修士学位論文

題目

コードクローン検出の後処理としての統一的ラベリング手法

指導教員
肥後 芳樹 教授

報告者
清水 ささら

令和 8 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 ソフトウェア工学講座

令和 7 年度 修士学位論文

コードクローン検出の後処理としての統一的ラベリング手法

清水 ささら

内容梗概

コードクローンはソースコード中に存在する，一致または類似するコード片であり，バグの修正漏れを引き起こす原因となる．種々のコードクローン検出ツールが提案されているが，それぞれが独自の手法を用いて検出を行っている．コードクローンは類似度に応じて Type1，Type2，Type3，Type4 の 4 種類に分類されるが，これらの定義は必ずしも厳密ではなく，特に Type3 と Type4 の境界は曖昧である．その結果，ある検出ツールでは Type3 として検出されたコードクローンが，別の検出ツールでは Type4 として扱われるといった不整合が生じる．また，多くのコードクローン検出ツールは検出のみを行い，どの Type のクローンであるかのラベル付けを行わない．このことから，複数のツールの検出結果を単一のデータセットとして扱えない問題が発生する．

本研究では，これらの問題を解決するために，コードクローン検出後の後処理として統一的なラベリング手法を提案する．提案手法を用いることで，使用するコードクローン検出ツールに依存することなく，検出されたコードクローンに対して一貫した基準に基づくラベリングが可能となる．近年，ソフトウェア工学分野ではコードクローン情報を大規模言語モデル（LLM）の学習データとして利用する研究が増加しており，本研究で提案するラベリング手法は，高品質にラベル付けされたコードクローンデータの構築にも寄与する．提案手法の有効性を評価するため，広く利用されているコードクローンデータセットである BigCloneBench に含まれるコードクローンを対象に再ラベリングを行った．そして，BigCloneBench における既存のラベリングと提案手法によるラベリングについて被験者実験を実施し，どちらがコードクローンの定義により適合しているかを比較・分析した．

評価実験の結果，BigCloneBench で Type4 と判定されていたクローンペアを提案手法が Type3 と分類したケースでは，提案手法の方が人間の感覚に近いことが確認された．しかし，その逆のケースでは既存のラベルの方が支持される傾向にあり，提案手法の判定ルールをより実態に合わせて細かく調整していく必要性が示唆された．また，被験者によって判断が分かれた事実は，クローンの境界が曖昧さを示しており，本研究が目的とする一貫したラベリング基準を確立することの重要性を再確認した．

主な用語

コードクローン

BigCloneBench

抽象構文木

目 次

1　はじめに

コードクローンとは，プログラムテキスト中の一致または類似するコード片である [42]. コードクローンにバグが含まれているとバグの修正漏れを引き起こす原因になり，ソースコードの保守性が低下する要因の一つとなる [24]．コードクローンは，構文的な類似度に基づいて Type1，Type2，Type3，Type4 の 4 種類に分類される [26]．ソースコードの規模が大きくなると，手動でコードクローンを管理することは困難となる．そのため，コードクローンを自動で検出するツールの研究が行われている．

現状では，ソースコード中のコードクローン検出は，単一のツールを用いて行われる．様々なコードクローン検出ツールが存在するが，それらは，検出する際に用いる中間表現や計算アルゴリズムが異なる．例えば，NiCad [25] は行単位での検出，CCFinder [15]，NIL [22] は字句単位での検出を行う．単一のコードクローン検出ツールを用いた検出では，ソースコードから検出できるコードクローンに偏りが生じる．この課題解決のため，一つのソースコードに対し複数のコードクローン検出ツールを実行することが考えられる．複数のツールを利用することで可能な限りコードクローンを検出する．しかし，各ツールは，分類の基準が異なるため，コードクローンとして検出されたコードに対して異なるラベリングを行う場合がある．また，コードクローンの検出は行うものの，分類は行わないツールも存在する．ラベリング結果が異なることや，ラベリングが行われないことは，複数ツールから得られたクローン検出結果を用いて大規模なコードクローンのデータセットを作成する際に，単一のデータセットとして統合ができないという問題が発生する．

この問題解決のため，本研究では，コードクローン検出後の後処理として統一的なラベリング手法を提案する．提案手法を用いることで，複数の検出ツールから検出されたコードクローンに対し，統一されたラベリングを行うことができる．統一的なラベリングが行われたコードクローンは単一のデータセットに統合が可能となり，コードクローンデータを他の研究や開発に利用しやすくなる．

また，既存のコードクローンデータセットである BigCloneBench [30] は最大規模のデータセットであり，Type3 および Type4 の分類に行単位の類似度を用いているが，こうした類似度に基づく分類は解析の単位粒度に結果が左右されるという課題がある．したがって，本研究では類似度という数値的な判定ではなく，プログラムの論理的な構造を直接反映する抽象構文木の構造的特徴に基づいた新たな Type 分類の定義を導入する．これにより，解析の単位粒度に左右されない，一貫性のある客観的な分類を実現する．

本研究では, 提案手法を評価するために被験者実験を行った．被験者実験では, BigCloneBench に含まれるクローンペアを対象とし，BigCloneBench による分類と，提案手法による分類のどちらが人間の感覚に近いかを調査した．その結果，BigCloneBench で Type4 に分類さ

れ，提案手法で Type3 と分類されたクローンペアについては，提案手法による分類の方が，人間の感覚に近いことがわかり，BigCloneBench の分類が人間の直感と反するケースが存在することが確認できた．一方，BigCloneBench で Type3 に分類され，提案手法で Type4 と分類されたクローンペアについては，BigCloneBench の分類の方が，人間の感覚に近いことがわかり，分類の定義について修正を行う必要性が示唆された．定義の修正を行うことで，人間の感覚により近い分類が可能となる．また，被験者ごとの傾向から，Type3 および Type4 のコードクローンの分類の定義の曖昧さが示唆され，本研究の目的である，統一的なラベリング手法の必要性が改めて確認された．

以降，2 章では，本研究で用いる技術について述べる．3 章では，本研究の動機について述べる．4 章では，提案手法の詳細について述べる．5 章では，評価実験の詳細について述べ，6 章でその結果と考察を述べる．7 章では，妥当性の脅威を述べ，8 章では，まとめと今後の課題について述べる．

2　準備

2.1　コードクローン

コードクローンとは，プログラムテキスト中の一致または類似するコード片である [42]. コピーとペーストによるプログラミングや，意図的に同一の処理を繰り返して書くことにより発生する．複数のコードクローンがソフトウェアプログラム中に存在した場合，あるコードクローン上にバグが見つかると，他のコードクローンにもバグが含まれる可能性が出てくる．そのため，開発者は，全てのコードクローンを追跡する必要がある．これは，バグの修正漏れを引き起こす原因になり，ソースコードの保守性が低下する要因の一つとなる [24]. また，互いにクローン関係であるソースコード片のペアを，クローンペアと呼ぶ.

一般にコードクローンは，構文的な類似度に基づき，以下の 4 種類に分類される [26].

Type1 改行，コメント，空白を除いて一致するコードクローン

Type2 Type1 に加えて，リテラル，識別子，型の違いを除いて一致するコードクローン

Type3 Type2 に加えて，文の挿入，削除，変更を除いて一致するコードクローン

Type4 構文は異なるが同じ機能を提供するコードクローン

上記の Type3，Type4 の境界は曖昧であるため，コードクローン検出ツールによって分類の基準が異なる.

2.2　コードクローン検出ツール

ソースコードの規模が大きくなると，ソースコード中のコードクローンの量も増加するため，手動でコードクローンを管理することは困難となる．そのため，ソースコード中から自動的にコードクローンを検出するツールが研究されている [4, 9, 14, 15, 21, 22, 25, 28, 35, 41].

行単位で解析を行う手法として代表的な NiCad [25] は，ソースコードの整形と正規化を行い，行の一致率を見ることでコードクローンの検出を行う．検出だけでなく，Type1 から Type3 までのラベリングを行う.

字句単位で解析を行う手法は, ソースコードをトークン列に変換して比較を行う．CCFinder [15] や iClones [9]，LVMapper [34] は大規模なプロジェクトから高速にクローンを抽出することに長けているが，出力結果はコードクローンがどこに含まれるかの情報のみであり，どの Type に分類されるかのラベリングは行われない．一方で，NIL [22] や SourcererCC [28] は，字句単位の類似度計算に基づいたラベリングを行う.

