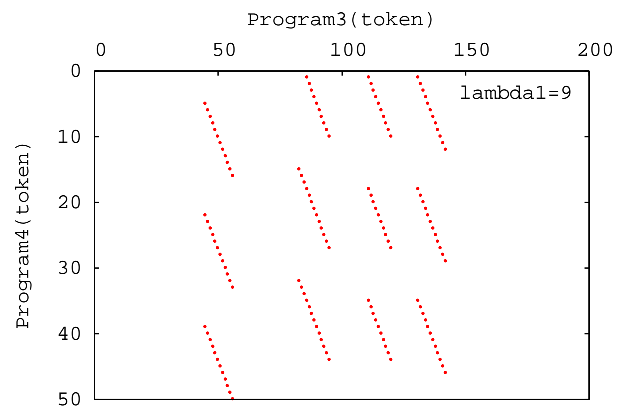
図 8 不正コピーを正しく判定したときのコードクローン検出の様子

合の FN が少なくなり，適合率が上昇した．

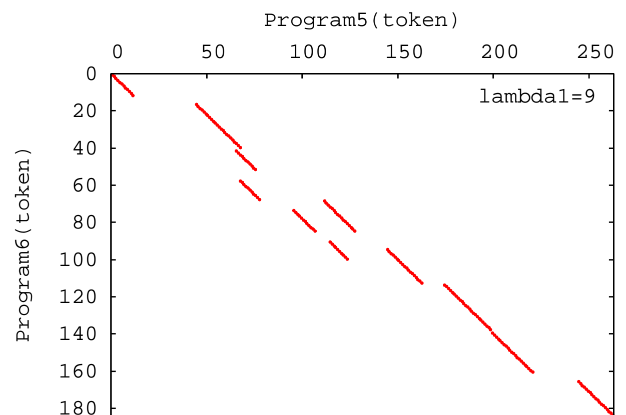
次に，名前と数値の置換の有無について，図 7(a) と 7(c)，あるいは図 7(b) と図 7(d) を比較すると，置換なしとした場合，λ_1 が 7 から 15 程度に小さくないと，0.8 の再現率が確保できないことが分かる．逆に，名前や数値を置換した場合，λ_1 は 9 から 25 程度まで大きくしてもよい．これは，図 6 の結果とも一致し，TypeA に対するコードクローン検出がうまく機能していることを意味する．TypeD と TypeE の特徴を持つ不正コピーの検出については有効性を検証する十分なデータが集まっていない．しかしながら，行の入れ換え工作により目視検査で発見できなかった不正コピーを，提案するアルゴリズムは高いコードクローン率を示して発見しており，不正コピーの正解グループに追加した．このことから，TypeD や TypeE の特徴についても，提案手法は有効に機能している．

今回検出されたもののうち，TP の例を図 8 に示す．横軸がオリジナルのプログラム，縦軸がコピーと判定されたプログラムのトークン列を示している．一つの点は，その位置のトークンが同一のものであることを意味する．すなわち，この点の連続は，2 つのプログラムに同一のトークン列が存在するということを意味する．図では，トークンが 9 つ以上連続する $\lambda_1 = 9$ の場合を示している．

図 8 では，最初のトークンから最後のトークンまでほぼ斜めに線が伸びていることから，完全な不正コピーである．一方，図 9 は FP の例である．図 9(a) は，自己クローンにより FP となった．自己クローンとは，プログラム内で自分自身のソースコードの一部を同一プログラム中の 2 ヶ所以上にコピーしたものであり，初心者のソースコードによく見られる特徴である．図では，平行な複数の線として観測できる．図 9(a) にあげた自己クローンによる FP を防ぐ方法としては，自己クローンをクローンと見なさない方法があるが，その場合，コピーした他人のプログラムをさらに自己コピーした場合に対応できない．図 9(b) は，不正コピーでないものを誤ってコピーと判定した場合のコードクローン検出の様子である．ここでは，大部分でコードクローンが検出されているものの，一つ一つは短く，図 8 ほどの直線とはならない．



(a) 自己クローンの場合



(b) 短いコードクローンが偶然連続した場合

図 9 不正コピーでないものを誤ってコピーと判定したときのコードクローン検出の様子

6. おわりに

学生の不正コピーには，関数の定義場所やプログラムの行単位の入れ替え，変数名や数値の付け替え，コメントや文字列定数の変更，ソースコード整形ツールを使用した変形などの特徴がある．本稿では，このような不正コピーの検出に特化した，トークンベースのアルゴリズムを提案した．目視検査による判定の結果と，提案アルゴリズムによる判定結果を比較したところ，学生の作成した非常に短いプログラムにおいて，再現率 0.8 を実現したしきい値では，14,042 通りの組合せに含まれる 36 件の不正コピーのうち 32 件を検出することができた（TP）．このとき，間違って不正コピーと判定した件数が 72 件（適合率 0.302）であった（F P）．以上のことから，TypeA および TypeC のブレース記号に関しては，実験により提案するアルゴリズムの有用性を確認した．また，TypeB や TypeC, TypeF の特徴は，提案するアルゴリズム自体がコメントやインデント，空行などを書き換えることにより，コードクローン判定には影響を及ぼさないことが分かっている．さらに，TypeD と TypeE については，人間の目視検査で見落とし

