修士学位論文

題目

TF-IDF 法と LSH アルゴリズムを用いた高速な関数単位のコードクローン検出法

指導教員

井上 克郎 教授

報告者

山中 裕樹

平成 26 年 2 月 5 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

平成 25 年度 修士学位論文

TF-IDF 法と LSH アルゴリズムを用いた高速な関数単位のコードクローン検出法

山中 裕樹

内容梗概

ソフトウェア保守における問題の 1 つとしてコードクローンが指摘されている．コードクローンとは，ソースコード中に存在する同一または類似した部分を持つコード片を意味している．コードクローンを検出し，共通する処理に対してライブラリ化などの集約を行うことによって，ソフトウェアの保守性や可読性を向上させることが可能となる．

これまでの研究において様々なコードクローン検出手法が提案されてきたが，その多くの手法がプログラムの構文上の類似性のみに着目している．また，プログラムの意味的な処理の類似性に着目した手法では，検出時間に膨大な時間がかかるという問題点がある．

そこで本研究では，情報検索技術を利用した関数クローン（関数単位のコードクローン）の検出手法を提案する．関数単位のコードクローンは処理の内容がまとまっているため，コード片単位のコードクローンに比べてライブラリ化などの集約の対象になりやすいと考えられる．

本手法では，情報検索において一般的に利用されている TF-IDF 法を用い，ソースコード中の識別子や予約語に利用される単語に対して重み付けを行うことによって，関数を特徴ベクトルに変換する．そして，特徴ベクトル間の類似度を求めることによって，関数クローンの検出を行う．また，近似最近傍探索アルゴリズムの一種である LSH（Locality-Sensitive Hashing）アルゴリズムを用いて，特徴ベクトルをクラスタリングすることによって，検出の高速化を図っている．

評価実験では，2 つの Java プロジェクトに対して，90%以上の適合率で関数クローンの検出を行うことができた．さらに，既存のコードクローン検出手法と比較を行い，検出精度と検出時間の観点から本手法の有用性を確認することができた．

主な用語

コードクローン
ソフトウェア保守

TF-IDF

LSH（Locality-Sensitive Hashing）

目 次

1 はじめに 5

2 背景 7

2.1 コードクローン 7

2.2 コードクローン検出 9

2.2.1 行単位の検出 9

2.2.2 字句単位の検出 10

2.2.3 抽象構文木を用いた検出 10

2.2.4 メトリックを用いた検出 11

2.2.5 プログラム依存グラフを用いた検出 11

2.2.6 メモリベースの検出 12

2.3 既存のコードクローン検出手法の問題点 14

3 検出手法 16

3.1 STEP1:ワードの抽出 17

3.2 STEP2:TF-IDF 法を利用した特徴ベクトルの計算 18

3.3 STEP3:LSH アルゴリズムを用いた特徴ベクトルのクラスタリング 19

3.4 STEP4:コードクローンの判定 21

4 評価実験 22

4.1 検出精度の評価 23

4.1.1 Tempero らのコーパスを用いた評価 23

4.1.2 Roy らのベンチマークを用いた評価 24

4.2 既存手法との比較 25

4.2.1 検出精度の比較 25

4.2.2 検出時間の比較 27

4.3 関数クローンの実例 27

5 考察 35

5.1 本手法の有用性 35

5.1.1 検出精度 35

5.1.2 検出時間 39

5.2 本手法の拡張性 39

5.3 評価実験の妥当性 39

6　まとめと今後の課題 **41**

謝辞 **42**

参考文献 **43**

1　はじめに

ソフトウェア保守における問題の 1 つとしてコードクローンが指摘されている．コードクローンとは，ソースコード中に存在する同一または類似した部分を持つコード片のことであり，コピーアンドペーストなどの様々な理由により生成される [10]．一般的に，コードクローンの存在はソフトウェアの保守を困難にすると言われている．例えば，あるコード片を編集する場合，対応する全てのコードクローンに対しても一貫した編集が必要となる可能性がある．

ソースコードの規模が大きくなると，ソースコード中に含まれるコードクローンも膨大な量となり，手作業でそれらを管理することは困難となる．従って，コードクローンを自動的に検出することを目的とした様々な研究が行われている [23]．コードクローンを検出することによって，一貫した編集がなされておらず，不具合を引き起こす危険性があるコード片を検出することが可能である．また，コードクローン中の共通処理に対して，親クラスへの引上げやライブラリ化といった集約を行うことによって，ソフトウェアの保守性や可読性を向上させることが可能となる [8, 9]．

これまでに様々なコードクローン検出手法が提案されてきたが，その多くの手法がプログラムの構造的な類似性のみに着目している [3, 6, 7, 13, 14, 15, 21]．そのため，同一の処理を実装しているにも関わらず，for 文と while 文の違いなど構文上の実装が異なる場合はコードクローンとして検出することができる手法は少ない．また，プログラムの意味的な処理の類似性に着目した手法もいくつか提案されているが，検出時間に膨大な時間がかかるという問題点がある [16, 17, 20]．

そこで本研究では，情報検索技術 [2] を利用することによって，意味的に処理が類似した関数クローン（関数単位のコードクローン）を検出する手法を提案する．コード片単位で検出を行う場合，構文の不完全な部分で終了するコード片など，集約を行うことが困難であるコードクローンが多く検出されることがある [27]．一方，関数単位のコードクローンは処理の内容がまとまっているため，開発者にとって集約の対象になりやすいコードクローンを検出することができると考えられる．

本手法では，情報検索で一般的に利用されている TF-IDF 法 [2] を用いて，ソースコード中の識別子や予約語に利用される単語に対して重み付けを行う．そして，重み付けに基づいて各関数を特徴ベクトルに変換し，それらの類似度を計算することによって，関数クローンの検出を行う．また，近似最近傍探索アルゴリズムの一種である LSH（Locality-Sensitive Hashing）アルゴリズム [12] を用いて，特徴ベクトルをクラスタリングすることによって，検出の高速化を図っている．

評価実験では，2 つの Java プロジェクトに対して適用を行い，90%以上の適合率で関数ク

ローンの検出を行うことを確認することができた．さらに，Kim らが開発したコードクローン検出ツールである MeCC[16] と検出精度と検出時間の観点から比較を行った．MeCC は，構文の類似性に依存せずに，意味的に処理が類似した関数単位のコードクローンを検出することが可能である．3 つの C プロジェクトに対して適用した結果，本手法の方が高い精度で，より多くのコードクローンを検出することができた．また，本手法を用いた場合の検出にかかる時間は 5 分以下となり，MeCC よりも高速に関数クローンを検出することができた．さらに，ほぼ同一の処理を実装しているにも関わらず，条件分岐処理や繰り返し処理の実装が異なる関数クローンや，文が並び替えられている関数クローンを検出することができた．

以降，2 章では，本研究の背景について述べる．3 章では，本研究で提案する関数クローン検出手法について述べる．4 章では，本手法の評価実験について述べる．5 章では，本手法と評価実験の考察について述べる．最後に，6 章でまとめと今後の課題について述べる．

2 背景

本章では，本研究の背景としてコードクローン，および，既存のコードクローン検出手法について述べる．

2.1 コードクローン

コードクローンとは，ソースコード中に存在する同一または類似した部分を持つコード片のことであり，コピーアンドペーストなどの様々な理由により生成される [10]．一般的に，コードクローンの存在はソフトウェアの保守性を悪化させる要因の一つと言われている．例えば，あるコード片を編集する場合，そのコード片に対応する全てのコードクローンに対しても一貫した編集の是非を検討する必要がある．

文献 [16, 24] では，コードクローン間の違いの度合いに基づき，コードクローンを以下の 4 つの定義に分類している．

タイプ 1

空白やタブの有無，括弧の位置などのコーディングスタイル，コメントの有無などの違いを除き完全に一致するコードクローン

タイプ 2

タイプ 1 のコードクローンの違い加えて，変数名や関数名などのユーザ定義名，変数の型などが異なるコードクローン

タイプ 3

タイプ 2 のコードクローンの違いに加えて，文の挿入や削除，変更などが行われたコードクローン

タイプ 4

同一の処理を実行するが，構文上の実装が異なるコードクローン

タイプ 4 のコードクローンとして，以下のものが挙げられる．

- 条件分岐処理の制御構造の実装が異なる．(図 10 参照)
- 中間媒介変数の利用の有無が存在している．(図 11 参照)
- 繰り返し処理の制御構造の実装が異なる．(図 12, 図 13 参照)
- 文の並び替えが発生している．(図 14 参照)

```
public void add1(ResourceCollection rc){
    if (rc == null) {
        return;
    }
    if (resources == null) {
        resources=new Union();
    }
    resources.add(rc);
}
```

```
public void add2(ResourceCollection rc){
    // ※ nullチェック漏れ
    if (resources == null) {
        resources=new Union();
    }
    resources.add(rc);
}
```

```
public void add3(ResourceCollection rc){
    // ※ nullチェック漏れ
    resources=resources
                == null ? new Union() : resources;
    resources.add(rc);
}
```

図 1: タイプ 3・タイプ 4 のコードクローンの例

図 1 は，タイプ 3，および，タイプ 4 のコードクローンの実例である．関数 add1 では引数 rc に対する null チェックが行われているのに対して，関数 add2 では null チェックが欠如している．従って，関数 add1 と関数 add2 はタイプ 3 のコードクローンとなる．また，関数 add2 では if 文を用いて条件分岐処理を実装しているのに対して，関数 add3 では三項演算子を用いて全く同一の条件分岐処理を実装してる．従って，関数 add2 と関数 add3 はタイプ 4 のコードクローンとなる．

図 1 の例では，関数 add2 と関数 add3 は引数 rc に対する null チェックが欠如しており，不具合を引き起こす危険性がある．これらの関数をコードクローンとして検出することによって，開発者は，不具合を引きを超す危険性がある関数を認識することが可能となる．さらに，共通する処理に対して，親クラスへの引上げやライブラリ化といった集約を行うことによって，ソフトウェアの保守性や可読性を向上させることが可能となる [8, 9].

2.2 コードクローン検出

これまでの研究において，様々なコードクローン検出手法が提案されてきた [10, 23]．コードクローン検出では，主に以下の種類が存在する．

- 行単位の検出 [6, 7, 14]
- 字句単位の検出 [15, 21, 25]
- 抽象構文木を用いた検出 [3, 13]
- メトリックを用いた検出 [18, 19, 22]
- プログラム依存グラフを用いた検出 [11, 17, 20]
- メモリベースの検出 [16]

以降，それぞれの検出手法について説明する．

2.2.1 行単位の検出

この検出手法では，ソースコードを行単位で比較し，閾値以上の長さで重複している行をコードクローンとして検出する．

Johnson や Ducasse らの検出手法では，各行に含まれる空白やタブを取り除いた後，全ての行を比較し，共通した行をコードクローンとして検出している．従って，プログラミング言語に依存せずにコードクローンを検出することが可能であるが，タイプ 1 のコードクローンのみが検出の対象である [6, 7, 14].

