特別研究報告

題目

クローンペアマッピングに基づく
複数コードクローン検出結果の比較法

指導教員

井上 克郎 教授

報告者

松島 一樹

平成 31 年 2 月 12 日

大阪大学 基礎工学部 情報科学科

平成 30 年度 特別研究報告

クローンペアマッピングに基づく複数コードクローン検出結果の比較法

松島 一樹

内容梗概

コードクローンとは, ソースコード中に存在する「全く同一のコード片」もしくは「類似したコード片」のことである. コードクローンは, 主に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によって生じる. もし, 欠陥が含まれたコード片にコードクローンが存在する場合, それらすべてにも欠陥が含まれている可能性が高い. 開発者はすべてのコードクローンに対して修正を行うか検討する必要があり, 欠陥の修正に大きなコストが必要となる. そのため, コードクローンはソフトウェア開発の保守工程における大きな問題の 1 つとして指摘されている. 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, 大規模なソフトウェアにおいては, 手作業ですべてのコードクローンを探し出すことは非現実的である. そこで, コードクローンを自動的に検出するための様々な手法が開発され, コードクローン検出ツールとして実装されている.

一方 Bellon らや Wang らの研究から, 同一のソースコード集合に対してであっても, コードクローン検出ツールごとの特性や検出パラメータによって検出されるコードクローンは大きく変化することが明らかになっている. そのため, 様々なコードクローン検出ツールやパラメータでコードクローンを検出し, 得られた結果の共通部分や差異を知ることは重要である.

しかし, これらの研究を始めとしてコードクローン検出結果はベンチマークに対する再現率や適合率といった基準で比較されてきたが, 含まれているコードクローンの共通部分や差異といった基準での比較は行われていなかった.

そこで, 本研究では同一ソースコード集合を対象とした複数のコードクローン検出結果に対してクローンペア間の対応に注目し, 定性的・定量的な評価を容易にする比較法を提案する.

評価実験では, Java で実装された 3 つのソフトウェアに対して異なる 3 つのコードクローン検出ツールをそれぞれ適用し, 得られたコードクローン検出結果を本ツールを用いて比較した. また, 同じ 3 つのソフトウェアに対してあるコードクローン検出ツールを 3 つの異なるパラメータでそれぞれ適用し, 得られたコードクローン検出結果を本ツールを用いて比較した. そして, 提案手法によりコードクローン検出結果の比較が容易に行えることを確認した.

主な用語

コードクローン
ソフトウェア保守
クローンペアマッピング

目 次

1　まえがき

コードクローンとは, ソースコード中に存在する「全く同一のコード片」もしくは「類似したコード片」のことである. コードクローンの発生原因として, 主に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や, コード生成ツールによる自動生成が挙げられる [1]. もし, 欠陥が含まれたコード片にコードクローンが存在する場合, それらすべてにも欠陥が含まれている可能性が高い. 開発者はすべてのコードクローンに対して修正を行うか検討する必要があり, 欠陥の修正に大きなコストが必要となる. そのため, コードクローンはソフトウェア開発の保守工程における大きな問題の 1 つとして指摘されている.

開発者がコードクローンの管理や修正を行うことで, 潜在的な欠陥の増大の抑制やソースコード量の削減につながり, ソフトウェアの保守性を高く保つことができる. そのため, 開発者がコードクローンに関する情報を認識することは重要である. しかし, 大規模なソフトウェアにおいては, 手作業ですべてのコードクローンを探し出すことは非現実的である. そこで, コードクローンを自動的に検出するための様々な手法が開発され, コードクローン検出ツールとして実装されている [2].

一方 Bellon ら [3] や Wang ら [4] の研究から, 同一のソースコード集合に対してであっても, コードクローン検出ツールごとの特性や検出パラメータによって検出されるコードクローンは大きく変化することが明らかになっている. そのため, 様々なコードクローン検出ツールやパラメータでコードクローン検出を行い, 得られた結果の共通部分や差異を知ることは重要である.

