修士学位論文

題目

プログラム依存グラフを用いたコードクローンのインクリメンタルな検出手法

指導教員
楠本　真二　教授

報告者
西野　稔

平成 23 年 2 月 7 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

平成 22 年度 修士学位論文

プログラム依存グラフを用いたコードクローンのインクリメンタルな検出手法

西野　稔

内容梗概

ソフトウェアの保守作業を困難にする要因の 1 つとして，コードクローンが指摘されている．コードクローンの存在により，ソフトウェアの理解が難化し，不具合等に対する一貫した修正が困難になる．そのような影響を軽減するには，ソフトウェアに存在するコードクローンの把握が必要であるため，コードクローンの自動検出を行う手法が多数提案されている．特に近年は，インクリメンタルな検出手法が注目されている．この種の手法は，対象ソフトウェアに対する初回の検出処理は全てのファイルを解析するため長時間を要するものの，2 度目以降は過去に解析した情報を再利用することで，短時間でコードクローンを検出することができる．現在までに，行単位や字句単位のコードクローンをインクリメンタルに検出する手法が提案されている．しかし，プログラム依存グラフを用いたコードクローン検出においては，インクリメンタルな処理は行われていない．本研究では，他の手法で検出できないコードクローンを発見できる一方で，処理時間が他の手法に比べて長くなるプログラム依存グラフを用いた検出を高速化し，実用性を高めるため，インクリメンタルな処理を行う手法を提案する．その際にはそれに適したデータ構造が必要になるため，新たに “ユニット” というデータ構造を定義している．本手法を実装したツールを用いた実験の結果，大規模なソフトウェアに対し，インクリメンタルな処理が可能な状況において，非常に短い時間でコードクローンを検出することができた．また，検出されたコードクローンはプログラム依存グラフを用いた既存の検出手法と極めて類似しており，検出結果も適切であるといえる．

主な用語

コードクローン, プログラム依存グラフ, データベース, 大規模ソフトウェア

目 次

図目次

表 目 次

1　まえがき

ソフトウェアの保守作業を困難にする要因の 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，ソースコード中の互いに一致または類似したコード片であり，コピーアンドペーストなどによって発生するといわれている．ソフトウェアの開発においてコピーアンドペーストを繰り返し行うと，類似した機能が複数の箇所に存在することになり，ソフトウェアに対する理解が難しくなると考えられる．さらに，そのようなコード片の 1 つに不具合が発生した場合や，環境の変化などにより変更が必要になった場合には，それらのコード片の全てに対して修正が必要になる可能性が高い．

保守作業におけるコードクローンの影響を低減するには，コードクローンに関する情報を把握する必要がある．しかし，大規模で複雑なソフトウェアシステムにおいて，コードクローンの情報を手作業で収集し，常に最新の情報に保つのは不可能である．そこで，コードクローンを自動的に検出するためのさまざまな手法が提案されている．それらは行単位，字句単位など，互いに異なったコードクローンの定義を持ち，検出結果がそれぞれ異なる．その中の 1 種として，プログラム依存グラフを用いた手法がある．この手法の特徴は，ソースコード中での順序が入れ換わっているコードクローンなど，他の手法では検出できないコードクローンを検出できることである．その一方で，グラフに対する処理は計算量が多くなるため，他の手法と比較して検出にかかる時間が非常に長いという弱点がある．

また，近年はコードクローンの自動検出において，インクリメンタルな処理を行う手法が注目されている．インクリメンタルな検出手法とは，2 回目以降の検出処理の時間を短縮するため，検出処理で得た情報をデータベースなどに登録して永続化し，その後の検出処理において再利用する手法である．今日までに，行単位や字句単位でインクリメンタルにコードクローン検出を行う手法が提案されている．

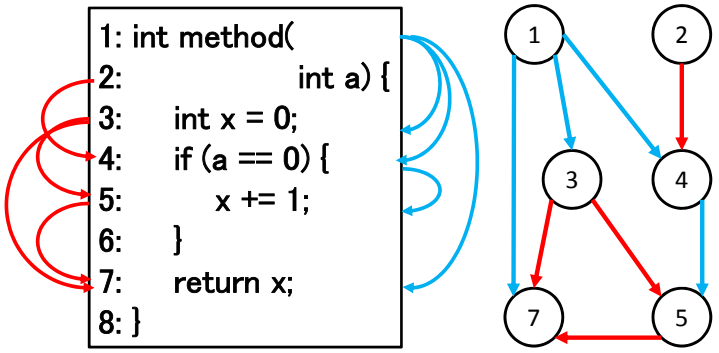
本研究では，プログラム依存グラフを用いたコードクローンの検出において，インクリメンタルな処理を行う手法を提案する．プログラム依存グラフを用いた手法は処理時間が長いため，大規模なソフトウェアに対する適用は現実的でないと考えられているが，それに対してインクリメンタルな手法を用いることで，初回の検出には長時間を要するものの，2 回目以降の検出時間を短縮することを考えた．このような処理を行うためには，永続化に適したデータ構造が必要であるため，プログラム依存グラフの辺と頂点を統一して扱うことのできる “ユニット” というデータ構造を定義した．そして，ユニットを用いたクローンペアの定義を行い，それを検出するアルゴリズムを提案し，さらに高速化のための様々な工夫を施して実装している．

インクリメンタルな処理は様々な状況において効果的である．例えばソフトウェアの開発履歴における，コードクローンの変化を調査する場合，従来の手法では各リビジョンに対してソースコード全体を解析する必要があった．しかし，インクリメンタルな処理を行えば解析対象は各リビジョンで更新のあった部分のみに絞られ，検出時間を大幅に削減することができる．また，ソフトウェアの一部

分に対するコードクローンを調査する場合は，従来の手法でも検出結果を保存しておき必要に応じて確認するという方法を取ることができるものの，そのソフトウェアに対する更新があった時点で全体に対する再検出が必要になるため，継続的に更新が行われているソフトウェアに対しては効率が悪い．それに対してインクリメンタルな手法では，更新のあった時点で更新部分に対してのみ解析を行って解析情報を蓄積し，コードクローンの情報が必要な時に蓄積した解析情報を用いて検出処理を行うという方法を取る事で，全体を解析する場合と比較して極めて高速な検出が可能である．本手法はこのような状況において，プログラム依存グラフを用いた手法が求められる場合に，有効であると考えられる．

提案手法を実装したツールを用いた評価実験の結果，実規模ソフトウェアに対する開発履歴の調査において，既存の字句単位のコードクローン検出ツールと同等以上の時間で，処理を終了することが確認された．また，100 万行以上の大規模ソフトウェアにおいて 1 つのファイルに対するコードクローンを検出する場合，平均して 2 秒から 6 秒で検出できることが確認された．さらに，提案手法と他の検出手法の検出結果を比較し，提案手法がプログラム依存グラフを用いた既存手法に対して，95% の再現率と 73% の適合率をもつことが分かった．

以降，2 章で基礎知識や既存研究について紹介し，3 章で提案手法を説明する．4 章では提案手法の実装とその際の工夫点を述べ，その実装を用いた評価実験を 5 章で説明する．最後に，6 章でまとめと今後の課題を記す．



(a) ソースコード　　(b) プログラム依存グラフ

図 1: プログラム依存グラフの例

2 準備

2.1 プログラム依存グラフ

プログラム依存グラフとは，プログラム内の文の間に存在する依存関係を表す有向グラフである [1]．プログラム依存グラフの頂点はプログラム内の文であり，辺で結ばれた頂点間に依存関係があることを表す．依存関係としては，以下の二種類が用いられる．

データ依存 文 A で値を定義した変数を，再定義なしに文 B で用いる可能性がある場合，文 A から文 B へのデータ依存があるという．

制御依存 条件式である文 A の評価結果によって，文 B を実行するか否かが直接決まる場合，文 A から文 B への制御依存があるという．

図 1 は，ソースコードとそれに対するプログラム依存グラフの例である．ただし，データ依存を赤色，制御依存は青色で表し，頂点と文の対応は行番号により示している．ソースコードの 1 行目は文ではないが，プログラム依存グラフ上にはメソッドの入り口を表す頂点として追加される．また，2 行目の引数についても，変数の入力として追加される．このソースコードでは，2 行目の引数 `a` を 4 行目の文 `a == 0` で用いている．そのため，これらの文の間にデータ依存があり，対応する頂点の間にデータ依存辺が引かれる．また，5 行目の `x += 1` において定義された変数 `x` が，再定義なしに 7 行目の `return x` で用いられることは明らかであるが，4 行目の評価結果によっては 3 行目の `int x = 0` で定義された変数 `x` を 7 行目において再定義なしに用いることになる．そのため，3 行目および 5 行目から 7 行目へのデータ依存があり，各頂点の間にデータ依存辺が引かれる．また，制御依存については，4 行目の評価結果によって 5 行目を実行するか否かが直接決まるため，これらの文の間に制御依存がある．また，メソッドの入り口にあたる 1 行目からは，このメソッドが呼び出された際に無条件で実行される各行に対して，制御依存があると判断する．

なお，有効グラフ G は一般的に，頂点集合 V，辺集合 E，および辺から頂点の組への写像 $f : E \to V \times V$ を用いて，$G = (f, V, E)$ と表される．これを用いて，あるプログラムに対するプログラム依存グラフを $G = (f, V, E)$ と記述すると，$v \in V$ はプログラム中の文と対応し，$e \in E$ は依存関係と対応する．f は $v_1, v_2 \in V$ に対して v_1 から v_2 への依存関係 $e \in E$ があるとき，$f(e) = (v_1, v_2)$ となる写像である．

2.2 コードクローン

コードクローンとは，ソースコード中の互いに一致または類似したコード片である [2]．コードクローンとなる 2 つのコード片の組をクローンペアといい，コードクローンとなるコード片の集合をクローンセットという．

コードクローンが発生する原因として，以下のようなものが挙げられる [3][4][5].

既存コードのコピーとペーストによる再利用

近年のソフトウェア設計手法を利用すれば，構造化や再利用可能な設計が可能である．しかし，一からコードを書くよりも既存コードを流用して部分的な変更を加える方が信頼性が高いこともあり，実際にはコピーアンドペーストによる再利用が多く存在する．

コーディングスタイル

規則的に必要なコードはスタイルとして同じように記述される場合がある．例えば，ユーザインターフェース処理を記述するコードなどである．

定型処理

定義上簡単で頻繁に用いられる処理はコードクローンになる傾向がある．例えば，キューの挿入処理や，データ構造へのアクセス処理などである．

プログラミング言語の機能的な制限

抽象データ型や，ローカル変数を用いられない場合には，同じようなアルゴリズムを持った処理を繰り返し書かなくてはならないことがある．

パフォーマンス改善

リアルタイムシステムなど時間制約のあるシステムにおいて，インライン展開などの機能が提供されていない場合に，特定のコード片を意図的に繰り返し書くことによってパフォーマンスの改善を図ることがある．

コード生成ツールの生成コード

コード生成ツールにおいて，類似した処理を目的としたコードの生成には，識別子名等の違いはあろうとも，あらかじめ決められたコードをベースにして自動的に生成されるため，類似したコードが生成される．

複数のプラットフォームに対応したコード

複数の OS(Linux, FreeBSD, HP-UX や AIX など) や CPU(i386 系, amd64 系, alpha や sparc64 など) に対応したソフトウェアは，各プラットフォーム用のコード部分に重複した処理が存在する傾向が強い．

偶然

偶然に，開発者が同一のコード片を書いてしまう場合もあるが，大きなコードクローンになる可能性は低い．

一般に，コードクローンはソフトウェアの保守性を悪化させる要因であると考えられている．例えば，あるクローンセットに含まれるコード片に不具合が発見された場合，同じクローンセットに含まれる他のコード片にも同様の不具合が存在し，修正が必要になる可能性がある．しかし，特に大規模なシステムにおいては，全てのコード片に対して一貫した修正を行うのは非常に困難な作業となる．門田らは，コードクローンとソースファイルの改版数の関係を調査し，多くのコードクローンを含むソースファイルや，巨大なコードクローンを含むソースファイルは，改版数が多くなる傾向があると述べている [6]．また，Lozano らはメソッド単位で同様の調査を行い，コードクローンの存在期間が長いメソッドは，保守コストが急激に増加すると述べている [7].

一方で，コードクローンを用いた開発が望ましい場合もあるとの報告もされている．例えば Kapser らは，ハードウェアドライバを作成する場合に，既存のドライバからのコピーアンドペーストが有効であると述べている [8]．また，佐野らは様々な言語および規模のソフトウェアに対して，行単位でコードクローンとコードクローンでない箇所の修正頻度の比較を行い，開発期間の長いソフトウェアに関してコードクローンに含まれる行の方が修正頻度が低いと報告している [9].

なお，コードクローンに関しては様々な研究が行われているが，それらはどれも異なった定義をもつ．つまり，コードクローンの厳密で普遍的な定義は存在しないといえる．

2.3 コードクローンの検出手法

コードクローンの情報を手作業で収集および管理するのは，現実的には困難である．そこで，コードクローンを自動で検出するための，さまざまな手法が提案されている．それらの手法はコードクローン検出の単位によって，大まかに以下の 5 つに分類することができる [2].

- 行単位の検出
- 字句単位の検出
- 抽象構文木を用いた検出
- プログラム依存グラフを用いた検出
- メトリクスなどを用いた検出

各分類について，提案されている既存手法と合わせて説明する．

行単位の検出

ソースコードを行単位で比較して，連続して重複する行をコードクローンとして出力する手法である．他の検出手法に比べて高速であるが，コーディングスタイルが異なる場合にコードクローンとして検出できないという欠点がある．この分類に属する手法として，Duccase らは各行を空白やタブを取り除いた上で比較するという，プログラミング言語に依存しない手法を提案している [10]．また，Baker は変数名などを特殊文字に置換した上で，接尾辞木 [11] を用いて行単位の比較を行うことにより．変数名などの違いを無視したコードクローンを高速に検出する手法を提案している [12]．Wettel らは，行単位のコードクローン検出結果のうち，近接する複数のコードクローンを 1 つにまとめることで，間にコードクローンにならない行が含まれる大きなコードクローンとして検出する手法を提案している [13].