2.2.2　字句単位の検出

字句単位の検出手法では，検出の前処理としてソースコードを字句の列に変換する．そして，閾値以上の長さで一致している字句の部分列をコードクローンとして検出する．

Kamiya らが開発した CCFinder[15] では，字句解析と構文解析を行うことによって，ソースコードをトークンの列に変換する．このとき，変数名や関数名などのユーザ定義名を同一のトークンに変換する．そして，閾値以上の長さで共通したトークン列をコードクローンとして検出する．このツールは C/C++，Java，COBOL，FORTRAN など広く用いられている複数のプログラミング言語に対応しており，タイプ 2 までのコードクローンが検出の対象である．

佐々木らが開発した FCFinder[25] では，ソースコード中のトークンの列に基づいて，ソースコード中の各ファイルをハッシュ値に変換する．そして，ハッシュ値が一致したファイルをコードクローンとして検出する．この手法は，大規模ソースコードに対しても高速に検出を行うことが可能であり，タイプ 2 までのファイル単位のコードクローンが検出の対象である．

また，Li らが開発した CP-Miner[21] では，字句解析と構文解析を行い，ソースコード中のユーザー定義名を特殊文字に置換する．そして，ソースコード中の各文をハッシュ値に変換する．最後に，ハッシュ値のシーケンスに対して頻出系列マイニングを適用することによって，コードクローンの検出を行う．頻出系列マイニングでは，連続でないシーケンス（不一致部分を含むシーケンス）であっても検出することが可能である．従って，タイプ 1 からタイプ 3 までのコードクローンを検出することが可能である．

2.2.3　抽象構文木を用いた検出

抽象構文木とは，ソースコードの構文構造を木構造で表したグラフのことを意味する．図 2 に抽象構文木の例を示す．この検出手法では，検出の前処理としてソースコードに対して構文解析を行うことによって，抽象構文木を構築する．そして，抽象構文木上の同形の部分木をコードクローンとして検出する．

Baxter らが開発した CloneDR[3] では，抽象構文木の各部分木からハッシュ値を生成する．そして，ハッシュ値が同一の部分木の対をコードクローンとして検出する．このツールは，大規模ソフトウェアに対しても高速に検出を行うことが可能であり，タイプ 2 までのコードクローンが検出の対象である．

また，Jiang らが開発した DECKARD[13] では，抽象構文木の各部分木を特徴ベクトルに変換する．そして，LSH(Locality-Sensitive Hashing) アルゴリズム [12] を用い，特徴ベクトル間の類似度を求めることによって，コードクローンの検出を行う．このアルゴリズムで

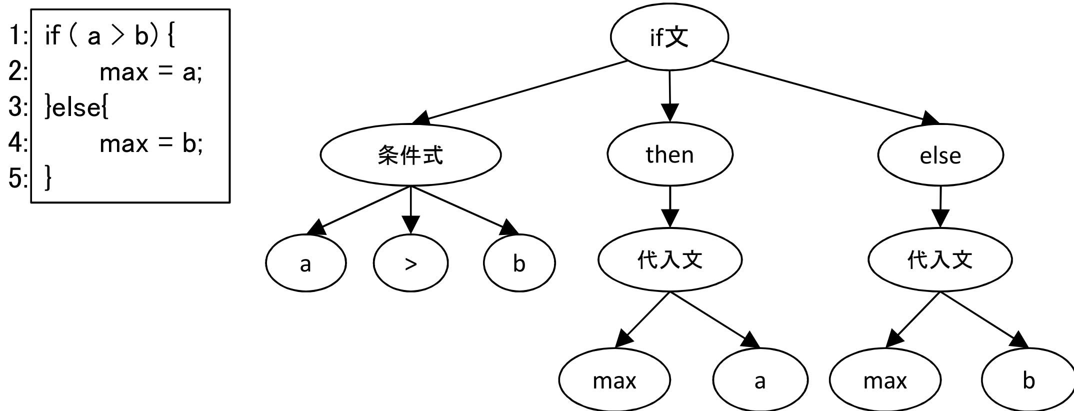

図 2: 抽象構文木の例

は，ある程度特徴ベクトルに違いがあっても検出することが可能である．従って，タイプ 1 からタイプ 3 までのコードクローンを検出することが可能である．

2.2.4 メトリックを用いた検出

この検出手法では，ファイル，クラス，メソッドなどのプログラムのモジュールに対してメトリックスを計測し，それらの類似度を計算することによって，コードクローンの検出を行う．

Mayrand らの手法 [22] では，関数に対して 21 種類のメトリックスを計測し，コードクローンの検出を行っている．また，Kontogiannis らの手法 [18, 19] では，抽象構文木の部分木に対して 5 種類のメトリックスを計測し，ユークリッド距離を用いて類似度を計算している．そして，閾値より短い距離に位置する部分木をコードクローンとして検出している．

これらの検出手法では，タイプ 1 からタイプ 3 のコードクローンを検出することが可能である．

2.2.5 プログラム依存グラフを用いた検出

プログラム依存グラフとは，ソースコードの文や式といった要素間の依存関係を表した有効グラフのことを意味する．図 3 に，プログラム依存グラフの例を示す．この例では，関数内の各文をノードとしている．そして，ノード間のデータ依存（実線）と制御依存（点線）の関係を表している．

この検出手法では，検出の前処理としてソースコードに対して意味解析を行うことによって，プログラム依存グラフを構築する．そして，グラフ上の同形の部分グラフをコードク

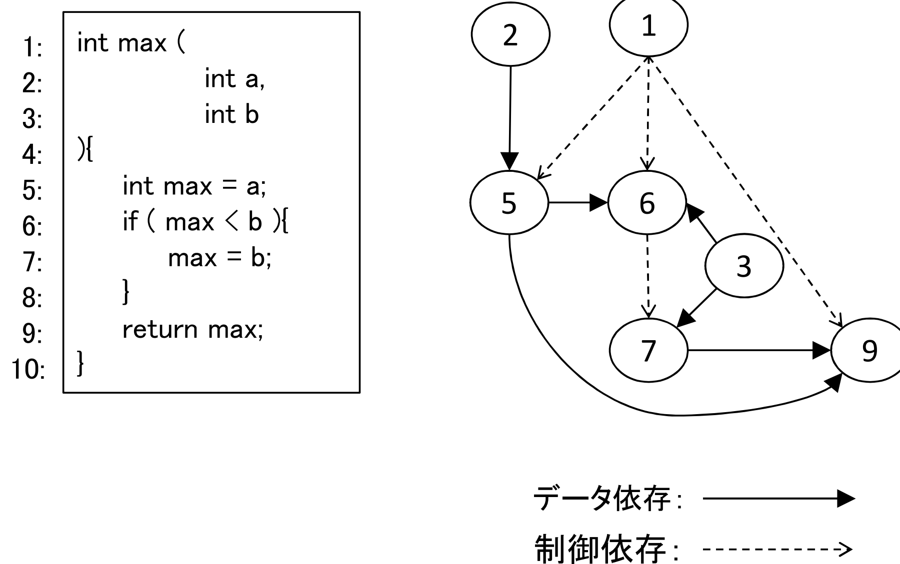

図 3: プログラム依存グラフの例

ローンとして検出する.

Komondoor らの手法 [17] では，プログラム依存グラフ上での各ノードの比較を行い，コードクローンの検出を行っている．また，Krinke の手法 [20] では，変数や演算子などの字句をノードとするプログラム依存グラフからコードクローンの検出を行っている.

また，肥後らは Komondoor らの手法を拡張し，Scorpio[11] を開発している．この手法では，実行依存（文の実行順序の関係）を追加した拡張プログラム依存グラフを利用し，コードクローンの検出を行っている.

これらの検出手法は，プログラムの意味的な処理の類似性に着目しているため，文の並び替えが発生したコードクローンなど，タイプ 4 のコードクローンを検出することが可能である.

2.2.6 メモリベースの検出

メモリベースの検出ツールとして，Kim らが開発した MeCC[16] が挙げられる.

MeCC では，静的解析を行うことによって，ソースコード中の各関数が終了した時点における抽象的なメモリの状態の予測を行う．そして，メモリの状態が類似した関数をコードクローンとして検出する．ここでメモリとは，ADDR（大域変数，仮引数，局所変数などの関数で利用されている変数）と GV（ADDR が取り得る値の集合）の組の集合で表される.

メモリの状態の類似度の計算例を図 4 に示す．この例では，2 つの関数 Function A と

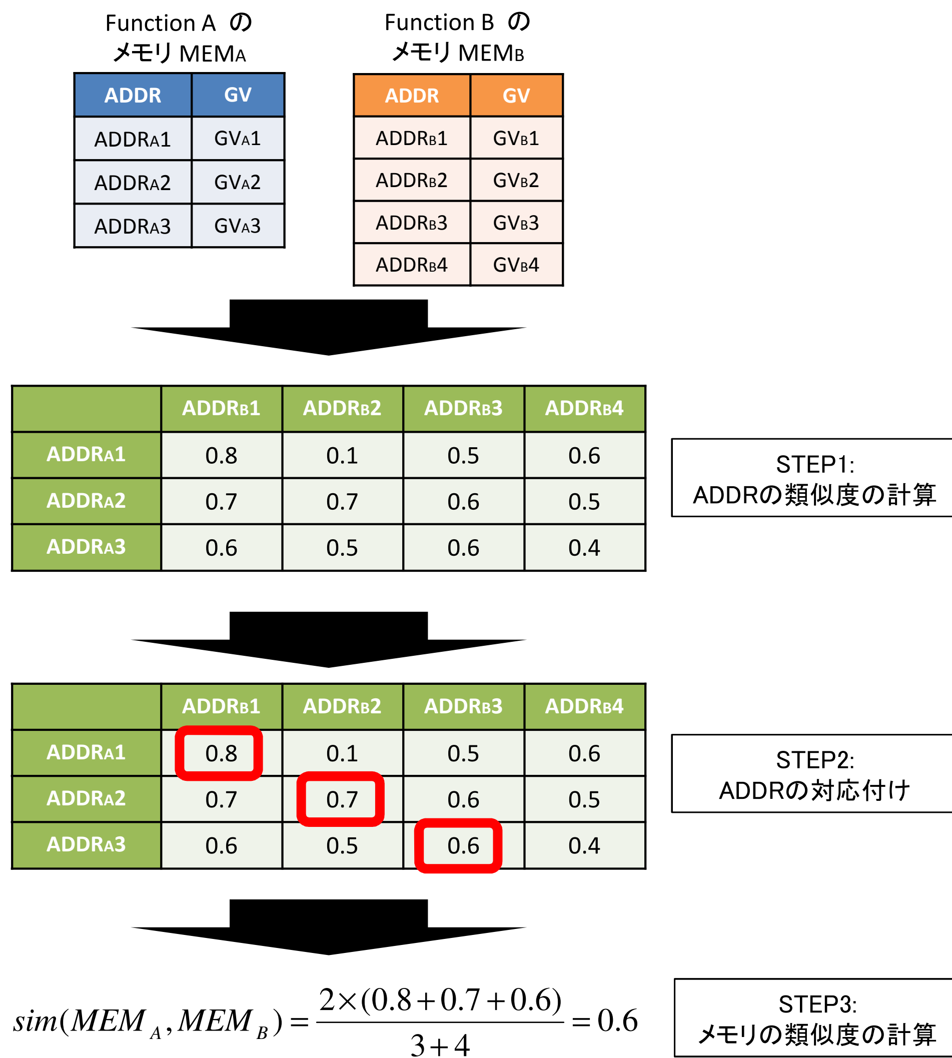

図 4: メモリ間の類似度の計算手法（MeCC）

Function B のメモリ MEM_A と MEM_B の類似度を求めている．類似度の計算手法は以下の 3 つのステップから構成される．

STEP1：ADDR 間の類似度の計算

まず最初に，2 つの関数中の ADDR の全組み合わせに対して類似度を計算する．以下に，$ADDR_X$ と $ADDR_Y$ の類似度の計算式を示す．

$$sim_{addr}(ADDR_X, ADDR_Y) = \frac{2 \times |GV_X \cap GV_Y|}{|GV_X| + |GV_Y|}$$