しかし, これらの研究を始めとしてコードクローン検出結果はベンチマークに対する再現率や適合率といった基準で比較されてきたが, 含まれているコードクローンの共通部分や差異といった基準での比較は行われていなかった.

そこで, 本研究では同一ソースコード集合を対象とした複数のコードクローン検出結果に対してクローンペア間の対応に注目し, 定性的・定量的な評価を容易にする比較法を提案する.

評価実験では, Apache-Ant[5], fastjson[6] および Joda-Time[7] のそれぞれのソフトウェアに対して, CCFinderX[22], CCVolti[11] および NiCAD[9] のそれぞれのコードクローン検出ツールを適用し, 得られたコードクローン検出結果を提案手法を用いて比較した. また, Apache-Ant, fastjson および Joda-Time に対して, 異なる 3 つのパラメータを設定した NiCAD を適用し, 得られたコードクローン検出結果を提案手法を用いて比較した. そして, 提案手法によりコードクローン検出結果の比較が容易に行えることを確認した.

2 章では本研究の背景として, コードクローンと, パラメータおよびコードクローン検出ツールによるコードクローン検出結果の変化について述べる. 3 章では同一ソースコード集

合を対象とした複数のコードクローン検出結果に対してクローンペア間の対応に注目し, 定性的・定量的な評価を容易にする比較法について説明する. 4 章ではソフトウェア Apache-Ant, fast-json および Joda-Time とコードクローン検出ツール CCFinderX, CCVolti および NiCAD を用いた提案手法の適用実験について述べる. 最後に 5 章では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の定義と, コードクローン検出ツールおよび検出パラメータの違いがコードクローン検出結果に与える影響について述べる.

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「全く同一のコード片」もしくは「類似したコード片」のことである. コードクローンは, 主に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やコード生成ツールによる自動生成によって生じる.

図 1 は 2 つのファイルに存在するコードクローンの例である. 一般的に互いに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呼び, コードクローンの同値類をクローンセットと呼ぶ.

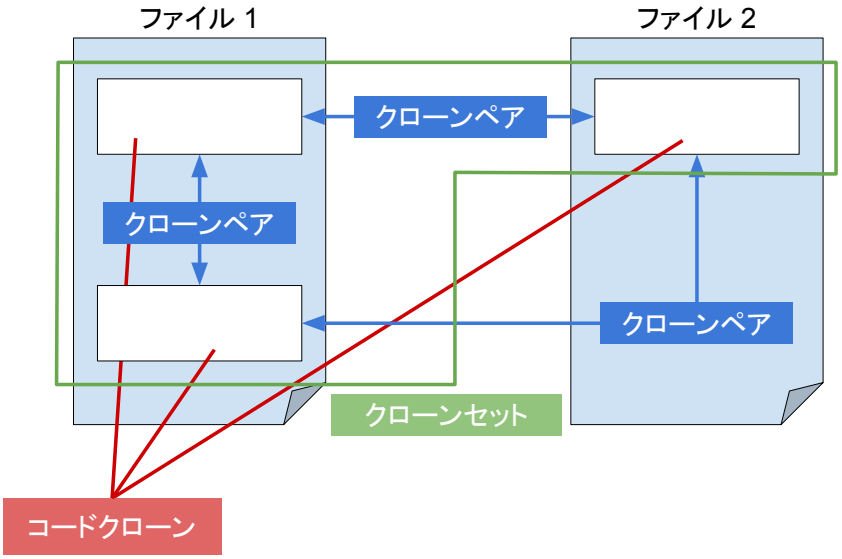

図 1: コードクローンの例

2.1.1 コードクローンの分類

コードクローンには普遍的定義が存在せず, コードクローン検出ツールごとに異なった定義を持つ. 本論文では, Roy らがコードクローンの差異の度合いに基づいて定義した, 4 つのタイプの分類を用いる [8].

タイプ 1

空白, 改行, コメントなどの違いを除いて一致するコードクローン.

タイプ 2

タイプ 1 に加えて識別子, リテラル, 型の違いを除いて一致するコードクローン.