字句単位の検出

検出の前処理としてソースコードを字句の列に変換し，連続して重複する字句の列をコードクローンとして出力する手法である．行単位の検出手法には劣るものの比較的高速であり，検出結果がコーディングスタイルに依存しないという特徴を持つ．Kamiya らは，接尾辞木を用いてトークン単位の検出を行う手法を提案し，多数の言語に対応した検出ツール CCFinder を開発している [4]．Li らは，ソースコードに対して字句解析と構文解析を行った後，文ごとにハッシュ値を計算してそのハッシュ値の列に対して頻出系列マイニング [14] を行う手法を提案し，検出ツール CP-Miner を開発している [15].

抽象構文木を用いた検出

検出の前処理としてソースコードから抽象構文木を構築し，抽象構文木上の同型の部分木をコードクローンとして検出する手法である．行単位や字句単位の検出手法と比較して，検出に必要な時間的および空間的なコストが高くなるものの，複数の関数にまたがるなど，プログラムの構造を無視したコードクローンを検出しないという特徴がある．Baxter らは，部分木の同型判定の調査に時間がかかることから，ハッシュ値を用いて調査対象を削減する手法を提案し，検出ツール CloneDR を開発している [3].

プログラム依存グラフを用いた検出

検出の前処理として，2.1 節で説明したプログラム依存グラフを構築し．グラフ上の同型部分グラフをコードクローンとして出力する手法である．図 2 に示すような順序の入れ換わったコードクローンなど，他の手法で検出できないコードクローンを検出できるが，プログラム依存グラフの構築や同型部分グラフの探索に高い計算コストを必要とする．Komondoor らは，ソースコード中の文を頂点とするプログラム依存グラフを構築し，プログラムスライス [1] を用いて同一のグラフ構造を発見する手法を提案している [16]．Krinke は，より粒度の高い，字句を頂

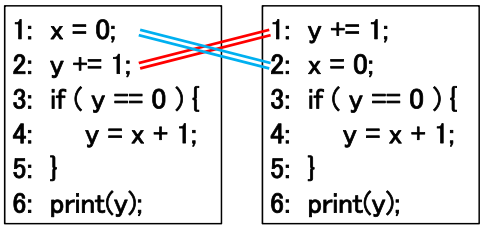


図 2: 順序入れ換わりコードクローン

点とするプログラム依存グラフを定義し，それを用いたコードクローン検出を行う手法を提案している [17].

メトリクスなどを用いた検出

プログラムのファイルやクラス，メソッドに対してメトリクスを計算し，その値が近似するものをコードクローンとして出力する手法である．Mayrand らは，関数に対してユーザ定義名やコーディングスタイルなど，21 種類のメトリクスを定義し，それらを用いて 8 段階の類似度を定義している [18].

また，この分類とは別に，近年インクリメンタルなコードクローン検出手法が注目されている．インクリメンタルな検出手法とは，2 回目以降の検出処理の時間を短縮するため，検出処理で得た情報をデータベースなどに蓄積し，その後の検出処理において再利用する手法である．このような手法は，特に大規模なソフトウェアに対して継続的にクローン検出を行う際に有効である．

Hummel らは，各行の内容と位置情報を蓄積して，同内容の行を検索することでコードクローンを検出する手法を提案し，検出ツール ConQAT に実装している [19]．この手法では，まずソースコード中の各行に対して，変数名などに対する置換を行った上で，ハッシュ値を計算する．そして，そのハッシュ値とファイル名や行番号を組にした情報を蓄積する．すると，ある行に対してコードクローンとなる行は，この情報をその行のハッシュ値を用いて検索することで，取得することができる．検出結果として出力するのは複数行からなるコードクローンであるが，それについてはあるコード片の各行とコードクローンになる行の集合から容易に構築することができる．

Göde らは，接尾辞木を改良し，情報の追加や削除が容易な汎用接尾辞木[1]というデータ構造を定義して，それを用いた字句単位のコードクローン検出を行う手法を提案している [20]．接尾辞木とは，文字列に対して定義される文字数と等しい個数の葉を持つ木構造であり，根からそれぞれの葉 i への経路が文字列の i 文字目から最後の文字までの部分文字列 (接尾部) に対応する．接尾辞木を用いることで，文字列中の繰り返し部分を高速に検出することができる．コードクローン検出においては，コードクローンがソースコード中の繰り返し部分であることから，プログラム全体を 1 つの文字列として接尾辞木を構築し，検出を行う手法が用いられている [12]．汎用接尾辞木は，1 つの文字列で

[1]Generalized Suffix Tree

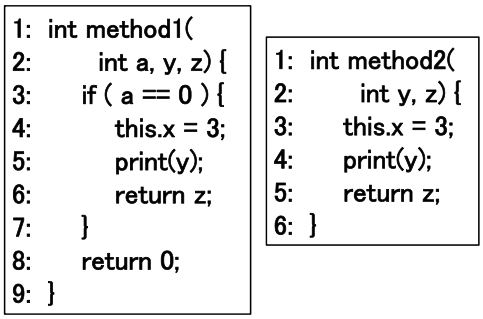


図 3: 連続コードクローン

はなく文字列の集合に対して定義され，根から葉への経路は各文字列における接尾部に対応する．各ファイルをそれぞれ 1 つの文字列として汎用接尾辞木を構築することにより，ファイルの追加や削除が容易になり，インクリメンタルな検出が可能になる一方，コードクローン検出の精度には影響を与えないことを示している．

Jiang らは，抽象構文木の部分木を配列表現に変換して，局所感度ハッシュアルゴリズム [21] を用いて類似する部分木を検出する手法を提案し，検出ツール DECKARD を開発している [22]．この手法では，部分木に含まれる字句を変数名やリテラルなどに分類し，各分類の個数を要素とする配列を，部分木の特徴配列[2]と定義している．局所感度ハッシュアルゴリズムは，配列間のある程度の違いを無視して同一か否かを判定することができるアルゴリズムであり，これを特徴配列に対して適用することで，各分類の字句の個数が類似している部分木同士を，コードクローンとして検出することができる．この手法は，多数の特徴配列の中から特定の特徴配列と類似するものを探し出す手法であるといえるため，特徴配列の情報を蓄積しておくことで，インクリメンタルな検出処理が可能であると考えられる．また，この手法に対する改良として，Gabel らはプログラム依存グラフの情報の一部を特徴配列に含めることで，意味的な類似性も考慮したコードクローンを検出する手法を提案している [23]．また Lee らは，アクセスの局所性なども考慮した効率的なインデックスを作成し，類似する部分木をより高速に発見する手法を提案している [24]．Lee らの改良により，この手法のインクリメンタルな検出に対する適用可能性が，さらに高まったといえる．

なお，プログラム依存グラフを用いたインクリメンタルな検出手法は提案されておらず，メトリクスなどを用いた手法は，その値を記録しておくだけで，容易にインクリメンタルな処理が可能である．

2.4 本手法で用いるプログラム依存グラフ

Bellon らの調査によると，2.3 節で紹介した文献 [17] のプログラム依存グラフを用いた手法は，他の手法で検出できないコードクローンを検出できる一方，図 3 に示すような連続コードクローンの検出能力が劣る [25]．同じくプログラム依存グラフを用いた文献 [16] の手法も，同様の問題点をも

[2]Characteristic Vector

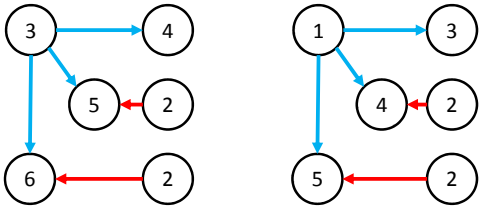


図 4: 図 3 のプログラム依存グラフの部分グラフ

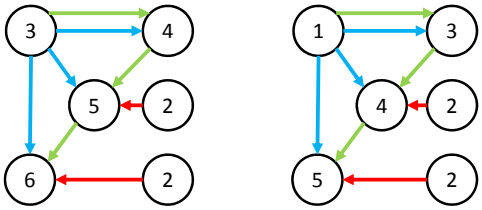


図 5: 実行依存辺を付加したプログラム依存グラフ

つことが知られている．図 3 において，左のコード片の 4 行目から 6 行目と右のコード片の 3 行目から 5 行目は明らかに同一であるため，行単位や字句単位の手法であれば，これらをコードクローンとして検出する．一方，プログラム依存グラフを用いた手法では，次の理由により検出できない．説明に当たり，このソースコードに対するプログラム依存グラフの一部を図 4 に示す．これらのグラフは一見すると同一の構造を持つが，左のグラフの頂点 3 が条件文 `a == 0` を表すのに対し，右のグラフの頂点 1 はメソッドの入り口である．そのため，これらのグラフは異なる構造となり，文献 [16] や文献 [17] の手法ではこのようなコードクローンを検出することができない．

この問題に対して肥後らは，プログラム依存グラフに実行依存という新たな依存関係を定義することで連続コードクローンの検出を可能にした手法を提案し，検出ツール Scorpio を開発している [26]．実行依存は，以下のように定義される．

実行依存 文 A が実行された直後に，文 B が実行される可能性がある場合，文 A から文 B への実行依存があるという．

図 4 のプログラム依存グラフに対して実行依存を付加したものを，図 5 に示す．これにより，左のコード片の 4 行目から 6 行目と右のコード片の 3 行目から 5 行目が，それぞれ依存関係を持つようになり，この 3 つの頂点による共通の構造を持つ部分グラフが構成される．よって，これらのコード片の組をコードクローンとして出力することができる．

本研究では，文献 [16] と同様に，ソースコード中の文を頂点としたメソッド内部の構造を表すプログラム依存グラフを用いてコードクローン検出を行う．また，連続コードクローンを検出するため，依存関係としてデータ依存と制御依存に加え，実行依存を用いる．さらに，プログラム依存グラ

フは孤立頂点を持たないものと仮定する．これは，例外としてメソッドの引数がメソッド内部で一切用いられていない場合，その引数に対応する頂点が孤立頂点となるが，このような頂点をコードクローンとして検出する必要はないと考えられるためである．以降，単にプログラム依存グラフと表記した場合，このようなグラフを指すものとする．なお，肥後らはさらに，繰り返し部分から検出されるコードクローンは人間が必要とすることがほとんどないことを示し [27]，そのようなコードクローンを出力する頂点を集約することで，計算コストを削減する手法を提案している [28]．しかし，本研究では現時点でこのような頂点の集約は行っていない．

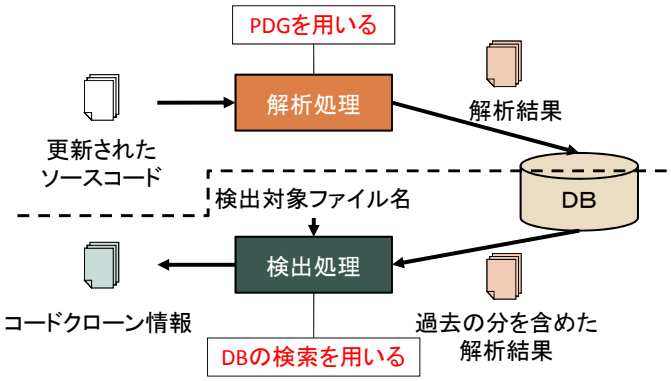


図 6: 提案手法の概要

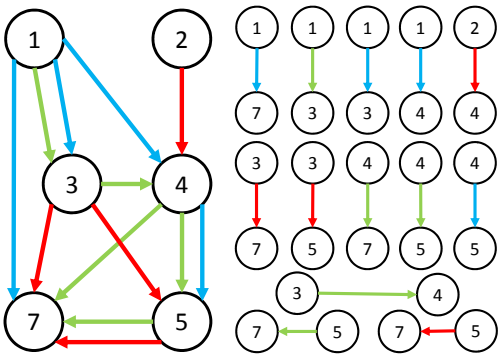
3　提案手法

この章では，提案手法について述べる．まず 3.1 節で概要を述べ，3.2 節で本研究で用いる定義および定理を提示し，3.3 節と 3.4 節で詳細部分の説明を行う．最後に 3.5 節で，この手法の時間計算量について議論する．

3.1　提案手法の概要

2.3 項で紹介したように，現在までに様々なコードクローン検出手法が提案されている．本手法はその中で，プログラム依存グラフを用いた検出手法に分類される．プログラム依存グラフを用いた手法は，他の手法で検出できないコードクローンが検出できるものの，検出処理に長時間を要する点が課題である．これに関して，文献 [28] のような時間短縮の方法も研究されているが，プログラム依存グラフを用いる限り，コストの削減には限界があると考えられる．そこで本研究では，プログラム依存グラフを用いた検出において，他の分類では行われているインクリメンタルな検出手法を提案する．これにより，初回の検出に限っては高い計算コストを要するものの，2 回目以降の検出における処理時間を大幅に短縮できると考えられる．

図 6 に，提案手法の概要を示す．この手法は，解析処理と検出処理に分けられ，解析処理でソースコードを解析した結果を用いて，検出処理でコードクローンの情報を構築する．処理間の解析結果の受け渡しにはデータベースを用いる．これにより，解析処理の結果を永続的に保持することで，インクリメンタルな検出が実現できる．それと同時に，検出処理においてデータベースの検索を適切に用いることで，処理の対象が削減されるため，処理時間の短縮が期待できる．データベースに登録する情報としては，プログラム依存グラフの辺と頂点を統一して扱うための，“ユニット” というデータ構造を用いる．ユニットに関する定義と，これを用いたクローンペアの定義は，3.2 節で行う．