ここで GV_X と GV_Y は，それぞれ，$ADDR_X$ と $ADDR_Y$ が取り得る値の集合を意味する．GV は，関数中の条件分岐などによって，単一の ADDR に対して複数の値を取り得る可能性がある．従って，GV が全く同一の値の集合を持つ場合，類似度は 1 となる．

STEP2：ADDR の対応付け

次に，STEP1 で計算した類似度に基づいて，2 つの関数中の ADDR の対応付けを行う．この手法では，貪欲法を用いて対応付けを行っている．すなわち，高い類似度から順番に，ADDR を対応させていく．

STEP3：メモリの類似度の計算

最後に，以下の計算式を用いて，メモリ MEM_A と MEM_B の類似度の計算を行う．

$$sim(MEM_A, MEM_B) = \frac{2 \times SUM(sim_{addr})}{|MEM_A| + |MEM_B|}$$

ここで，$SUM(sim_{addr})$ は，2 つの関数間で対応する ADDR の類似度の合計値を意味する．図 4 の例では，$sim_{addr}(ADDR_A1, ADDR_B1)$，$sim_{addr}(ADDR_A2, ADDR_B2)$，$sim_{addr}(ADDR_A3, ADDR_B3)$ の合計値となる．また，$|MEM_A|$ と $|MEM_B|$ は，それぞれ Function A と Function B における ADDR と GV の組の総数を意味する．そして，これらの類似度が閾値以上であれば，コードクローンとして検出を行う．

この検出手法では，条件分岐処理や繰り返し処理の実装が異なる関数クローンや，文の並び替えが発生している関数クローンを検出することが可能である．評価実験では，3 つの大規模ソフトウェアに対して適用を行っている．そして，字句単位の検出手法，抽象構文木を用いた検出手法，プログラム依存グラフを用いた検出手法と比較を行い，より多くのタイプ 3 とタイプ 4 のコードクローンを検出することができることを示している．

2.3 既存のコードクローン検出手法の問題点

2.2 節では，既存のコードクローン検出の代表的な手法について説明した．

それらの多くの手法は，行単位の検出（2.2.1 節），字句単位の検出（2.2.2 節），抽象構文木を用いた検出（2.2.3 節）など，ソースコードの構文上の類似性のみに着目している．これらの手法は，比較的高速にコードクローンの検出を行うことが可能である．しかし，図 1 で示した例などのようなタイプ 4 のコードクローンの検出を行うことが困難である．さらに，メトリックを用いた検出手法（2.2.4 節）では，検出精度の問題点も挙げられる．

一方，意味的な処理の類似性に着目しているプログラム依存グラフを用いた検出手法（2.2.5 節）では，文の並び替えが発生しているコードクローンなどのタイプ 4 のコードクローンを検出することが可能である．しかし，プログラム依存グラフの構築や比較に対する計算コストが高く，大規模ソフトウェアに対して適用することが困難である．また，プログラミング言語毎にプログラム依存グラフを構築する機構を用意する必要がある．

そこで，Kim らの研究で，メモリベースの検出手法（2.2.6 節）が提案された．この検出手法では，他の検出手法に比べて，より多くのタイプ 3 とタイプ 4 のコードクローンを検出することが可能である．さらに，大規模ソフトウェアに対して適用を行い，実時間で検出することが可能である．しかし，静的解析に膨大な時間がかかり，コードクローンの検出に 1 時間以上の検出時間が必要である．また，C 言語のみが対象であり，プログラミング言語毎に静的解析を行う機構を用意する必要がある．

本研究で提案する手法では，TF-IDF 法を利用し．識別子や予約語などに基づいてソースコード中の各関数を特徴ベクトルに変換する．そして，それらの類似度を計算することによって，全タイプのコードクローンの検出を行う．従って，プログラム言語に依存せずにコードクローンの検出を行うことが可能である．また，LSH アルゴリズムをを用いて特徴ベクトルをクラスタリングすることによって，既存手法に比べて高速な検出が可能であると考えられる．なお，本研究における評価実験では，MeCC と比較を行うことによって本手法の有用性を示している．

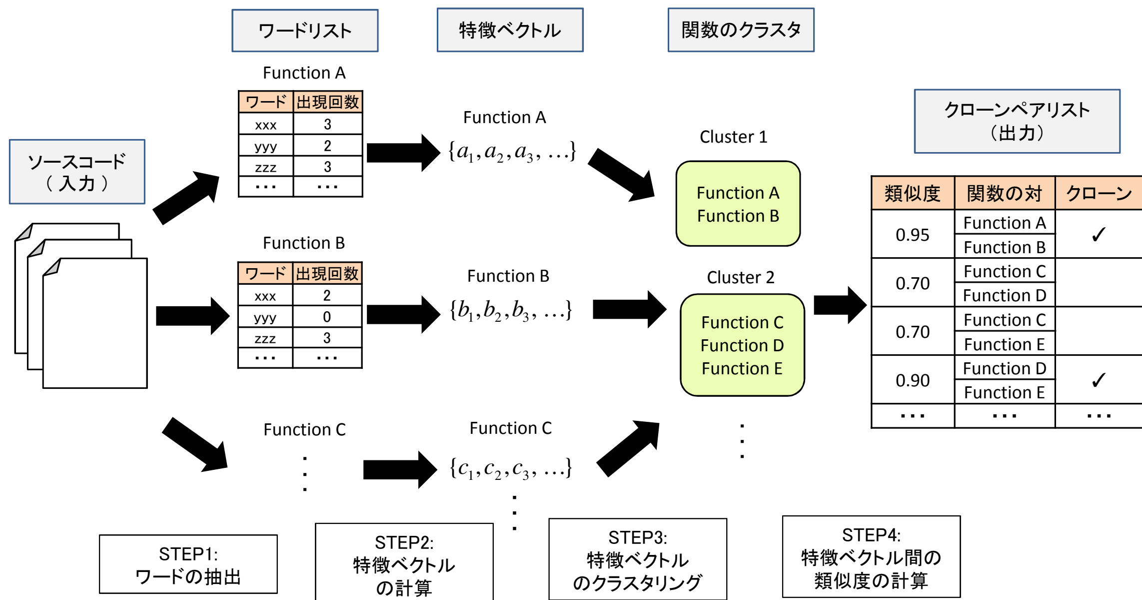

図 5: 本手法の概要

3　検出手法

本章では，本研究で提案する関数クローン検出手法について述べる．本研究では，情報検索技術を利用し，2.1 節で説明した全てのタイプの関数クローンを高速に検出することが目的である．本手法では，TF-IDF 法 [2] を用い，入力されたソースコード中の**ワード**に基づいて各関数を特徴ベクトルに変換する．ここでワードとは，以下の 2 つを対象とする．

- 変数や関数などに付けられた識別子名を構成する単語
- 条件文や繰り返し文などの構文に利用される予約語

そして，特徴ベクトル間の類似度を求めることによってクローンペア（互いに処理が類似した関数クローンの対）の集合をリストとして出力する．また，類似度の計算の直前に LSH(Locality-Sensitive Hashing) アルゴリズム [12] を利用し，特徴ベクトルのクラスタリングを行うことによって，検出の高速化を図っている．もし，全関数の特徴ベクトル間の類似度を計算する場合，ソースコードの規模が大きくなると検出時間が膨大になると考えられる．

本手法の概要を図 5 に示す．本手法は，主に以下の 4 つのステップから構成される．

STEP1

ソースコード中の各関数からワードの抽出を行う．

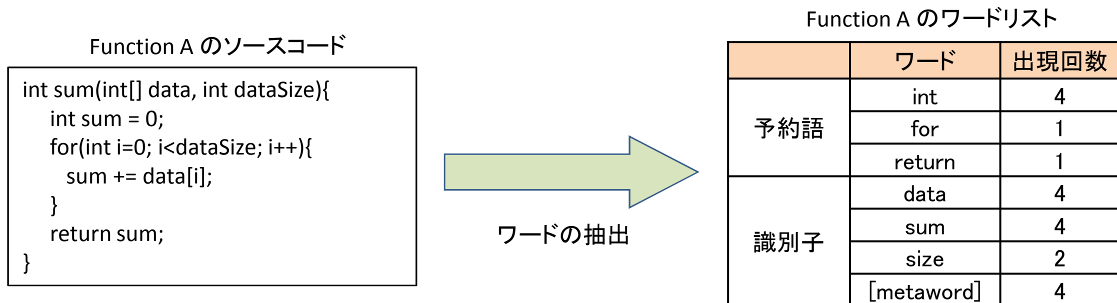

図 6: ワードの抽出の例

STEP2

TF-IDF 法を利用し，STEP1 で抽出したワードに重み付けを行い，各関数を特徴ベクトルに変換する．

STEP3

LSH アルゴリズムを利用し，STEP2 で求めた各関数に対する特徴ベクトルのクラスタリングを行う．

STEP4

STEP3 で求めた関数の各クラスタの中で，特徴ベクトル間の類似度の計算を行い，関数クローンを検出する．

以降の節で，それぞれのステップの詳細について説明する．

3.1 STEP1:ワードの抽出

まず最初に，ソースコードの各関数に含まれる識別子や構文に利用される予約語から，ワードの抽出を行う．

識別子名が複数の単語から構成される場合，以下の方法でワード単位に分割する．

- ハイフンやアンダースコアなどの区切り記号（デリミタ）による分割
- 識別子名中の大文字になっているアルファベット（キャメルケース）による分割

また，2 文字以下の識別子に対してはそれらをまとめて 1 つのワードとして扱う．この理由は，繰り返し文などによく利用される i や j といった意味情報が込められない変数を全て同一のものとして扱うためである．識別子の情報を利用することによって，各関数が実装する機能を表すことができると考えられる．

