特別研究報告

題目

TF-IDF 法と LSH アルゴリズムを用いた コードブロック単位のクローン検出法

指導教員
井上 克郎 教授

報告者
横井 一輝

平成 29 年 2 月 14 日

大阪大学 基礎工学部 情報科学科

平成 28 年度 特別研究報告

TF-IDF 法と LSH アルゴリズムを用いたコードブロック単位のクローン検出法

横井 一輝

内容梗概

ソフトウェアの保守工程における問題の 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，ソースコード中に含まれる互いに一致または類似した部分を持つコード片のことであり，主に既存コード片のコピーアンドペーストが原因で生成される．コードクローンに対する保守作業の 1 つとして，コードクローンの集約がある．これにより，ソフトウェアの保守性や可読性の向上が可能となる．

山中らは，情報検索技術に基づいて関数単位のコードクローンを検出する手法を提案し，検出精度と検出時間の観点からその有用性が確認されている．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．しかし，関数単位のコードクローン検出では，長い関数の一部にコードクローンが含まれた場合に，検出漏れが生じる可能性がある．

そこで本研究では，関数単位より検出粒度を小さくした，コードブロック単位のコードクローン検出法を提案する．ここではコードブロックを，関数と，関数内部の if，for，while 文等の中括弧で囲まれた部分と定義する．本手法では，ソースコードに対して構文解析を行い，コードブロックの抽出を行う．その後，抽出した各コードブロックに対して情報検索技術の 1 つである 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，コードブロック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，コードブロック単位のクローン検出を行う．また，本手法では，特徴ベクトル間の類似度を求める際に，計算の並列化を行った．そして，特徴ベクトルが疎である点を利用し，非 0 要素のみを保持することで空間計算量の削減を行った．また，LSH（Locality-Sensitive Hashing）アルゴリズムの一種であり，メモリ使用量削減の改良を行った Multi-Probe LSH を用いて特徴ベクトルをクラスタリングすることで，少ないメモリ使用量で検出の高速化を実現した．

評価実験では，3 つの C 言語のプロジェクトに対して，既存のコードクローン検出手法との比較評価を行った．その結果，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 まえがき **4**

2 背景 **6**

2.1 コードクローン 6

2.1.1 コードクローンの定義 6

2.1.2 コードクローン検出 7

2.2 関数クローン検出法 8

2.3 関数クローン検出法の問題点 10

3 提案する検出手法 **12**

3.1 用語の定義 13

3.1.1 コードブロック 13

3.1.2 ワード 14

3.1.3 ブロッククローン 14

3.2 コードブロックの抽出 15

3.3 特徴ベクトルの計算 17

3.4 計算処理の並列化 17

3.5 特徴ベクトルの実装方法 18

3.6 特徴ベクトルのクラスタリング 18

4 評価実験 **20**

4.1 ベンチマークの作成方法 20

4.2 検出精度の指標の定義 22

4.3 関数クローン検出法と CCFinder との比較 23

4.4 ブロッククローンの実例 26

4.5 考察 31

5 まとめと今後の課題 **33**

謝辞 **34**

参考文献 **35**

1　まえがき

ソフトウェアの保守工程における大きな問題の 1 つとして，コードクローンが指摘されている [4]．コードクローンとは，ソースコード中に含まれる互いに一致または類似した部分を持つコード片のことであり，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．コードクローンに対する様々な保守や管理の方法が提案されているが，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．そこで，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コードクローン検出手法が提案されている [7, 16]．

山中らは情報検索技術 [3] を利用することによって，意味的に処理が類似した関数単位のコードクローン（関数クローン）を検出する手法を提案した [19]．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なコードクローンが多く検出されることがある [20]．一方，関数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できる．山中らの手法では，情報検索技術の 1 つである TF-IDF 法 [3] 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，特徴ベクトル間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 [9] を用いて，特徴ベクトルをクラスタリングすることにより，検出の高速化を行っている．

しかし，山中らの手法では検出粒度が関数単位のため，長い関数内の一部にコードクローンが含まれた場合，検出漏れが生じる可能性がある．このような検出漏れを減らすためには，関数単位より小さい粒度でコードクローンの検出を行うべきである．そこで，本研究ではコードブロック単位のコードクローン（ブロッククローン）を検出する手法を提案する．ここではコードブロックを，関数と，関数内部の if，for，while 文等の中括弧で囲まれた部分と定義する．本手法では，まずソースコードに対して構文解析を行い，コードブロックの抽出を行う．その後，抽出した各コードブロックに対して TF-IDF 法を用いて各コードブロックを特徴ベクトルに変換し，特徴ベクトル間の類似度を計算することでブロッククローンの検出を行う．また，本手法では検出の高速化とメモリ使用量の削減を行った．検出の高速化の手法として，コードブロックの特徴ベクトルと類似度を求める際に，計算処理の並列化を行った．メモリ使用量の削減の手法としては，TF-IDF 法を用いた特徴ベクトルが疎である特性を利用した．疎なベクトル（疎ベクトル）とは，ほとんどの要素が 0 のベクトルのことである．疎ベクトルの場合，非 0 要素のみを保持することで空間計算量の削減ができる．また，Multi-Probe LSH[14] を使用して特徴ベクトルのクラスタリングを行った．Multi-Probe

LSH とは，従来の LSH のメモリ使用量が多いという問題点の改良を行ったアルゴリズムである.（以降，単に“LSH”と表記した場合は LSH アルゴリズム全般を指し，LSH アルゴリズムを区別する際は“Multi-Probe LSH”，“従来の LSH”という表記を用いる）このことで，より少ないメモリ使用量で高速な検出を実現した.

評価実験では，関数クローン検出法 [19] と CCFinder[11] の 2 つと検出精度と検出時間の観点から比較を行った．CCFinder は神谷らが開発したコードクローン検出ツールであり，字句単位のコードクローン検出が可能である．3 つの C 言語のプロジェクトに対して適用した結果，本手法が総合的に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，本手法の検出にかかる時間は 3 分以下となり，関数クローン検出法と CCFinder よりも高速にコード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，本研究で提案するブロッククローン検出法について述べる．4 章では，本手法の評価実験について述べる．最後に，5 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，および山中らの関数クローン検出法と，その問題点について述べる．

2.1　コードクローン

コードクローンとは，ソースコード中に含まれる互いに一致または類似した部分を持つコード片のことであり，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 [7]．コードクローンの主な発生要因は，既存のソースコードのコピーアンドペーストによる再利用である．類似した処理を行うソースコードを書く際には，一から書くより既存のソースコードを再利用することが多い．他の発生要因としては，定型処理による発生，コード自動生成ツールによる発生，偶然の一致による発生等も挙げられる [4]．また，互いにコードクローンになるコード片の対をクローンペアと呼び，クローンペアにおいて推移関係が成り立つコードクローンの集合をクローンセットと呼ぶ．

2.1.1　コードクローンの定義

コードクローンには，普遍的定義は存在しない．Roy らはコードクローンの定義として，コードクローン間の違いの度合いに基づき以下の 4 つのタイプに分類している [17]．

タイプ 1

空白やタブの有無，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 の違いに加えて，変数名などのユーザー定義名，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 の違いに加えて，文の挿入や削除，変更などが行われているコードクローン．

タイプ 4

類似した処理を実行するが，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．

- 条件分岐処理や繰り返し処理などの制御構造の実装が異なる．
- 中間媒介変数の利用の有無が存在している．
- 文の並び替えが発生している．

2.1.2 コードクローン検出

コードクローンの存在がソフトウェアの保守を困難にするため，集約や同時修正などコードクローンに対する保守や管理が行われる [6, 8]．しかし，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も困難となる．そこで，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コードクローン検出手法が提案されている [7, 16]．現在まで，コードクローン検出の例として，字句単位の検出，抽象構文木を用いた検出，識別子名に基づく検出が提案されている．以下では，この 3 つの検出手法についての概要を説明する．