タイプ 3

タイプ 2 に加えて, 文の変更・挿入・削除などの違いを除いて一致するコードクローン.

タイプ 4

構文上の実装は異なるが, 同様の処理を行うコードクローン.

2.1.2 コードクローン検出ツール

大規模なソフトウェアにおいては, 手作業ですべてのコードクローンを探し出すことは非現実的である. そこで, コードクローンを自動的に検出するための様々な手法が提案されている. 以下にその代表的なものを 4 つ示す.

テキストベースの検出

テキストベースの検出では, 入力ソースコード文字列を直接比較し, 類似度が閾値以上であるものをコードクローンとして検出する. コーディングスタイルが検出結果に影響を与えてしまう可能性もあるため, 比較の際にソースコード文字列の正規化や, 一部変形を行う手法も存在する. テキストベースの検出を行うコードクローン検出ツールの 1 つとして NiCAD[9] がある. NiCAD は行単位でソースコードを比較しコードクローンを検出するが, オプションにより変数名や関数名などの識別子を無視したりソースコードを変形したりすることで, タイプ 3 までのコードクローンを検出することができる.

トークンベースの検出

トークンベースの検出では, 字句解析で入力ソースコードをトークン列に変換し, トークン列の類似度が閾値以上であるものをコードクローンとして検出する. テキストベースの検出のようにコーディングスタイルが検出結果に影響を与えることは無いが, プ

ログラミング言語の文法規則を無視したコードクローンを検出してしまうことがある．トークンベースの検出を行うコードクローン検出ツールの 1 つとして CCFinder[10] がある．CCFinder は字句解析時に変数名や関数名などの識別子をある 1 つのトークンに置き換えることで, タイプ 2 までのコードクローンを検出することができる．

ベクトルベースの検出

ベクトルベースの検出では, コード片をベクトル表現に変換し, ベクトル表現の類似度が閾値以上であるものをコードクローンとして検出する．ベクトルベースで検出することで, トークンや構文単位で類似していないコード片でもベクトル空間上で距離が近ければコードクローンとして検出できる．ベクトルベースの検出を行うコードクローン検出ツールの 1 つとして CCVolti[11] がある．CCVolti では TF-IDF[12] を用いてトークン列をベクトル表現に変換し, LSH (Locality-Sensitive Hashing)[13] を用いてクラスタリングすることでコードクローンを検出する．TF-IDF を用いることでトークン列や構文の類似度に依存せずに検出を行えるため, タイプ 4 までのコードクローンを検出することができる．

抽象構文木ベースの検出

抽象構文木ベースの検出では, 構文解析で入力ソースコードを抽象構文木に変換し, 各部分木の類似度が閾値以上であるものをコードクローンとして検出する．抽象構文木をもとにしているため, 検出されるコードクローンはすべて文法規則に沿う形になっている．抽象構文木ベースの検出を行うコードクローン検出ツールの 1 つとして Deckard[14] がある．Deckard では抽象構文木の各部分木から特徴ベクトルを計算し, LSH を用いてクラスタリングすることでコードクローンを検出する．LSH は近傍の特徴ベクトルに対して高確率で同じハッシュ値を出力するため, タイプ 3 までのコードクローンを検出することができる．

2.2 コードクローン分析の困難さ

開発者はコードクローン検出結果を分析することで, コードクローンに対して保守作業を検討する．保守作業の例として以下の 2 つが挙げられる．

集約

クローンセット中のコード片と同様の処理を実装するサブルーチンを作り, 各コード片をそのサブルーチンの呼び出し文に置き換えることである．集約により, コードクローンの数とソースコード量を削減する．

同時修正
: あるコード片を修正する際に, そのコード片のコードクローンに対しても一貫した修正を行うことである.