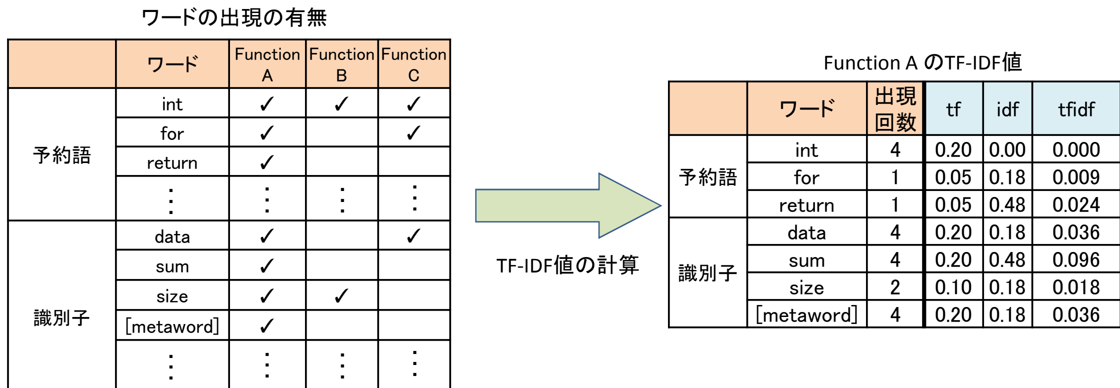

図 7: TF-IDF 法による特徴ベクトルの計算例

さらに，構文に利用される単語とは，条件文に用いられる if や switch，繰り返し文に用いられる for や while といった予約語のことを示している．本手法では，このような単語もワードとして扱う．

なお，各ワードの大文字と小文字による違いは区別せずに，同一のもとして扱う．

図 6 にワードの抽出例を示す．Function A は与えられたデータ配列の要素の合計値を計算し，その値を返す処理を行っている．この関数には，3 つの予約語 (int, for, return) と，4 つの識別子（sum, data, dataSize, i）が含まれている．この例では，変数 dataSize をワード data とワード size に分割している．また，for 文の中で利用されている変数 i は 2 文字以下であるため，単一のワード metaword として扱っている．

3.2 STEP2:TF-IDF 法を利用した特徴ベクトルの計算

次に，STEP1 で抽出したワードに重み付けを行うことによって，各関数を特徴ベクトルに変換する．

ここでは，TF-IDF 法 [2] を利用して各ワードの重みを計算し，その値を特徴量として利用する．TF-IDF 法は情報検索において，自然言語で書かれた文書の要約や類似性の判定などに利用されており，計算コストが小さいため大規模なソースコードにも適用できると考えられる．TF-DIF 法による値は tf 値（関数中のワードの出現頻度）と idf 値（ソースコード全体のワードの希少さ）の積で与えられる．ワード x の重み $tfidf_x$ の計算式を以下に示す．

$$tf_x = \frac{\text{関数中のワード}\, x\, \text{の出現回数}}{\text{関数中に出現する全ワードの出現回数の合計}}$$

$$idf_x = \log \frac{\text{全関数の数}}{\text{ワード}\, x\, \text{が出現する関数の数}}$$

$$tfidf_x = tf_x \times idf_x$$

本手法では，全関数中の各ワードに対して重みを計算し，それらを特徴量として用いることによって特徴ベクトルを求める．従って，各関数の特徴ベクトルの次元はソースコード中に存在する全ワードの数となる．

図 7 に TF-IDF 法の計算例を示す．この例は，図 6 の Function A の $tfidf$ 値を求めるものである．

図中の左の表は，ソースコード中の全関数に対して，Function A で出現したワードが含まれているか否かを表す表である．この表では，Function A の他に，Function B と Function C の合計 3 つの関数が存在すると仮定している．この表と図 6 の各ワードの出現回数から $tfidf$ 値を求めることが可能となる．

例として，変数 sum の $tfidf$ 値の計算方法について説明する．tf 値は Function A における変数 sum の出現頻度を意味している．Function A における全ワードの出現概数の合計値は 20 であるため，以下の式で tf 値を求めることができる．

$$tf_{sum} = 4/20 = 0.20$$

また，変数 sum は 3 つの関数の中で Function A にしか存在していないため，以下の式で求めることができる．

$$idf_{sum} = \log(3/1) = 0.48$$

$tfidf$ 値はこれらの値の積であるため，以下の値となる．

$$tfidf_{sum} = tf_{sum} \times idf_{sum} = 0.096$$

Function A 中の他のワードについても同様に $tfidf$ 値を求めることが可能である．これらの値を特徴量として，各関数を特徴ベクトルに変換する．従って，Function A は以下の特徴ベクトルに変換することができる．

$$V_A = (0.00, 0.009, 0.024, 0.036, 0.096, 0.018, 0.036, ...)$$

なお，特徴ベクトルの他の要素は，Function A で出現しなかったワードに対する重みである．従って，それらのワードの tf 値は 0 となり，特徴量も 0 となる．

3.3 STEP3:LSH アルゴリズムを用いた特徴ベクトルのクラスタリング

このステップでは，STEP2 で計算した各関数の特徴ベクトルに対してクラスタリングを行うことによって，クローンペアと成り得る候補を絞ることを目的とする．

ここでは，近似最近傍探索アルゴリズムの一種であるLSHアルゴリズム[12]を用いて特徴ベクトルのクラスタリングを行う．LSHアルゴリズムは，(p_1, p_2, r, c)-Sensitive Hashingと(r, c)-Approximate Neighborを用いたクラスタリング手法である．このアルゴリズムを利用することによって，クエリとして1つの特徴ベクトルを与えると，特徴ベクトル集合からそのクエリと近似した特徴ベクトル集合のクラスタを取得することができる．(p_1, p_2, r, c)-Sensitive Hashingと(r, c)-Approximate Neighborの定義を以下に示す．

(p_1, p_2, r, c)-Sensitive Hashing

実数$p_1, p_2 (0 \le p_1, p_2 \le 1)$，実数$r (0 < r)$，実数$c (1 < c)$，$\mathbf{R}^d$空間に対するベクトル集合$V$上の任意の点$v_i$, v_jが与えられたとき，

$$\begin{cases} if\ D(v_i, v_j) < r\ \ then\ Prob[h(v_i) = h(v_j)] > p_1 \\ if\ D(v_i, v_j) > cr\ then\ Prob[h(v_i) = h(v_j)] < p_2 \end{cases}$$

を満たすハッシュ関数hを(p_1, p_2, r, c)-Sensitive Hashingと呼ぶ．ここで，Dは$\mathbf{R}^d$上の距離関数，$Prob$は条件式が真となる確率を表している．

(r, c)-Approximate Neighbor

$\mathbf{R}^d$空間に対するベクトル集合V上の任意の点（クエリ）vが与えられたとき，

$$U = \{u \in V | D(v, u) \le cr\}$$

で定義されるVの部分集合Uをクエリvに対する(r, c)-Approximate Neighborと呼ぶ．ここで，cとrの定義は(p_1, p_2, r, c)-Sensitive Hashingで利用した定義と共通である．

$\mathbf{R}^d$空間に対する特徴ベクトル集合Vが与えられ，(p_1, p_2, r, c)-Sensitive Hashingを用いた関数の族$h_{l,k} (1 \le l \le L, 1 \le k \le K)$からなる関数$H_l$が以下の式で与えられたとき，

$$H_l = (h_{l,1}(v), h_{l,2}(v), ..., h_{l,K}(v)) \in \mathbf{R}^K$$

V上の任意の点vに対してL個のK次元のベクトルが得られる．LSHアルゴリズムでは，これらのベクトルをそれぞれハッシュテーブルの鍵とすることで高速に近傍点を求めることができる．

本手法では，STEP2で求めた全関数の特徴ベクトルに対して，各関数の特徴ベクトルをクエリとして与えることによって，互いに類似した関数のクラスタの集合を取得している．なお，本手法ではLSHアルゴリズムの実装であるE^2LSH[1][1]を利用している．

[1] http://www.mit.edu/~andoni/LSH/

特徴ベクトルの次元を d，特徴ベクトル集合の大きさを n，確率に関するパラメータを $\rho = log_{p_2}p_1$ としたとき，LSH のクラスタリングの時間計算量は $O(dn^{\rho}\log n)$ と表される．一方，全関数に対して特徴ベクトル間の類似度を計算する場合の時間計算量は $O(dn^2)$ となる．従って，本ステップであらかじめクラスタリングを行い，クローンペアと成り得る候補を絞ることによって，検出時間にかかる計算コストを削減できると考えられる．

3.4 STEP4:コードクローンの判定

最後に，STEP3 で求めた各クラスタ中の関数の対に対して，コサイン類似度を用いてクローンペアであるか否かの判定を行う．コサイン類似度は多次元ベクトルの類似度を測定するものであり，次元が d である 2 つの特徴ベクトル $\vec{a}, \vec{b}$ 間の類似度は以下の式で表すことができる．

$$sim(\vec{a}, \vec{b}) = cos(\vec{a}, \vec{b}) = \frac{\sum_{i=1}^{d} a_i b_i}{\sqrt{\sum_{i=1}^{d} {a_i}^2}\sqrt{\sum_{i=1}^{d} {b_i}^2}}$$

TF-IDF 法の計算式から分かるように，特徴量は常に正の値となるため，コサイン類似度は 0 から 1 の範囲となる．もし，コサイン類似度が閾値以上であれば，それら 2 つの関数はクローンペアであると判定する．

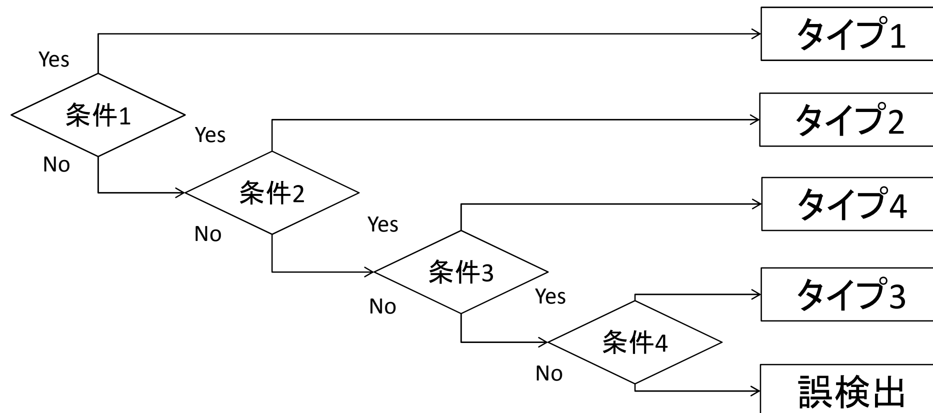

図 8: 関数クローンの分類方法

4 評価実験

本章では，3 章で述べた関数クローン検出手法の評価実験について述べる．評価実験では，コーパスとベンチマークを用いた検出精度の評価と，既存のタイプ 4 の関数クローン検出ツールとの比較実験を行った．本実験で適用の対象としたプロジェクトを表 1 に示す．また，本実験では，図 8 に基づいて関数クローンのタイプの分類を手作業で行っている．文献 [16]