ⓒ 2012 Information Processing Society of Japan

たコードクローンを発見したことで，有効性を確認できた．

本稿で提案したアルゴリズムは，不正コピーの常習者を探すことを目的としている．課題演習を何度か繰り返すことで，不正コピー常習者と判定する可能性は極めて低い．むしろ，このような機能を用いたチェックをすることを事前にアナウンスすることで，学生の安易な不正コピーを防止する狙いがある．そのため，FP や適合率については取り扱ってこなかった．しかしながら，FP を減らすことができれば，不正コピー常習者だけでなく，時おり不正コピーをする学生を検出できる．

今後の課題として，FP を減少させることができないか，検討をしている．自己クローンによる FP では，コピー率が高くなおかつコピー元と見なしているプログラムのトークンの大部分がコピー先のソースコードに含まれる場合に不正コピーであると判定する方法がある．コピー元のプログラムの一部のみがコピー先に多数のクローンとして検出された場合，偶然同じトークン列が現れた可能性が高い．また，短い同一トークン列が多数存在するような FP への対応として，各トークン列の長さを考慮したコードクローン検出を検討している．他にも，一度コードクローンと判定されたトークン列を 2 度以上重複してコードクローンと見なしているものがいくつかあり，これを防ぐため，トークン列の長い方からマッチングを行うなどの手法が考えられる．また，今回は実験データ数が少なく，特に，TypeD, TypeE については，十分に検証したとはいえない．今後，データ数を増やすことで，詳細に検討したい．さらに，提案アルゴリズムの本来の目的は，不正コピーを減らすことにある．そのため，不正コピーをチェックしている事実を学生に伝えることで，実際に不正コピーの提出についての学生の意識調査を実施するなどした客観的なデータが必要であろう．

7. 謝辞

本稿を作成するにあたり，提出した課題プログラムの研究使用を承諾して頂いた熊本高専八代キャンパス 3 年機械知能システム工学科，建築社会デザイン工学科，生物化学システム工学科の学生諸君に感謝申し上げます．

参考文献

[1] Baker, B. S.: On finding duplication and near-duplication in large software system, *WCRE*, IEEE Computer Society, pp. 86–95 (1995).

[2] Baxter, I. D., Yahin, A., Moura, L., Anna, M. S. and Bier, L.: Clone Detection Using Abstract Syntax Trees (1998).

[3] Bellon, S., Koschke, R., Antoniol, G., Krinke, J. and Merlo, E.: Comparison and Evaluation of Clone Detection Tools, *IEEE Transactions on Software Engineering*, Vol. 33, pp. 577–591 (2007).

[4] Kamiya, T., Kusumoto, S. and Inoue, K.: CCFinder: A Multilinguistic Token-Based Code Clone Detection System for Large Scale Source Code, *IEEE Transactions on Software Engineering*, Vol. 28, pp. 654–670 (2002).

[5] Krinke, J.: Identifying Similar Code with Program Dependence Graphs, *8th Working Conference On Reverse Engineering*, Stuttgart, Germany, IEEE, pp. 301–309 (2001).

[6] Mayrand, J., Leblanc, C. and Merlo, E. M.: Experiment on the Automatic Detection of Function Clones in a Software System Using Metrics, *ICSM*, IEEE Computer Society, pp. 244–253 (1996).

[7] Navarro, G.: A guided tour to approximate string matching, *ACM Computer Surveys(CSUR)*, Vol. 33, No. 1, pp. 31–88 (2001).

[8] Tamada, H., Nakamura, M., Monden, A. and ichi Matsumoto, K.: Java Birthmarks –Detecting the Software Theft–, *IEICE Trans. and Syst.*, Vol. E88-D, pp. 2148–2158 (2005).

[9] Yang, W.: Identifying Syntactic Differences Between Two Programs, *Software - Practice and Experience*, Vol. 21, pp. 739–755 (1991).

[10] 岡原　聖，真鍋雄貴，山内寛己，門田暁人，松本健一：ソースコード流用のコードクローンメトリクスに基づく検出手法 (ソフトウェア解析)，電子情報通信学会技術研究報告. KBSE, 知能ソフトウェア工学， Vol. 109, No. 307, pp. 73–78 (2009).

[11] 肥後芳樹，楠本真二，井上克郎：コードクローン検出とその関連技術，電子情報通信学会論文誌 D， Vol. J91-D, No. 6, pp. 1465–1481 (2008).

[12] 武田隆之，牛窓朋義，山内寛己，門田暁人，松本健一：コーディングスタイルの特徴量とソースコード盗用との関係の分析，情報処理学会研究報告. ソフトウェア工学研究会報告， Vol. 2010, No. 8, pp. 1–8 (2010).

ⓒ 2012 Information Processing Society of Japan