(a) プログラム依存グラフ　　(b) ユニット

図 7: ユニットの例

3.2　定義および定理

提案手法を説明する上で，用いる用語を定義し，それを用いたコードクローンの形式的な定義を示す．まず，3.1 節で述べたユニットというデータ構造を定義する．

定義 3.1 (ユニット) プログラム依存グラフ $G = (f, V, E)$ において，$e \in E$ および $f(e) = (v_1, v_2)$ となる (v_1, v_2) に対して，

$$unit(e, v_1, v_2) := (f, \{v_1, v_2\}, \{e\})$$

図 7(a) は図 1(b) に示したプログラム依存グラフに実行依存を付加したものである．これに含まれるユニットを図 7(b) に示す．このように，ユニットとはプログラム依存グラフ G において，辺 e とその始点である頂点 v_1，終点である頂点 v_2 により構成される部分グラフであり，プログラム依存グラフの各辺に対して定義される．また，プログラム依存グラフ G はユニットの集合 $\{u_1, u_2, \ldots, u_n\}$ を用いて表すことができ，G の孤立頂点を持たない任意の部分グラフ S は，$\{u_1, u_2, \ldots, u_n\}$ の部分集合により表される．

ユニットの隣接関係を以下のように定義する．

定義 3.2 (ユニットの隣接) ユニット $u_1 = unit(e_1, v_1, v_2)$，$u_2 = unit(e_2, v_3, v_4)$ に対して，

$$neighbor(u_1, u_2) := \bigvee_{i \in \{1,2\}, j \in \{3,4\}} v_i = v_j$$

すなわち，いずれかの頂点を共有するユニット u_1 と u_2 を隣接関係にあるといい，$neighbor(u_1, u_2)$ はこのようなユニットの組に対してのみ真となる．

次に，ユニット間の経路を定義する．ただし，一般に任意の 2 つのユニット間の経路は複数存在するため，集合として定義する．

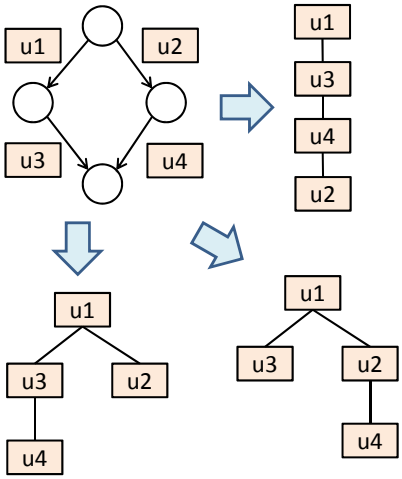


図 8: ユニット木の例

定義 3.3 (ユニット間の経路)

$$PATH(u_0, u_n) := \{(u_0, u_1, \ldots, u_n) \mid \forall i(neighbor(u_i, u_{i+1})) \wedge \forall i \forall j(i \neq j \rightarrow u_i \neq u_j)\}$$

すなわちユニット間の経路とは，重複する要素を持たず，かつ各要素が隣の要素との隣接関係を持つユニット列をいう．

この定義を用いると，以下の定理は自明である．

定理 3.4 (連結グラフ)

$$\text{グラフ } G = \{u_1, u_2, \ldots, u_n\} \text{ が連結グラフ} \Leftrightarrow \forall i \forall j(PATH(u_i, u_j) \neq \phi)$$

ここで，新たなデータ構造を定義する．

定義 3.5 (ユニット木) 連結グラフ $G = \{u_1, u_2, \ldots, u_n\}$ に対して，$\{u_1, u_2, \ldots, u_n\}$ を頂点集合とし，隣接関係を辺とする根付き木を，ユニット木という．

図 8 は，あるプログラム依存グラフに対するユニット木の例である．左上のグラフは 4 つのユニットを含むプログラム依存グラフである．このプログラム依存グラフに対するユニット木は多数存在するが，そのうち u_1 を根とするグラフ 3 種を示した．ユニット木の頂点集合は連結グラフを構成するユニットの集合であるが，図から自明であるように，それらの間の隣接関係を全て辺として持つわけではない．そのため，1 つのプログラム依存グラフに対するユニット木には，まず根とする頂点の自由度があり，それに加えて，辺として選択する隣接関係の自由度があることになる．

それを踏まえて，任意の連結グラフ G に対するユニット木の集合を以下のように定義する．

定義 3.6 (ユニット木の集合) 連結グラフ G に対して，

$$TREE(G) := G \text{ から生成可能な全てのユニット木の集合}$$

また，ユニット木における経路を以下のように定義すると，定理 3.8 は自明である．

定義 3.7 (ユニット木における経路) ユニット木 t の根 u_r と，任意の頂点 u について，

$$treepath(t, u) := t \text{ における } u_r \text{から } u \text{ への経路}$$

定理 3.8 (ユニット間の経路とユニット木) ユニット木 t の根 u_r と，任意の頂点 u について，

$$treepath(t, u) \in PATH(u_r, u)$$

また，定理 3.4 より連結グラフにおいて $PATH(u_r, u) \neq \phi$ なので，任意の連結グラフ G とユニット $u_r \in G$ に対して，u_r から G の各頂点への少なくとも 1 つの経路が存在する．よって，G に対して，u_r を根とする少なくとも 1 つのユニット木を構成することができる．

次に，ユニットの同値関係を定義する．

定義 3.9 (ユニットの同値関係) ユニット $u_1 = unit(e_1, v_1, v_2)$，$u_2 = unit(e_2, v_3, v_4)$ について，

$$u_1 \sim u_2 := e_1 \sim e_2 \wedge v_1 \sim v_3 \wedge v_2 \sim v_4$$

ただし，$e_1 \sim e_2$ は，e_1 と e_2 の依存関係の種類が一致することを表す．$v_1 \sim v_2$ は，コードクローン検出の目的に応じた定義をする必要がある．すなわち，ソースコード上の文字列が完全に一致するコードクローンのみ検出を検出する場合は，$v_1 \sim v_2$ を 2 つの頂点に対応する文の文字列が完全に一致することと定義する．一方，変数名やリテラルが異なる場合も，構造が同じであればコードクローンであるとみなす場合などは，それに応じた $v_1 \sim v_2$ の定義を用いる．

この定義を用いて，ユニット間の経路の同値関係を定義する．

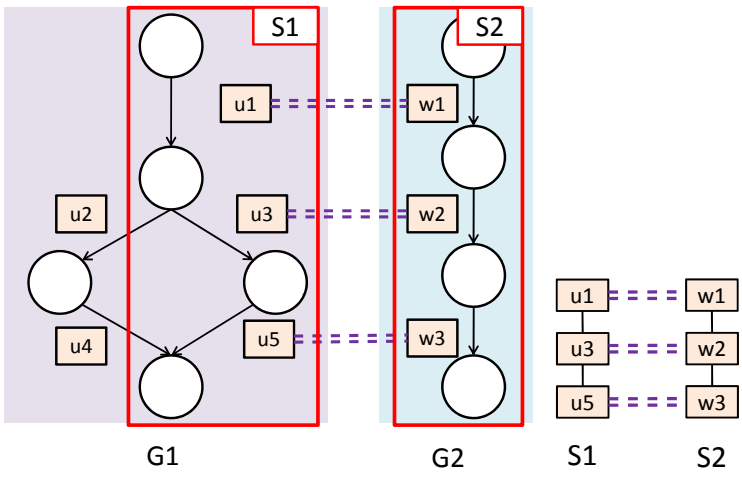
定義 3.10 (ユニット間の経路の同値関係) ユニット間の経路 $p_1 = (u_0, u_1, \ldots, u_m)$ および $p_2 = (w_0, w_1, \ldots, w_n)$ について，

$$p_1 \sim p_2 := |p_1| = |p_2| \wedge \forall i (u_i \sim w_i)$$

すなわちユニット間の経路が同値関係にあるとは，経路上の全てのユニットが同値関係にあることを意味する．さらに，ユニット木の同値関係を定義する．

定義 3.11 (ユニット木の同値関係) ユニット木 $t_1 \in TREE(G_1)$ と $t_2 \in TREE(G_2)$ について，

$$\begin{aligned} t_1 \sim t_2 \quad := \quad & |G_1| = |G_2| \\ \wedge \quad & \exists(u_1, u_2, \ldots, u_{|G_1|}) \exists(w_1, w_2, \ldots, w_{|G_2|}) \forall k \\ & \Big(\bigcup_{1 \leq i \leq |G_1|} \{u_i\} = G_1 \wedge \bigcup_{1 \leq j \leq |G_2|} \{w_j\} = G_2 \wedge treepath(t_1, u_k) \sim treepath(t_2, w_k) \Big) \end{aligned}$$



(a) クローンペア　　(b) ユニット木

図 9: クローンペアの例

すなわち，ユニット木が同値関係にあるとは，頂点数の等しい 2 つのユニット木において，根からの経路が同値関係にある頂点の，1 対 1 の対応関係があることを意味する．

以上の定義を用いて，クローンペアを以下のように定義する．

定義 3.12 (クローンペア) プログラム依存グラフ G_1 の連結部分グラフ S_1，G_2 の連結部分グラフ S_2 について，

$$pair(S_1, S_2) := S_1 \cap S_2 = \phi \wedge \exists t_1 \exists t_2 (t_1 \in TREE(S_1) \wedge t_2 \in TREE(S_2) \wedge t_1 \sim t_2)$$

すなわちこの手法では，同値関係にあるユニット木をもつ S_1 と S_2 を，クローンペアとして検出する．ただし，ユニット木 $t_1 \in TREE(S_1)$ の頂点集合はユニットの集合 S_1 であるため，この定義は $t_1 \sim t_2$ であるユニット木の組 (t_1, t_2) に対して，それぞれの頂点集合の組 (S_1, S_2) をクローンペアとして検出することと等価である．

この定義に基づくクローンペアを，図 9(a) に示す．この例では，プログラム依存グラフ G_1 および G_2 において，$u_1 \sim w_1$，$u_3 \sim w_2$，$u_5 \sim w_3$ が成り立っている．この場合に，それぞれの部分グラフ S_1 と S_2 はクローンペアであり，同値関係にあるユニット木は図 9(b) である．

しかし，コードクローンの検出結果として定義 3.12 によるクローンペアを全て出力するのは不適切である．そこで，出力すべきクローンペアを以下のように定義する．

定義 3.13 (出力すべきクローンペア) プログラム依存グラフの連結部分グラフ S_1，S_2 について，

$$outputpair(S_1, S_2) := pair(S_1, S_2) \wedge \neg \exists (S_1', S_2') (pare(S_1', S_2') \wedge S_1' \supset S_1 \wedge S_2' \supset S_2)$$

すなわち出力すべきクローンペアは，それ以上ユニットの組を追加できないクローンペアである．

3.3 解析処理

解析処理の入出力は，以下の通りである．

入力 更新されたソースコード

出力 ユニットの情報 (データベースの更新)

この処理を実現するには，まずソースコードから各メソッドのプログラム依存グラフを構築する必要があるが，それには文献 [29] や文献 [30] のような既存の手法を利用する．プログラム依存グラフからのユニットの情報の構築は，定義 3.1 よりユニットはプログラム依存グラフの各辺に対して定義されるので，プログラム依存グラフから各辺とその両端の頂点の情報を取得できれば，容易に実現可能である．

解析処理が終了した時点で，更新の無かったソースコードを含めた全てのソースコードの解析結果が，データベースに登録された状態になる．

3.4 検出処理

検出処理の入出力を以下に示す.

入力 全てのソースコードに対するユニットの情報，検出対象メソッドの集合

出力 検出対象メソッドに関連する出力すべきクローンペアの集合

この処理では，ユニットの情報からクローンペアの情報を構築するアルゴリズムが必要である．入力は検出対象メソッドの集合であるが，あるメソッドに対してクローン情報を出力するアルゴリズムがあれば，それを全てのメソッドに対して適用することで，全てのメソッドに対するクローン情報が出力できる．そこで，あるメソッド m に対するクローン情報構築アルゴリズム $detect(m)$ を示す.

inputs
m: 検出対象メソッド.
outputs
C: m に含まれるコード片に対するクローンペアの集合
do
$C \leftarrow \phi$
for all u_1 such that $u_1 \in m$ **do**
for all u_2 such that $u_2 \sim u_1 \wedge u_2 \neq u_1$ **do**
$C \leftarrow C + create(u_1, u_2, \phi)$
end for
end for
return C

まず最初に，出力 C を空集合で初期化する．次に，メソッド m に含まれるユニットの集合を取得する．$u_1 \in m$ とは，正確には m に対応するプログラム依存グラフ G_m について，$u_1 \in G_m$ であることを指す．本研究で扱うプログラム依存グラフは各メソッドに対して構築されるので，この表記でも曖昧性は生じない．このようなユニットの集合は，データベースに登録するユニットの情報にメソッドの情報を含めば，検出処理においてプログラム依存グラフの構築を行わなくても，データベースの検索によって容易に取得することができる．その後，取得した各ユニットに対して，同値関係にあるユニット u_2 の集合を取得する．この集合の取得にも，データベース検索を用いることができる．これらの処理により，同値関係にあるユニットの組 (u_1, u_2) が取得できるので，その組からアルゴリズム $create$ によって出力すべきクローンペアを構築する．

出力すべきクローンペアの構築は，定義 3.12 および定義 3.13 より，2 つのユニットが属するプログラム依存グラフに対する，それぞれのユニットを根とする極大のユニット木の組を求め，それらの頂点集合の組を出力する問題であるといえる．そのため，木構造に対する深さ優先探索のように，再帰的な処理で実現することができる．ただし，ユニット木の辺の集合は，ユニット間の隣接関係の集合であるが，図 8 に示したようにユニット木は隣接関係の全てを辺として含むことはできないため，現在探索しようとしている隣接関係をユニット木に追加することで，閉路ができないか確認する必要がある．また，定義 3.12 より，ユニットの共有を回避する必要がある．これら 2 つの問題を解決するために，探索済みの頂点 (ユニット) の集合 U をアルゴリズムの入力に加え，その集合に含まれるユニットに対しては処理を行わないこととした．なお，前掲のアルゴリズムからこの処理を呼び出す場合には，探索済みのユニットが存在しないことを表す，空集合 ϕ を入力とする．これを踏まえて，アルゴリズム $create(u_1, u_2, U)$ を以下に示す．

inputs

(u_1, u_2): $u_1 \sim u_2$ かつ $u_1 \neq u_2$ であるユニットの組.