このように, コードクローン検出結果を分析しコードクローンに対して保守作業を行うことで, ソフトウェアの品質を向上させることができる. しかし, 同一のソースコード集合に対してであっても, コードクローン検出ツールごとの特性や検出パラメータによって検出されるコードクローンは大きく変化することが明らかになっている. そのため, コードクローン検出結果によっては着目すべきコードクローンが含まれていない可能性がある. 開発者がソースコードに対する保守作業を検討する際, 着目すべきコードクローンがコードクローン検出結果に含まれていなければ正確な分析は困難である.

本節では Bellon ら [3] や Wang ら [4] の研究 から, パラメータおよびコードクローン検出ツールの違いによるコードクローン検出結果の変化について述べる.

2.2.1 パラメータによる検出結果の変化

一般的に, コードクローン検出ツールにはいくつかのパラメータが存在する. 例えば NiCAD では以下のようなパラメータが存在する.

threshold
: コードクローンとして検出するコード片の非類似度の閾値.

minsize
: コードクローンとして検出するコード片の行数の最小値.

maxsize
: コードクローンとして検出するコード片の行数の最大値.

rename
: ソースコード文字列中の識別子を別の識別子に置き換える際の置換方法.

filter
: 指定された非終端記号に対応するソースコードを含むコードクローンを除外する.

これ以外にもツールごとにパラメータの数や種類は様々である. ほとんどのツールではデフォルトパラメータが用意されているが, パラメータ設定はコードクローンの検出結果に大きな影響を与えるため, 望ましい結果を得るためには適切なパラメータを選択する必要がある.

表 1: コードクローン数の変化

	検出されたコードクローン数	コードクローンと判断された数
デフォルト	5,552	43
調整後	1,543	43

パラメータがコードクローン検出結果に大きな影響を与える例として Bellon らによる CCFinder を用いた実験を挙げる [3]. この実験では Java で実装されたプログラム netbeans-javadoc のソースコードに対して CCFinder を用いて, デフォルトパラメータと調整したパラメータでコードクローンを検出した. そして, 検出されたコードクローン数とそのうちコード片を目視で確認し実際にコードクローンであると判断した数を比較した.

その結果を表 1 に示す. デフォルトパラメータから調整後検出されたコードクローン数は 3 分の 1 以下にまで減少している. しかし, 実際にコードクローンだった数はどちらも 43 個で横ばいである. よってこの例では, パラメータの調整により再現率が変化しないまま適合率が 3 倍以上向上している.

このようにコードクローン検出結果はパラメータ設定により変化する. しかし, 望ましい結果を得るためにどのようにパラメータを設定すれば良いか理解すること, つまり, パラメータから検出結果を予測することは非常に困難である. あるコードクローン検出ツールには一般的に複数のパラメータが存在し, それぞれのパラメータがコードクローン検出結果に大きな影響を与えるためである.

表 2: 対象となったコードクローン検出ツール

ツール名	検出手法	検出可能なコードクローンタイプ
PMD's CPD 5.0[15]	トークンベース	1, 2
IClones 0.1[16]	トークンベース	1, 2, 3
CCFinder 10.2.7.4[17]	トークンベース	1, 2, 3
ConQAT 2011.9[18]	トークンベース	1, 2
Simian 1.5.0.13[19]	テキストベース	1, 2
NiCAD 3.2[9]	テキストベース	1, 2, 3

表 3: 対象となったソフトウェア

略称	ソフトウェア名	言語	ファイル数	LOC
weltab	weltab	C	65	11,700
cook	cook	C	590	80,408
snns	snns	C	625	120,764
psql	postgresql	C	612	234,971
javadoc	netbeans-javadoc	Java	101	14,360
ant	eclipse-ant	Java	178	34,744
jdtcore	eclipse-jdtcore	Java	741	147,634
swing	j2sdk1.4.0-javax-swing	Java	538	204,037

2.2.2 コードクローン検出ツールによる検出結果の変化

Wang らは, 6 つのコードクローン検出ツールと 8 つのソフトウェアを用いてコードクローン検出を行い, それぞれの検出結果を比較した [4]. 対象となったコードクローン検出ツールを表 2 に, ソフトウェアを表 3 に示す. 表 3 中の LOC はソフトウェアごとの全ソースコードの物理行数を表す.