2.4 節で述べる抽象構文木を利用する手法として，Deckard [14] が挙げられる．Deckard はクローンの検出だけでなく，分類に応じたラベリングも行う．

グラフ形式の中間表現を用いる手法は，主に Type4 クローンの検出を目的とする．Yang らの手法 [35]，StoneDetector [4]，GroupDroid [21]，CCGraph [41] などがこれに該当する．これらの手法は，グラフの構造的一致を判定するため，記述形式に依存しない高度な解析が可能である．一方で，グラフの同型判定アルゴリズムは高い計算コストを伴う点が課題となる．また，これらのツールは，機能の等価性に焦点を当てているため，Type1 から Type3 への詳細なラベリングまでは行わない．

それぞれのツールについて表 1 に示す．表 1 では，コードクローン検出のための手法と，ツールがコードクローンを検出する際，Type 分類のラベルを出力するか否かを示す．コードクローン検出ツールはそれぞれ独自の中間表現 (抽象構文木，グラフ，トークン列など) や類似度計算アルゴリズムを用いているため，同じソースコードからコードクローンを検出する場合でも検出されるコードクローンが異なったり，同じ箇所がコードクローンとして検出された場合でもその分類が異なったりする．また，コードクローン検出ツールには，検出のみを行い，分類を行わないツールも存在する．

表 1 に示したコードクローン検出ツールの他に，機械学習を用いたコードクローン検出手法も存在する [5, 8, 19, 20, 27, 38, 40]．

表 1: 既存のコードクローン検出ツール

手法	ツール名/著者名	ラベルの有無
行単位での検出	Nicad [25]	あり
字句単位での検出	NIL [22]	あり
	CCFinder [15]	なし
	iClones [9]	なし
	SoucererCC [28]	あり
	LVMapper [34]	なし
抽象構文木を用いた検出	Deckard [14]	あり
グラフを用いた検出	Yang [35]	なし
	StoneDetector [4]	なし
	GroupDroid [21]	なし
	CCGraph [41]	なし

2.3 コードクローンのデータセット

既存のコードクローンデータセットを表 2 に示す．BigCloneBench [30] は，Java プロジェクトを対象とした大規模なデータセットである．詳細については，2.3.1 項で述べる．

近年では，同一の処理を行うものの，構文が大きく異なる Type4 に特化したデータセットや，多言語を対象としたデータセットが作成されている．SemanticCloneBench [1] は，Java, Python, C, C#の 4 言語にまたがる 4,000 件のクローンペアを提供する．また，GPT-CloneBench [2] は，近年の大規模言語モデル（LLM）の進展を背景に構築された新しいデータセットであり，Java, Python, C#における 3 万件以上のクローンペアが含まれる．

特定の言語において高い精度で検証を行うためのデータセットも存在する．FEMPDataset [10, 11] は Java の機能等価なメソッドペアに焦点を当てており，高精度な解析が求められる研究で利用されている [13, 29]．一方で，Clone oracle [18] は 66 件と小規模ではあるが，C/C++ におけるクローンペアを提供している．

2.3.1 BigCloneBench

本研究で用いる，BigCloneBench は，コードクローン検出器の性能評価で用いられる Java の大規模なベンチマークである [30]．複数のソフトウェアプロジェクトから収集した大規模データセット IJDataset2.0 から，特定の機能を実装している可能性のあるソースコードを自動で特定し，対象の機能を正しく実装しているか否かが手動で判断された．BigCloneBench に含まれるクローンペアは，いずれもメソッド単位のコード片から構成される．

BigCloneBench では，Type1 および Type2 のコードクローンは，ソースコードの正規化後テキストの一致を確認することで判定している．一方，Type3 および Type4 のコードクローンは，行単位での構文的類似度に基づいて 0 以上 1 未満の範囲で，次の 5 種類に分類される．この構文的類似度は，行の一致する割合で求められる．

Strongly-Type3 0.7 以上 1.0 未満

表 2: 既存のコードクローンのデータセット

データセット名	言語	データ数
BigCloneBench [30]	Java	8,612,826
Clone oracle [18]	C/C++	66
SemanticCloneBench [1]	Java,Python,C,C#	4,000
GPTCloneBench [2]	Java,Python,C#	37,149
FEMPDataset [10, 11]	Java	1,342

Moderately-Type3 0.5 以上 0.7 未満

Weakly-Type3 0.0 以上 0.5 未満

コードクローンの研究において，Weakly-Type3 クローンは Type4 クローンとして用いられている [3, 19, 20, 23, 33, 37, 39]．また，機械学習を用いたコードクローン検出手法において，BigCloneBench は学習データとして用いられている [19, 20, 33, 37, 38]．しかし，BigCloneBench は分類が不正確で，Type4 コードクローンとして扱うには不適切であることが指摘されている [16, 17]. BigCloneBench を Type4 クローンの検出性能評価に用いる場合，誤った分類の影響で結果の妥当性が脅されるため，Type4 クローンの検出を目標とする，機械学習を用いたコードクローン検出の評価結果に対する信頼性が低下する [16, 17, 36].

BigCloneBench に含まれるそれぞれの Type に分類されたクローンペアの数は表 3 に示す通りである．

2.4 抽象構文木 (AST)

AST とはソースコードの構文構造を木構造で表現したデータ構造である．図 1 に Java で書かれたソースコード例とそれに対応する AST を示す．AST において，各頂点 (ノード) はプログラムを構成する構文要素を表し，各辺はそれら要素間の論理的な親子関係や入れ子構造を表す．図 1 の例では，method_declaration ノードに，五つの子ノードが存在する (modifiers, integral_type, identifier, formal_parameters, block)．これら五つのノードは，method_declaration ノードを親として持つ．この親子関係に基づき，AST は階層的な構造を持つ.

以下に AST に関する用語を示す.

根ノード 親を持たないノードを指す [7]．図 1 では赤く示された method_declaration ノードである.

葉ノード 子を持たないノードを指す [7]．図 1 では緑で示されたノードである．斜体で示されているのがそのノードの持つ値である.

本研究では，AST 解析ライブラリとして Tree-sitter [1]を用いた．既存の Java 言語を対象

表 3: BigCloneBench に含まれるコードクローンの分類

Type1	Type2	Strongly-Type3	Moderately-Type3	Weakly-Type3	合計
48,116	4,234	21,966	88,306	8,450,204	8,612,826

[1]https://github.com/tree-sitter/tree-sitter

```
public int ret(int a) {
    return a
}
```

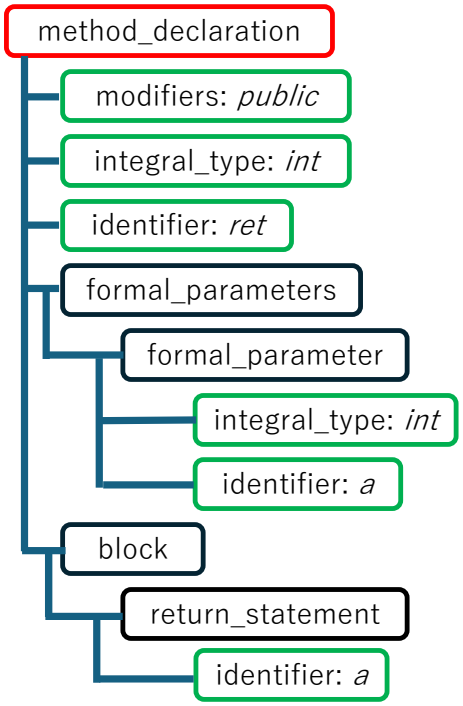

図 1: ソースコード例と対応する AST

とした研究では，AST 生成に Eclipse JDT[2] が広く利用されている [6, 31]．JDT は Java 言語専用の解析基盤であるのに対し，Tree-sitter は多言語共通のインターフェースを提供している．本研究では，Java 言語を対象としているが，将来的に他言語への拡張を考えており，他言語への拡張性が高いことから Tree-sitter を選択した．

[2]https://github.com/eclipse-jdt/eclipse.jdt.core

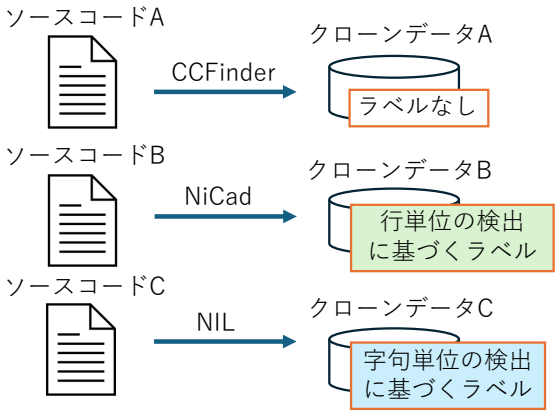

図 2: コードクローン検出の現状

3　本研究の動機

本章では，既存のコードクローン検出手法の問題点，および既存のデータセットのラベリング基準の不明瞭さという二つの観点から，本研究の動機について述べる．

3.1　既存のコードクローン検出手法の問題点

ソースコードに含まれるコードクローンを検出するため，現在では様々なコードクローン検出ツールが存在する．現状では，それらのツールは様々なソースコードに対して単一で用いられており，コードクローンの検出を行っている (図 2)．各コードクローン検出ツールは，異なるアルゴリズムを利用してコードクローン検出を行う．例えば，CCFinder [15], NIL [22] は，字句単位の比較を行うことで検出を行う．Nicad [25] は行単位の比較を行うことで検出を行う．これは，識別子の正規化やコードの整形といった前処理を適用した後の各行を比較することでコードクローンを検出する．しかし，単一のコードクローン検出ツールを用いた検出では，偏りがあり，対象ソースコード中に存在する全てのコードクローンを検出できるわけではない．

検出されるコードクローンの偏りを軽減するため，複数の検出ツールを対象ソースコードに実行することが考えられる (図 3)．複数のツールを実行することで可能な限り多様なコードクローンの検出を試みる．しかし，各ツールにおけるコードクローン分類の基準が異なるため，検出後にどの Type に分類されるかのラベリング結果が統一されない問題が起こる．例えば，ある関数 A とある関数 B がツール 1 によって Type3 として検出された場合に，別のツール 2 では，Type4 として検出される場合がある．また，コードクローンの大規模データベース作成の際にも，この問題が発生する．ツールごとにラベリング結果が異なるため，データの統合ができない．さらには，コードクローンの検出は行うものの，ラベリングを行