U: 探索済みのユニットの集合

outputs

(S_1, S_2): $u_1 \in S_1, u_2 \in S_2$ である，出力すべきクローンペア.

do

$(S_1, S_2) \leftarrow (\{u_1\}, \{u_2\})$

$U \leftarrow U \cup \{u_1, u_2\}$

for all u_x such that $neighbor(u_1, u_x) \wedge u_x \notin U$ **do**

for all u_y such that $u_x \sim u_y \wedge neighbor(u_2, u_y) \wedge u_y \notin U$ **do**

$(S_x, S_y) \leftarrow create(u_x, u_y, U)$

$(S_1, S_2) \leftarrow (S_1 \cup S_x, S_2 \cup S_y)$

end for

end for

return (S_1, S_2)

まず，(S_1, S_2) および U に対して，入力のユニットを追加する．ただし (S_1, S_2) に追加可能であることは，呼び出しの時点で確認済みである．入力のユニットとそれぞれ隣接し，かつ同値関係にあるユニットの組を検索する．これはユニット木において，共通に追加可能な子頂点を検索することになる．そのような頂点が見つかった場合には，それらの頂点を根とする部分木を再帰的に探索し，その全ての出力結果を (S_1, S_2) に追加することで，入力のユニットを根とする部分木を構築できる．この部分木の頂点集合が，出力すべきクローンペアである．

3.5 計算時間

提案手法の計算時間について考察する．解析処理については，プログラム依存グラフの構築時間は利用する既存手法に依存するため，これを $O(P)$ とする．データベースへのユニットの登録は，ユニット数を u として，$O(u)$ と表される．これを検出対象の行数 n を用いて，文の数を $O(n)$ と仮定した上で表現すると，最悪時は以下のようなソースコードにおいて，$O(n^3)$ となる．

```
void method(boolean b1, boolean b2, ..., bm) {
    int a1 = 0;
    …
    int am = 0;
    int x = 0;

    if(b1) x += a1++ + a2++ + ⋯ + am++;
    if(b2) x += a1++ + a2++ + ⋯ + am++;
    …
    if(bm) x += a1++ + a2++ + ⋯ + am++;

    return x + a1 + a2 + ⋯ + am;
}
```

このメソッドにおいて，文の数は引数部分が m，変数宣言部分が $m+1$，if 文と代入文で $2m$，return 文が 1 で，合計 $4m+2$ である．プログラム依存グラフの制御依存辺と実行依存辺の数も m に比例する．しかし，データ依存辺を考えると，x に対して加算代入を行っている文のうち，任意の 2 つの間にデータ依存辺が存在する．さらに，それぞれの組に x および各 ak に対するデータ依存辺が引かれるため，これらの文に関するデータ依存辺の総数は ${}_mC_2(m+1) = (m^3 - m)/2$ となる．よって，このときのユニット数 u は，$O(n^3)$ となる．しかし，これは極めて特殊な例であり，実際には $O(n)$ であると期待され，多くとも $O(n^2)$ 程度であると予想される．

以上より，解析処理の計算時間は最悪時に $O(n^3 + P)$ であるが，実際には $O(n + P)$ であると期待される．なお，プログラム依存グラフの構築時間は文献 [1] において，制御依存辺の構築が最悪時

に $O(n^2)$ であることが示されている．データ依存辺を含めると，ユニット数 u が辺の数と一致するため，前掲のソースコードのような最悪時には $O(n^3)$ になると考えられる．

次に，検出処理の計算時間を考える．まず，データベースに対する検索処理の時間は，データベースの実装に依存する．例えばハッシュ表によるインデックスを用いれば，検索時間は理論上 $O(1)$ となる．しかし，二分木を用いたインデックスであれば，検索時間はソフトウェアに含まれるユニットの総数を N として $O(\log N)$ である．これについて，以降 $O(D)$ という記号を用いて表すこととする．

アルゴリズム *create* の計算時間を考える．再帰処理の呼び出し回数は，出力するクローンペアに含まれるユニット数 s と一致する．各回の呼び出しにおいては，内側のループでは検出対象のユニット u_x と同値関係にあり，かつユニット u_2 と同じメソッドに含まれるユニット u_y を取得し，それぞれに対して u_2 との隣接関係を調査する．これらの処理は，ユニットの取得が $O(D)$ であり，隣接関係の調査は u_x と同値関係にあり u_2 と同じメソッドに含まれるユニットの数 w_m を用いて $O(w_m)$ と表せるため，合計で $O(D + w_m)$ となる．外側のループでは，検出対象のユニット u_1 と隣接するユニットを取得するが，検出対象のユニットの集合は *detect* において取得済みであるため，各頂点からそれを含むユニットの集合への対応関係を記憶するなど，適切なデータ構造を用いれば $O(1)$ で実行可能である．ループの繰り返し回数は隣接関係にあるユニット数であり，これを b と表記すると，このループの計算時間は $O(b(D + w_m))$ である．以上より，アルゴリズム *create* の計算時間は $O(sb(D + w_m))$ である．

これを用いて，アルゴリズム *detect* の計算時間を考える．内側のループでは，全てのメソッドにおいて同値関係にあるユニットの取得と，それらに対する *create* の実行が必要である．そのため，計算時間はあるユニットと同値関係にある全てのユニットの数を w_a として $O(D + w_a sb(D + w_m))$ となるが，$D < w_a sb(D + w_m)$ より，これは $O(w_a sb(D + w_m))$ である．外側のループでは，検出対象のユニットを取得し，各ユニットに対してループ処理を行う．これも同様に D を消去できるため，検出対象のユニット数 u を用いて $O(uw_a sb(D + w_m))$ となる．その他の処理は定数時間であるため，1 つのメソッドに対する検出処理のアルゴリズムはデータベースの検索時間を $O(D)$ とすると，検出対象のユニット数 u，あるユニットと同値関係にある全ユニット数の平均 w_a，あるユニットと同値関係にあるユニット数の 1 メソッド当たりの平均 w_m，出力するクローンペアに含まれるユニット数の平均 s，あるユニットと隣接関係にあるユニット数の平均 b を用いて，$O(uw_a sb(D + w_m))$ と表すことができる．

次に，これらの値をソフトウェアの総行数 N，検出対象の行数 n を用いて表現する．まず u は，前述の通り最悪時に $O(n^3)$ であるが，実際には $O(n)$ か多くとも $O(n^2)$ 程度であると予想される．w_a については，明らかに $O(N)$ である．しかし，実際には同値関係にあるユニット数は N と比較して非常に少なくなり，全体の計算時間に与える影響は小さいと考えられる．w_m は，1 メソッド当たりの平均行数に比例するが，これはソフトウェアの規模によらず一定であると考えられるため，$O(1)$ である．s は明らかに $O(u)$ であるため，最悪時はやはり $O(n^3)$ となる．b について考えると，前掲

のソースコードにおける代入文について，他の代入文との間に存在するデータ依存辺の数は，自身以外の代入文の数 $m-1$ と変数の数 $m+1$ の積となる．そのため，このとき b は $O(n^2)$ となる．しかし実際のソースコードにおいては，$O(1)$ であると期待される．

以上を踏まえると，このアルゴリズムは最悪時に $O(n^8ND)$ となる．しかし実際の検出においては，$w_a \ll N$ より w_a をそのまま用いて，$O(n^2w_aD)$ 程度であると考えられる．

4 実装

この章では，3 章で述べた提案手法の実装について説明する．まず 4.1 節で実装の概要を述べ，4.2 節では実装の際の工夫を紹介する．その後，これらを実現するためのデータベースの設計について，4.3 節で説明する．

4.1 実装の概要

提案手法を Java 言語を用いて実装する．実装に当たって，まず手法において規定していない項目について定める必要がある．具体的には以下の 4 点である．

1. 利用する既存手法
2. 頂点の同値関係の定義
3. 入出力の形式
4. データベースの設計

これらのうち 1 から 3 については，4.1.1 項から 4.1.4 項で説明する．4 については前述のとおり，4.3 節で述べる．

4.1.1 利用する既存手法

提案手法の実装においては，データベースに対する操作と，プログラム依存グラフの構築が必要であるが，これらの処理については既存手法を用いることとした．まず，データベース処理には SQLite[31] を利用した．SQLite はオープンソースのデータベース管理システムであり，サーバ・クライアントモデルでなく全てローカルでの処理として実装されていることなどから，ローカルでの利用においては非常に高速に動作するため，動作速度の向上という本手法の目的に合致すると考えられる．プログラム依存グラフの構築には，MASU[29] を利用した．MASU は汎用的なソースコード解析ライブラリであり，実行依存辺を含むプログラム依存グラフの構築に加えて，4.1.2 項で説明する頂点の同値関係の判定にも活用できるため採用した．

4.1.2 頂点の同値関係の定義

定義 3.9 に基づいてユニットの同値関係を判定するための，頂点の同値関係 $v_1 \sim v_2$ を定義する必要がある．そこで，まず文 s に対して必要な変換を行う $normalize(s)$ という処理を考え，それを用いて頂点の同値関係を定義する．

定義 4.1 (頂点の同値関係) プログラム依存グラフの頂点 v_1，v_2 に対して，それぞれに対応する文 s_1，s_2 を用いて，

$$v_1 \sim v_2 := normalize(s_1) = normalize(s_2)$$

$normalize(s)$ においては，4.1.1 項で紹介した MASU の解析結果を用いて，変数名などの正規化を行った．まず，文“`int x = x + y + 1`”に対する正規化の例を表 1 に示し，さらにこの文と正規化の結果が一致する文の例を表 2 に示す．

この正規化では，文に現れる正規化の対象を，前から順に “id$n$” という文字列に置換する．このとき対象と同じ文字列がそれ以前に出現していれば，それらは同一の文字列に置換する．その結果，正規化対象に関する違いを無視して，文を比較することが可能である．ただし，リテラルを正規化する場合は型名に対応する正規化文字列の後に “L” をつけた文字列で表す．これは，表 3 のような例において，正規化の結果を一致させるためである．

なお，どの対象を正規化するかについては，コードクローン検出を行う状況によって必要な正規化が変化すると考えられるので，ユーザが初期設定として与えることとした．

4.1.3 入力の形式

提案手法に対する入力は，解析処理の入力である前回の検出以降に更新されたソースコードと，検出処理の入力である検出対象のメソッドである．

まず検出対象のメソッドについて，これをメソッドの一覧としてユーザが指定するには，ユーザ自身が全てのメソッドを把握している必要があるため，開発者以外には困難であると考えられる．そこで，より粒度の低いファイルやディレクトリの単位で指定することを考えた．するとファイルからメソッドへのマッピングを行う必要があるが，それには以下の方法がある．

1. 解析処理でファイルとメソッドの対応関係をデータベースに登録

表 1: 正規化の例

正規化の対象	結果
なし	`int x = x + y + 1`
変数名	`int id0 = id0 + id1 + 1`
変数名, リテラル	`int id0 = id0 + id1 + id2L`
型名, 変数名, リテラル	`id0 id1 = id1 + id2 + id0L`

表 2: 表 1 と正規化結果が一致する文の例

正規化の対象	結果
変数名	`int i = i + j + 1`
変数名, リテラル	`int i = i + j + 2`
型名, 変数名, リテラル	`String a = a + b + "abc"`

2. 検出処理の前処理でファイルの中身を解析

これらを比較すると，1 は解析処理においてプログラム依存グラフを作成する際にファイル内のメソッドの情報を取得するため，データベースにテーブルを追加するだけで，容易に実現できる．一方の 2 は，追加の処理が必要となるため，実装の手間がかかり，処理時間も増加する．そのため，1 の方法を採用した．

これにより，解析処理と検出処理の入力は，いずれもファイルの集合として指定することになるが，その指定方法として，以下の 2 つを実装した．

1. ユーザによる一覧の記述

2. タイムスタンプを用いた自動判定

1 については，単純にユーザ自身が対象の一覧を指定するものであり，検出処理において，ある特定のファイルやメソッドに対するコードクローンを知りたい場合などに有効である．また，解析処理においても，バージョン管理システムへのコミットと同時に更新のあったファイルを解析する場合など，更新されたファイルの集合が既知であれば，この方法による指定が可能である．

しかし，特に大規模なシステムにおいて，ユーザが更新部分を手動で解析するのは現実的ではないため，2 のような自動判定の方法も必要であると考えられる．これを実装するには，検出を行った時刻を記憶する方法と，各ファイルごとに前回検出時のタイムスタンプを記憶する方法があるが，正確性を期すため，後者の方法を用いた．ただし，この自動判定を用いた場合でも，ファイルシステムに含まれる全てのファイルを対象にはできないので，処理対象のファイルを含むディレクトリは，ユーザが指定することになる．