表 1: 適用プロジェクト

プロジェクト名	バージョン	言語	規模
Apache Ant[2]	1.8.4	Java	109KLOC
ArgoUML[3]	0.34	Java	192KLOC
Apache HTTPD[4]	2.2.14	C/C++	343KLOC
Python[5]	2.5.1	C/C++	435KLOC
PostgreSQL[6]	8.5.1	C/C++	937KLOC

[2] http://ant.apache.org/
[3] http://argouml.tigris.org/
[4] http://www.python.org/
[5] http://httpd.apache.org/
[6] http://www.postgresql.org/

の評価実験においても，本実験同様に，タイプ 4 とタイプ 3 の定義を同時に満たす関数クローンは全てタイプ 4 に分類している．

なお，本実験では結果の信頼性を重視するため，関数クローンとして検出する類似度の閾値を 0.9 とした．また，比較的小さい関数を検出の対象から除外するため，30 トークン以上の関数を対象に実験を行った．

以降，4.1 節では，検出精度の評価実験について述べる．4.2 節では，既存手法との比較実験について述べる．最後に 4.3 節で，本手法で検出することができた関数クローンの実例を示す．

4.1 検出精度の評価

検出精度の評価では，Tempero らのコーパス [26][7]を用いた評価実験と，Roy らのベンチマーク [24] を用いた評価実験を行った．なお，一般的に，コードクローン検出手法の評価実験では Bellon らのベンチマーク [4][8]が用いられている．しかし，このベンチマークはタイプ 3 までのコード片単位のコードクローンを対象としているため，本実験では上記のコーパスとベンチマークを対象に評価実験を行っている．以降，それぞれの実験内容と結果について述べる．

4.1.1 Tempero らのコーパスを用いた評価

最初に，Tempero らのコーパス [26] を用いた評価実験について述べる．このコーパスは，コードクローン検出ツール CMCD[28] の出力結果と著者らの手作業による判断に基づいて作成されおり，112 個の Java プロジェクトに存在するタイプ 1 からタイプ 4 のメソッド単位のコードクローンを対象としている．

本実験では 2 つの Java プロジェクト（Apache Ant, ArgoUML）に対して適合率と再現率の評価を行った．適合率（precision）と再現率（recall）は以下の式で求められる．

$$precision = \frac{|FC_{result} \cap FC_{corpus}|}{|FC_{result}|}$$

表 2: コーパスを用いた検出精度の評価

	適合率	再現率	タイプ 1	タイプ 2	タイプ 3	タイプ 4
Apache Ant	92.2%	62.3%	56	139	220	22
ArgoUML	96.4%	52.7%	222	219	371	33

[7] http://www.qualitascorpus.com/clones/

[8] http://softwareclones.org/

$$recall = \frac{|FC_{result} \cap FC_{corpus}|}{|FC_{corpus}|}$$

上記の式で，FC_{result} は本手法で正解集合と判定したクローンペアの集合を表している．また，FC_{corpus} はコーパスで用意されている正解クローンペアの集合を表している．

結果を表 2 の適合率と再現率に示す．実験の対象とした 2 つの Java プロジェクトに対して再現率は 60%前後であるが，適合率は 90%を超えており，誤検出をほとんど含まずに関数クローンを検出することができた．

さらに，表 2 は，クローンペアを手作業で各タイプに分類した結果を示している．この表から分かるように，本手法を用いることによって，タイプ 1 からタイプ 4 までの全ての関数クローンの検出を行うことを確認することができた．タイプ 4 の関数クローンとしては，以下の種類を検出することができた．

- 文の並び替えが存在する関数クローン
- if 文と三項演算子など条件分岐処理の実装が異なる関数クローン
- for 文と while 文など繰り返し処理の実装が異なる関数クローン
- 中間媒介変数の利用の有無が異なる関数クローン

4.1.2 Roy らのベンチマークを用いた評価

次に，Roy らのベンチマーク [24] を用いた再現性の評価実験について述べる．このベンチマークでは，タイプ 1 からタイプ 4 までの合計 16 個の関数単位のクローンペアが用意されており，文献 [16] の評価実験でも利用されている．

表 3 は，ベンチマークで用意されているクローンペアのタイプ毎の個数と，本手法で検出することができたクローンペアのタイプ毎の個数を示している．結果として，全体で 16 個中 14 個のクローンペアを検出することができた．また，タイプ 1，タイプ 3，タイプ 4 において，本手法を用いて全てのクローンペアを検出することができた．タイプ 4 では，文の並び替えや for 文と while 文の繰り返し処理の実装が異なるクローンペアが用意されており，本手法を用いることによって，それらをコードクローンとして検出することができた．

表 3: Roy らのベンチマークを用いた再現性の評価

	タイプ 1	タイプ 2	タイプ 3	タイプ 4
ベンチマーク	3	4	5	4
検出結果	3	2	5	4

一方で，タイプ 2 では，2 つのクローンペアの検出を行うことができなかった．これらのクローンペアは，元の変数名が 1 文字のアルファベットに省略されており，意味をもたない変数名に変換されていたため，本手法ではコードクローンとして検出することができなかったと考えられる．

4.2　既存手法との比較

次に，既存手法との比較実験について述べる．本実験では，Kim らが開発したコードクローン検出ツールである MeCC[16][9]に対して比較を行った．MeCC は C 言語が対象であるため，その評価実験で適用の対象としている 3 つの C プロジェクト（Apach HTTPD, Python, PostgreSQL）に対して本手法を適用した．以降，検出精度と検出時間の比較について述べる．

4.2.1　検出精度の比較

検出精度の比較では，3 つのプロジェクトに対して検出した関数クローンを，手作業でタイプ 1 からタイプ 4，および，誤検出に分類し，タイプ毎の検出数と誤検出の割合を比較した．なお，MeCC ではクローンセット（互いに類似した関数クローンの集合）単位でタイプの分類を行っている．そのため，本実験においても，クローンペアをクローンセット単位に変換し分類を行っている．

本手法と MeCC における検出精度とタイプ毎の検出クローンセット数の比較を表 4 に示す．MeCC では全検出結果に対して 10%以上の誤検出を含んでいるが，本手法では誤検出の割合が 10%未満であることを確認することができた．また，クローンセットの検出数においても，MeCC よりも多くの関数クローンを検出している．特に，タイプ 1，タイプ 3，タイプ 4 の検出は全てのプロジェクトにおいて MeCC を上回っていることを確認することができた．

さらに，本実験では MeCC に対する本手法の再現性について評価を行った．再現性とは，MeCC と本手法で検出することができた関数クローンの和集合に対する，本手法の検出数の割合を示している．この評価指標は文献 [4] でも用いられている．結果を表 5 に示す．なお，本手法では 30 トークン以上の関数のみを対象としている．そのため，MeCC の全検出結果に対する再現性と，30 トークンでフィルタリングした場合について評価を行った．その結果，全検出結果に対して再現性は約 50%前後であるが，30 トークン以上のより大きな関数では 62%〜98%と高い再現性であることを確認することができた．

表 4: 検出精度と検出クローンセット数の比較

		誤検出の割合	正解集合			
			タイプ 1	タイプ 2	タイプ 3	タイプ 4
本手法	Apache HTTPD	4.6%	71	100	190	11
	Python	6.5%	19	103	159	21
	PostgreSQL	5.3%	57	230	341	17
	合計	5.4%	147	433	690	59
MeCC	Apache HTTPD	12.5%	2	84	71	10
	Python	14.7%	3	127	82	13
	PostgreSQL	16.9%	9	120	88	14
	合計	15.0%	14	331	241	37

表 5: 再現性の評価

	30 トークン以上	全関数
Python	97.5%	61.2%
Apache HTTPD	62.3%	40.0%
PostgreSQL	88.1%	72.7%

表 6: 検出時間の比較

	本手法		MeCC
	パラメータ計算有り	パラメータ計算無し	
Apache HTTPD	4m30s	1m43s	310m34s
Python	4m39s	2m13s	65m26s
PostgreSQL	8m51s	4m39s	428m32s

4.2.2　検出時間の比較

検出時間の比較では，MeCC の評価実験と同様の環境で本手法を実行し，3 つのプロジェクトに対する検出時間を測定した．本実験で用いたワークステーションの環境を以下に示す．

- OS: Ubuntu 64-bit
- CPU: Intel Xeon 2.40GHz
- RAM: 8.0 GB

結果を表 6 に示す．表中のパラメータ計算とは，特徴ベクトルのクラスタリングで利用している E^2LSH に対するものである．E^2LSH は LSH のパラメータを自動的に決定する機能を持つ．この機能はデータセットの中からいくつかのデータをランダムに選択し，その値の傾向を見て適当なパラメータを自動的に設定する．従って，1 つの適用対象に対してパラメータの値を一度計算すれば，その後の検出においてもその値を再利用することが可能であると考えられる．結果として，パラメータ計算を行う場合において，9 分以下で関数クローンを検出することができた．さらに，パラメータ計算を省略した場合では，5 分以下で関数クローンを検出することができた.

一方で，MeCC ではソースコードの静的解析に膨大な時間がかかってしまうため，検出に 60 分以上かかってしまう．従って，本実験では MeCC に比べて本手法の方が関数クローンを高速に検出を行うことを確認することができた.

4.3　関数クローンの実例

本節では，5 つのプロジェクトに対して本手法によって検出することができた関数クローンの実例を紹介する.

図 9 は，本手法によって検出したタイプ 3 の関数クローンの例である．青色のコード片が，関数クローン間で異なっている部分である．図 9(b) では変数 listenerList の null チェックが行われているが，図 9(a) では null チェックが行われていない．この例のように，本手法を用いて null チェック漏れなど，不具合を引き起こす危険性がある関数クローンを検出することができた.

また，検出されたタイプ 4 の関数クローンとしては，文の並び替え，中間媒介変数の利用の有無，if 文と三項演算子を用いた条件分岐処理の置き換え，for 文と while 文を用いた繰り返し処理の置き換えなどの違いが存在する関数クローンを検出することができた．図 10, 図 11，図 12，図 13, 図 14, に本手法で検出したタイプ 4 の関数クローンの例を示す.