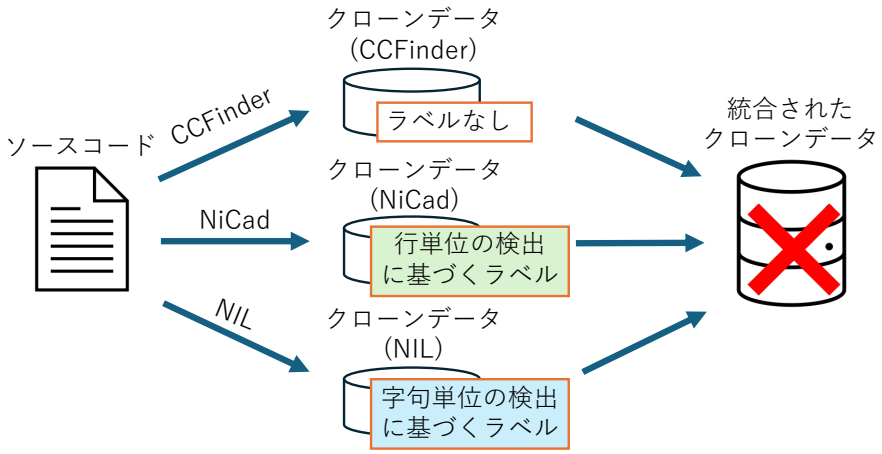

図 3: コードクローン検出の問題

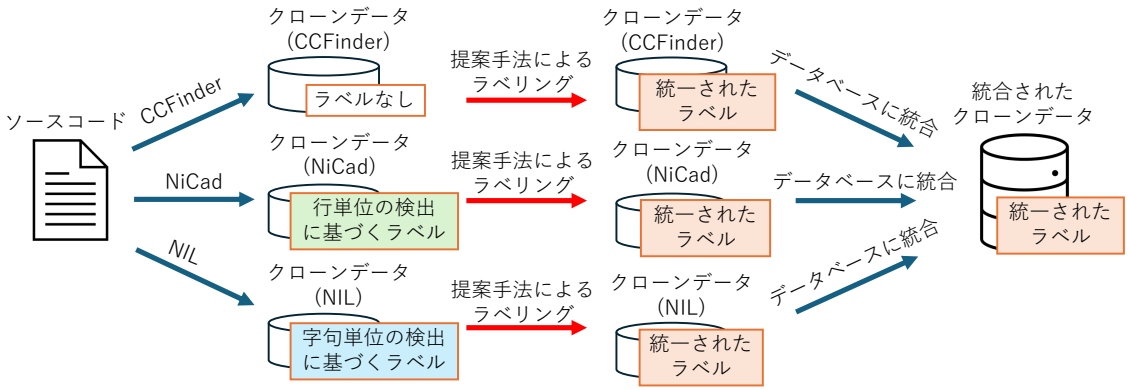

図 4: 提案手法による解決策

わないツールも存在する．そのため，複数のクローン検出結果をそのラベリングも含めて単一のデータセットとして扱うことはできない．

この問題を解決するため，提案手法では，コードクローン検出の後処理としての統一的なラベリングを行う (図 4)．これにより，複数ツールから検出されたコードクローンに一貫した分類に基づいたラベリングを行うことができる．統一的なラベリングを行うことで，それぞれのツールの検出結果を単一のデータセットに統合することが可能となる．これにより，コードクローン情報の質が向上し，コードクローンデータを他の研究や開発に利用しやすくなる．例えば，LLM の学習用データとしての利用が挙げられ，LLM ベースの超高精度なクローン検出の実現が見込まれる．

提案手法を用いて，ソースコード中のコードクローンの検出を行うわけでなく，コードクローン検出後のラベリングを行ったのは，実行時間の問題からである．対象関数の数を n

とした場合，提案手法の実行回数は n^2 となる．提案手法では，抽象構文木を用いて分類を行っており，二つの木を比較すること自体が非常に重い処理である．全ての関数のペアを比較して結果を得ることは時間的に現実的ではない．また，提案手法は入力をクローンペアと想定しているため，一定以上の差異がある場合に Type4 として判定している．そのため，提案手法はクローンペアではない関数のペアが与えられた場合にそれがクローンペアではないのか，もしくは Type4 クローンなのかが判定できない．これらのことから，提案手法ではコードクローン検出後に限定した統一的なラベリングを行う．

3.2 既存データセットにおけるラベリングの問題点

BigCloneBench [30] は，既存のコードクローンデータセットの中で最大規模のデータセットである．BigCloneBench では，Type1 および Type2 のコードクローンは，ソースコードの正規化後テキストの一致を確認することで判定している．一方，Type3 および Type4 のコードクローンは，行単位での構文的類似度に基づいて 0 以上 1 未満の範囲で，3 種類に分類される．この構文的類似度は，行の一致する割合で求められる．具体的には，行単位の類似度が 0.7 以上 1.0 未満のペアを Strongly-Type3，0.0 以上 0.5 未満のペアを Weakly-Type3 と定義している．

しかし, 類似度という数値指標に基づく分類は, 解析の単位粒度に結果が大きく左右される．そこで本研究では, 行単位の類似度による分類結果が, 字句という異なる単位を用いた場合にどの程度変動するかを調査した．具体的には，BigCloneBench に含まれる Strongly-Type3，Moderately-Type3，Weakly-Type3 のコードクローンについて，字句単位で比較を行い，一致率を計算した．字句単位の類似度算出にあたっては，二つのトークン列間における最長共通部分系列 (LCS: Longest Common Subsequence) [12, 32] を用いた．LCS は，トークンの出現順序を維持したまま，共通して現れる最長の系列を特定する手法である．単純な語彙の集合比較とは異なり，ソースコードの論理的な記述順序を反映した類似度を計測できるため，本分析の指標として採用した．具体的には，クローンペアの各トークン列を S_1, S_2 としたとき，以下の式により類似度 Sim_{LCS} を算出した．

$$Sim_{LCS}(S_1, S_2) = \frac{|LCS(S_1, S_2)|}{\max(|S_1|, |S_2|)}$$

算出結果を図 5 に示す．解析の結果，行単位の分類と字句レベルの類似度の間には顕著な乖離が確認された．例えば，Strongly-Type3 に分類されていたクローンペアについて，字句単位の類似度が閾値である 0.7 より小さくなるペアが存在し．また，Weakly-Type3 についても，字句単位の類似度が閾値である 0.5 より大きくなるペアが存在した．

以上の知見は，行単位や字句単位の類似度に基づく分類基準が単位粒度に依存して一貫性がないことを示唆しており，類似度を用いない新たな分類基準が必要であることを示す．し

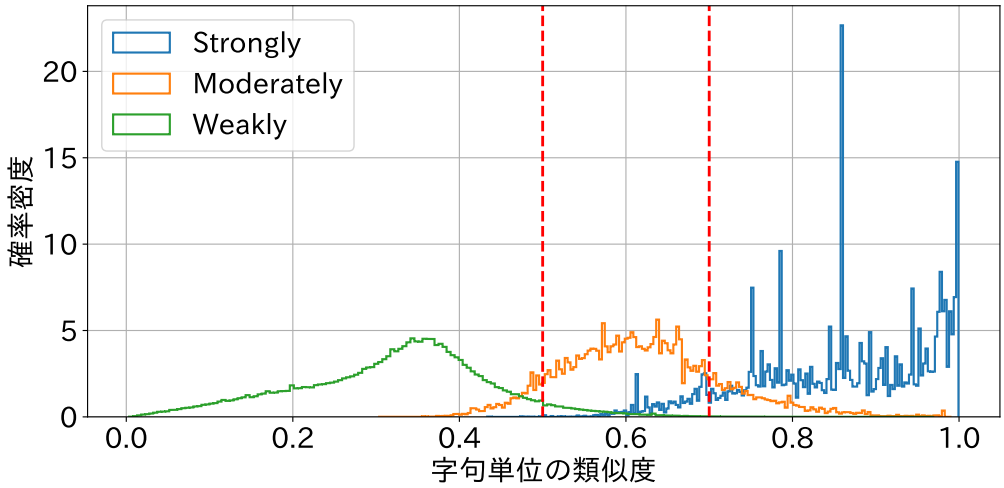

図 5: 字句単位の類似度における行一致率の分布

注： 赤い破線は，BigCloneBench における Weakly-Type3 と Moderately-Type3 の閾値である 0.5 と，Moderately-Type3 と Strongly-Type3 の閾値である 0.7 を表している

たがって，本研究では類似度という数値的な閾値判定に依存せず，AST の構造的特徴に基づいた新たな Type 分類の定義を導入する．AST に基づいた分類基準を設けることで，解析の単位粒度に結果が左右されることなく，プログラムの論理的な構造に根ざした客観的な分類が可能となる．

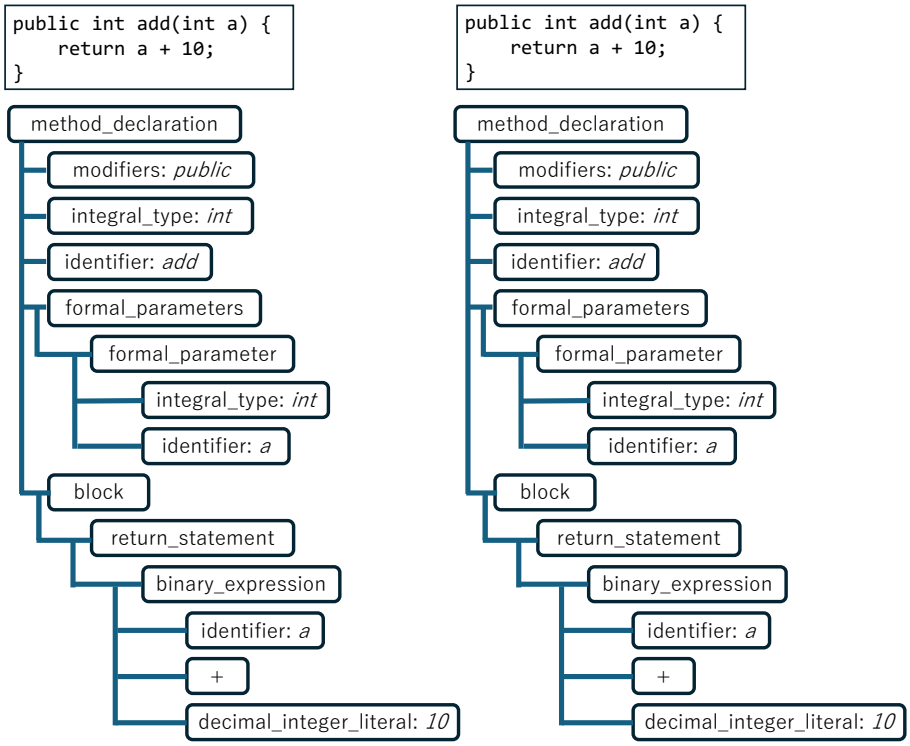

図 6: Type1 の例

4 提案手法

本章では，本研究で提案するコードクローン検出の後処理としての統一的なラベリング手法について述べる．提案手法では，クローンペアを入力し，そのクローンペアがどの Type に分類されるかのラベルを出力とする．