4.1.4 出力の形式

提案手法は，クローンペアとしてユニットの集合の組を出力するが，人間に対してそのまま出力しても，ソースコードとの関係が把握しにくく，理解が難しい．そこで，クローンペアをプログラム依存グラフの頂点の集合として出力することにした．プログラム依存グラフの頂点はソースコード中の文と対応するため，これによりユーザにとって理解が容易になると考えられる．具体的には，クローンペア (S_1, S_2) に対して，S_1 の頂点集合 V_1，S_2 の頂点集合 V_2 の組 (V_1, V_2) を出力する．ただし，頂点集合はユニットの集合から，以下の自明な定理により求められる．

表 3: リテラルの正規化の例

文	`x = x * 2 + 1`	`x = x * 2 + 2`
他と同様の置換	`x = x * id0 + id1`	`x = x * id0 + id0`
型 +“L” に置換	`x = x * id0L + id0L`	`x = x * id0L + id0L`

定理 4.2 (頂点集合) 連結グラフ G の頂点集合 V について，

$$V = \{v \mid \exists e \exists v'(unit(e, v, v') \in G \vee unit(e, v', v) \in G)\}$$

さらに，サイズの小さいコードクローンを多数出力しても，ユーザにとってそれほど利益はないと考えられる．そこで，ユーザの指定した閾値以上の大きさの頂点集合のみを，コードクローンとして出力することにした．

頂点集合の出力形式としては，以下の 3 種類を実装している．

1. データファイルによる出力
2. テキストファイルによる出力
3. ソースコード上での可視化

1 のデータファイルは，内部のデータ構造をそのままファイルに出力したものであり，本システムをライブラリとして用いるシステム間のデータの受け渡しに適している．この形式はファイルサイズが小さく，出力にかかる時間が短いという長所をもつが，結果の解釈には検出に用いたデータベースへのアクセスが必要になるという欠点があり，データベースが更新された時点でそのファイルは意味を成さなくなる．2 のテキストファイルは，各クローンペアのメソッド名[3]と頂点の位置情報をテキスト形式で出力したものであり，他のシステムとのデータの受け渡しには，主にこの形式を用いると想定している．しかし，人間が解釈するのは不可能ではないが難しい．そのため，3 のソースコード上での可視化が必要になる．この可視化の例を図 10 に示す．

まず下段に，検出されたクローンペアの一覧を表示する．この一覧には，2 つの部分グラフが属するメソッドの情報を含んでいる．そこから表示したいクローンペアを選択すると，上段に 2 つのソースコードが示され，クローンペアとなる箇所をハイライトして表示する．

4.2 高速化のための工夫

本研究の主眼は，インクリメンタルな検出手法によって余分な解析処理を回避する点にあるが，目的は高速化であるため実装の際にはそれ以外の点でも工夫が必要となる．この節では，そのような工夫点を紹介する．

4.2.1 検出処理の単純化

ある部分グラフの組がクローンペアであることを示すには，同値関係にある部分木が 1 つでも発見できればよい．しかし，3.4 節で説明したアルゴリズムでは，同値関係にある全てのユニットの組に対して，それらのユニットを根とするユニット木の探索を行うため，ユニット数 s のクローンペアに対して s 回の検出処理を行うことになり，非効率である．そこで，次の仮定を置く．

[3]クラス名の情報も含む

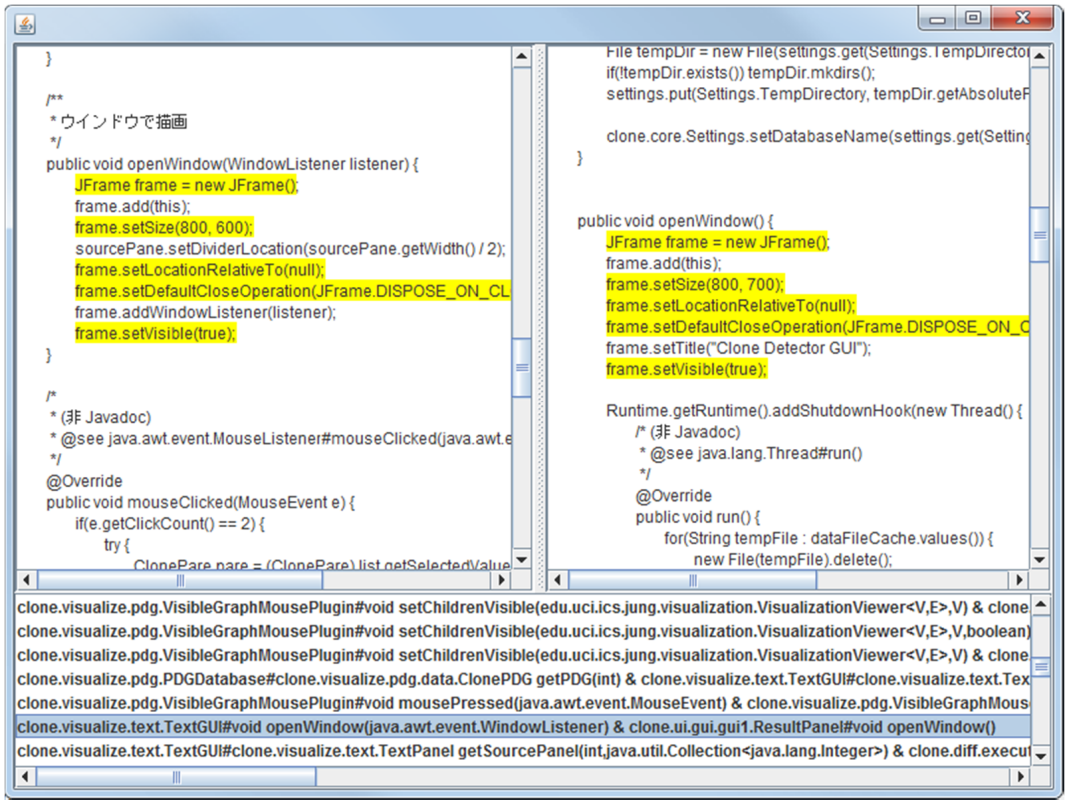


図 10: ソースコード上での可視化

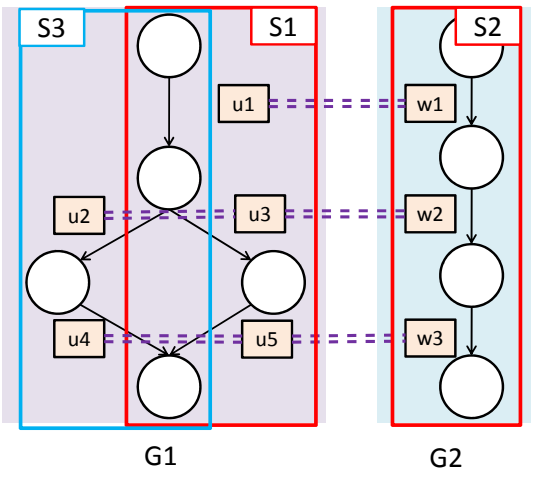


図 11: 実装において検出されないクローンペア

仮定 4.3 (ユニットの組に対する出力すべきクローンペアの一意性) $u_1 \sim u_2$ であるユニットの組 (u_1, u_2) に対して，$u_1 \in S_1$ かつ $u_2 \in S_2$ となる出力すべきクローンペア (S_1, S_2) は，一意に定まる．

仮定 4.3 により，発見済みの出力すべきクローンペアに含まれるユニットの組に対しては，再調査をする必要がない．そこで，そのようなユニットの組の集合 $DONE$ を考え，3.4 節のアルゴリズムにおいて，以下の修正を行う．

修正前　：　$u_1 \sim u_2$

修正後　：　$u_1 \sim u_2 \wedge (u_1, u_2) \notin DONE$

この修正により，同じクローンペアを複数回探索することが回避され，3.5 節で説明した計算時間は $O(uw_ab(D + w_m))$ となる．これは最悪時に $O(n^5ND)$ であり，平均的には $O(nw_aD)$ 程度であると期待される．

ただし，この仮定には例外があり，図 11 のようなクローンペアが出力されない．この図は，3.2 節で紹介した図 9(a) に対して，$u_2 \sim w_2$，$u_4 \sim w_3$ の同値関係を追加したものである．これにより，定義 3.13 からは (S_1, S_2) および (S_3, S_2) の組も出力すべきクローンペアとなる．しかし，これらのクローンペアはいずれも (u_1, w_1) という同値関係にあるユニットの組を含むため，仮定 4.3 によりこれらのクローンペアはどちらか片方しか検出されない．

この問題について，評価実験に基づいた具体例および考察は，5.3.2 項で述べる．

4.2.2　ID 番号の利用

3.1 節で説明した通り，本手法ではユニットの情報をデータベースに登録する．ユニットの情報は帰属するプログラム依存グラフ G と，ユニット $unit(e, v_1, v_2)$ によって表される．ただし e は G の辺，v_1 および v_2 は G の頂点である．これらについて含まれる情報を，表 4 に示す．

メソッド名については，3.4 節で述べた通り，G をメソッド名により識別する必要がある．依存関係の種類と頂点の文字列は，定義 3.9 によるユニットの同値関係の判定に必要である．頂点の位置情報は，ユニットの隣接判定における頂点の識別に必要である．

これらの情報を単純にデータベースに格納すると，問題となるのがメソッド名と頂点の文字列である．これらの情報は複数のユニットで共通となるが，全てのユニットに対して文字列を登録すれば，データ量の肥大化と検索速度の低下が懸念される．そこで，メソッド名と頂点をそれぞれ一意に識別する ID 番号を定義した．これにより，メソッド名のテーブルと頂点情報のテーブルが新たに必要になるが，ユニットの情報は G, v_1, v_2 がそれぞれ ID 番号となり，データ量の削減ができる．3.4 節のアルゴリズムで用いる検索の速度については，あるメソッドに含まれるユニットの検索はメソッドの ID 番号で検索するため，高速化が期待できる．一方で，一致するユニット集合の取得に対しては，現時点では頂点の ID 番号から頂点の文字列の情報を取得する必要があり，逆に効率が下がる可能性がある．この問題への対処は，4.2.3 項で説明する．

4.2.3 ハッシュ関数の利用

定義 3.9 に従えば，ユニットの同値関係の判定には，依存関係の種類の一致判定と，両端の頂点に対応する文の同値関係の判定が必要である．しかし文の同値関係の判定を単純に文字列の比較によって実装すると，2 つの問題を生ずる．まずは，判定に文字数に比例した時間がかかることである．もう 1 つは，頂点の文字列をデータベースに登録する必要があるが，一般に文の文字数が無制限であることから，文字列をそのまま登録するのは不適切であるということである．そこで，ユニット u に対してハッシュ関数 $h(u)$ を定義し，データベースには依存関係の種類や頂点に対応する文の文字列の代わりにハッシュ値を登録し，そのハッシュ値が一致するユニットどうしを同値関係であると判定することで，これらの問題に対処する[4]．なお，4.1.2 項で述べた正規化は，ハッシュ値の計算前に行う必要がある．

このハッシュ関数に対する要求は，以下の通りである．

条件 1 $u_1 \sim u_2$ ならば，$h(u_1) = h(u_2)$ である．

条件 2 $u_1 \not\sim u_2$ のとき，$h(u_1) = h(u_2)$ となる確率は十分に低い．

表 4: ユニットに含まれる情報

G	メソッド名
e	依存関係の種類
v_1, v_2	頂点の情報 (文字列，位置情報)

[4] クローン検出においてハッシュ値を用いて一致判定を行う手法として，2.3 項で紹介した文献 [19] や文献 [15] がある．

これらの条件を満たすハッシュ関数として，MD5 アルゴリズム [32] を用いることにした．このアルゴリズムは，任意のバイト列に対して固定長 (128 ビット) のバイト列を返すもので，暗号化通信や電子署名，ファイルの破損および改竄の検出などに用いられている．

本研究の実装では，ハッシュ値を数値として扱うため，MD5 ハッシュ値の上位 64 ビットのみを使用している．この場合でも，条件 1 が成立することは自明であり，条件 2 に関しても 2^{64} 通りの値域があれば，十分に低確率であると考えられる．以上を踏まえ，バイト列 X に対する MD5 ハッシュ値を $MD5(X)$ とすると，本研究で用いるハッシュ関数 $h(u)$ は，以下のように定義できる．

$$h(unit(e, v_1, v_2)) = MD5(e \text{ の依存関係の種類} + normalize(v_1) + normalize(v_2)) \gg 64$$

ただし，“+” は文字列の結合を表し，“≫” は右シフト演算を表す．また，$normalize(v_1)$ は，正確には v_1 に対応する文の文字列を正規化した結果である．

4.2.2 項で述べた ID 番号と，この項で述べたハッシュ値を用いた結果，ユニットについて必要な情報は表 5 の通りになる．辺に関する情報は，依存関係の種類がハッシュ値に含まれるため，不要になる．また，頂点に関する情報も，ユニットの一致判定をハッシュ値で行うことにより文字列が不要になり．頂点の識別には ID 番号を使うことで位置情報が不要になる．よってコードクローン検出において頂点の情報をデータベースに登録する必要がなくなるが，実際には検出結果の出力のために，頂点の ID 番号と位置情報の対応関係は登録することとした．なお，メソッド名についても，検出処理では用いないが同様の理由でデータベースに登録する．

4.2.4 巨大な対象に対する検出

3.5 節で説明した通り，提案手法は特定のソースコードに対して，計算時間が極めて長くなるという問題がある．実際のソフトウェアにおいても，3.5 節の例ほどではないが，TV-Browser[33] などでこのような問題を発生させるソースコードが発見された．具体的には，以下のような例である．

```
switch(x)
  case 1:
    S1;
  case 2:
    S2;
```

表 5: ユニットについて登録が必要な情報

G	メソッド ID
h	$h(unit(e, v_1, v_2))$
v_1, v_2	頂点 ID

```
  …
  case 1000:
    S1000;
}
```