字句単位の検出

字句単位の検出では，検出の前処理としてソースコードを字句（トークン）の列に変換する．そして，閾値以上連続して一致しているトークンの部分列をコードクローンとして検出する．また，ソースコードを検出用の中間表現に変換する必要がないため，高速にコードクローンを検出可能である．

字句単位の検出手法の代表的なツールとして，神谷らが開発した CCFinder[11] がある．CCFinder では，字句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．この時，変数名や関数名などのユーザー定義名を特殊文字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．このツールはタイプ 2 までのコードクローンが検出可能である．

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 1 に抽象構文木の例を示す．

この検出手法は，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．この検出手法は，抽象構文木を構築する必要があるため，字句単位の検出に比べて時間的，空間的計算量は増えるが，実用的な方法として知られている．

抽象構文木を用いた検出手法の代表的なツールとして，Jiang らが開発した DECKARD[10] がある．DECKARD 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）[9] 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．このツールはタイプ 3 までのコードクローンが検出可能である．

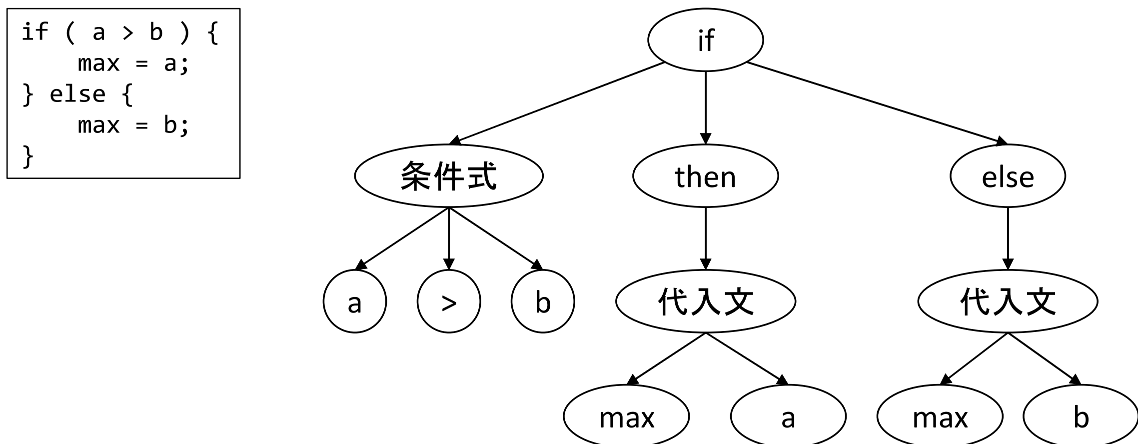

図 1: 抽象構文木の例

識別子名に基づく検出

識別子とは，ソースコード内の変数，関数などを識別するトークンのことである．識別子名に基づいた特徴メトリクスを用いて比較し，類似度が閾値以上となった箇所をコードクローンとして検出する．字句解析や構文解析を行うなどコンパイラの技術を利用した検出手法とは異なり，情報検索技術を利用して検出を行う．

識別子名に基づく検出の代表的な手法として，山中らが開発した関数クローン検出法 [19] がある．山中らの関数クローン検出法では，情報検索技術の 1 つである TF-IDF 法 [3] を利用し，関数を特徴ベクトルに変換し，意味的に処理が類似したコードクローンを検出する．この手法はタイプ 4 までのコードクローンが検出可能である．

2.2 関数クローン検出法

この節では，関数クローン検出法 [19] について説明する．

山中らは情報検索技術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を検出する手法を提案した．コード片単位でコードクローンの検出を行う場合，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恐れがある [20]．一方，関数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が検出できる．また関数クローン検出法は，タイプ 1 からタイプ 4 までのコードクローンを検出可能である．この手法は，まず，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．

- 変数や関数などにつけられた識別子名を構成する単語

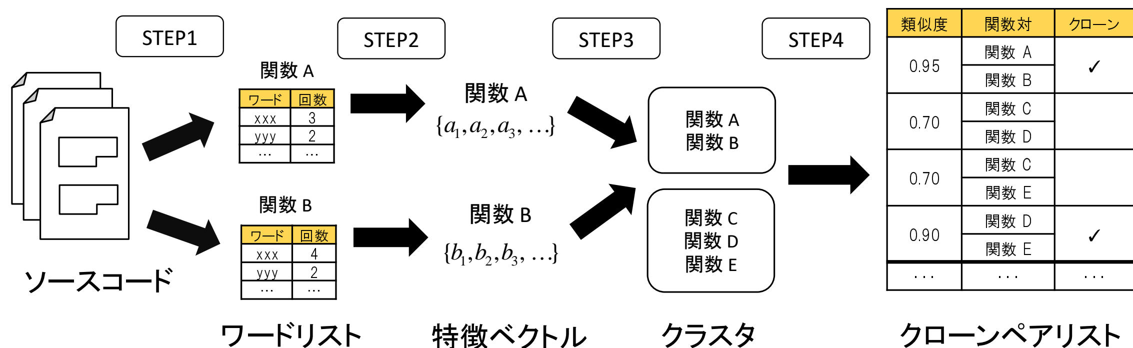

図 2: 関数クローン検出法の概要

- 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[9] 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．関数クローン検出法のコードクローン検出手法は以下の 4 つのステップに分けられる．図 2 はその関数クローン検出法の概要を表している．

STEP1:ワードの抽出

ソースコードの各関数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，新たに複数のワードとして分割する．分割方法は，ハイフン等の区切り記号による分割と，識別子名中の大文字になっているアルファベットによる分割である．また，繰り返し文等でよく用いられる i や j 等の 2 文字以下の識別子は，意味情報が込められていない変数として扱うために，すべて同一のメタワードとして認識する．

STEP2：特徴ベクトルの計算

STEP1 で抽出したワードに対し，TF-IDF 法 [3] を利用して重みを計算し，その値を特徴量として各関数を特徴ベクトルに変換する．TF-IDF 法とは，文書中のワードの出現頻度を表す tf 値と，文書全体の単語の希少さを表す idf 値の積で与えられる．この手法での tf 値と idf 値は以下の計算式で与えられる．

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

$$idf_X = \log \frac{\text{全関数の数}}{\text{ワード } X \text{ が出現する関数の数}}$$

この手法では，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ている．よって．各関数の特徴ベクトルの次元数はソースコード中に存在する全ワードの種類数となる．

STEP3：特徴ベクトルのクラスタリング

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，クローンペアになりうる候補を絞る．ここでは，近似最近傍探索アルゴリズムの一種である LSH[9] を用いて特徴ベクトルのクラスタリングを行う．LSH を用いることによって，クエリとして 1 つの特徴ベクトルを与えると，その特徴ベクトルと類似した特徴ベクトル集合のクラスタを取得でき，クローンペアになりうる候補を絞ることによって，検出時間を短縮している．なお，LSH が実装されている E2LSH[1][1]をこの手法では利用している．

STEP4：特徴ベクトルの類似度の計算

STEP3 で求めた関数の各クラスタの中で，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表す尺度であり，次元が V 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．

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

TF-IDF 法の計算式より，特徴量は常に正の値をとるため，コサイン類似度は 0 から 1 の範囲となる．コサイン類似度は大きいほど類似度が高いことを表す．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．

関数クローン検出法の検出精度の評価では，Tempero のベンチマーク [18] の 2 つの Java プロジェクト（Apache Ant，ArgoUML）に対して 90%以上と高い適合率で関数クローンを検出可能であることが分かった．また，タイプ 1 からタイプ 4 の関数クローンを検出できるツール MeCC[12] との比較を行い，MeCC より高速にコードクローンを検出することが確認されている．よって，検出精度と検出時間の観点から，関数クローン検出法の有用性が確認されている [19]．