[9] http://ropas.snu.ac.kr/mecc/

図 10 は，条件分岐処理が置き換わっている関数クローンである．図 10(a) では if-else 文を用いて標準出力と標準エラー出力の条件分岐処理を実装しているが，図 10(b) では三項演算子を用いて同一の条件分岐処理を実装している．

図 11 は，中間媒介変数の利用の有無が存在する関数クローンである．図 11(a) では 9 行目で変数 df を新たに定義し，それを 10-12 行目で中間媒介変数として利用している．一方，図 11(b) では中間媒介変数を利用せずに実装を行っている．

図 12 と図 13 は，繰り返し処理が置き換わっている関数クローンである．図 12(a) では for 文を用いて繰り返し処理を実装しているが，図 12(b) では while 文を用いてほぼ同一の繰り返し処理を実装している．図 13 も同様に，ほぼ同一の処理を実装しているが，繰り返し処理の実装が異なっている．図 13(a) では 7-9 行目で for 文を用いて繰り返し処理を実装しているが，図 13(b) では 5-8 行目で do-while 文を用いてほぼ同一の繰り返し処理を実装している．さらに，図 13(a) では 4-5 行目で変数 pool に対する例外のチェックが行われているのに対して，図 13(b) では行われていない．このような文の挿入の有無は，不具合を引き起こす危険性があると考えられる．

図 14 は，文の並び替えが発生している関数クローンである．図 14(a) の最初の if 文 (赤色のコード片) と 2 番目の if 文（青色のコード片）が図 14(b) では入れ替わっている．プログラム依存グラフを用いた手法では，このような関数は依存グラフそのものが異なってしまうため，検出することができない可能性が高い．一方，本手法では関数を特徴ベクトルに変換するため，文が並び替えられているタイプ 4 の関数クローンの検出を行うことは容易であると考えられる．

```
1:  private void fireTargetSet(TargetEvent targetEvent){
2:      // ※nullチェック漏れの可能性がある
3:      Object[] listeners=listenerList.getListenerList();
4:      for (int i=listeners.length - 2; i >= 0; i-=2) {
5:          if (listeners[i] == TargetListener.class) {
6:              ((TargetListener)listeners[i+1]).targetSet(targetEvent);
7:          }
8:      }
9:  }
```

(a) argouml/ui/DetailsPane.java

```
1:  private void fireTargetSet(TargetEvent targetEvent){
2:      if (listenerList == null) {
3:          listenerList=collectTargetListeners(this);
4:      }
5:      Object[] listeners=listenerList.getListenerList();
6:      for (int i=listeners.length - 2; i >= 0; i-=2) {
7:          if (listeners[i] == TargetListener.class) {
8:              ((TargetListener)listeners[i+1]).targetSet(targetEvent);
9:          }
10:     }
11: }
```

(b) argouml/uml/ui/PropPanel.java

図 9: タイプ 3 の関数クローン（ArgoUML）

```
 1: public void log(String message,int loglevel){
 2:     if (managingPc != null) {
 3:         managingPc.log(message,loglevel);
 4:     }else {
 5:         if (loglevel > Project.MSG_WARN) {
 6:             System.out.println(message);
 7:         }else {
 8:             System.err.println(message);
 9:         }
10:     }
11: }
```

(a) ant/util/ConcatFileInputStream.java（if 文を用いた条件分岐処理の実装）

```
1: public void log(String message,int loglevel){
2:     if (managingPc != null) {
3:         managingPc.log(message,loglevel);
4:     }else {
5:         (loglevel > Project.MSG_WARN ? System.out :
                                System.err).println(message);
6:     }
7: }
```

(b) ant/util/ConcatResourceInputStream.java（三項演算子を用いた条件分岐処理の実装）

図 10: 条件分岐処理の実装が異なるタイプ 4 の関数クローン（Apache Ant）

```
1:  static ap_conf_vector_t *
2:  create_default_per_dir_config(apr_pool_t *p)
3:  {
4:      void **conf_vector = apr_pcalloc(p, sizeof(void *) *
5:                               (total_modules +DYNAMIC_MODULE_LIMIT));
6:      module *modp;
7:
8:      for (modp = ap_top_module; modp; modp = modp->next) {
9:          dir_maker_func df = modp->create_dir_config;
10:         if (df)
11:             conf_vector[modp->module_index] = (*df)(p, NULL);
12:     }
13:     return (ap_conf_vector_t *)conf_vector;
14: }
```

(a) server/config.c（中間媒介変数を利用した実装）

```
1:  static ap_conf_vector_t *
2:  create_server_config(apr_pool_t *p, server_rec *s)
3:  {
4:      void **conf_vector = apr_pcalloc(p, sizeof(void *) *
5:                              (total_modules + DYNAMIC_MODULE_LIMIT));
6:      module *modp;
7:
8:      for (modp = ap_top_module; modp; modp = modp->next) {
9:          if (modp->create_server_config)
10:             conf_vector[modp->module_index]
11:                           = (*modp>create_server_config)(p, s);
12:     }
13:     return (ap_conf_vector_t *)conf_vector;
14: }
```

(b) server/config.c（中間媒介変数を利用しない実装）

図 11: 中間媒介変数の利用が異なるタイプ 4 の関数クローン（Apache HTTPD）

```
 1: int
 2: pg_strcasecmp(const char *s1, const char *s2)
 3: {
 4:     for (;;)
 5:     {
 6:         unsigned char ch1 = (unsigned char) *s1++;
 7:         unsigned char ch2 = (unsigned char) *s2++;
 8:         if (ch1 != ch2)
 9:         {
10:             if (ch1 >= 'A' && ch1 <= 'Z')
11:                 ch1 += 'a' - 'A';
12:             else if (IS_HIGHBIT_SET(ch1) && isupper(ch1))
13:                 ch1 = tolower(ch1);
14:
15:             if (ch2 >= 'A' && ch2 <= 'Z')
16:                 ch2 += 'a' - 'A';
17:             else if (IS_HIGHBIT_SET(ch2) && isupper(ch2))
18:                 ch2 = tolower(ch2);
19:
20:             if (ch1 != ch2)
21:                 return (int) ch1 - (int) ch2;
22          }
23:         if (ch1 == 0)
24:             break;
25:     }
26:     return 0;
27: }
```

(a) port/pgstrcasecmp.c（for 文を用いた繰り返し処理の実装）

```
 1: int
 2: pg_strncasecmp(const char *s1, const char *s2, size_t n)
 3: {
 4:     while (n-- > 0)
 5:     {
 6:         unsigned char ch1 = (unsigned char) *s1++;
 7:         unsigned char ch2 = (unsigned char) *s2++;
 8:         if (ch1 != ch2)
 9:         {
10:             if (ch1 >= 'A' && ch1 <= 'Z')
11:                 ch1 += 'a' - 'A';
12:             else if (IS_HIGHBIT_SET(ch1) && isupper(ch1))
13:                 ch1 = tolower(ch1);
14:
15:             if (ch2 >= 'A' && ch2 <= 'Z')
16:                 ch2 += 'a' - 'A';
17:             else if (IS_HIGHBIT_SET(ch2) && isupper(ch2))
18:                 ch2 = tolower(ch2);
19:
20:             if (ch1 != ch2)
21:                 return (int) ch1 - (int) ch2;
22          }
23:         if (ch1 == 0)
24:             break;
25:     }
26:     return 0;
27: }
```

(b) port/pgstrcasecmp.c（while 文を用いた繰り返し処理の実装）

図 12: 繰り返し処理の実装が異なるタイプ 4 の関数クローン（PostgreSQL）

```
1:  static const XML_Char *
2:  poolCopyStringN(STRING_POOL *pool, const XML_Char *s, int n)
3:  {
4:      if (!pool->ptr && !poolGrow(pool))
5:          return NULL;
6:      for (; n > 0; --n, s++) {
7:          if (!poolAppendChar(pool, *s))
8:              return NULL;
9:       }
10:      s = pool->start;
11:      poolFinish(pool);
12:      return s;
13: }
```

(a) Modules/expat/xmlparse.c（for 文を用いた繰り返し処理の実装）

```
1:  static const XML_Char * FASTCALL
2:  poolCopyString(STRING_POOL *pool, const XML_Char *s)
3:  {
4:      // ※例外処理漏れの可能性がある
5:      do {
6:          if (!poolAppendChar(pool, *s))
7:              return NULL;
8:      }while (*s++);
9:      s = pool->start;
10:     poolFinish(pool);
11:     return s;
12: }
```

(b) Modules/expat/xmlparse.c（do-while 文を用いた繰り返し処理の実装）

図 13: 繰り返し処理の実装が異なるタイプ 4 の関数クローン（Python）

```
1:  static PyObject *
2:  dequereviter_next(dequeiterobject *it)
3:  {
4:      PyObject *item;
5:      if (it->counter == 0)
6:          return NULL;
7:      if (it->deque->state != it->state) {
8:          it->counter = 0;
9:          PyErr_SetString(PyExc_RuntimeError,
10:                                         "deque mutated during iteration");
11:         return NULL;
12:     }
13:     assert (!(it->b == it->deque->leftblock &&
14:                                         it->index < it->deque->leftindex));
15:
16:     item = it->b->data[it->index];
17:     it->index--;
18:     it->counter--;
19:     if (it->index == -1 && it->counter > 0) {
20:         assert (it->b->leftlink != NULL);
21:         it->b = it->b->leftlink;
22          it->index = BLOCKLEN - 1;
23:     }
24:     Py_INCREF(item);
25:     return item;
26: }
```

(a) Modules/collectionsmodule.c

```
1:  static PyObject *
2:  dequeiter_next(dequeiterobject *it)
3:  {
4:      PyObject *item;
5:      if (it->deque->state != it->state) {
6:          it->counter = 0;
7:          PyErr_SetString(PyExc_RuntimeError,
8:                                          "deque mutated during iteration");
9:          return NULL;
10:     }
11:     if (it->counter == 0)
12:         return NULL;
13:     assert (!(it->b == it->deque->rightblock &&
14:                                 it->index > it->deque->rightindex));
15:
16:     item = it->b->data[it->index];
17:     it->index++;
18:     it->counter--;
19:     if (it->index == BLOCKLEN && it->counter > 0) {
20:         assert (it->b->rightlink != NULL);
21:         it->b = it->b->rightlink;
22          it->index = 0;
23:     }
24:     Py_INCREF(item);
25:     return item;
26: }
```

(b) Modules/collectionsmodule.c

図 14: 文の並び替えが存在するタイプ 4 の関数クローン（Python）

5　考察

本章では，3 章の提案手法，および，4 章の評価実験の結果について議論する．

5.1　本手法の有用性

本節では，検出精度と検出時間の観点から本手法の有用性について考察する．

5.1.1　検出精度

検出精度の評価では，2 つの Java プロジェクトに対して 90%以上の非常に高い適合率で関数クローンを検出することができた．既存手法との比較実験においても，3 つの C プロジェクトに対して，MeCC よりも高い適合率で関数クローンの検出を行うことを確認することができた．一般的に，コードクローン検出においては適合率が高いことは非常に重要であるため，検出精度の観点から，本手法は有用であると言える．