Wang らは $AgreedLOC[i]$ をちょうど i 種類のコードクローン検出ツールで検出された行数の合計と定義した. 例えば $AgreedLOC[1]$ は 6 つのコードクローン検出ツールのうちどれか 1 つだけで検出された行数の合計を表し, $AgreedLOC[6]$ は 6 つすべてのコードクローン検出ツールで検出された行数を表す. そして, 表 2 の 6 つのコードクローン検出ツールのそれぞれのデフォルトパラメータを用いて, 表 3 の 8 つのソフトウェアに対してコードクローン検出を行った. ソフトウェアごとの $AgreedLOC[i]$ の値を表 4 に示す.

表 4: ソフトウェアごとの $AgreedLOC[i]$ の値

i	*weltab*	*cook*	*snns*	*psql*	*javadoc*	*ant*	*jdtcore*	*swing*
6	6,941	2,156	6,197	6,598	725	227	11,002	5,774
5	909	2,094	7,409	4,523	448	296	5,222	4,001
4	452	3,214	6,161	6,690	574	459	6,853	7,769
3	355	3,773	4,706	7,501	470	519	6,878	5,558
2	393	7,964	9,231	17,822	800	832	11,714	9,780
1	937	9,996	13,225	28,014	2,124	3,976	25,657	33,007
$\sum$	9,987	29,197	46,929	71,148	5,141	6,309	67,326	65,889

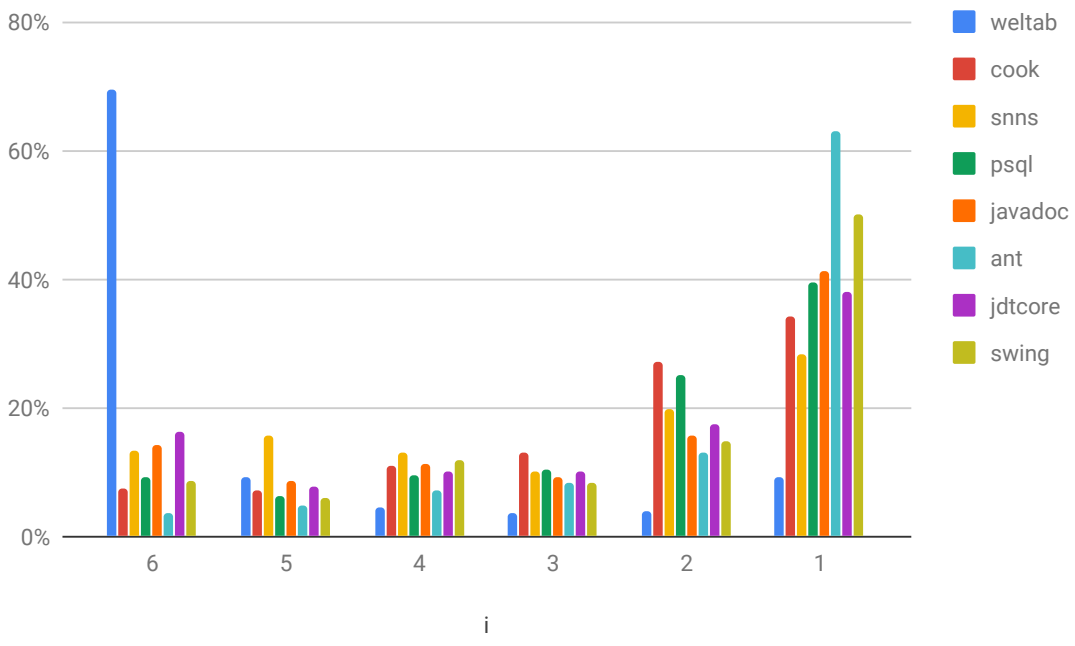

図 2: 各 $AgreedLOC[i]$ が検出されたコードクローンの行数の合計に占める割合

また, 各 $AgreedLOC[i]$ が検出されたコードクローンの行数の合計 (表 4 の $\sum$) に占める割合を図 2 に示す.

weltab 以外のすべてのソフトウェアで $AgreedLOC[1]$ がおおよそ 30 % から 60 % を占めている. 更に, $AgreedLOC[6]$ が占める割合は 10 % 程度である. つまり, ある 1 つのコードクローン検出ツールで検出されたコードクローンの多くはほかのコードクローン検出ツールでは検出されないもので, 全ツールで共通して得られるコードクローンは 10 % 程度である.