2.3 関数クローン検出法の問題点

2.2 節では，関数クローン検出法の概要とその有用性について説明した．しかし，関数クローン検出法に対して以下の 2 つの問題点が挙げられる．

1 つ目は，関数単位の検出による問題点である．この手法は，関数全体ではなく，一部のみがコードクローンになっているものを検出することができない．例えば図 3 のように，長い関数内の一部にコードクローンが含まれる場合，検出漏れが生じる可能性がある．

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

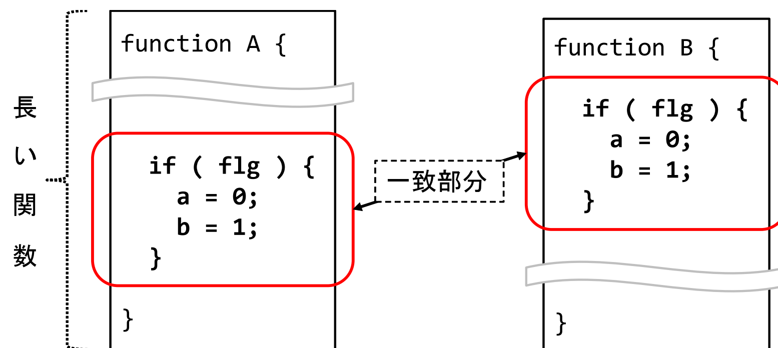

図 3: 長い関数内の一部にコードクローンを含む例

2 つ目は，メモリ使用量が多いという問題点である．関数を特徴ベクトルに変換する方法として関数クローン検出法は TF-IDF 法を用いているが，TF-IDF 法では全関数で出現したワードの種類数が次元数となるため，特徴ベクトルの次元数が非常に大きくなる傾向にある．特徴ベクトルは各関数に 1 つずつ与えるため，次元数の大きい特徴ベクトルを多数保持することになり，メモリ使用量が大きくなる．また，高速に検出を行うために LSH を用いてクラスタリングを行っているが，LSH は精度を上げるとメモリ使用量が大きくなるアルゴリズムである．よって，大規模プロジェクト（Linux Kernel 等）に対してコードクローンの検出を行った場合，メモリ不足で検出を完了できない恐れがある．

そこで，上の 2 つの問題点を踏まえた新しいコードクローン検出法の必要性が考えられる．本研究では関数単位より検出粒度を小さくした，コードブロック単位のコードクローン検出法を提案した．コードブロック単位で検出を行うことで，関数クローン検出法では検出できなかったコードクローンを検出が可能になる．また，空間計算量の少ない特徴ベクトルの実装方法や LSH に変更することで，大規模プロジェクトへの適用が可能となる．

3　提案する検出手法

本研究では，2.2 節で説明した山中らの関数クローン検出法を基に，コードブロック単位のクローン検出に対応するよう変更した手法（ブロッククローン検出法）を提案する．本手法の概要を図 4 に示す．本手法は主に以下の 5 つのステップで実行される．

STEP A

構文解析を行いソースコードから抽象構文木を生成し，生成した抽象構文木からコードブロック（3.1.1 節参照）を取り出す．

STEP B

STEP A で抽出した各コードブロックから，ワード（3.1.2 節参照）の抽出を行う．

STEP C

TF-IDF 法を利用し，STEP B で抽出したワードに重み付けを行い，各コードブロックを特徴ベクトルに変換する．

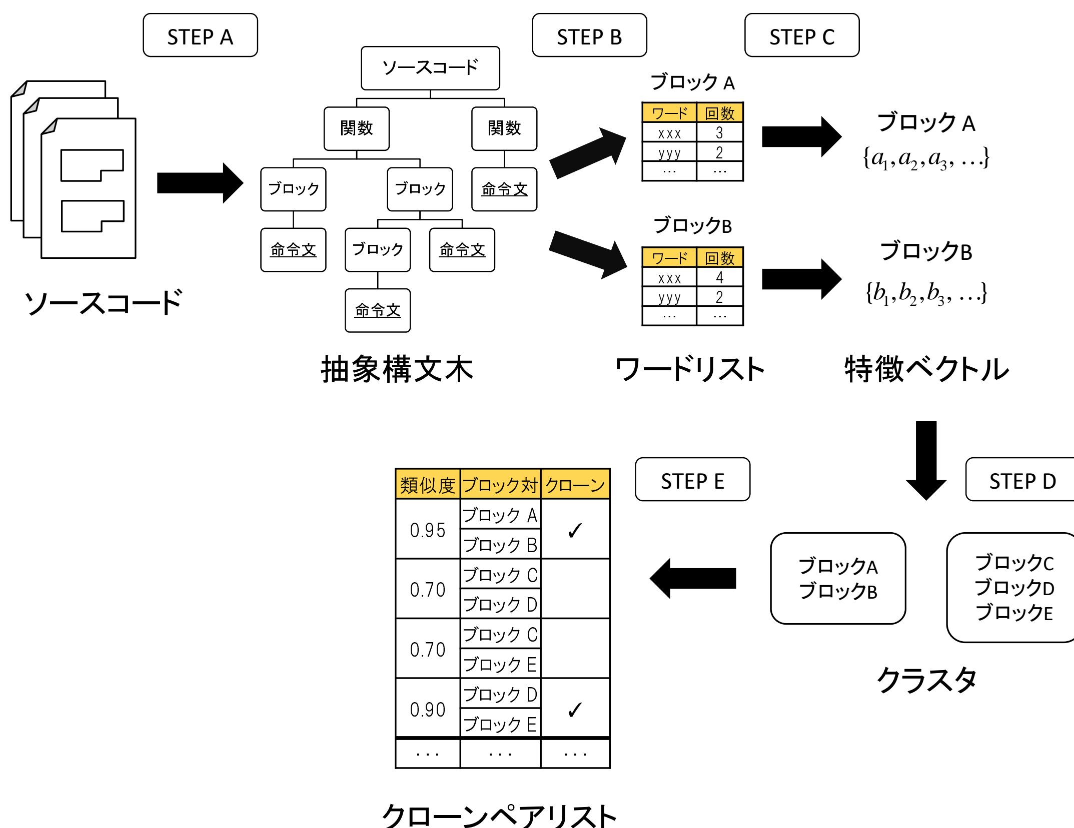

図 4: 本手法の概要図

STEP D

LSH を利用し，STEP C で求めた各コードブロックに対する特徴ベクトルのクラスタリングを行う．

STEP E

STEP D で求めたコードブロック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，ブロッククローン（3.1.3 節参照）を検出する．

図 2（関数クローン検出法の概要）と図 4 の比較から分かるように，本手法と関数クローン検出法の主な相違点は，コードクローンの検出粒度である．関数クローン検出法では関数の検出のみを行うが，本手法では特徴ベクトルの計算方法も変更し，関数と関数内のコードブロックの両方を検出する．

しかし，本手法は検出粒度を小さくすることで検出対象数が増え，それに伴い検出時間，メモリ使用量が増大する．そのため，計算処理の並列化や特徴ベクトルの実装方法の工夫を行うことで，検出時間の高速化，メモリ使用量の削減を行った．また，特徴ベクトルのクラスタリングでは，LSH[9] の一種であり，空間計算量の改良を行った Multi-Probe LSH[14] を適用した．

以降の節では，本手法で用いる用語の定義と，本手法と関数クローン検出法の相違点の詳細について説明する．

3.1 用語の定義

3.1.1 コードブロック

プログラミング言語において，複数の命令文を一括りにまとめたものをコードブロックという．多くのプログラミング言語では，コードブロックを入れ子構造にすることができ，変数のスコープとしての意味を持つことがある．

本手法では，以下の 2 つの条件のいずれかを満たすコードブロックを検出対象とする．対象言語は C 言語と Java 言語とする．

条件 1 関数の“{ }”で囲まれた範囲

条件 2 if，else，for，while，do-while，switch 文の“{ }”で囲まれた範囲

ただし，後に“{ }”が現れない単文の命令文はコードブロックとしての纏まりがないため検出対象としない．また図 5 の Block A に対する Block B や Block C のように，入れ子構造の内側のコードブロックも検出可能であり，検出対象を再帰的に探索する．