この例では，switch 文の評価式から，S1, S2, ..., S1000 に対する制御依存がある．ここで S1, S2, ..., S1000 が全て同値関係にあるとき，各制御依存辺に対応するユニットは互いに同値関係にあり，かつ隣接している．4.2.1 項の単純化により，計算時間は $O(uw_ab(D+w_m))$ となり，これは $O(nw_aD)$ 程度であると期待されるが，この例では u，b，w_m がそれぞれ $O(n)$ であり，w_a は最低でも $O(n)$ である．よって，計算時間は最低でも $O(n^4)$ となり，しかもその n の値が極めて大きいため，このソースコードに対する計算時間は非常に長くなる．

このようなソースコードについては，ユーザが見ればコードクローンであることは自明であり，ユーザに対してはこのメソッドが多数のコードクローンが含まれるメソッドであるということのみ出力すれば十分であると考えられる．そこで，このようなソースコードが見つかった場合には，あらかじめユーザによって選択された以下のいずれかの対応を取ることとした．

1. そのメソッド内部に対する検出処理を中止する
2. そのメソッドに対する検出処理を全て中止する

1 の方法を取った場合，そのメソッドと他のメソッドの間のコードクローンは検出する．この処理は，w_m が $O(1)$ となり，w_a も大幅に減少するため，計算時間は w_a を用いて $O(n^2w_aD)$ となる．しかし，n が大きい場合はやはり計算時間が非常に長くなるため，コードクローン検出の目的によっては 2 の方法も有効であると考えられる．

なお，このようなメソッドか否かの判断は，メソッド内の同値関係にあるユニットの数を用いて行うこととした．これは，多数の同値関係にあるユニットが含まれる場合は，このソースコードのように類似した処理の繰り返しである可能性が高く，逆に同様の構造を持つソースコードであっても，ユニット数が少なければ問題にはならないためである．

4.3 データベースの設計

以上の説明より，データベースに対して必要な検索処理は表 6 の通りである．

メソッド ID およびハッシュ値からのユニット集合の取得については，3.4 節で説明したものに，4.2.2 項および 4.2.3 項の改良を加えたアルゴリズムの実現に必要である．それ以外の検索処理は，4.1.3 項および 4.1.4 項で説明した入出力に必要である．これらの処理を実現するために，5 つのテーブルからなるリレーショナルデータベースを作成した．それぞれのテーブルの登録内容を，表 7 に示す．

ユニットテーブルに関しては 4.2.3 項で説明した通りであり，メソッド ID またはハッシュ値で検索することで，表 6 に示した検索処理が実現できる．ファイルテーブルについて，まずファイル名で

検索することで，ファイル ID と更新時刻を得ることができ，検索処理の 1 つが実現される．次にメソッド ID の取得であるが，その前にファイルとメソッドの関係を考える．まずファイルからメソッドへの 1 対多の対応関係があることは自明である．一方，メソッドからファイルへの対応関係は，基本的には 1 対 1 であるが，ファイルの移動などにより 1 対多の対応関係が発生する可能性がある．よって，ファイルからメソッドの関係は，多対多であることを前提とする必要がある．そこで，ファイルとメソッドの双方から 1 対多の関係になる対応関係テーブルを作成し，ファイル ID とメソッド ID の組を登録することにした．このテーブルをファイル ID で検索することにより，そのファイルに含まれるメソッド ID を取得することができる．メソッドおよび頂点に対しても，表 6 に示した検索に必要な情報を登録している．ただし頂点 ID については，検出処理においてユニットの隣接判定に用いる際，帰属するメソッドが異なる頂点同士を比較する必要がないため[5]，各メソッド内で一意の値をとるようにした．そのため，特定の頂点を取得するためには，頂点 ID とメソッド ID の組を指定する必要がある．

表 6: データベースに対する検索処理

入力	出力
メソッド ID	メソッドに含まれるユニットの集合
ユニットのハッシュ値	同値関係にあるユニットの集合
ファイル名	更新時刻
ファイル名	メソッド ID
メソッド ID	メソッド名
頂点 ID	頂点の位置情報

表 7: データベースの登録内容

テーブル名	内容
ユニット	メソッド ID，ハッシュ値，始点の頂点 ID，終点の頂点 ID
ファイル	ファイル ID，ファイル名，更新時刻
対応関係	ファイル ID，メソッド ID
メソッド	メソッド ID，メソッド名
頂点	頂点 ID，メソッド ID，開始位置，終了位置

[5]そのようなユニットの組の場合，ユニット情報に含まれるメソッド ID で隣接しないことが判定できる．

5 評価

この章では，4 章で紹介した提案手法の実装を用いて行った評価実験について説明する．

5.1 実験の概要

5.1.1 実験目的

今回の実験の目的は，提案手法の実用性を示すことである．実用的なコードクローン検出手法であるためには，短い時間で質の高いコードクローンを検出する必要があると考えられる．そこで，以下の 2 点について評価を行った．

- インクリメンタルな検出における，提案手法の実行時間の評価
- 提案手法の出力するコードクローンに対する評価

実行時間の評価を 5.2 節で，コードクローンに対する評価を 5.3 節で行う．

5.1.2 実験環境

実験に用いた計算機は，以下の 2 台である．

計算機 1

CPU : Intel Xeon E5405 (2.0GHz * 4)
メモリ : 8GB
OS : Microsoft Windows 7 Enterprise (64bit)

計算機 2

CPU : Intel Core 2 Duo U9300 (1.2GHz * 2)
メモリ : 2GB
OS : Microsoft Windows 7 Enterprise (32bit)

実行時間の評価には，計算速度が要求されるため，計算機 1 を用いた．それに対して出力するコードクローンの評価では，より性能の低い計算機においても実行できることを示すため，計算機 2 を用いている．

5.1.3 実験対象

実験対象として用いたソフトウェアを表 8 に示し，提案手法の比較対象として用いた既存のコードクローン検出ツールと，2.3 節における分類を表 9 に示す．対象ソフトウェアについて，開発言語はいずれも Java 言語であるが，これは利用したライブラリである MASU が Java 言語にしか対応していないためであり，本手法そのものはプログラム依存グラフの構築が可能であれば，あらゆる言語

に対して適用することができる．また，ファイル数は Java のソースコードのファイル数を表し，総行数は本手法がメソッド単位でコードクローン検出を行うことを踏まえ，全ファイルの行数の総和ではなく全メソッドの行数の総和とした．

5.2 実行時間の評価

提案手法の実行時間を評価するにあたり，インクリメンタルな検出を用いることのできる，以下の 2 つの状況を考えた．

状況 1 あるソフトウェアに対して，開発履歴におけるコードクローンの変遷を調査する．

状況 2 継続的に更新があるソフトウェアに対して，特定のファイルとコードクローンになる箇所を検出する．

状況 1 は，主にコードクローンに対する研究において発生する．例えば 2.2 節で紹介した文献 [6][7][9] のように，コードクローンと保守コストの関係を調査する場合には，対象ソフトウェアの複数のバージョンに対してコードクローンの検出を行うことになる．また，コードクローンの可視化手法としても，Adar らは複数のバージョンからコードクローンを検出し，その遷移の様子を表示するツール SoftGUESS を開発している [38]．状況 2 は，実際のソフトウェア開発において発生すると考えられる．例えば，開発者があるコード片にバグを発見して修正を行う場合に，そのコード片とコードクローンになる箇所を検出し，それぞれに対して必要に応じた修正を行うことで，コードクローンに対する一貫した修正を行うことができる．

表 8: 実験対象ソフトウェア

名称	開発言語	ファイル数	総行数
ant[34]	Java	804	89,309
TV-Browser[33]	Java	1,099	125,885
e4[35]	Java	5,532	1,004,682

表 9: 比較対象ツール

名称	分類
Simian[36]	行単位
CCFinder[4]	字句単位
CCFinderX[37]	字句単位
Scorpio[28]	プログラム依存グラフ

これらの状況において，コードクローンの検出に要する時間を評価した．状況 1 に対応する実験を 5.2.1 項で，状況 2 に対応する実験を 5.2.2 項で説明し，5.2.3 項で実行時間に関する考察を述べる．なお，いずれの実験においても，変数およびリテラルに対して 4.1.2 項で説明した正規化を行っている．

5.2.1 開発履歴に対する検出

この実験では，実験対象のバージョン管理システムのリポジトリをネットワーク経由で利用して，以下の手順を適用する．

1. 最初のリビジョンをチェックアウトする．
2. コードクローン検出を実行する．(全てのファイルを解析)
3. 次のリビジョンをチェックアウトする．
4. コードクローン検出を実行する．(更新のあったファイルのみを解析)
5. 全リビジョンに対する検出処理が終わるまで，手順 3 と手順 4 を繰り返す．

これらの処理の総実行時間と，ツール実行時間を述べる．ツール実行時間は手順 2 および手順 4 の時間の総和であり，バージョン管理システムからのチェックアウトに要する時間は含まれない．

なお，手順 4 において出力するコードクローンは，更新のあったファイルに対するコードクローンのみとする．これは，あるリビジョンにおける全てのコードクローンの集合は，直前のリビジョンにおける全てのコードクローンの集合と，更新のあったファイルの集合，およびそれに対するコードクローンの集合により，求めることができるためである．具体的には，C_r をリビジョン r における全てのコードクローンの集合，F_r をリビジョン r において更新のあったファイルの集合，$C_r(F_r)$ を C_r に含まれる F_r に対するコードクローンの集合として，以下の漸化式により定義される．

$$C_r = (C_{r-1} - C_{r-1}(F_r)) \cup C_r(F_r)$$

また，手順 4 における更新のあったファイルの判定は，4.1.3 項で説明した，タイムスタンプにより自動的に判定する方法を用いた．これは実験の容易さを重視しての選択であるが，一般的にチェックアウトの時点で更新のあったファイルの一覧が取得できるので，これを用いればタイムスタンプの判定を回避することができ，実行時間はより短縮されると考えられる．

この実験は，ant および TV-Browser に対して行った．対象となったリビジョン数と実行時間を表 10 に示す．なお，表 8 で示した総行数は，最終リビジョンでの総行数である．また，ant に対する各リビジョンでのツール実行時間を，図 12 に示す[6]．さらに，比較対象のツールを用いて行った同様の実験のツール実行時間を，表 11 に示す．

[6]ただしこの図のリビジョン番号は ant の最初のリビジョンを 0 とし，更新の度に 1 ずつ増加させている．これは開発履歴上での位置を示すものであるが，バージョン管理システムのリビジョン番号とは必ずしも一致しない．

表 10: 開発履歴に対する検出時間

名称	ant	TV-Browser
対象リビジョン数	5,903	5,207
総実行時間	22 時間 02 分	44 時間 54 分
ツール実行時間	14 時間 29 分	19 時間 43 分
ツール以外の実行時間	7 時間 33 分	25 時間 11 分

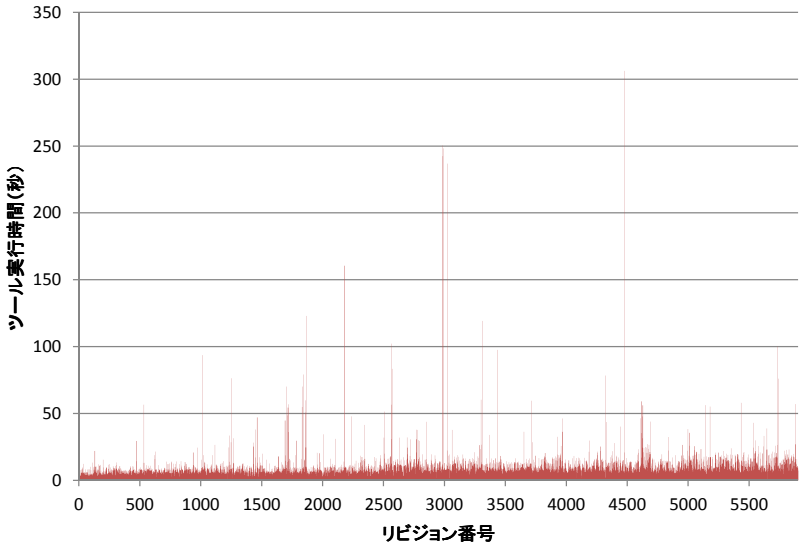


図 12: ant に対する各リビジョンのツール実行時間

表 11: 比較対象ツールによる TV-Browser の開発履歴に対するツール実行時間

Simian	5 時間
CCFinder	10 時間
CCFinderX	24 時間
Scorpio	測定中止

5.2.2 単一ファイルに対する検出

この実験では，ローカルに存在する対象ソフトウェアのソースコードに対して，以下の手順を適用する．

1. 全てのファイルに対して解析処理を行う．
2. 一つ一つのファイルに対して検出処理を行う．
3. 比較対象のツールで全てのファイルに対して検出処理を行う．

これらの各処理について，実行時間を計測する．解析処理は手順 1 において完了しており，その後のファイルの更新もないため，手順 2 では解析処理を行わない．これは，状況 2 の具体例で述べたような状況では，ファイルの修正を行う前にコードクローン検出を行い，その結果を用いて修正すべき個所を特定する必要があるため，実際の利用状況に近いといえる．なお，実際にはこれらの処理に加えて，ソースコードに更新があった場合の解析処理が必要であるが，それに関しては 5.2.1 項の実験と内容が重複するため，今回は調査を行っていない．手順 3 では，この状況でインクリメンタルでない検出手法を用いた場合に，どの程度の時間を要するか調査する．比較対象は表 9 節で紹介した手法のうち，最も高速な行単位の検出ツールである Simian と，提案手法と同様にプログラム依存グラフを用いる検出ツールである Scorpio を用いた．

この実験は，大規模なソフトウェアに対する処理時間を示すため，e4 に対してのみ行った．その結果について，各手順での処理時間をまとめたものを表 12 に示す．また，単一ファイルに対する検出処理における，実行時間とファイル数のヒストグラムを図 13 に示す．