さらに，各タイプの関数クローンの検出数においても MeCC を上回っていることを確認することができた．MeCC の評価実験では，字句単位の検出手法，抽象構文木を用いた検出手法，プログラム依存グラフを用いた検出手法との比較を行っており，MeCC の方がより多くのタイプ 3 とタイプ 4 の関数クローンを検出することができることを示している．従って，タイプ 3 とタイプ 4 のコードクローンの検出数は MeCC 以外の他のコードクローン検出手法に比べても高いと言える．

4.3 節では，5 つのプロジェクトで検出することができた関数クローンの実例を紹介した．本手法で検出することができた関数クローンの中には，図 10 や図 13 のように，不具合を引き起こす危険性があるものを多く検出することができた．このようなバグは，コピーアンドペーストなどによって作られた関数クローンに対して，一貫した修正が行われなかったために発生したものであると考えられる．

表 7: Exploitable Bug と Code Smell の検出数の比較

		Exploitable Bug	Code Smell	全検出数に対する割合
本手法	Apache HTTPD	29	35	31.8%
	Python	27	25	28.9%
	PostgreSQL	39	53	25.6%
MeCC	Apache HTTPD	8	27	43.2%
	Python	26	23	51.6%
	PostgreSQL	21	20	40.2%

```
1:  APR_DECLARE(void *)
2:  apr_palloc_debug(apr_pool_t *pool, apr_size_t size,  const char  *file_line)
3:  {
4:      void *mem;
5:
6:      apr_pool_check_integrity(pool);
7:
8:      mem = pool_alloc(pool, size);
9:      memset(mem, 0, size);
10:
11: #if (APR_POOL_DEBUG & APR_POOL_DEBUG_VERBOSE_ALLOC)
12:         apr_pool_log_event(pool, "PALLOC", file_line, 1);
13: #endif
14:     /* (APR_POOL_DEBUG &  APR_POOL_DEBUG_VERBOSE_ALLOC) */
15:
16:     return mem;
17: }
```

(a) memset 関数の呼び出しが存在する

```
1:  APR_DECLARE(void *)
2:  apr_palloc_debug(apr_pool_t *pool, apr_size_t size,  const char  *file_line)
3:  {
4:      void *mem;
5:
6:      apr_pool_check_integrity(pool);
7:
8:      mem = pool_alloc(pool, size);
9:      //※ memset関数呼び出し漏れの可能性がある
10:
11: #if (APR_POOL_DEBUG & APR_POOL_DEBUG_VERBOSE_ALLOC)
12:         apr_pool_log_event(pool, "PALLOC", file_line, 1);
13: #endif
14:     /*(APR_POOL_DEBUG & APR_POOL_DEBUG_VERBOSE_ALLOC)*/
15:
16:     return mem;
17: }
```

(b) memset 関数の呼び出しが存在しない

図 15: Code Smell の例

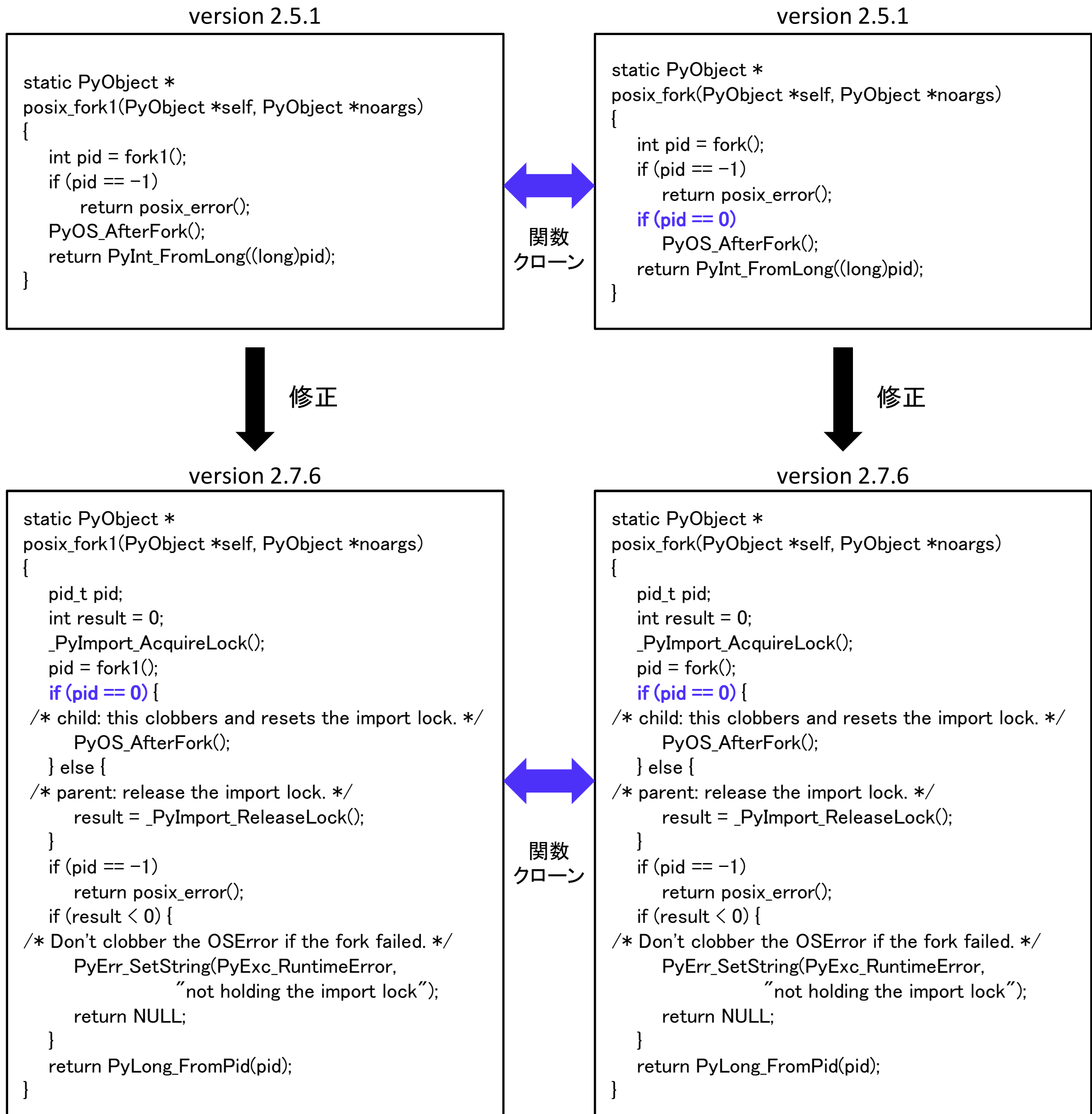

図 16: Exploitable Bug の不具合修正の実例

表 7 は，3 つの C プロジェクトに対して，不具合を引き起こす危険性や問題があるクローンセットがどれぐらい含まれているかを調べたものである．ここでは，それらのクローンセットを Exploitable Bug と Code Smell に手作業で分類した．それぞれの定義を以下に示す．

Exploitable Bug: 特定の値の場合における条件分岐が欠落していることが原因で，不具合を引き起こす危険性がある関数

Code Smell: Exploitable Bug に分類されないクローンセットで，修正が一貫していないことが原因で，不具合を引き起こす危険性がある関数

すなわち，Code Smell の条件を満たすクローンセットで，修正の一貫性が条件分岐処理に関係しているものは，全て Exploitable Bug に分類される．図 15 は Code Smell の例であり，図 15(b) では関数 memset の呼び出しが欠如している可能性がある．また，図 10 や図 13 は Exploitable Bug の例であり，例外に対する処理が欠如している可能性がある．このような関数は，クローンセット間で表現を一致させたり，リファクタリングを行う必要があると考えられる．

結果として，3 つのプロジェクトにおいて，全体で約 30%の関数クローンが，Exploitable Bug と Code Smell に含まれていることを確認することができた．Exploitable Bugs と Code Smell については，MeCC でも議論されている．MeCC と比較すると，Exploitable Bugs と Code Smell に含まれる関数クローンの割合は本手法の方が小さいが，本手法の方がより多くの Exploitable Bugs や Code Smell の候補を検出することができている．本手法で発見した Exploitable Bug の中には，3 つのプロジェクトの最新バージョンにおいて，4 つの関数に対して不具合の修正が行われていることを確認することができた．

図 16 は，Python において Exploitable Bug が修正された実例である．関数 fork1 と関数 fork は，生成した子プロセスのスレッドモデル毎に用意された関数クローンである．この例では，親プロセスと子プロセスの条件分岐による不具合が修正されている．バージョン 2.5.1 の関数 fork1 では，変数 pid の値が 0（子プロセス）の場合における if 文が欠如しており，子プロセスだけでなく親プロセスの場合も関数 PyOS_AfterFork を呼び出すことになる．しかし，バージョン 2.7.6 では if 文が追加されており，子プロセスのみが関数 PyOS_AfterFork を呼び出すことができる[10]．本手法では，このような条件分岐の欠落がある関数クローンも柔軟に検出することが可能である．

一方で，MeCC で検出することができ，本手法では検出できない関数クローンも多く存在した．特に，タイプ 2 の関数クローンについては本手法は弱い部分がある．意味のある識別

[10] この不具合はリビジョン 64452 で修正されている．http://svn.python.org/view/python/trunk/Modules/posixmodule.c?r1=64452&r2=64944

子名から頭文字だけに省略された場合は，コードクローンとして検出を行うことができない可能性が高い．字句単位の検出などの高速なタイプ 2 の検出ツールとの併用や，略語を考慮したワード抽出部分の改善など，さらに手法の改良を行っていく必要がある．

5.1.2 検出時間

検出時間の評価では，3 つの C プロジェクトに対して適用し，MeCC に比べて高速に関数クローンの検出を行うことを確認することができた．MeCC では，抽象的なメモリの状態を予測するための静的解析に膨大な時間がかかるため，検出時間が長くなってしまうと考えられる．

一方，本手法では，関数を特徴ベクトルに変換し，あらかじめ LSH アルゴリズムを用いたクラスタリングを行っている．そのため，アルゴリズムの時間計算量から，ソースコードに対してほぼ比例に近い時間で検出時間が増えていくと考えられる．評価実験からもソースコードの規模と検出時間との間に線形的な関係がある結果が得られた．従って，100 万行を超える大規模ソフトウェアに対しても十分適用することが可能であると考えられる．

5.2 本手法の拡張性

本手法の実装は，現在 Java 言語と C 言語にのみに対応している．しかし，本手法では字句解析を行い識別子と予約語の判定を行うだけで，関数を特徴ベクトルに変換することができる．従って，COBOL などの他の言語についても容易に本手法を適用することができると考えらえる．さらに，コンパイルができないソースコードに対しても関数クローンの検出を行うことが可能である．