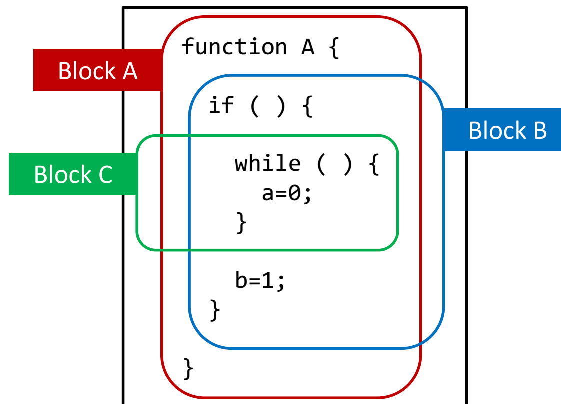

図 5: 入れ子構造にあるコードブロック

3.1.2 ワード

本手法では以下の条件 3，4 のいずれかを満たすものをワードとして定義する．

条件 3 予約語

条件 4 識別子名を構成する単語

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による分割

また，2 文字以下の識別子は，それらをまとめて同一のメタワードとして扱う．例えば，繰り返し文等でよく利用される i や j といった変数は，意味情報が込められていない変数として扱うためである．さらに，条件分岐に用いられる if や while，繰り返しに用いられる for や while 等の予約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区別はつけず，同一のワードとして扱う．

3.1.3 ブロッククローン

あるソースコード中に存在する 2 つのコードブロック CB_1，CB_2 が以下の条件 5，6 全て満たすとき，ペア (CB_1, CB_2) をブロッククローンペアと呼ぶ．

条件 5 コードブロック間の類似度が閾値以上

$$sim(CB_1, CB_2) \geq p \quad (0 \leq p \leq 1)$$

条件 6 コードブロック間に共通部分がない

$$CB_1 \cap CB_2 = \phi$$

CB_1，CB_2 それぞれを真に包含する如何なるコードブロックもブロッククローンペアでないとき，CB_1，CB_2 を極大ブロッククローンと呼ぶ．本手法では，極大ブロッククローンをブロッククローンと定義する．

条件 6 で示したように，ブロッククローンペアはコードブロック間に共通部分がないことが条件である．コードブロック間に共通部分が存在する場合，一方のコードブロックが他方を包含していることを示している．例えば，図 6 のコードブロック A と B は共通部分が存在し，包含関係にあるためブロッククローンペアでない．

また，極大ブロッククローンをブロッククローンと定義するとは，言い換えるとそれぞれ入れ子関係にあるコードブロックの類似度が閾値以上の場合，最も外側のコードブロックペアをブロッククローンとするという意味である．例えば，図 7 のコードブロック A と C，B と D それぞれの類似度が閾値以上となる場合，最も外側のコードブロック A と C をブロッククローンとする．

3.2 コードブロックの抽出

本手法では構文解析を行い，コードブロックの抽出を行う．コードブロック抽出の手法は以下の 6 つのステップに分けられる．本手法では，構文解析に ANTLR（ver. 4.5.3）[2]を利用している．

STEP I ソースコードに対して構文解析を行い，抽象構文木を生成する．

STEP II 抽象構文木から関数（3.1.1 章の条件 1 を満たすコードブロック）の部分木を取り出す．

STEP III STEP II で取り出した部分木を最も外側のコードブロックとして抽出する．

STEP IV STEP II で取り出した部分木から，コードブロック（3.1.1 章の条件 2 を満たすコードブロック）の部分木を取り出す．

STEP V STEP IV で取り出した部分木を入れ子関係にあるコードブロックとして抽出する．

STEP VI 以降，深さ優先探索で抽象構文木からコードブロックを抽出する．

[2]http://www.antlr.org/

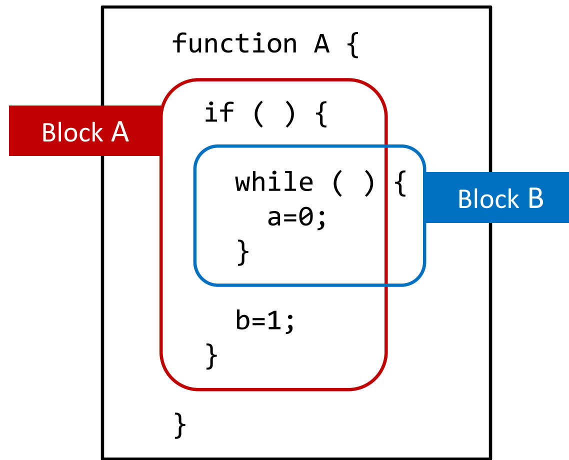

図 6: 共通部分があるコードブロックペア

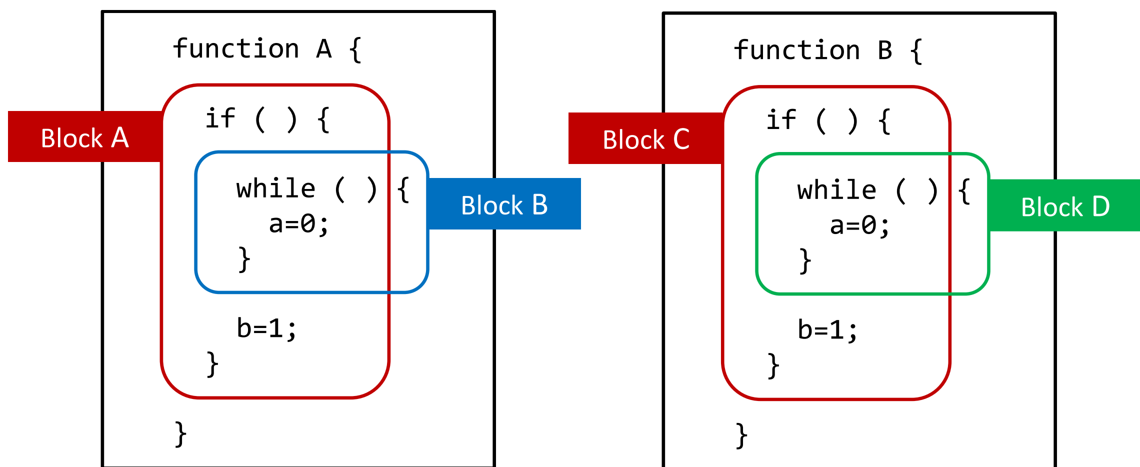

図 7: 極大コードブロックと重複したコードブロックペア

3.3 特徴ベクトルの計算

特徴ベクトルの計算では，ワードに対し TF-IDF 法 [3] を利用して重みを計算し，その値を特徴量として各コードブロックを特徴ベクトルに変換する．本手法では，tf 値はコードブロック中のワードの出現頻度を，idf 値はソースコード中のワードの希少さを表している．コードブロック Y 内におけるワード X の tf 値と idf 値は以下の計算式で与えられる．

$$tf_{X,Y} = \frac{\text{コードブロック}\,Y\,\text{内におけるワード}\,X\,\text{の出現回数}}{\text{コードブロック}\,Y\,\text{内における全ワードの出現回数の合計}} \quad (1)$$

$$idf_{X} = \log \frac{\text{全関数の数}}{\text{ワード}\,X\,\text{が出現する関数の数}} \quad (2)$$

関数クローン検出法と比較して，tf 値の求め方をコードブロック単位に変更したが，idf 値の求め方はコードブロック単位に変更せず関数単位のままである．なぜなら，コードブロック単位で idf 値を求めるとワードの重み付けに偏りが生じてしまうからである．あるワードがいくつのコードブロックに含まれるかは出現場所によって異なる．例えば，図 6 の関数内の変数 a はブロック A と B の 2 個のコードブロックに含まれるが，変数 b はブロック A のみにしか含まれない．そのため，ソースコード中の出現回数は同じにも関わらず，出現するコードブロックの数が異なってしまう．idf 値はワードの希少性に基づいて重み付けを行っているため，偏りを無くすために関数単位で求めることにした．