4.1 分類基準

本研究では AST に基づいて新たな分類基準を定義した．定義は以下の通りである．すでにコードクローンとして検出されたコードが対象で，与えられたクローンペアをそれぞれ AST に変換し，二つの AST の差に応じて分類を行う．

Type1 AST が完全一致するコードクローン

Type2 葉ノードを除いて AST が一致するコードクローン

Type3 葉ノードに加え，単文以下のノードを除いて AST が一致するコードクローン

Type4 上記に分類されないコードクローン

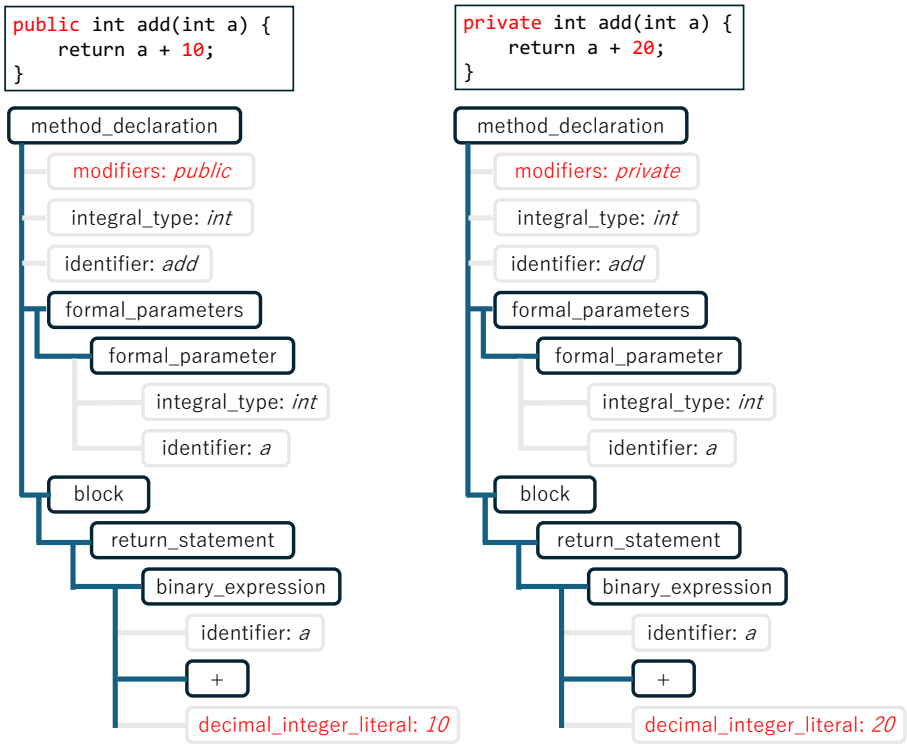

図 7: Type2 の例

Type1〜3 の分類基準について詳細を述べる．

Type1 では葉ノードの持つ値を含む AST 全体が完全に一致する (図 6)．AST の木の形状と，斜体で示された値が完全に同一である場合，Type1 と判断される．

Type2 では葉ノードを除いた AST が一致する．本手法において，演算子は葉ノードとして扱わない．この理由は，計算処理の種類を厳密に区別するためである．Type2 のコードクローンは識別子やリテラルの置換を許容するが，演算子の変更は計算手順そのものの変容を意味する．例えば，$a+b$ と $a \times b$ は AST の形状は同一であるが，計算アルゴリズムとしては異なり，演算子が他の葉ノードと同様に除かれ比較されると，これらは同一の Type2 クローンとして判定されてしまう．したがって，本手法では演算子を識別子やリテラル値と区別して扱い葉ノードから除いた．図 7 に示した例では，赤く示した部分がソースコード上で異なる部分である．葉ノードとして除かれるノードを灰色で，比較対象として残るノードを黒で示している．黒く示されたノードを持つ AST の形状が同一であるため，Type2 と判断される．

Type3 では単文以下のノードを除いた AST が一致する．Java において，単文として定義される文を表 4 に示す．図 8 に示した例では，葉ノードに加え，return_statement ノード，expression_statement ノードが単文にあたるため，これらのノード以下のノードが除かれる．

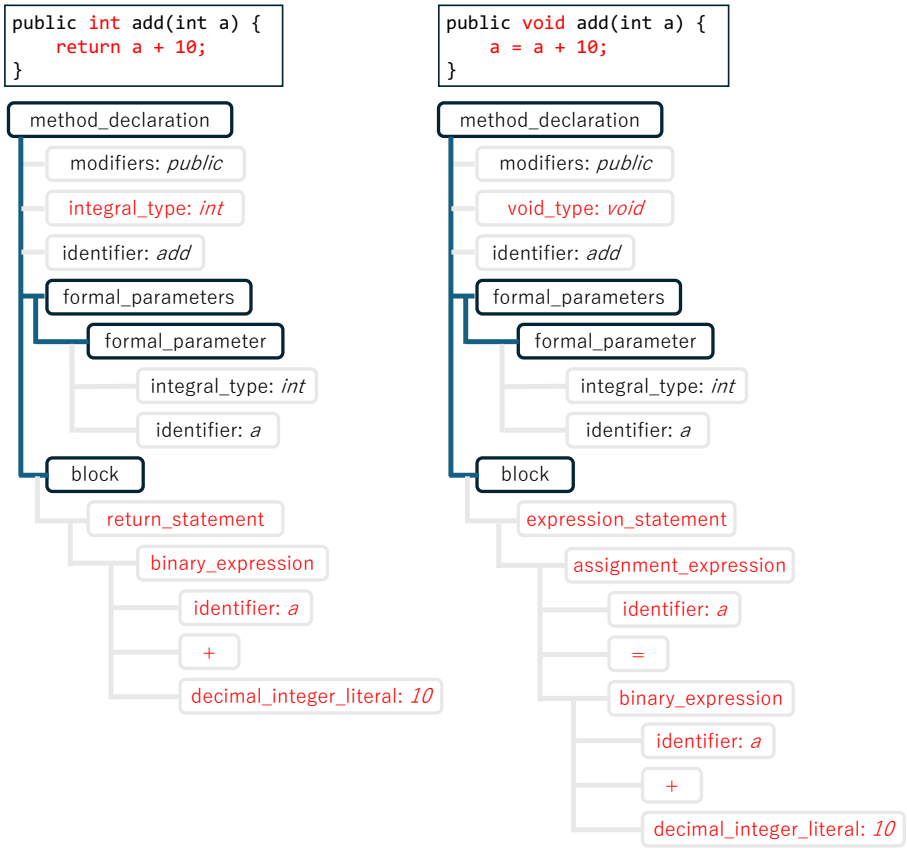

図 8: Type3 の例

黒く示されたノードを持つ AST の形状が同一であるため，Type3 と判断される．

表 4: Java において単文として定義される文の例

単文	例
式文	x = a + b; System.out.println(s);
変数宣言文	int count = 0;
return 文	return result;
throw 文	throw new IllegalArgumentException();
break 文	break;
continue 文	continue;
yield 文	yield 100;
assert 文	assert list != null;