また，本手法は現在，関数単位のコードクローンの検出のみを行っている．今後の拡張では，ブロック単位の検出など，さらに細かい粒度で検出を行うことも可能であると考えられる．

5.3 評価実験の妥当性

本実験では，2 つの Java プロジェクト，3 つの C プロジェクトのみに対して比較を行うことによって本手法の有用性を示した．しかし，今後，他の言語で実装された多くのプロジェクトに対して適用し，一般性を示す必要がある．また，検出時間の評価では，3 つの C プロジェクトに対してのみ比較・評価を行っている．従って，さらに規模が大きいオープンソースソフトウェアに対して適用を行い，検出時間やスケーラビリティの評価を行う必要がある．

また，評価実験の中では，手作業で関数クローンの正解集合・誤検出の判定や，タイプ毎の分類を行っている．そのため，データ公開などを行うことによって，結果の正確性を証明

する必要がある．しかし，ベンチマークやコーパスを用いた評価実験も行っているため，一部の検出精度においては信頼することができると言える．

6 まとめと今後の課題

本研究では，情報検索技術を利用した関数クローン検出手法の提案を行った．本手法では，ソースコード中の識別子や予約語に利用されている単語からワードを抽出し，各ワードに対して計算した重みを特徴量として各関数を特徴ベクトルに変換する．ワードの重みの計算には，情報検索技術で一般的に利用されている TF-IDF 法を利用した．そして，特徴ベクトル間の類似度を計算することによって，意味的に処理内容が類似した関数クローンの検出を行う．また，LSH アルゴリズムを用いてあらかじめ特徴ベクトルのクラスタリングを行うことによって，高速な関数クローンの検出を実現した．

評価実験では，2 つの Java プロジェクトに対して適用を行い，90%以上の適合率で関数クローンの検出を行うことを確認することができた．さらに，Kim らが開発したコードクローン検出ツールである MeCC と検出精度と検出時間の観点から比較を行った．3 つの C プロジェクトに対して適用した結果，本手法の方が高い精度で，より多くの関数クローンを検出することができた．また，本手法を用いた場合の検出にかかる時間は 5 分以下となり，MeCC よりも高速に関数クローンを検出することができた．さらに，ほぼ同一の処理を実装しているにも関わらず，条件分岐処理や繰り返し処理の実装が異なる関数クローンや，文が並び替えられている関数クローンを検出することができた．

今後の課題として，以下が挙げられる．

- ワードの重みの計算に，LSI（Latent Semantic Indexing）[2] や LDA（Latent Dirichlet Allocation）[5] を用いた手法と比較を行う必要がある．LSI を用いた場合，特徴ベクトルの次元を射影によって削減するため，検出精度が向上する可能性がある．また，LDA を用いた場合，ワードの持つ潜在的な意味を考慮することができるため，検出精度が向上する可能性がある．しかし，どちらの手法も TF-IDF 法に比べて計算コストが大きいため，検出精度と検出時間の観点から評価を行う必要がある．

- 類義語や同位語の関係を用いたワードのクラスタリングなどを行うことによって，リネームが行われた場合のタイプ 2 のコードクローンをより多く検出し，本手法の再現性を向上させる．

- 本手法は C 言語と Java 言語のみを対象としているが，COBOL などの他の言語についても対応させる必要がある．

- 他の大規模プロジェクトに対して適用し，本手法の有用性を評価する必要がある．さらに，MeCC 以外の様々なコードクローン検出手法との比較実験を行う必要がある．

謝辞

本研究を進めるにあたり，多くの方々に御指導及び御助言頂きました．ここに謝意を添えて御名前を記させて頂きます.

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 井上克郎 教授には，ご多忙にも関わらず，本研究に関して個別の打ち合わせにも御参加して頂き，その中で常に本質的な御指導及び御助言を頂きました．また，本研究に限らず，学部から修士修了まで 3 年間充実した生活を送ることができたことは，先生の御指導と御人柄によるものと確信しております．心より深く感謝を申し上げます.

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，グループミーティングや中間報告を通して，本研究に関して常に適切な御指導及び御助言を頂きました．また，本研究に限らず，企業との打ち合わせなどにも積極的に参加させて頂き，大変貴重な経験をさせて頂きました．深く感謝を申し上げます.

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 石尾隆 助教には，中間報告などを通して，本研究に関する様々な論文をご紹介して頂き，逐一適切な御指導及び御助言を頂きました．本研究に限らず，幅広い知識が得られたことは，先生の御指導によるものと確信しております．深く感謝を申し上げます.

奈良先端科学技術大学院大学情報科学研究科ソフトウェア設計学講座 吉田則裕 助教には，本研究を進めるにあたって，積極的に打ち合わせを行って頂き，その中で終始適切な御指導及び御助言を頂きました．本研究に限らず，論文投稿時や学会発表に関して常に適切な御助言をして頂き，研究室に配属されてから 3 年間，大変お世話になりました．国際会議での発表など貴重な経験をさせて頂き，充実した生活を送ることができたことは，先生の御指導と御人柄によるものと確信しております．心より深く感謝を申し上げます.

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 崔恩瀞 氏には，本研究を進めるにあたって，積極的に打ち合わせに御参加して頂き，その中で終始適切な御指導及び御助言を頂きました．特に，論文投稿時や学会発表に関して終始適切な御助言をして頂き，大変お世話になりました．本研究に限らず，研究室に配属されてから 3 年間，研究や学生生活を送るにあたって，非常に熱心に御指導頂きました．心より深く感謝を申し上げます.

日本電気株式会社 岩崎新一 氏，三橋二彩子 氏，佐野建樹 氏，鈴木明彦 氏，前田直人 氏には，本研究に限らず，企業のソフトウェア開発者の観点から様々な御助言を頂きました．心より深く感謝を申し上げます.

最後に，その他様々な御指導及び御助言等を頂い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井上研究室の皆様に深く感謝いたします.

参考文献

[1] A. Andoni and P. Indyk. Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. *Communications of the ACM*, Vol. 51, No. 1, pp. 117–122, 2008.

[2] R. A. Baeza-Yates and B. A. Ribeiro-Neto. *Modern information retrieval: the concepts and technology behind Search (2nd Edition)*. Addison-Wesley Professional, 2011.

[3] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier. Clone detection using abstract syntax trees. In *Proceedings of the 14th International Conference on Software Maintenance*, ICSM '98, pp. 368–377, 1998.

[4] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. Merlo. Comparison and evaluation of clone detection tools. *IEEE Transactions on Software Engineering*, Vol. 33, No. 9, pp. 577–591, 2007.

[5] D. M. Blei, A. Y. Ng, and M. I. Jordan. Latent dirichlet allocation. *The Journal of Machine Learning Research*, Vol. 3, pp. 993–1022, 2003.

[6] S. Ducasse, O. Nierstrasz, and M. Rieger. On the effectiveness of clone detection by string matching: research articles. *the Journal of Software Maintenance and Evolution*, Vol. 18, No. 1, pp. 37–58, 2006.

[7] S. Ducasse, M. Rieger, and S. Demeyer. A language independent approach for detecting duplicated code. In *Proceedings of the 15th International Conference on Software Maintenance*, ICSM '99, pp. 109–118, 1999.

[8] M. Fowler. *Refactoring: improving the design of existing code*. Addison-Wesley Longman Publishing Co., Inc., 1999.

[9] 肥後芳樹, 神谷年洋, 楠本真二, 井上克郎. コードクローンを対象としたリファクタリング支援環境. 電子情報通信学会論文誌, Vol. 88, No. 2, pp. 186–195, 2005.

[10] 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, Vol. J91-D, No. 6, pp. 1465–1481, 2008.

[11] 肥後芳樹, 楠本真二. プログラム依存グラフを用いたコードクローン検出法の改善と評価. 情報処理学会論文誌, Vol. 51, No. 12, pp. 2149–2168, 2010.

[12] P. Indyk and R. Motwani. Approximate nearest neighbors: towards removing the curse of dimensionality. In *Proceedings of the 30th annual ACM Symposium on Theory of Computing*, STOC '98, pp. 604–613, 1998.

[13] L. Jiang, G. Misherghi, Z. Su, and S. 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.

[14] J. H. Johnson. Substring matching for clone detection and change tracking. In *Proceedings of the 10th International Conference on Software Maintenance*, ICSM '94, pp. 120–126, 1994.

[15] T. Kamiya, S. Kusumoto, and K. 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] H. Kim, Y. Jung, S. Kim, and K. Yi. Mecc: memory comparison-based clone detector. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pp. 301–310, 2011.

[17] R. Komondoor and S. Horwitz. Using slicing to identify duplication in source code. In *Proceedings of the 8th International Symposium on Static Analysis*, SAS '01, pp. 40–56, 2001.

[18] K. A. Kontogiannis. Evaluation experiments on the detection of programming patterns using software metrics. In *Proceedings of the 4th Working Conference on Reverse Enginnering*, WCRE '97, pp. 44–55, 1997.

[19] K. A. Kontogiannis, R. Demori, E. Merlo, M. Galler, and M. Bernstein. Pattern matching for clone and concept detection. *Automated Software Engineering*, Vol. 3, pp. 77–108, 1996.

[20] J. Krinke. Identifying similar code with program dependence graphs. In *Proceedings of the 8th Working Conference on Reverse Engineering*, WCRE '01, pp. 301–309, 2001.

[21] Z. Li, S. Lu, S. Myagmar, and Y. Zhou. CP-Miner: finding copy-paste and related bugs in large-scale software code. *IEEE Transactions on Software Engineering*, Vol. 32, No. 3, pp. 176–192, 2006.

[22] J. Mayrand, C. Leblanc, and E. Merlo. Experiment on the automatic detection of function clones in a software system using metrics. In *Proceedings of the 12th International Conference on Software Maintenance*, ICSM '96, pp. 244–253, 1996.

[23] D. Rattan, R. Bhatia, and M. Singh. Software clone detection: a systematic review. *Information and Software Technology*, Vol. 55, pp. 1165–1199, 2013.

[24] C. K. Roy, J. R. Cordy, and R. 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.

[25] 佐々木裕介, 山本哲男, 早瀬康裕, 井上克郎. 大規模ソフトウェアシステムを対象としたファイルクローンの検出. 電子情報通信学会論文誌, Vol. J94-D, No. 8, pp. 1423–1433, 2011.

[26] E. Tempero. Towards a curated collection of code clones. In *Proceedings of the 7th International Workshop on Software Clones*, IWSC '13, pp. 53–59, 2013.

[27] Y. Yamanaka, E. Choi, N. Yoshida, K. Inoue, and T. Sano. Applying clone change notification system into an industrial development process. In *Proceedings of the 21th International Conference on Program Comprehension*, ICPC '13, pp. 199–206, 2013.

[28] Y. Yuan and Y. Guo. CMCD: count matrix based code clone detection. In *Proceedings of the 8th Asia Pacific Software Engineering Conference*, APSEC '11, pp. 250–257, 2011.