3.4 計算処理の並列化

計算処理では，特徴ベクトルを求める際と，特徴ベクトル間の類似度を求める際に並列化を行い，高速化を行っている．本手法では，Java の ExecutorService インターフェースを利用し，マルチスレッド処理を行うことで並列化を行っている．

特徴ベクトルを求める際の並列化

本手法では各コードブロックごとに特徴ベクトルを求める．上の式 (1) より，tf 値はコードブロック間で独立した値のため，コードブロックごとに処理を分割し並列計算で求める．上の式 (2) より，idf 値はコードブロック間に依存関係があるため，処理の分割を行わずあらかじめ求めておく．

類似度を求める際の並列化

類似度計算は各クラスタ内で行われるため，クラスタが異なればデータに依存性はない．よって，類似度は各クラスタごとに処理を分割し並列計算で求める．

1	0	2	0	0	0	0	3	0	0

図 8: 配列を用いた実装

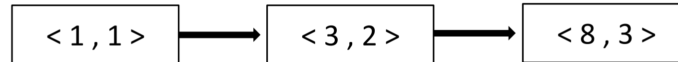

図 9: リストを用いた実装

3.5 特徴ベクトルの実装方法

TF-IDF を用いた特徴ベクトルは，非常に高次元かつ疎となる特性がある [2]．疎なベクトル（疎ベクトル）とは，ほとんどの要素が 0 のベクトルのことである．逆にほとんどの要素が 0 以外のベクトルを密なベクトル（密ベクトル）という．

ベクトルの実装方法として，以下の 2 種類の格納方式がある．

配列を用いた実装

ベクトルのすべての要素を格納する方式．配列を使用することで実装する．密ベクトルの場合効率の良い実装方法．

リストを用いた実装

ベクトルの 0 以外の要素とその添え字のペアを格納する方式.（添え字，値）のペアのリストを使用することで実装する．疎ベクトルの場合効率の良い実装方法.

実際に $\vec{v} = (1\ 0\ 2\ 0\ 0\ 0\ 0\ 3\ 0\ 0)$ を実装した場合，配列を用いた実装では図 8 のように全ての要素を格納する．それに対し，リストを用いた実装の場合は図 9 のように非 0 の値の 3 要素のみを格納する． リストを用いた実装の場合，添え字と値の両方を格納する必要があるため 1 要素当たりのコストは大きい．しかし，非 0 要素のみを格納するため，次元数に対して非 0 要素の割合が少ない疎ベクトルの場合は効率の良い実装方法である．したがって本手法では，TF-IDF を用いた特徴ベクトルが疎であるという特性を利用し，リストを用いて実装した．このことによって，メモリ使用量の削減を行った．

3.6 特徴ベクトルのクラスタリング

特徴ベクトルのクラスタリングとして，関数クローン検出法と同様に LSH を用いた．クラスタリングを行うことによって，クローンペアとなりうる候補を絞ることができ，高速なクローンペアの検出が可能となる．しかし，大規模のデータセットを LSH アルゴリズムを用いて高い精度で求めようとすると，メモリ使用量が非常に大きくなるという問題点がある [13, 14].

LSH のアルゴリズムは，空間的に近接した二点が同じハッシュ値になる確率が高くなるようなハッシュ関数を用い，同じハッシュ値を取る点を同じバケットに入れることでクラスタリングを行う．しかし LSH は確率的手法であるため，近接した二点が偶然に別のバケットに入る可能性がある．

従来の LSH では複数のハッシュ関数を用いることで確率的な誤差を少なくし精度を上げている．そのため，大規模なデータセットに対してはより多数のハッシュ関数が必要となり，これにより LSH のメモリ使用量の増大につながっている．

そこで，Lv らはメモリ使用量の改良を行った Multi-Probe LSH[14] を提案した．Multi-Probe LSH は，ある点が入るバケットだけでなく，空間的に近接したバケット群も調べるというものである．これにより，少ないハッシュ関数でも偶然別のバケットに入った点の見落としを防いでいる．実際に，192 次元のデータセットに対して同じ再現率 (0.90) を得るために，従来の LSH は 49 個のハッシュ関数が必要だったのに対し，Multi-Probe LSH は 3 個のハッシュ関数で検索時間をほぼ落とさずに達成したという結果が報告されている [14].

本手法ではメモリ使用量の改良を行うため，Multi-Probe LSH を用いてクラスタリングを行った．なお，Multi-Probe LSH の実装として FALCONN[2][3] ライブラリを利用している．

[3] https://falconn-lib.org/

4 評価実験

本章では，本研究で提案したブロッククローン検出法の評価実験について述べる．評価実験では，対象プロジェクトから作成したベンチマークをに対する検出精度と検出時間の観点から，関数クローン検出法と CCFinder[11] との比較を行った．CCFinder は井上研究室で開発され，国内外の企業・大学で使用されているため，比較対象に追加した．本実験で検出対象としたプロジェクトの一覧を表 1 に示す.

以降，4.1 節では，ベンチマークの作成方法について述べる．4.2 節では，検出精度の指標の定義について述べる．4.3 節では，関数クローン検出法と CCFinder との比較実験の結果を述べる．4.4 節では，本手法で検出することができたブロッククローンの実例を示す．最後に 4.5 節では，評価実験に対する考察を述べる．

4.1 ベンチマークの作成方法

ベンチマークの作成は以下の 3 ステップで行った．

1. 表 1 の 3 つのプロジェクトに対し，本手法，関数クローン検出法，CCFinder の 3 つの手法でコードクローンを検出.
2. それぞれの手法が各プロジェクトから検出したクローンペアから，30 個のクローンペアをランダムサンプリングし，合計 270 個のクローンペア集合を作成.
3. 2 で作成した 270 個のクローンペアに対し，目視で集約または同時修正の保守対象となるコードクローンかの判断を行い，ベンチマークを作成.

表 1: 検出対象プロジェクト

プロジェクト	バージョン	言語	規模
Apache HTTPD[4]	2.2.14	C/C++	343 KLOC
PostgreSQL[5]	8.5.1	C/C++	937 KLOC
Python[6]	2.5.1	C/C++	435 KLOC

[4] http://httpd.apache.org/
[5] http://www.postgresql.org/
[6] http://www.python.org/

なお，ベンチマークに客観性を持たせるため，アンケートにより第三者にコードクローンの判断を依頼した．アンケートの概要を以下に示す．

調査対象 以下の 3 名に依頼

- コードクローンの研究者 1 名
- コードクローンの研究に従事している大学院生 2 名

質問内容 集約または同時修正の保守対象となるか

回答方式 二択（はい/いいえ）

上記の質問を，検出結果からサンプリングした 270 個のクローンペアに対して行った．アンケートの集計結果を表 2 に示す．上記の質問で「はい」と回答した人数を賛成者数と表し，ランダムサンプリングした 30 個のクローンペアに対し，賛成者数ごとのクローンペアの個数を示している．

そして，過半数である 2 人以上が「はい」と回答したクローンペアを正解とし，本実験で用いる正解クローンペア集合としてベンチマークを作成した．作成したベンチマークの正解クローンペア数を表 3 に示す．本実験では，各プロジェクトの正解クローンペア数は，Apache HTPPD が 74 個，PostgreSQL が 46 個，Python が 62 個となった．

表 2: アンケート集計結果

検出手法	検出対象	賛成者数毎の度数分布				標本数
		3 人	2 人	1 人	0 人	
本手法	Apache HTTPD	13	14	2	1	30
	PostgreSQL	8	9	5	8	30
	Python	9	18	1	2	30
関数クローン検出法	Apache HTTPD	14	12	4	0	30
	PostgreSQL	13	12	2	3	30
	Python	4	5	6	15	30
CCFinder	Apache HTTPD	11	10	2	8	30
	PostgreSQL	1	3	15	11	30
	Python	2	24	0	4	30