5.2.3 考察

まず，5.2.1 項の実験に関して考察する．表 10 より ant の 1 リビジョン当たりのツール実行時間の平均を算出すると 8.8 秒であり，図 12 より開発履歴のほとんどのリビジョンに対して，その程度の時間で処理を終了していることが分かる．一方で，一部のリビジョンに対しては非常に長い時間を要している．具体的には，1 分以上を要したリビジョンが 20 あり，最大では 5 分を超えている．これ

表 12: 単一ファイルに対する検出時間

解析処理	17 分 27 秒
1 ファイル検出処理 (平均値)	5.0 秒
1 ファイル検出処理 (中央値)	3.9 秒
1 ファイル検出処理 (最大値)	1 分 47 秒
Simian による全ファイル検出処理	2 分 20 秒
Scorpio による全ファイル検出処理	1 時間以上

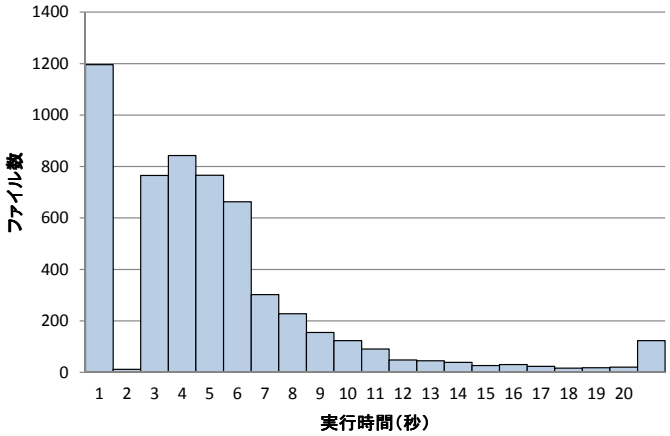


図 13: 単一ファイルに対する検出時間のヒストグラム

らのリビジョンの共通点は，一度に多数のファイルが更新されたことであり，特に 5 分を要したリビジョンでは 733 ものファイルが同時に更新されていた．このような場合には，本手法でも更新されたファイル全てを調査する必要があるため，インクリメンタルな処理による利益が得られず，検出時間が長くなると考えられる．

表 10 のツール実行時間と表 11 を比較すると，提案手法は行単位の検出ツールである Simian の 4 倍，字句単位の検出ツールである CCFinder の 2 倍の時間がかかっているものの，CCFinderX よりは高速であった．提案手法と同じくプログラム依存グラフを用いた Scorpio に関しては，1 リビジョンに数分から十数分の時間がかかるため実験を中止した．提案手法の TV-Browser に対するツール実行時間をリビジョン数で割ると 13.6 秒となり，提案手法の方がかなり高速であることは明らかである．ただし，この実験における比較対象はいずれもインクリメンタルな検出手法ではないため，文献 [19] や文献 [20] のような手法を用いれば，より短時間で処理が完了すると考えられる．

なお，この実験ではバージョン管理システムのリポジトリをネットワーク経由で利用したため，チェックアウトの実行時間はネットワークの状況などに左右され，表 10 の総実行時間やツール以外の実行時間を用いた評価は本質的ではない．しかし，ソフトウェアの開発履歴を調査する場合には必ずこのような処理を行うことになるため，どのような検出ツールを用いても，ツール以外の実行時間は最低限必要になる．そこで，表 11 の各ツールの実行時間に表 10 のツール以外の実行時間を加えた時間を考えると，総実行時間は Simian でも約 30 時間となり，提案手法はその 1.5 倍に過ぎない．

以上の点から，提案手法は状況 1 においてプログラム依存グラフを用いたコードクローンを利用する場合に，十分に実用的であると考えられる．

次に，5.2.2 項の実験について考察する．表 12 より，提案手法は 20 分弱の前処理を一度行えば，それ以降は何度でも，100 万行のソースコードの中から 1 つのファイルを指定した際に，平均的には約 5 秒，最大でも 2 分以内にコードクローンを出力することができる．インクリメンタルな手法を用いない場合は，比較対象のうち最も高速な Simian を用いても 2 分 20 秒を要するため，提案手法はこ

れと比較して平均的に非常に短い時間で結果を出力しており，最悪時でもこれより短時間で検出を終了していることが分かる．また，Simian は行単位のコードクローンを出力するため，検出できないコードクローンが存在する．そのようなコードクローンを検出するためにプログラム依存グラフを用いた手法である Scorpio を用いた場合には，1 時間以上経過しても処理が終了しなかった．よって，状況 2 においてプログラム依存グラフを用いた検出処理を行う場合は，提案手法のインクリメンタルな処理が必要であるといえる．

また，図 13 は横軸を実行時間 (t 秒) とし，縦軸に実行時間が $t-1$ 秒以上 t 秒未満であったファイル数をとったヒストグラムである．例えば横軸が 4 秒であるときの縦軸は 800 強であるが，これは実行時間が 3 秒以上 4 秒未満のファイル数を表す．このヒストグラムから，まず 1 秒以内に処理が終了したファイルが 1,200 以上あったことがわかる．それに対して 1 秒以上 2 秒未満の処理時間であったファイルは非常に少ない．その原因を調査するため 1 秒以内に処理が終了したファイルのいくつかを確認すると，いずれもインタフェースか，メソッド本体を含まない抽象クラスであった．本手法ではメソッドに対してコードクローンを検出するため，このようなファイルに対しては検出処理を一切行わない．その結果，極めて短い時間で処理が終了したと考えられる[7]．1 秒以内に処理が終了したファイルを除くと，2 秒以上 6 秒未満の範囲に多くのファイルが含まれる一方で，20 秒以上の処理時間を要したファイルも 100 以上ある．これらは，表 12 の内容を裏付ける結果であるといえる．

以上の点から，提案手法が状況 2 において有効であることは明らかである．また，これらの考察を踏まえると，インクリメンタルな検出処理を行える状況において，提案手法がプログラム依存グラフを用いたコードクローンを高速に検出できていることが分かる．ただし，この結果には 4.2.1 項で説明した高速化が影響しており，これを行わない場合の検出時間はかなり悪化すると考えられる．

5.3 検出したコードクローンの評価

本手法により検出したコードクローンについて，他の検出ツールの出力結果と比較することで，プログラム依存グラフを用いた手法として適切なコードクローンが検出されていることを確認する．実験内容は以下の通りである．

1. 提案手法を用いてクローンペアの検出を行う
2. 比較対象ツールの検出結果との差異を調査する

検出対象は ant を用いた．比較対象のツールは Scorpio と Simian を用いる．Scorpio はプログラム依存グラフを用いた検出ツールであるため，これによる検出結果をプログラム依存グラフを用いた検出手法の正解集合であると考える．Simian は，行単位のコードクローンを検出するツールであり，提案手法と Scorpio の結果がどの程度類似しているかの目安として，Simian による検出結果と提案手法や Scorpio の検出結果を比較するために用いる．

[7]なお，このようなファイルを除外した場合の処理時間の平均値は 6.2 秒であった．表 12 より増加するが，前述の議論には影響を与えないと考えられる．

1 のクローン検出においては，各検出ツールの設定をできる限り一致させる必要がある．具体的には，コードクローンとみなすサイズの閾値と，4.1.2 節で説明したような正規化である．まず，コードクローンとみなすサイズは，6 行以上を基準とすることにした．しかし，閾値の指定方法は検出ツールによって異なり，Simian は行数，Scorpio と提案手法はプログラム依存グラフの頂点数で指定する．そのため，Scorpio と提案手法に対しては頂点数 6 以上のコードクローンを検出した上で，出力されたコードクローンの中から 6 行未満のものを除去した．正規化は，どのツールにも実装されているものの，ツールによって微妙に定義が異なる可能性を考え，正規化を一切行わずに検出を行うこととした．

2 については，2 つの検出結果の集合から両方の集合に含まれるクローンペアを除去することで，2 つの検出結果の差異を特定できるが，2.2 節で述べたように，コードクローンに普遍的な定義は存在しない．そのため，実際には完全に一致するクローンペアが出力されることは稀であり，閾値以上の類似度を持つクローンペアどうしを，一致していると判断するのが妥当である．クローンペア間の類似度としては，文献 [25] において $good$ 値および ok 値が定義されている．これらの値の定義を以下に示す．ただし，説明に当たりクローンペア p_1 を構成するコード片 f_1 および f_2 を，$p_1.f_1$ および $p_1.f_2$ と表記し，$lines(f_1)$ を f_1 に含まれる行の集合とする．

定義 5.1 ($good$ 値) *2 つのクローンペア p_1 と p_2 に対して，*

$$good(p_1, p_2) := min(overlap(p_1.f_1, p_2.f_1), overlap(p_1.f_2, p_2.f_2))$$

ただし，

$$overlap(f_1, f_2) := \frac{|lines(f_1) \cap lines(f_2)|}{|lines(f_1) \cup lines(f_2)|}$$

定義 5.2 (ok 値) *2 つのクローンペア p_1 と p_2 に対して，*

$$\begin{aligned} ok(p_1, p_2) \quad := \quad & min(max(contain(p_1.f_1, p_2.f_1), contain(p_2.f_1, p_1.f_1)), \\ & max(contain(p_1.f_2, p_2.f_2), contain(p_2.f_2, p_1.f_2))) \end{aligned}$$

ただし，

$$contain(f_1, f_2) := \frac{|lines(f_1) \cap lines(f_2)|}{|lines(f_1)|}$$

簡潔に説明すると，$good$ 値は 2 つのクローンペアの重なりの程度を表し，ok 値は一方のクローンペアが他方のクローンペアに含まれている程度を表す．これらの値の相違点は，一方のクローンペアが他方に完全に含まれている場合，ok 値は必ず最大になるが，$good$ 値は双方の大きさが大きく異なる場合は低い値になることである．これらの値は，2 つのクローンペアが類似しているほど高くなる．そこで，これらの値が閾値以上になるクローンペアに対して，一致していると判定する．閾値としては，文献 [25] では 0.7 が用いられており，本実験でもそれに倣った．

5.3.1 実験結果

表 13 は，各ツールにより検出されたクローンペアの数と，*good* 値を用いて判断した各ツール間の検出結果の差異である．表 14 は，同様の調査を *ok* 値を用いて行った結果である．“検出数” はそのツールによって検出されたクローンペアの総数を表し，“My”，“Sc”，“Si” はそれぞれ，そのツールで検出されて提案手法 (MyTool)，Scorpio，Simian で検出されなかったクローンペアの個数である．

5.3.2 考察

表 13 より，まず提案手法と Simian を比較すると，検出したクローンペアのほとんどが異なっていることが分かる．具体的には，提案手法で検出されたクローンペアの 98% が Simian で検出されず，Simian で検出されたクローンペアの 99% が提案手法で検出されていない．表 14 では，包含関係にあるクローンペアを一致するものと見なしているが，それでも提案手法と Simian を比較して，提案手法の検出結果の 75%，Simian の検出結果の 83% が，他方の結果と一致しない．Scorpio と Simian を比較しても同様の傾向がみられ，表 13 ではそれぞれ 97%，98% であり，表 14 では 75%，87% である．これらと比較すると，提案手法は Scorpio に近い検出結果を出力しているといえる．

また，Simian で検出されて提案手法で検出されなかったクローンペアのうち，無作為に 10 個を抽出してその内容を調査したところ，そのうちの 5 個は複数のメソッドにまたがるコードクローンであった．このようなコードクローンは，例えば 5.2 節で挙げた状況 1 においては，必要か否かは研究者の考え方に依存する．しかし，コードクローンを 1 つのメソッドとして集約する場合には不要であり，状況 2 のようにバグ修正のためコードクローンとなる箇所を知りたい場合にも，重要ではないと考えられる．

次に，Scorpio の検出結果を正解集合として，それぞれの指標について提案手法の再現率と適合率

表 13: *good* 値を用いたクローンペア数の差異

ツール名	検出数	My	Sc	Si
提案手法	520	–	146	508
Scorpio	398	18	–	385
Simian	838	826	824	–

表 14: *ok* 値を用いたクローンペア数の差異

ツール名	検出数	My	Sc	Si
提案手法	520	–	139	389
Scorpio	398	10	–	297
Simian	838	695	730	–

を計算した結果を，表 15 に示す．ただし，再現率とは正解集合のうち検出結果に含まれるものの割合であり，適合率とは検出結果のうち正解集合に含まれるものの割合である．再現率は非常に高く，提案手法は Scorpio で検出されたクローンペアのほとんどを検出していることが分かる．4.2.1 項で説明した検出処理の単純化により，検出されないコードクローンの存在が懸念されたが，今回の実験においては再現率にそれほど影響を与えていないことが分かる．ただし，単純化により検出できないコードクローンは他のコードクローンと少なくとも 1 つのユニットを共有するため，必然的にそのようなクローンペアどうしの $good$ 値や ok 値は高くなると予想され，Scorpio の検出結果に含まれる提案手法では検出できなかったクローンペアも，細部が異なる別のクローンペアと一致すると判定された可能性がある．そのため，この結果は必ずしも 4.2.1 項の単純化が検出結果に悪影響を与えていないことを示すものではない．なお，同じユニットの組を含むクローンペアを検出しないことには長所と短所があり，例えばユーザが手作業でコードクローンの調査を行う場合には，近い位置にある複数のクローンペアを検出してもそれほど有益ではなく，逆に調査の効率を下げる可能性があるため，このような出力が有効であると考えられる．また，文献 [6] や文献 [7] のように，ファイルやメソッドの単位でコードクローンの有無を調査する場合には，調査結果に影響しないと考えられる．一方で，見逃しているクローンペアが存在することから，文献 [9] のように行単位でのコードクローンの有無を調査する場合など，正確な検出結果が要求される場合には，不適切であるかもしれない．そのような状況で提案手法を用いる場合は，4.2.1 項で説明した単純化を行わないことで，検出時間は増加するものの，より正確な結果を得られると期待できる．