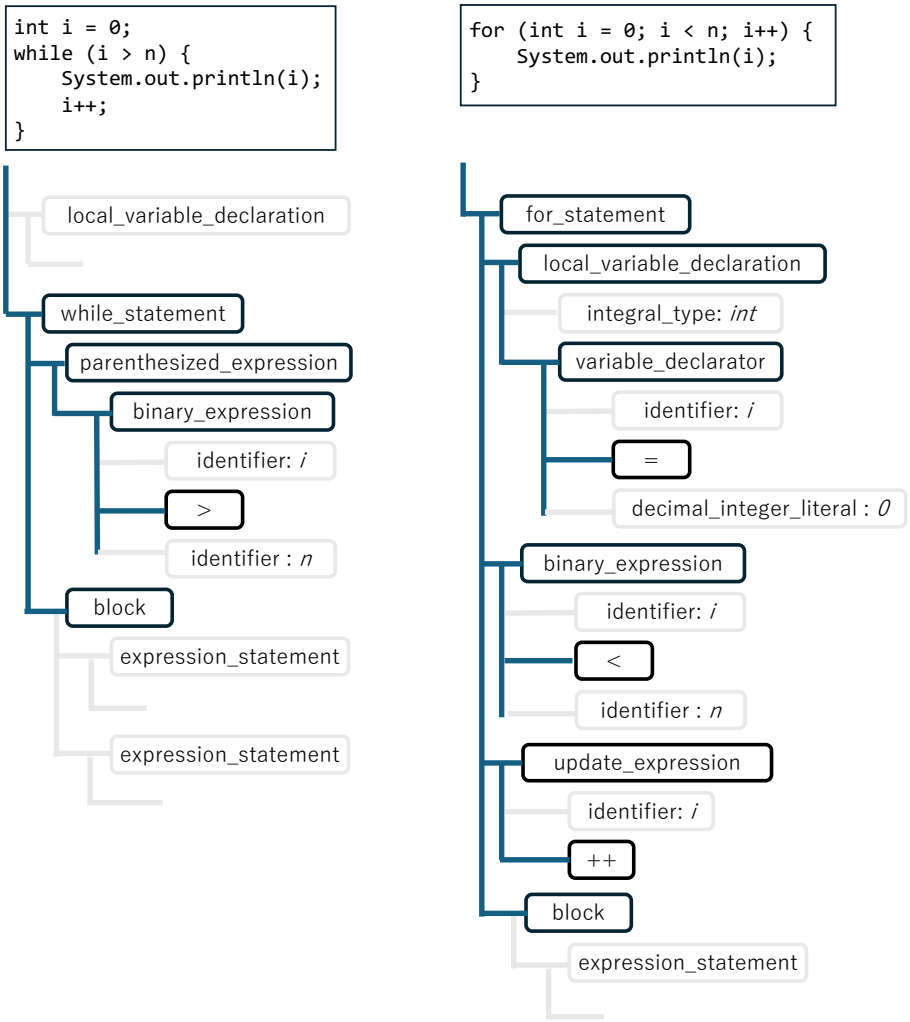

図 9: 制御構造を持つコードの例

提案手法の分類定義における if 文，while 文，for 文等の制御構造を持つソースコードの扱いについて述べる．図 9 に同じ機能を持つ Java プログラムと対応する AST を示す．簡単のため，一部のノードを省略している．この二つのコードでは，同じ処理を行うものの，一方は while 文を，他方は for 文を用いて処理を行っている．AST において，葉ノードおよび単文以下のノードとして除かれるノードを灰色で示す．黒く示すノードを持つ AST の形状が異なるため，Type4 に分類される．ここで，while 文および for 文の条件式内の文について，単文として扱われないため，葉ノード以外のノードは除かれることなく AST の形状の比較が行われる．

5　評価実験

本章では，評価実験について述べる．本実験は，BigCloneBench が行単位での構文的類似度に基づいて分類した場合と，提案手法で分類した場合のどちらがより人間の感覚に近いのかを評価することを目的としている．特に，分類の基準が明確でない Type3 および Type4 のコードクローンを対象とした．

5.1　実験対象データセット

実験対象データセットとして BigCloneBench を用いた．コードクローンの研究において，Strongly-Type3 は Type3 クローンとして，Weakly-Type3 クローンは Type4 クローンとして用いられている [3, 19, 20, 23, 33, 37, 39]．提案手法を用いて BigCloneBench に含まれるクローンペアに対して分類を行った．分類には，AppleM3 チップ (8 コア CPU)，メモリ 24GB の MacBookAir を使用した．Java のバージョンは 25.0.1 を利用した．提案手法を用いて BigCloneBench 内のクローンペアの分類を行った結果を表 5 に示す．分類に要した総時間は 35 時間 49 分であった．

5.2　実験手順

被験者はコンピュータサイエンスを専攻する博士前期課程の大学院生 6 人であり，いずれも Java を用いたプログラミングの経験がある．以下のいずれかの基準を満たすクローンペアを本研究の調査対象とした．ペアを構成する二つのメソッドの合計トークン数が少ない順にそれぞれ 50 個，合計 100 個のクローンペアを対象とした．同一のメソッドが複数のペアに含まれる場合は，そのうち一組のみを抽出対象とした．

表 5: 提案手法による BigCloneBench の分類結果

		BigCloneBench					
		Type1	Type2	Strongly-Type3	Moderately-Type3	Weakly-Type3	合計
提案手法	Type1	48,116	0	0	0	0	48,116
	Type2	0	4,234	3,637	12	1	7,884
	Type3	0	0	7,984	13,302	46,973	68,259
	Type4	0	0	10,345	74,992	8,403,230	8,488,567
	合計	48,116	4,234	21,966	88,306	8,450,204	8,612,826

- BigCloneBench で Strongly-Type3 に分類されるクローンペアのうち提案手法で Type4 と分類されたクローンペア
- BigCloneBench で Weakly-Type3 に分類されるクローンペアのうち提案手法で Type3 と分類されたクローンペア

被験者には，事前に，2.1 節に記述したコードクローン分類の定義を提示し，100 個のクローンペアがどちらに分類されるべきかを判断してもらった．また，判断に迷った場合にその理由と最終的な判断の理由についても回答してもらった．

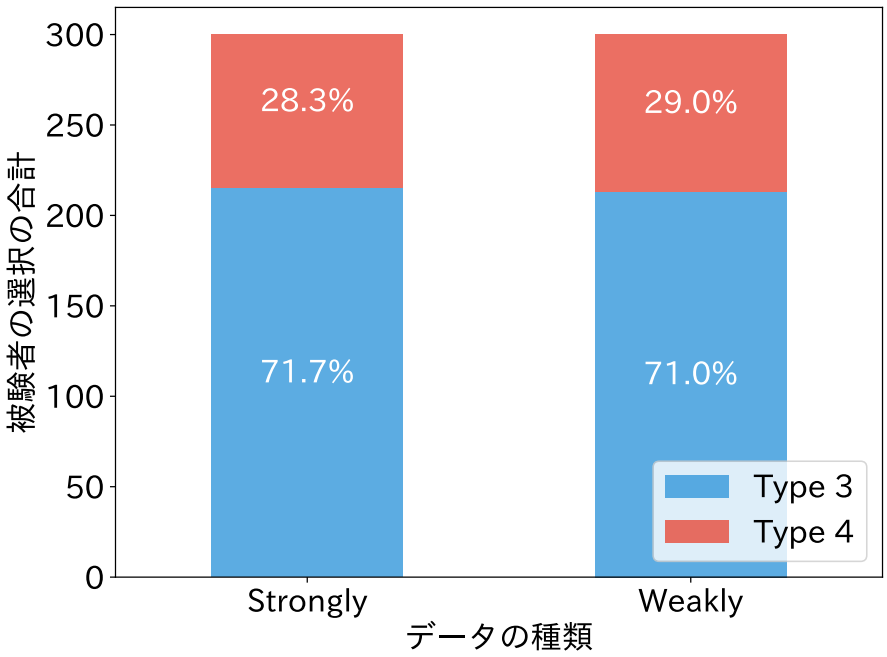

図 10: 評価実験の結果：被験者の判断の割合

注： Strongly は，BigCloneBench によって Strongly-Type3 と分類されるクローンペアのうち提案手法で Type4 と分類されたクローンペアを示し，Weakly は，BigCloneBench によって Weakly-Type3 と分類されるクローンペアのうち提案手法で Type3 と分類されたクローンペアを示す．

6 実験結果と考察

本章では評価実験の結果とそれに対する考察を述べる．

6.1 実験結果

結果を図 10 に示す．BigCloneBench で Strongly-Type3 に分類されるクローンペアのうち提案手法で Type4 と分類されたクローンペアは，28.3%が Type4 と判断され，提案手法による分類が人間の感覚に反するという結果になった．一方，Weakly-Type3 に分類されるクローンペアのうち提案手法で Type3 と分類されたクローンペアは，71.0%が Type3 と判断され，提案手法による分類が人間の感覚に近いという結果になった．また，全体的に Type3 と判断されたクローンペアの数が多かった．

提案手法の判定と一致した被験者数ごとのクローンペア分布を図 11 に，その具体的な内訳となるペア数を表 6 に示す．図 11 における分布の詳細は，表 6 の数値に対応している．Strongly-Type3 に分類されるクローンペアのうち提案手法で Type4 と分類されたクローンペアは，分布の偏りから，判断に迷いが少ないことがわかる．一方，Weakly-Type3 に分類

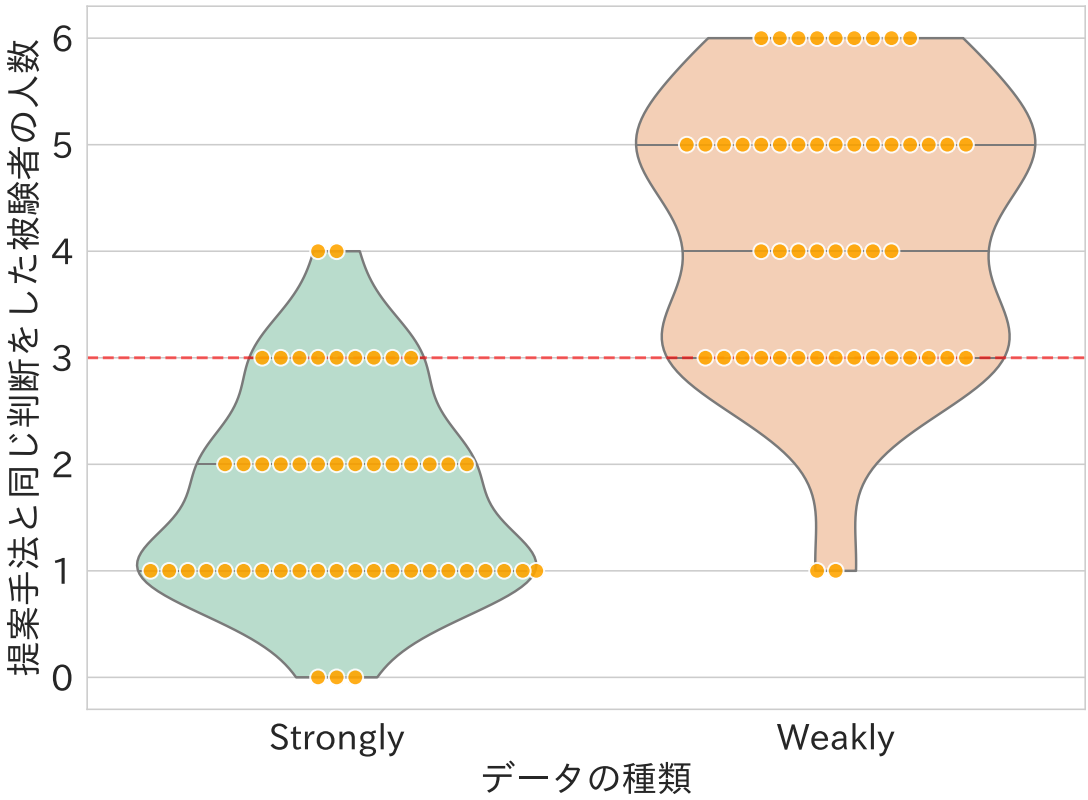

図 11: 評価実験の結果：被験者の分布

注： Strongly は，BigCloneBench によって Strongly-Type3 と分類されるクローンペアのうち提案手法で Type4 と分類されたクローンペアを示し，Weakly は，BigCloneBench によって Weakly-Type3 と分類されるクローンペアのうち提案手法で Type3 と分類されたクローンペアを示す．