表 3: ベンチマークの正解クローンペア数

検出手法	検出対象		
	Apache HTPPD	PostgreSQL	Python
本手法	27	17	27
関数クローン検出法	26	25	9
CCFinder	21	4	26
合計	74	46	62

4.2 検出精度の指標の定義

本実験では検出精度の指標として，適合率，再現率，F 値の 3 つの指標を用いて評価を行った．適合率と再現率と F 値の説明を以下に示す．

適合率

検出結果に対して本当に正しかった割合を指し，正確性に関する指標として用いられる．本実験では，各 30 個ずつランダムサンプリングしたクローンペア集合に対して，アンケートにて保守対象となるコードクローンと判断されたクローンペアの割合によって適合率を求めた．今回は過半数である 2 人以上がコードクローンと判断した場合を正解としている．

再現率

正解に対して実際に検出された割合を指し，網羅性に関する指標として用いられる．本実験では，アンケートによって作成したベンチマークの正解集合に対し，各手法が検出したクローンペアの割合によって再現率を求める．

F 値

適合率と再現率の総合的な評価として用いられ，適合率と再現率の調和平均によって求められる．

本評価実験における適合率（precision），再現率（recall），F 値（F-measure）は，それぞれ以下の式で求められる．

$$
\begin{aligned}
precision &= \frac{|CP_{bench} \cap CP_{sample}|}{|CP_{sample}|} \\
recall &= \frac{|CP_{result} \cap CP_{bench}|}{|CP_{bench}|} \\
F\text{-}measure &= \frac{2 * precision * recall}{precision + recall}
\end{aligned}
$$

CP_{bench} はベンチマークとして用意された正解クローンペア集合を表している．CP_{sample} はランダムサンプリングされたクローンペアの集合を表している．また，CP_{result} は各手法が検出した正解クローンペア集合を表している．

4.3　関数クローン検出法と CCFinder との比較

本節では，関数クローン検出法と CCFinder との比較実験の結果について述べる．本実験では，ベンチマークを用いた検出精度と検出時間の観点から比較を行った．検出精度の指標として，適合率，再現率，F 値の 3 つの指標を用いた．

適合率

アンケートによる適合率の評価結果を表 4 に示す．本手法では，Apache HTTPD と Python の 2 つのプロジェクトにおいて，それぞれ 27 個のクローンペアが保守対象のコードクローンと判断され，関数クローン検出法や CCFinder より高い適合率が得られた．また，PostgreSQL においては，17 個のクローンペアがコードクローンと判断され，CCFinder よりは高い適合率が得られたが，関数クローン検出法より低い適合率となった．

3 つの全てのプロジェクトより 30 個ずつサンプリングした合計 90 個のクローンペア集合に対して，本手法では 61 個のクローンペアが保守対象のコードクローンと判断された．よって適合率 0.68 と，関数クローン検出法と CCFinder より総合的に適合率が高いことが確認できた．

再現率

ベンチマークを用いた再現率の評価結果を表 5 に示す．本手法では，Apache HTTPD において 55 個，PostgreSQL において 40 個のクローンペアを正解集合から検出し，関数クローン検出法や CCFinder より高い再現率が得られた．また，Python においては，33 個のクローンペアを正解集合から検出し，関数クローン検出法よりは高い再現率が得られたが，CCFinder より低い再現率となった．

3 つの全てのプロジェクトより作成した合計 182 個の正解集合に対して，本手法では 128 個のクローンペアを検出できた．よって再現率 0.70 と，関数クローン検出法と CCFinder より総合的に再現率が高いことが確認できた．

F 値

本実験における F 値を表 6 に示す．本手法では，Apache HTTPD において F 値 0.81 と，関数クローン検出法や CCFinder より高い F 値が得られた．PostgreSQL においては F 値

0.69 と，CCFinder よりは高い F 値が得られたが，関数クローン検出法より低い F 値となった．また，Python においては F 値 0.67 と，関数クローン検出法よりは高い F 値が得られたが，CCFinder より低い F 値となった．

3 つの全てのプロジェクトの合計の F 値は 0.69 となり，関数クローン検出法と CCFinder より総合的に再現率が高いことが確認できた．

表 4: アンケートによる適合率の評価

検出手法	検出対象	適合率	正解数	標本数
本手法	Apache HTTPD	0.90	27	30
	PostgreSQL	0.57	17	30
	Python	0.90	27	30
	合計	0.68	61	90
関数クローン検出法	Apache HTTPD	0.87	26	30
	PostgreSQL	0.83	25	30
	Python	0.30	9	30
	合計	0.67	60	90
CCFinder	Apache HTTPD	0.70	21	30
	PostgreSQL	0.13	4	30
	Python	0.87	26	30
	合計	0.57	51	90

表 5: ベンチマークを用いた再現率の評価

検出手法	検出対象	再現率	検出数	正解集合数
本手法	Apache HTTPD	0.74	55	74
	PostgreSQL	0.87	40	46
	Python	0.53	33	62
	合計	0.70	128	182
関数クローン検出法	Apache HTTPD	0.53	39	74
	PostgreSQL	0.74	34	46
	Python	0.21	13	62
	合計	0.47	86	182
CCFinder	Apache HTTPD	0.55	41	74
	PostgreSQL	0.33	15	46
	Python	0.63	39	62
	合計	0.52	95	182

表 6: F 値の評価

検出手法	検出対象	F 値
本手法	Apache HTTPD	0.81
	PostgreSQL	0.69
	Python	0.67
	合計	0.69
関数クローン検出法	Apache HTTPD	0.66
	PostgreSQL	0.78
	Python	0.25
	合計	0.55
CCFinder	Apache HTTPD	0.62
	PostgreSQL	0.19
	Python	0.73
	合計	0.54

表 7: 検出時間の比較

検出対象	本手法	関数クローン検出法	CCFinder
Apache HTTPD	1m39s	4m07s	2m01s
PostgreSQL	2m27s	8m47s	5m30s
Python	1m15s	3m33s	3m10s

検出時間

検出時間の比較では，以下の環境で 3 つの手法をそれぞれ実行し，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 Windows 10 64-bit

CPU Intel Xeon 2.80GHz

コア 4（論理プロセッサ数：4）

メモリ 16.0GB

検出時間の比較結果を表 7 に示す．本手法では，検出対象全てのプロジェクトに対して 3 分以下でコードクローンを検出ができた．また，関数クローン検出法に対して 3～4 割程度，CCFinder に対して 4～8 割程度と，比較手法よりも短時間で検出することが確認できた．

4.4 ブロッククローンの実例

本節では，アンケートにて保守対象のコードクローンと判断されたクローンペアの中から，本手法によって検出したブロッククローンの実例を 4 つ示す．赤色のコード片と緑色のコード片が一致箇所を表している．

図 10 は，同じ関数内に存在するブロッククローンである．関数クローン検出法では関数単位の検出のため，同じ関数内でコピーアンドペーストを行うなどして関数内で発生したコードクローンを検出できなかったが，図 10 の例によって，本手法では検出可能であることを示せた．

図 11 は，長い関数（90 行以上）内の一部が一致するブロッククローンである．関数単位では異なる処理を行っているため，関数クローン検出法ではこのようなコードクローンを検出できなかった．しかし，図 11 の例によって，本手法では検出可能であることを示せた．

図 12 は，文の挿入が行われたタイプ 3 のブロッククローンである．関数単位では，文字列の大文字と小文字の変換を行うタイプ 4 のコードクローンであるが，関数クローン検出法