適合率は，再現率と比較すると低い値となった．これは，提案手法により出力されたクローンペアの中に，Scorpio で検出されないものが多く存在することを意味する．これについては，以下の 2 つの理由が考えられる．

1. 2.4 節で紹介した文献 [28] の頂点集約により，Scorpio は一部のコードクローンを検出しない
2. Scorpio は頂点の組を，提案手法は内部的にはユニットの組をコードクローンとみなす

1 については，本手法でも同様の頂点集約を行うことで，検出を回避することができると考えられる．2 については，単純化した例を図 14 に示す．これらの組は，左のコード片の `else` 節を除いた部分と，右のコード片がクローンペアであるのは明白だが，提案手法ではそれぞれのコード片の全体がコードクローンとして検出された．しかし Scorpio は頂点の組をコードクローンとみなすため，頂点数の異なるこれらのコード片をコードクローンとして検出することはない．

表 15: 提案手法の再現率と適合率

用いた指標	再現率	適合率
$good$ 値	0.955	0.729
ok 値	0.975	0.733

提案手法において，なぜこれらのコード片がコードクローンとして検出されるのか説明する．まず，これらのコード片に対するプログラム依存グラフを図 15 に示す．この図では，各頂点において対応する文の文字列の前に頂点名を記述し，データ依存辺を赤色，制御依存辺を青色，実行依存辺を緑色で示している．また，以下ではデータ依存辺に対するユニットを $D(v_xv_y)$，制御依存辺に対するユニットを $C(v_xv_y)$，実行依存辺に対するユニットを $E(v_xv_y)$ と表す．これらのコード片がクローンペアとなる理由は，頂点 $w2$ と $w3$，および $w3$ と $w4$ の間に，それぞれ 2 種類の依存関係があるためである．これにより，定義 3.10 による経路の同値関係を考えると，$(E(v_2v_3), E(v_3v_5)) \sim (E(w_2w_3), E(w_3w_4))$ および $(C(v_2v_4), D(v_4v_5)) \sim (C(w_2w_3), D(w_3w_4))$ が成り立つ．その結果，図 16 に示す 2 つの部分グラフにおいて，同値関係にあるユニット木が構築され，これらがクローンペアであると判定される．

このようなクローンペアが出力されることは，人間の直観に反するという意味では不適切である．しかし，これらのコード片が類似していることは事実であり，片方のコード片に修正を行う場合，もう一方のコード片にも修正が必要になる可能性が高い．そのため，クローン検出を行う状況によっては，この出力がむしろ適切である場合もあると考えられる．なお，提案手法でこのようなコードクローンの検出を回避するためには，以下の 2 つの方法が考えられる．

1. ユニット数だけでなく頂点数も等しい部分グラフの組のみをクローンペアと見なす
2. 依存関係の種類を無視し，頂点間に複数の辺がある場合は統合する

1 の方法を用いる場合には，クローンペアの定義が変化するため，それに伴い 3.4 節のアルゴリズムにも変更を加える必要がある．一方 2 の手法を用いた場合，ユニットの総数が減少するため計算時間の削減も期待できるが，依存関係の異なるユニット間に同値関係が発生するため，検出精度の低下が懸念される．

以上の結果より，提案手法は同じくプログラム依存グラフを用いた手法である Scorpio と類似したコードクローンを検出していることが示され，インクリメンタルな検出処理が可能な状況でプログラム依存グラフを用いたコードクローン検出を行いたい場合に，有効な手法であるといえる．

```
x = 0;
if (a == 0) {
    x += 1;
} else {
    x += 1;
}
return x;
```

```
x = 0;
if (a == 0) {
    x += 1;
}
return x;
```

図 14: 提案手法により検出されたクローンペア

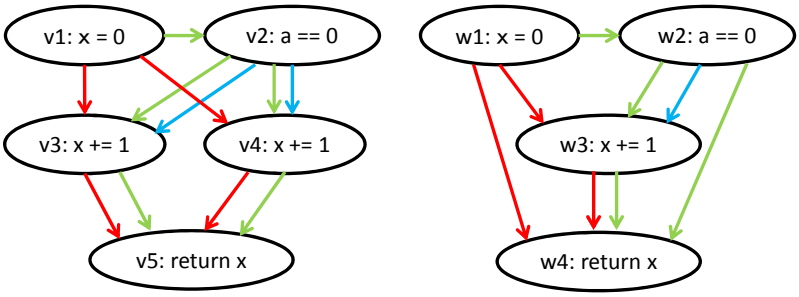


図 15: 図 14 のプログラム依存グラフ

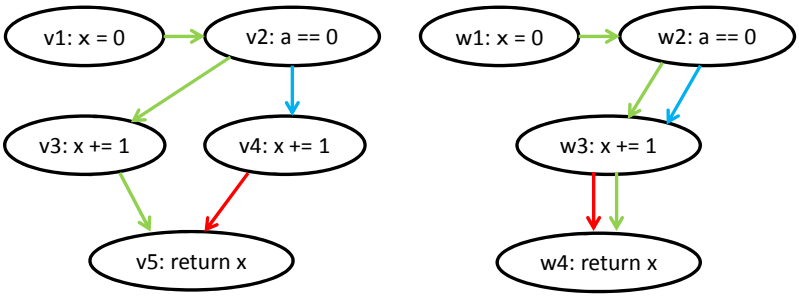


図 16: クローンペアとなる部分グラフ

6　あとがき

本研究ではプログラム依存グラフを用いたコードクローン検出において，インクリメンタルな検出手法を提案した．まず，データを永続化するための “ユニット” というデータ構造を定義し，それを用いたクローンペアを定義し，検出のためのアルゴリズムを紹介した．さらに様々な工夫を加えて提案手法を実装し，それを用いた実行時間と検出されたクローンペアの評価を行った．

今回は実装に Java 言語を用い，いくつかの外部ライブラリを利用したが，これらに関しては高速化において最適であるか検討しておらず，より時間短縮に適した選択をすれば，実行時間はさらに改善される可能性がある．

今後の課題としては，まず検出結果に対する，より詳細な評価が挙げられる．今回の実験では他の検出ツールの出力と比較して再現率や適合率を算出し，それによる評価を行ったが，出力したコードクローンの適切さは開発者や研究者を含む複数の人間が確認して判断することが最善であると考えられる．また，今回は実装において検出精度を犠牲にした高速化を行ったが，検出精度を維持したまま，高速化を行うアルゴリズムを考案することも今後の課題である．さらに，より実用性を高めるためには，表示方法の改良や Eclipse プラグインとしての実装など，ユーザインタフェースの充実も必要であると考えられる．

謝辞

本研究の全過程を通して，理解あるご指導を賜り，的確なご助言を頂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 楠本 真二 教授に心より感謝申し上げます.

本研究に関して，有益かつ的確なご助言を頂きました 同 岡野 浩三 准教授に深く感謝申し上げます.

本研究において，常に熱心かつ適切なご指導およびご助言を頂きました 同 肥後 芳樹 助教に深く感謝申し上げます.

最後に，その他様々なご協力を頂きました 楠本研究室の皆様に深く感謝致します.

参考文献

[1] J. Ferrante, K. Ottenstein and J. Warren: “The program dependence graph and its use in optimization”, ACM Transactions on Programming Languages and Systems, **9**, 3, pp. 319–349 (1987).

[2] 肥後芳樹, 楠本真二, 井上克郎：“コードクローン検出とその関連技術”, 電子情報通信学会論文誌 D, **J91-D**, 6, pp. 1465–1481 (2008).

[3] I. Baxter, A. Yahin, L. Moura, M. Sant’Anna and L. Bier: “Clone detection using abstract syntax trees”, Proceedings of the 14th International Conference on Software Maintenance, pp. 368–377 (1998).

[4] 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, pp. 654–670 (2002).

[5] S. Uchida, A. Monden, N. Ohsugi, T. Kamiya, K. Matsumoto and H. Kudo: “Software analysis by code clones in open source software”, The Journal of Computer Information Systems, **45**, 3, pp. 1–11 (2005).

[6] 門田暁人, 佐藤慎一, 神谷年洋, 松本健一：“コードクローンに基づくレガシーソフトウェアの品質の分析”, 情報処理学会論文誌, **44**, 8, pp. 2178–2188 (2003).

[7] A. Lozano and M. Wermelinger: “Assessing the effect of clones on changeability”, Proceedings of the 19th International Conference on Software Maintenance, pp. 227–236 (2008).

[8] C. Kapser and M. Godfrey: ““Cloning Considered Harmful” Considered Harmful”, Proceedings of the 13th Working Conference on Reverse Engineering, pp. 19–28 (2006).

[9] 佐野由希子, 肥後芳樹, 楠本真二：“重複コードと非重複コードにおける修正頻度の比較”, 電子情報通信学会技術研究報告, 第 109 巻, pp. 43–48 (2010).

[10] S. Ducasse, M. Rieger and S. Demeyer: “A language independent approach for detecting duplicated code”, Proceedings of the 15th International Conference on Software Maintenance, pp. 109–118 (1999).

[11] D. Gusfield: “Algorithms on strings, trees, and sequences: computer science and computational biology”, Cambridge University Press (1997).

[12] B. Baker: “Parameterized duplication in strings: Algorithms and an application to software maintenance”, SIAM Journal on Computing, **26**, 5, pp. 1343–1362 (1997).

[13] R. Wettel and R. Marinescu: “Archeology of code duplication: Recovering duplication chains from small duplication fragments”, Proceedings of the 7th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing, pp. 63–70 (2005).

[14] X. Yan, J. Han and R. Afshar: “CloSpan: Mining closed sequential patterns in large datasets”, Proceedings of the SIAM International Conference on Data Mining, pp. 166–177 (2003).

[15] 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, pp. 176–192 (2006).

[16] R. Komondoor and S. Horwitz: “Using slicing to identify duplication in source code”, Proceedings of the 8th International Symposium on Static Analysis, pp. 40–56 (2001).

[17] J. Krinke: “Identifying similar code with program dependence graphs”, Proceedings of the 8th Working Conference on Reverse Engineering, pp. 301–309 (2001).

[18] J. Mayrand, C. Leblanc and E. Merlo: “Experiment on the automatic detection of function clones in a software system using metrics”, Proceedings of the International Conference on Software Maintenance, Vol. 96, pp. 244–253 (1996).

[19] B. Hummel, E. Juergens, L. Heinemann and M. Conradt: “Index-Based Code Clone Detection: Incremental, Distributed, Scalable”, Proceedings of the 26th International Conference on Software Maintenance, pp. 1–9 (2010).

[20] N. Göde and R. Koschke: “Incremental clone detection”, Proceedings of the 13th European Conference on Software Maintenance and Reengineering, pp. 219–228 (2009).

[21] M. Datar, N. Immorlica, P. Indyk and V. Mirrokni: “Locality-sensitive hashing scheme based on p-stable distributions”, Proceedings of the 20th Symposium on Computational Geometry, pp. 253–262 (2004).

[22] L. Jiang, G. Misherghi, Z. Su and S. Glondu: “DECKARD: Scalable and accurate tree-based detection of code clones”, Proceedings of the 29th International Conference on Software Engineering, pp. 96–105 (2007).

[23] M. Gabel, L. Jiang and Z. Su: “Scalable detection of semantic clones”, Proceedings of the 30th international conference on Software engineering, pp. 321–330 (2008).

[24] M. Lee, J. Roh, S. Hwang and S. Kim: “Instant code clone search”, Proceedings of the 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering, pp. 167–176 (2010).

[25] S. Bellon, R. Koschke, G. Antoniol, J. Krinke and E. Merlo: “Comparison and evaluation of clone detection tools”, IEEE Transactions on Software Engineering, **33**, 9, pp. 577–591 (2007).

[26] Y. Higo and S. Kusumoto: “Enhancing Quality of Code Clone Detection with Program Dependency Graph”, Proceedings of the 16th Working Conference on Reverse EngineeringIEEE, pp. 315–316 (2009).

[27] Y. Higo, T. Kamiya, S. Kusumoto and K. Inoue: “Method and implementation for investigating code clones in a software system”, Information and Software Technology, **49**, 9-10, pp. 985–998 (2007).

[28] 肥後芳樹, 楠本真二：“コードクローン検出に必要な計算コストの削減を目的としたプログラム依存グラフ頂点集約手法の提案”, ソフトウェアエンジニアリング最前線 2010(ソフトウェアエンジニアリングシンポジウム 2010 予稿集), pp. 127–134 (2010).

[29] 三宅達也, 肥後芳樹, 楠本真二, 井上克郎：“多言語対応メトリクス計測プラグイン開発基盤 masu の開発”, 電子情報通信学会論文誌 D, **J92-D**, 9, pp. 1518–1531 (2009).

[30] “CodeSurfer”. available at http://www.grammatech.com/products/codesurfer/.

[31] “SQLite”. available at http://www.sqlite.org/.

[32] R. Rivest: “The MD5 Message-Digest Algorithm”, RFC 1321 (1992).

[33] “TV-Browser”. available at http://www.tvbrowser.org/.

[34] “the Apache Ant project”. available at http://ant.apache.org/.

[35] “e4 project”. available at http://www.eclipse.org/e4/.

[36] “Simian - Similarity Analyser”. available at http://www.redhillconsulting.com.au/products/simian/.

[37] “CCFinderX”. available at http://www.ccfinder.net/.

[38] E. Adar and M. Kim: “SoftGUESS: Visualization and exploration of code clones in context”, Proceedings of the 29th International Conference on Software Engineering (2007).