この結果から, コードクローン検出ツールの違いがコードクローン検出結果に大きな影響を与えることが明らかとなった.

2.3 コードクローン検出結果の可視化手法

コードクローン検出結果を可視化する既存の手法として, クローン散布図が挙げられる. クローン散布図とは, ソースコード中のどの位置にコードクローンが存在するかをプロットした図である.

CCFinderX に付属するコードクローン分析ツール GemX で実際に用いられるクローン散布図を図 3 に示す. GemX のクローン散布図では, 左上角を原点として, 横軸・縦軸共にソースコードのトークン列を表す. そして, 両軸の対応するトークンが一致している場合に

点が描画されている. また, 描画される点は左上角から右下角への対角線を軸として線対称となっている. クローン散布図を見ることで, ソースコード中のどの位置にどの程度の大きさのコードクローンが存在しているか容易に理解することができる.

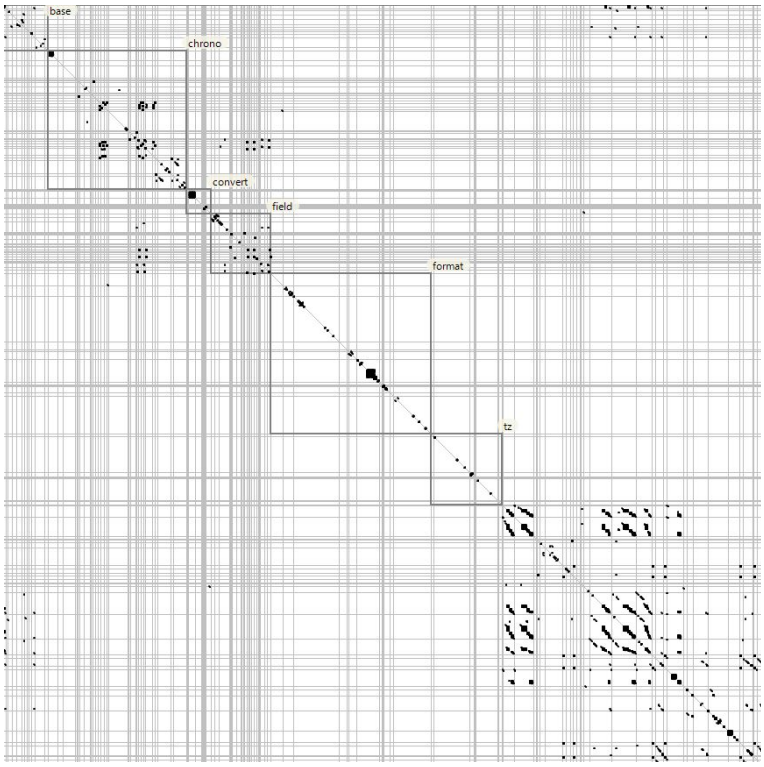

図 3: GemX の散布図

3　提案手法

本章では, 提案手法の処理概要とその各ステップの詳細について述べる. また, 提案手法を用いたツールの実装についても述べる. ツールに関する実装はすべて C++ で行った.

図 4 は提案手法の処理概要である. 複数のコードクローン検出結果を入力とし 4 つのステップで処理が行われる.