では実際に検出できなかった．しかし検出粒度を下げることで，本手法では図 12 の例のように検出可能であることを示せた．

図 13 は，ファイルの出力処理を行うタイプ 4 のブロッククローンである．どちらもファイルの入出力関連の処理を行う関数だが，図 13(b) の apr_file_sync 関数が，図 13(a) の apr_file_flush 関数に加えて独自の処理を行うため，関数クローン検出法では検出できなかった．しかし検出粒度を下げることで各関数の共通部分を見つけ出し，図 13 の例のように検出可能であることを示せた．

```
248: APU_DECLARE(apr_status_t) apr_rmm_destroy(apr_rmm_t *rmm)
249: {
250:     apr_status_t rv;
251:     rmm_block_t *blk;
252:
253:     if ((rv = APR_ANYLOCK_LOCK(&rmm->lock)) != APR_SUCCESS) {
254:         return rv;
255:     }
256:     /* Blast it all --- no going back :) */
257:     if (rmm->base->firstused) {
258:         apr_rmm_off_t this = rmm->base->firstused;
259:         do {
260:             blk = (rmm_block_t *)((char*)rmm->base + this);
261:             this = blk->next;
262:             blk->next = blk->prev = 0;
263:         } while (this);
264:         rmm->base->firstused = 0;
265:     }
266:     if (rmm->base->firstfree) {
267:         apr_rmm_off_t this = rmm->base->firstfree;
268:         do {
269:             blk = (rmm_block_t *)((char*)rmm->base + this);
270:             this = blk->next;
271:             blk->next = blk->prev = 0;
272:         } while (this);
273:         rmm->base->firstfree = 0;
274:     }
275:     rmm->base->abssize = 0;
276:     rmm->size = 0;
277:
278:     return APR_ANYLOCK_UNLOCK(&rmm->lock);
279: }
```

httpd/srclib/apr-util/misc/apr_rmm.c

図 10: 同じ関数内に存在するブロッククローン（タイプ 1）

```
1158: inleap(char **fields, int nfields)
1159: {

                          ...中略...

1212:         j = TM_JANUARY;
1213:         while (j != month)
1214:         {
1215:                 i = len_months[isleap(year)][j];
1216:                 dayoff = oadd(dayoff, eitol(i));
1217:                 ++j;
1218:         }
1219:         cp = fields[LP_DAY];
1220:         if (sscanf(cp, scheck(cp, "%d"), &day) != 1 ||

                          ...中略...

1277: }
```

(a) postgreSQL/src/timezone/zic.c （120 行）

```
2661: rpytime(const struct rule * rp, int wantedy)
2662: {

                          ...中略...

2689:         }
2690:         while (m != rp->r_month)
2691:         {
2692:                 i = len_months[isleap(y)][m];
2693:                 dayoff = oadd(dayoff, eitol(i));
2694:                 ++m;
2695:         }
2696:         i = rp->r_dayofmonth;
2697:         if (m == TM_FEBRUARY && i == 29 && !isleap(y))

                          ...中略...

2756: }
```

(b) postgreSQL/src/timezone/zic.c （96 行）

図 11: 非常に長い関数内の一部が一致したブロッククローン（タイプ 2）

```
690: strop_swapcase(PyObject *self, PyObject *args)
691: {
692:         char *s, *s_new;
693:         Py_ssize_t i, n;
694:         PyObject *newstr;
695:         int changed;
696:
697:         WARN;
698:         if (PyString_AsStringAndSize(args, &s, &n))
699:                 return NULL;
700:         newstr = PyString_FromStringAndSize(NULL, n);
701:         if (newstr == NULL)
702:                 return NULL;
703:         s_new = PyString_AsString(newstr);
704:         changed = 0;
705:         for (i = 0; i < n; i++) {
706:                 int c = Py_CHARMASK(*s++);
707:                 if (islower(c)) {
708:                         changed = 1;
709:                         *s_new = toupper(c);
710:                 }
711:                 else if (isupper(c)) {
712:                         changed = 1;
713:                         *s_new = tolower(c);
714:                 }
715:                 else
716:                         *s_new = c;
717:                 s_new++;
718:         }
719:         if (!changed) {
720:                 Py_DECREF(newstr);
721:                 Py_INCREF(args);
722:                 return args;
723:         }
724:         return newstr;
725: }
```

(a) python/Modules/stropmodule.c

```
2308: string_swapcase(PyStringObject *self)
2309: {
2310:         char *s = PyString_AS_STRING(self), *s_new;
2311:         Py_ssize_t i, n = PyString_GET_SIZE(self);
2312:         PyObject *newobj;
2313:
2314:         newobj = PyString_FromStringAndSize(NULL, n);
2315:         if (newobj == NULL)
2316:                 return NULL;
2317:         s_new = PyString_AsString(newobj);
2318:         for (i = 0; i < n; i++) {
2319:                 int c = Py_CHARMASK(*s++);
2320:                 if (islower(c)) {
2321:                         *s_new = toupper(c);
2322:                 }
2323:                 else if (isupper(c)) {
2324:                         *s_new = tolower(c);
2325:                 }
2326:                 else
2327:                         *s_new = c;
2328:                 s_new++;
2329:         }

2330:         return newobj;
2331: }
```

(b) python/Objects/stringobject.c

図 12: 文の挿入が行われたブロッククローン（タイプ 3）

```
334: APR_DECLARE(apr_status_t) apr_file_flush(apr_file_t *thefile)
335: {
336:     apr_status_t rv = APR_SUCCESS;
337:
338:     if (thefile->buffered) {
339:         file_lock(thefile);
340:         rv = apr_file_flush_locked(thefile);
341:         file_unlock(thefile);
342:     }
343:     /*
344:      * (コメント省略)
345:      */
346:     return rv;
347: }
```

(a) httpd/srclib/apr/file_io/unix/readwrite.c

```
349: APR_DECLARE(apr_status_t) apr_file_sync(apr_file_t *thefile)
350: {
351:     apr_status_t rv = APR_SUCCESS;
352:
353:     file_lock(thefile);
354:
355:     if (thefile->buffered) {
356:         rv = apr_file_flush_locked(thefile);
357:
358:         if (rv != APR_SUCCESS) {
359:             file_unlock(thefile);
360:             return rv;
361:         }
362:     }
363:
364:     if (fsync(thefile->filedes)) {
365:         rv = apr_get_os_error();
366:     }
367:
368:     file_unlock(thefile);
369:
370:     return rv;
371: }
```

(b) httpd/srclib/apr/file_io/unix/readwrite.c

図 13: ファイルの出力処理を行うブロッククローン（タイプ 4）

4.5 考察

本節では，提案手法，および評価実験の結果についての議論を行い，本手法の有用性，拡張性，および評価実験の妥当性についての考察を行う．

検出精度

山中らが行ったTemperoらのコーパスを用いた評価実験では，関数クローン検出法は適合率が90%を超えるという報告がされている[19]．また，沼田らが行ったバグを含むコード片に対する関数クローン検出手法とCCFinderの比較実験により，関数クローン検出法がCCFinderより高い適合率を得られることも報告されており，関数クローン検出法の有用性が確認できる．しかし同時に，関数クローン検出法はCCFinderより再現率が低いという報告もされている[15]．そこで，関数クローン検出法では関数単位の検出のために検出漏れが生じている可能性を考え，検出粒度をコードブロック単位に小さくした本手法を提案した．本実験の結果，3つのプロジェクトの合計に対して適合率，再現率，F値の3つの指標で関数クローン検出法やCCFinderより高い値が得られ，検出精度の観点で本手法の有用性を確認できた．

検出時間

本手法では並列可能な処理は並列計算を行うことで，検出の高速化を行った．また，Multi-Probe LSHを用いてクラスタリングを行うことで，より高速に検出を行うことができた．本実験の結果，比較手法より高速に検出できることが確認でき，検出時間の観点で本手法の有用性を確認できた．ただし，関数クローン検出法はパラメータ推定を行っているため，一度推定したパラメータを再利用することによって，より高速にコードクローンを検出できる可能性がある．

ブロッククローンの実例

ブロッククローンの実例によって，長い関数内の一部が一致するコードクローンや，同じ関数内に存在するコードクローンなど，関数クローン検出法では検出できなかったコードクローンを確認できた．よって，関数クローン検出法による検出漏れの削減を示せた．