されるクローンペアのうち提案手法で Type3 と分類されたクローンペアは，BigCloneBench の分類を選んだのが 3 人，提案手法の分類を選んだのが 3 人である場合，つまり意見が割れたクローンペアが多く存在することから，判断の迷いがあったと考える．

Strongly-Type3 に分類されるクローンペアのうち提案手法で Type4 と分類されたクローンペア，Weakly-Type3 に分類されるクローンペアのうち提案手法で Type3 と分類されたク

表 6: 提案手法と一致する判断を行った人数の分布

	0 人	1 人	2 人	3 人	4 人	5 人	6 人
Strongly	3	22	14	9	2	0	0
Weakly	0	2	0	15	8	16	9

ローンペア，それぞれについて，被験者による判断と提案手法による分類が一致した場合，異なった場合のコードを確認した．以降に詳細を述べる．

6.1.1 Strongly-Type3 に分類されるクローンペアのうち提案手法で Type4 と分類されたクローンペア

本項では，BigCloneBench によって Strongly-Type3 と分類されたものの，提案手法では Type4 と判定されたクローンペアについて詳細を述べる．実験の結果，提案手法の分類は人間の感覚に反するという結果になった．表 6 に示されているように，被験者 6 人のうち，6 人全員が Type4 と判断したクローンペアは存在せず，最大で 4 人が Type4 と判断したクローンペアは，50 ペア中 2 ペアに留まった．一方で，被験者全員が Type3 と判断したクローンペアは 3 ペア存在した．それぞれのクローンペアについて例を示して詳細を述べる．

ソースコード 1: 被験者 6 人中 4 人が提案手法と同じ Type4 と判断したクローンペアの例

```
public static final void randomShuffle (int [] v, Random r) {
    while (-- n > 0) {
        int k = r.nextInt (n + 1);
        int temp = v [n];
        v [n] = v [k];
        v [k] = temp;
    }
}
```

```
public static synchronized void shuffle (int [] anArray) {
    int n = anArray.length;
    for (int i = n - 1; i >= 1; i --) {
        int j = randomSource.nextInt (i + 1);
        int temp = anArray [j];
        anArray [j] = anArray [i];
        anArray [i] = temp;
    }
}
```

ソースコード 1 は被験者 6 人中 4 人が提案手法と同じ Type4 と判断したクローンペアである．このクローンペアでは，一方が while 文，他方が for 文という異なる制御構造を含んでいるため，AST の構造的差異に基づき，提案手法では Type4 と判定された．この例のように制御構造そのものが異なるクローンペアに対しては，提案手法による分類が被験者の主観評価に近いことが確認された．

ソースコード 2: 被験者全員が提案手法とは異なる Type3 と判断したクローンペアの例 (while 文の条件式の違い)

```
1 private void writeFile (FileInputStream inFile, FileOutputStream
      outFile) throws IOException {
2     byte [] buf = new byte [2048];
3     int read;
4     while ((read = inFile.read (buf)) > 0 && !
          stopped) outFile.write (buf, 0, read);
5
6     inFile.close ();
7 }
```

```
1 private void writeFile (FileInputStream inFile, FileOutputStream
      outFile) throws IOException {
2     byte [] buf = new byte [2048];
3     int read;
4     while ((read = inFile.read (buf)) > 0) outFile.write (buf, 0, read);
5
6     inFile.close ();
7 }
```

次に，被験者全員が提案手法と異なる Type3 と判断したクローンペアを示す．ソースコード 2 は，赤く示された while 文の条件式において AST に差異が生じたため，提案手法では Type4 と判断された．しかし，被験者の判断では，条件式のみの変更は，文の変更に相当し，Type3 の定義の範疇に収まるとして Type3 と判断していた．現在の定義では，while 文，if 文，for 文等の制御構造において，条件式内の文は単文に含まれない．そのため，内包される条件式の AST 構造が異なる場合は一律に Type4 と判定される．単文として定義される文に制御構造内の条件式を含めることで，より人間の直感に近い分類が可能になる．

ソースコード 3: 被験者全員が提案手法と異なる Type3 と判断したクローンペアの例 (仮引数の違い)

```
public static final void copy (InputStream is, OutputStream
    os) throws IOException {
     try {
         IOUtils.copy (is, os);
     } finally {
         IOUtils.closeQuietly (is);
         IOUtils.closeQuietly (os);
     }
}
```

```
public static void readFile (FOUserAgent ua, String uri, OutputStream
    output) throws IOException {
     InputStream in = getURLInputStream (ua, uri);
     try {
         IOUtils.copy (in, output);
     } finally {
         IOUtils.closeQuietly (in);
     }
}
```

ソースコード 3 は，メソッドの仮引数が異なっていたため，AST に差異が生じ，提案手法では Type4 と判断された．しかし，被験者は，仮引数の違いも Type3 における文の変更の一部であると判断した．引数の数や順序の変更についても，条件式と同様に，単文として定義される文に仮引数を含めることで，より人間の直感に近い分類が可能になる．

6.1.2 Weakly-Type3 に分類されるクローンペアのうち提案手法で Type3 と分類されたクローンペア

本項では，BigCloneBench で Weakly-Type3 に分類されるペアのうち，提案手法によって Type3 と判定されたケースについて述べる．実験の結果，提案手法の分類が人間の感覚に近いという結果になった．具体的には，被験者 6 人全員が Type3 と判断したペアが 9 ペア存在した．一方で，被験者全員が Type4 と判断したペアは存在せず，最大で 5 人が Type4 と判断したペアが 50 ペア中 2 ペア確認された．以下にその詳細を述べる．

ソースコード 4: 被験者全員が提案手法と同じ Type3 と判断したクローンペアの例

```
private byte [] generateHash (String s) throws
    NoSuchAlgorithmException {
     MessageDigest md = MessageDigest.getInstance ("MD5");
     md.update (s.getBytes ());
     return md.digest ();
}
```

```
private static byte [] finalizeStringHash (String loginHash) throws
     NoSuchAlgorithmException {
     MessageDigest md5Hasher;
     md5Hasher = MessageDigest.getInstance ("MD5");
     md5Hasher.update (loginHash.getBytes ());
     md5Hasher.update (LOGIN_FINAL_SALT);
     return md5Hasher.digest ();
}
```

ソースコード 4 は，被験者全員が提案手法と同じ Type3 と判断したクローンペアである．このクローンペアでは，互いに共通する文を多く含んでおり，被験者はその内容の類似性から Type3 と判断したと考えられる．しかし，BigCloneBench では Weakly-Type3 として扱われ，人間の感覚と反する．このように，BigCloneBench の分類が人間の直感と反するケースが存在することが確認できた．これは，BigCloneBench の分類基準である行単位の類似度という指標ではなく，AST を用いることで，プログラムの論理的な骨組みを直接評価できることが，人間の感覚に近い判定に寄与したと言える．提案手法の分類基準が，既存の類似度に基づくよりもコードクローンの実態をより正確に反映できる可能性を示している．

ソースコード 5: 被験者 6 人中 5 人が提案手法と異なる Type4 と判断したクローンペアの例

```
private void javaToHtml (File source, File destination) throws
    IOException {
     Reader reader = new FileReader (source);
     Writer writer = new FileWriter (destination);
     JavaUtils.writeJava (reader, writer);
     writer.flush ();
     writer.close ();
}
```

```
public static void copy (InputStream stream, OutputStream ostream)
    throws IOException {
     IOUtils.copy (stream, ostream, false);
}
```

一方，ソースコード 5 は，被験者 6 人のうち 5 人が提案手法と異なる Type4 と判断したクローンペアである．このクローンペアでは，機能は同一であるが，構文的に異なるとして被験者は Type4 と判断していた．提案手法は，それぞれのメソッド以下の文を全て単文として扱ったために，単文以下のノードを除いた後の AST の形状が同一となり，Type3 と判断され，人間の判断との乖離が発生していた．

6.2 考察

6.3 提案手法の分類定義

6.1.1 項の結果から，提案手法の厳密な構造比較は Type4 の特定に寄与する一方で，人間が，2.1 節に示した Type3 の定義として許容する範囲のコードクローンを Type4 と判定する可能性が示唆された．これらの知見に基づき，特定の構文要素に対する判定基準を柔軟に調整することが，今後の精度向上における重要な課題である．