本手法の拡張性

本手法の実装は，現在C言語とJava言語にのみ対応している．しかし，本手法ではANTLRを用いて構文解析を行っており，ANTLRにて構文解析を行うための文法ファイルが100種

類以上用意されていることから，C#や 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可能である．

また，本手法ではコードブロックを“{ }”に囲まれた範囲によって抽出している．しかし，字下げによるコードブロックの抽出や，ある一定行数のまとまりをコードブロックとして抽出するなど，抽出方法に他の手法を適用することも可能である．

評価実験の妥当性

本実験では，3 つの C 言語の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，一般性を示す必要性がある．また，今回はベンチマークの作成において，アンケートにて過半数である 2 人以上が保守対象となりうるコードクローンと判定した場合を正解として扱った．しかし，より正確性を上げるためには，意見が割れたクローンペアについて議論を行う必要性も考えられる．

5　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法を基に，より検出粒度を小さくしたブロッククローンの検出を行う手法を提案した．本手法では，構文解析を行いコードブロックの抽出を行い，コードブロック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する．そして，TF-IDF を利用して各ワードに対する重みを計算し，その重みを特徴量として各コードブロックを特徴ベクトルに変換する．その後，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ブロック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，高速なブロッククローンの検出を実現した．

評価実験では，3 つの C プロジェクトに対し，検出精度と検出時間の観点から，関数クローン検出法と CCFinder の 2 つの手法と比較を行った．その結果，本手法が総合的に高い精度で多くのコードクローンを検出できることが確認できた．また，検出時間は 3 分以下となり，関数クローン検出法と CCFinder より高速に関数クローン検出を行うことができた．さらに，関数クローン検出法では検出できなかったコードブロック単位のコード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）[3] や，LDA（Latent Dirichlet Allocation）[5] を用いた手法と比較を行う必要がある．LSI や LDA を用いた場合，ワードの潜在的な意味を考慮し，次元を削減することができる．意味的に類似する識別子を自動的にまとめることで，検出精度が向上する可能性がある．しかし，どちらも TF-IDF と比較し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比較を行う必要がある．

- 本手法は C 言語と Java 言語のみを対象としているが，他の言語についても対応させる必要がある．本手法では“{ }”によってコードブロックの定義を行っているが，Pascal のように“begin ”と“end ”で囲む言語や，Python のように字下げによりコードブロックを示す言語があるため，コードブロックの定義や抽出方法を再考する必要がある．

- 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 など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

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，研究に関する適切な御指導及び御助言を賜りました．井上 教授の御指導及び御助言のおかげで本論文を完成させることができました．井上 教授に心より深く感謝いたします．

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，研究の各段階において多くの御助言を賜りました．多くの御指導及び御助言を頂いた松下 准教授に心より深く感謝いたします．

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，研究においてたくさんの貴重な御意見を賜りました．多くの御助言を頂いた石尾 助教に心より深く感謝いたします．

名古屋大学大学院情報科学研究科附属組込みシステム研究センター / 情報システム学専攻 吉田 則裕 准教授には，研究に関する直接の御指導を賜りました．常に適切な御指導及び御助言を頂いたことにより，本論文を完成することができました．吉田 准教授に心より深く感謝いたします．

大阪大学大学院情報科学研究科 春名 修介 特任教授には，常に適切な御指導及び御助言を賜りました．多くの御助言を頂いた 春名 特任教授に心より深く感謝いたします．

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研究室 崔 恩瀞 助教には，研究に関する多くの貴重な御助言を賜り，評価実験の御協力もしていただきました．崔 恩瀞 助教に心より深く感謝いたします．

大阪大学大学院情報科学研究科 Kula Raula Gaikovina 特任助教には，研究において貴重な御意見を賜りました．Kula Raula Gaikovina 特任助教に心より深く感謝いたします．

日本電気株式会社 前田 直人 氏，渋谷 健介 氏には，研究に関して企業のソフトウェア研究者の観点から多くの御意見を頂きました．心より深く感謝いたします．

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 中村 勇太 氏，沼田 聖也 氏，石津 卓也 氏，堤 祥吾 氏には，研究に関する相談に乗っていただき，また本論文の修正や評価実験に御協力していただくなど研究の様々な場面で御助力いただきました．有意義な研究室生活を送りながら本論文を完成させることができたことは先輩方のおかげであると，心より深く感謝いたします．

最後に，御指導，御助言を頂き私を支えてくださった， 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，心より深く感謝いたします．

参考文献

[1] Alexandr Andoni and Piotr Indyk. Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. In *Proceeding of the 47th Annual IEEE Symposium on Foundations of Computer Science*, pp. 459–468, 2006.

[2] Alexandr Andoni, Piotr Indyk, Thijs Laarhoven, Ilya Razenshteyn, and Ludwig Schmidt. Practical and optimal lsh for angular distance. In *Proceedings of the 28th International Conference on Neural Information Processing Systems*, pp. 1225–1233, 2015.

[3] Ricardo Baeza-Yates and Berthier Ribeiro-Neto. *Modern information retrieval: The concepts and technology behind search.* Addison-Wesley, 2011.

[4] Ira D. Baxter, Andrew Yahin, Leonardo Moura, Marcelo Sant Anna, and Lorraine Bier. Clone detection using abstract syntax trees. In *Proceedings of International Conference on Software Maintenance*, pp. 368–377, 1998.

[5] David M Blei, Andrew Y Ng, and Michael I Jordan. Latent dirichlet allocation. *Journal of machine Learning research*, Vol. 3, No. Jan, pp. 993–1022, 2003.

[6] Martin Fowler. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley Longman Publishing Co., Inc., 1997.

[7] 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.

[8] 肥後芳樹, 吉田則裕. コードクローンを対象としたリファクタリング. コンピュータソフトウェア, Vol. 28, No. 4, pp. 43–56, 2011.

[9] Piotr Indyk and Rajeev Motwani. Approximate nearest neighbors: towards removing the curse of dimensionality. In *Proceedings of the 30th annual ACM symposium on Theory of computing*, pp. 604–613, 1998.

[10] 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*, pp. 96–105, 2007.

[11] 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.

[12] Heejung Kim, Yungbum Jung, Sunghun Kim, and Kwankeun Yi. Mecc: Memory comparison-based clone detector. In *Proceedings of the 33rd International Conference on Software Engineering*, pp. 301–310, 2011.

[13] 古賀久志. ハッシュを用いた類似検索技術とその応用. 電子情報通信学会 基礎・境界ソサイエティ Fundamentals Review, Vol. 7, No. 3, pp. 256–268, 2014.

[14] Qin Lv, William Josephson, Zhe Wang, Moses Charikar, and Kai Li. Multi-probe lsh: efficient indexing for high-dimensional similarity search. In *Proceedings of the 33rd international conference on Very large data bases*, pp. 950–961, 2007.

[15] Seiya Numata, Norihiro Yoshida, Eunjong Choi, and Katsuro Inoue. *On the Effectiveness of Vector-Based Approach for Supporting Simultaneous Editing of Software Clones*, pp. 560–567. Springer International Publishing, 2016.

[16] Dhavleesh Rattan, Rajesh Bhatia, and Maninder Singh. Software clone detection: A systematic review. *Information and Software Technology*, Vol. 55, No. 7, pp. 1165–1199, 2013.

[17] Chanchal K. Roy, James R. 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.

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

[19] 山中裕樹, 崔恩瀞, 吉田則裕, 井上克郎. 情報検索技術に基づく高速な関数クローン検出. 情報処理学会論文誌, Vol. 55, No. 10, pp. 2245–2255, 2014.

[20] Yuki Yamanaka, Eunjong Choi, Norihiro Yoshida, Katsuro Inoue, and Tateki Sano. Applying clone change notification system into an industrial development process. In *Proceedings of the 21st International Conference on Program Comprehension*, pp. 199–206, 2013.