6.1.2 項の例に対し，提案手法が Type3 と判定した要因は，単文以下のノードを除いた後の AST の形状が同一であったためである．しかし，極端な例として，メソッド本体が一つの単文のみで構成されるコードと，大量の単文を含むコードであっても，それらが同じ制御構造の中に配置されていれば，本手法では AST が同一であると見なされ，Type3 に分類されてしまう．実験の結果，このような規模の著しい乖離がある場合に Type3 と分類することは，人間の直感から離れる原因になると示唆された．したがって，単文以下のノードを除く際，何らかの制約を設ける必要があると考えられる．具体的な改善策として，次の 2 点が考えられる．

- 単文の数による量的制約：追加，削除または変更されたとする単文の数に閾値を設け，ソースコード全体の文の量が著しく異なる場合には，Type3 判定しないように制限する方法である．例えば，挿入された文の数が元の構造に対して一定の割合を超える場合，それを Type3 ではなく Type4 と判定する．

- 文の種類や文脈に応じたフィルタリング：ソースコード中の全ての単文を一律に取り除くのではなく，文の内部構造 (例：メソッド呼び出しの有無) に応じて，その単文を除くか，保持するかを判定する方法である．変更を許容する要素を文脈に応じて動的に選択することで，単なる変数への代入といった軽微な実装の差異は除き，外部ライブラリの呼び出しを伴うような重要なコードの変更を保持することが可能となる．

これらの変更により，より人間の直感に近い分類が可能になると考えられる．

6.3.1 被験者ごとの傾向

本実験における被験者ごとの傾向を調査するため，回答の分析を行なった．被験者ごとの Type3 と判断したペア数を表 7 に，被験者間の回答の一致率を表 8 に示す．

表 8 の被験者間の一致率に基づき，被験者を二つのグループに分類した．具体的には，P1，P2，P3 からなるグループ A と P4，P5，P6 からなるグループ B である．グループ A 内では，P1 と P2 の間で 70%以上の一致率が確認され，P3 も P1 および P2 とそれぞれ 60%以上

表 7: 被験者ごとの Type3 と判断したペア数 (100 ペア中)

	P1	P2	P3	P4	P5	P6
ペア数	75	66	53	78	82	74

表 8: 被験者間における実験回答の一致率

	P1	P2	P3	P4	P5	P6
P1	–	**75%**	64%	55%	63%	57%
P2	**75%**	–	63%	46%	52%	54%
P3	64%	63%	–	39%	45%	51%
P4	55%	46%	39%	–	**78%**	**76%**
P5	63%	52%	45%	**78%**	–	**74%**
P6	57%	54%	51%	**76%**	**74%**	–

注： セルの背景色は、一致率が 70%以上のものを濃色、60%以上のものを淡色で示している。

の一致率を示した．対してグループ B では，P4 から P6 の全被験者間でそれぞれ 70%以上の高い一致率を記録した．これら二つのグループについて回答の判断理由を分析した結果，両者の間でクローン判定の基準が明確に異なることが判明した．具体的には，処理機能の同一性を重視する立場と，評価の際に構文構造の差異を重視する立場の違いである．グループ A は，2.1 節の Type4 の定義に基づき，同一の処理を行う場合に Type4 と判断し，行わない場合に Type3 と判断していた．一方，グループ B は，2.1 節の Type3 の定義に基づき，クローンペア間で異なる部分が，文の挿入，削除，変更に留まる場合には Type3 と判断し，それ以上の構文の変更が見られる場合に Type4 と判断していた．このような被験者間における判断基準の不一致が，同一のクローンペアに対して異なる分類が行われる要因になったと考えられる．以上の結果は，コードクローンにおける Type3 と Type4 の定義の曖昧さを示唆しており，本研究が目的とする統一的なラベリング手法の必要性が改めて確認された．

7　妥当性への脅威

本章では，本研究における内部妥当性，外部妥当性，結論妥当性への脅威について述べる．

7.1　内部妥当性

内部妥当性への脅威としては，提案手法の実装および解析基盤として採用した Tree-sitter の精度が挙げられる．パーサによる構文解析に誤りが含まれる場合，正しくラベリングが行われない可能性がある．これに対し，本研究では解析対象の各文に対し，生成された AST ノードが意図した構文要素と一致しているかを事前にテストを用いて検証し，実装の正確性を確認している．

また，評価実験における被験者の主観性も脅威となり得る．考察で述べた通り，被験者の判断基準が，構文の変更を重視する立場と機能の同一性を重視する立場の二つに類別されることが確認された．実験前の教示においてコードクローンの定義を説明しているが，被験者個人の経験や直感に起因する判断のブレを完全に排除することは困難である．この影響を軽減するため，本研究では被験者間の回答一致率を算出し，主観による偏りを定量的に評価している．

7.2　外部妥当性

外部妥当性への脅威は，結果の汎用性に関するものである．本研究では評価対象を Java 言語に限定しており，他のプログラミング言語（Python や C++ 等）における有効性は未検証である．しかし，採用した Tree-sitter は多言語対応の解析基盤であり，他言語への拡張は比較的容易であると考えられる．Java 以外の言語への拡張と，拡張した提案手法の評価は今後の課題である．

また，サンプリングにおいてトークン数の合計が少ないペアを優先的に抽出したため，大規模なメソッドにおける精度については更なる検証が必要である．さらに，本研究では BigCloneBench のみを評価対象としたが, 特定のドメインや特定の開発手法（テストコード等）におけるクローン特性が，本手法の分類精度に影響を与える可能性がある．BigCloneBench 以外のデータセットを用いた提案手法の評価や，クローン特性による影響の調査は今後の課題である．

7.3　結論妥当性

結論妥当性への脅威として，評価実験におけるサンプルサイズが挙げられる．被験者 6 名という規模は，統計的に十分な一般性を担保しているとは言い難い．得られた知見（グルー

プ分けや判断傾向）が開発者全体の一般的な傾向を代表していると断定するには，今後，より多くの被験者を対象とした大規模な調査による検証が必要である．

8　おわりに

本研究では，複数のコードクローン検出ツールから得られる結果を統合し，一貫性のあるデータセットを構築するため，コードクローン検出の後処理としての統一的ラベリング手法の提案を行なった．

提案手法の妥当性を評価するために実施した被験者実験では，以下の知見が得られた．

既存の行単位の類似度ではType4と判定されていたクローンペアに対し，提案手法がType3と判定したケースでは，人間の感覚もType3を支持する傾向にあった．一方で，提案手法がType4と判定したクローンペアを人間がType3と見なすケースも確認された．

以上の結果から，類似度という単一の指標に頼るのではなく，構造的なアプローチによって統一的なラベルを付与する本手法の有用性が確認された．同時に，人間の感覚により近づけるためには，分類定義のさらなる見直しが必要であるという結論に至った．また，被験者ごとの傾向から，本研究が目的とする統一的なラベリング手法の必要性が改めて確認された．

今後の展望として，次の三つが考えられる．

一つ目は，提案手法における分類基準の最適化である．実験で明らかになった人間との感覚のズレを解消するため，文の種類や文脈に応じた動的なフィルタリングを導入することが考えられる．

二つ目は，提案手法を用いて統一的なラベリングが行われたデータセットの作成である．複数の既存検出ツールから得られた結果に対し，本手法を用いて統一的なラベリングを行うことで，ノイズの少ない高品質なデータセットの整備が可能となる．このようなデータセットをLLMの学習データとして活用し，クローン情報の付与がLLMのコード生成精度や理解能力の向上にどの程度寄与するかを詳細に検証することが考えられる．

三つ目は，Java以外のプログラミング言語への提案手法の拡張である．本研究で採用した解析ライブラリであるTree-sitterは多言語共通のインターフェースを提供しており，PythonやC++といった他言語への適用が容易であるという特長を持つ．したがって，今後は対象言語を拡大し，異なる言語仕様においても本研究と同様のラベリング精度および一貫性が得られるかについて，評価実験を通じて手法の汎用性を実証していく必要がある．

謝辞

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肥後芳樹教授には，研究の方向性の検討から研究活動の直接の御指導，論文の執筆, 発表資料の作成に至るまで，研究活動全ての場面で手厚く御指導，御助言を賜りました．肥後芳樹教授の適切な御指導により，研究活動を行え，研究会での発表や論文の執筆をすることができました．心より深く感謝申し上げます．

I would like to express my deepest gratitude to Professor Kula Gaikovina Raula from the Department of Computer Science, Graduate School of Information Science and Technology, Osaka University, for his invaluable advice during presentations within our laboratory.

I would like to express my deepest gratitude to Assistant Professor Olivier Nourry from the Department of Computer Science, Graduate School of Information Science and Technology, Osaka University, for his invaluable advice during presentations within our laboratory.

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松下誠准教授には，研究室内の発表機会にて貴重なご意見，ご助言賜りました．

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楠本真二教授には，研究室内の発表機会にて貴重なご意見，ご助言賜りました．

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柗本真佑准教授には，研究室内の発表機会にて貴重なご意見，ご助言賜りました．

事務職員軽部瑞穂氏，宮崎貴羅氏は研究活動に関わらず，多くのサポートをしていただきました．

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肥後研究室のメンバーには，被験者実験の参加や普段の日常生活において，様々な場面で助けられました．心より深く感謝申し上げます．

最後に，長年にわたり私の学業を温かく見守り，多大なる支援を賜りました家族に深く感謝いたします．

参考文献

[1] Farouq Al-Omari, Chanchal Kumar Roy, and Tonghao Chen. SemanticCloneBench: A semantic code clone benchmark using crowd-source knowledge. In *Proceedings of the 14th IEEE International Workshop on Software Clones (IWSC)*, pp. 57–63, 2020.

[2] Ajmain I. Alam, Palash R. Roy, Farouq Al-Omari, Chanchal K. Roy, Banani Roy, and Kevin A. Schneider. GPTCloneBench: A comprehensive benchmark of semantic clones and cross-language clones using GPT-3 model and SemanticCloneBench. In *Proceedings of the 39th IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pp. 1–13, 2023.

[3] Ajmain Alam, Palash Roy, Farouq Al-omari, Chanchal Roy, Banani Roy, and Kevin Schneider. Are classical clone detectors good enough for the AI era? In *Proceedings of the 41st IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pp. 295–307, 2025.

[4] Wolfram Amme, Thomas S. Heinze, and André Schäfer. You look so different: Finding structural clones and subclones in Java source code. In *Proceedings of the 37th IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pp. 70–80, 2021.

[5] Nghi D. Q. Bui, Yijun Yu, and Lingxiao Jiang. InferCode: Self-supervised learning of code representations by predicting subtrees. In *Proceedings of the 43rd IEEE/ACM International Conference on Software Engineering (ICSE)*, pp. 1186–1197, 2021.

[6] Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc, Matias Martinez, and Martin Monperrus. Fine-grained and accurate source code differencing. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering (ASE)*, pp. 313–324, 2014.

[7] Yuanrui Fan, Xin Xia, David Lo, Ahmed E. Hassan, Yuan Wang, and Shanping Li. A differential testing approach for evaluating abstract syntax tree mapping algorithms. In *Proceedings of the 43rd International Conference on Software Engineering (ICSE)*, pp. 1174–1185, 2021.

[8] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Ting Liu, Daxin Jiang, and Ming Zhou. CodeBERT: A pre-trained

model for programming and natural languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pp. 1536–1547, 2020.

[9] Nils Göde and Rainer Koschke. Incremental clone detection. In *Proceedings of the 13th European Conference on Software Maintenance and Reengineering (CSMR)*, pp. 219–228, 2009.

[10] Yoshiki Higo. Dataset of functionally equivalent Java methods and its application to evaluating clone detection tools. *IEICE Transactions on Information and Systems*, Vol. E107.D, No. 6, pp. 751–760, 2024.

[11] Yoshiki Higo, Shinsuke Matsumoto, Shinji Kusumoto, and Kazuya Yasuda. Constructing dataset of functionally equivalent Java methods using automated test generation techniques. In *Proceedings of the 19th IEEE/ACM International Conference on Mining Software Repositories (MSR)*, pp. 682–686, 2022.

[12] James W. Hunt and Thomas G. Szymanski. A fast algorithm for computing longest common subsequences. *Communications of the ACM*, Vol. 20, No. 5, pp. 350–353, 1977.

[13] Ryutaro Inoue and Yoshiki Higo. Improving accuracy of LLM-based code clone detection using functionally equivalent methods. In *Proceedings of the 22nd IEEE/ACIS International Conference on Software Engineering Research, Management and Applications (SERA)*, pp. 24–27, 2024.

[14] Lingxiao Jiang, Ghassan Misherghi, Zhendong Su, and Stephane Glondu. DECKARD: Scalable and accurate tree-based detection of code clones. In *Proceedings of the 29th International Conference on Software Engineering (ICSE'07)*, pp. 96–105, 2007.

[15] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. CCFinder: A multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering*, Vol. 28, No. 7, pp. 654–670, 2002.

[16] Jens Krinke and Chaiyong Ragkhitwetsagul. BigCloneBench considered harmful for machine learning. In *Proceedings of the 16th IEEE International Workshop on Software Clones (IWSC)*, pp. 1–7, 2022.

[17] Jens Krinke and Chaiyong Ragkhitwetsagul. How the misuse of a dataset harmed semantic clone detection. *arXiv preprint arXiv:2505.04311*, 2025.

[18] Daniel E. Krutz and Wei Le. A code clone oracle. In *Proceedings of the 11th Working Conference on Mining Software Repositories (MSR)*, pp. 388–391, 2014.

[19] Chunguang Li, Jessada Konpang, Adisorn Sirikham, and Yan Wang. Nuanced code clone detection through LLM-based code revision and AST graph modeling. *IEEE Access*, Vol. 13, pp. 191024–191036, 2025.

[20] Liuqing Li, He Feng, Wenjie Zhuang, Na Meng, and Barbara Ryder. CCLearner: A deep learning-based clone detection approach. In *Proceedings of the 33rd IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pp. 249–260, 2017.

[21] Niccolò Marastoni, Andrea Continella, Davide Quarta, Stefano Zanero, and Mila Dalla Preda. GroupDroid: Automatically grouping mobile malware by extracting code similarities. In *Proceedings of the 7th ACM Conference on Data and Application Security and Privacy (CODASPY)*, pp. 1–12, 2017.

[22] Tasuku Nakagawa, Yoshiki Higo, and Shinji Kusumoto. NIL: large-scale detection of large-variance clones. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pp. 830–841, 2021.

[23] Subroto Nag Pinku, Debajyoti Mondal, and Chanchal K. Roy. On the use of deep learning models for semantic clone detection. In *Proceedings of the 40th IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pp. 512–524, 2024.

[24] Chanchal Kumar Roy and James Ronald Cordy. A survey on software clone detection research. Technical Report TR 2007-541, School of Computing, Queen's University, 2007.

[25] Chanchal Kumar Roy and James Ronald Cordy. NICAD: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization. In *Proceedings of the 16th IEEE International Conference on Program Comprehension (ICPC)*, pp. 172–181, 2008.

[26] Chanchal Kumar Roy, James Ronald Cordy, and Rainer Koschke. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Science of Computer Programming*, Vol. 74, No. 7, pp. 470–495, 2009.

[27] Vaibhav Saini, Farima Farmahinifarahani, Yadong Lu, Pierre Baldi, and Cristina V. Lopes. Oreo: detection of clones in the twilight zone. In *Proceedings of the 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pp. 354–365, 2018.

[28] Hitesh Sajnani, Vaibhav Saini, Jeffrey Svajlenko, Chanchal K. Roy, and Cristina V. Lopes. SourcererCC: Scaling code clone detection to big-code. In *Proceedings of the 38th IEEE/ACM International Conference on Software Engineering (ICSE)*, pp. 1157–1168, 2016.

[29] Kota Someya, Lei Chen, Michael J. Decker, and Shinpei Hayashi. How much can a behavior-preserving changeset be decomposed into refactoring operations? In *Proceedings of the 41st IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pp. 809–814, 2025.

[30] Jeffrey Svajlenko, Judith Fatema Islam, Iman Keivanloo, Chanchal Kumar Roy, and Mohammad Mamun Mia. Towards a big data curated benchmark of inter-project code clones. In *Proceedings of the 30th IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pp. 476–480, 2014.

[31] Nikolaos Tsantalis, Matin Mansouri, Laleh Eshkevari, Davood Mazinanian, and Danny Dig. Accurate and efficient refactoring detection in commit history. In *Proceedings of the 40th IEEE/ACM International Conference on Software Engineering (ICSE)*, pp. 483–494, 2018.

[32] Robert A. Wagner and Michael J. Fischer. The string-to-string correction problem. *Journal of the ACM*, Vol. 21, No. 1, pp. 168–173, 1974.

[33] Yuekun Wang, Yuhang Ye, Yueming Wu, Weiwei Zhang, Yinxing Xue, and Yang Liu. Comparison and evaluation of clone detection techniques with different code representations. In *Proceedings of the 45th IEEE/ACM International Conference on Software Engineering (ICSE)*, pp. 332–344, 2023.

[34] Ming Wu, Pengcheng Wang, Kangqi Yin, Haoyu Cheng, Yun Xu, and Chanchal K. Roy. LVMapper: A large-variance clone detector using sequencing alignment approach. *IEEE Access*, Vol. 8, pp. 27986–27997, 2020.

[35] Yanming Yang, Zhilei Ren, Xin Chen, and He Jiang. Structural function based code clone detection using a new hybrid technique. In *Proceedings of the 42nd IEEE Annual Computer Software and Applications Conference (COMPSAC)*, pp. 286–291, 2018.

[36] Hao Yu, Xing Hu, Ge Li, Ying Li, Qianxiang Wang, and Tao Xie. Assessing and improving an evaluation dataset for detecting semantic code clones via deep learning. *ACM Transactions on Software Engineering and Methodology*, Vol. 31, No. 4, pp. 62:1–62:25, 2022.

[37] Tianchen Yu, Li Yuan, Liannan Lin, and Hongkui He. A multiple representation Transformer with optimized abstract syntax tree for efficient code clone detection. In *Proceedings of the 47th IEEE/ACM International Conference on Software Engineering (ICSE)*, pp. 281–293, 2025.

[38] Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, Kaixuan Wang, and Xudong Liu. A novel neural source code representation based on abstract syntax tree. In *Proceedings of the 41st IEEE/ACM International Conference on Software Engineering (ICSE)*, pp. 783–794, 2019.

[39] Zixian Zhang and Takfarinas Saber. Assessing the code clone detection capability of large language models. In *Proceedings of the 4th International Conference on Code Quality (ICCQ)*, pp. 75–83, 2024.

[40] Gang Zhao and Jeff Huang. DeepSim: deep learning code functional similarity. In *Proceedings of the 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pp. 141–151, 2018.

[41] Yue Zou, Bihuan Ban, Yinxing Xue, and Yun Xu. CCGraph: a PDG-based code clone detector with approximate graph matching. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 931–942, 2020.

[42] 井上克郎, 神谷年洋, 楠本真二. コードクローン検出法. コンピュータ ソフトウェア, Vol. 18, No. 5, pp. 529–536, 2001.