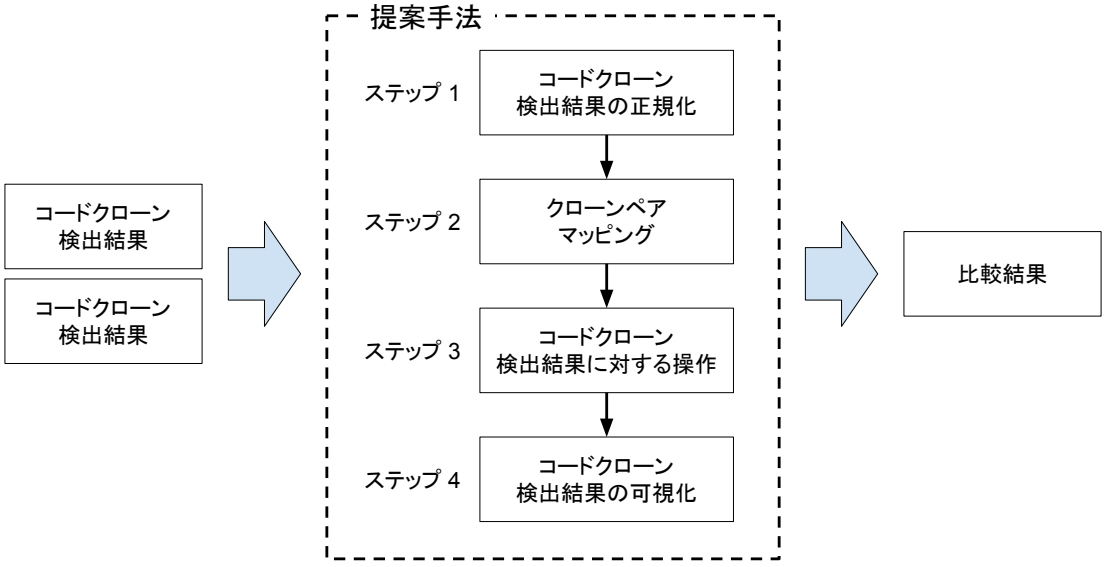

図 4: 提案手法の処理概要

3.1 ステップ 1: コードクローン検出結果の正規化

コード片は一般的に以下の 3 つの要素から定義される.

- コード片が含まれるファイル
- 開始位置
- 終了位置

しかし, これらの出力形式はコードクローン検出ツールにより様々である. 提案手法を用いて実装したツールが対応しているコードクローン検出ツール CCFinderX, CCVolti および NiCAD の出力形式について表 5 にまとめた.

提案手法ではコード片をファイル ID, 開始行番号および終了行番号の 3 つの要素から定義し, この形式に合うように各コードクローン検出結果を変換する. ファイル ID はファイルの出現順に 0, 1, $\cdots$ のように割り当てられる整数値である. また, 開始行番号と終了行番号は 1 以上の整数値で表される.

以降, コード片 f を構成する 3 つの要素についてファイル ID を $file\_id$, 開始行番号を $begin$, 終了行番号を end とし, それぞれ $f.file\_id$, $f.begin$, $f.end$ のように表す.

クローンペアは以下の 3 つの要素から定義される.

- 類似度
- コード片 1
- コード片 2

以降, クローンペア p を構成する 3 つの要素について類似度を $similarity$, コード片 1 を f_1, コード片 2 を f_2 とし, それぞれ $p.similarity$, $p.f_1$, $p.f_2$ のように表す.

また, コード片に関して次のような順序関係を定義する.

表 5: ツールごとのコード片の出力形式

ツール名	ファイル	開始位置	終了位置
CCFinderX	ファイル ID	トークン番号	トークン番号
CCVolti	ファイルパス	行番号	行番号
NiCAD	ファイルパス	行番号	行番号

任意のコード片 f_1, f_2 に対して,

$$
\begin{aligned}
f_1 < f_2 \Leftrightarrow & (f_1.file\_id < f_2.file\_id) \vee \\
& (f_1.file\_id = f_2.file\_id \wedge f_1.begin < f_2.begin) \vee \\
& (f_1.file\_id = f_2.file\_id \wedge f_1.begin = f_2.begin \wedge f_1.end < f_2.end)
\end{aligned}
$$

以降, 任意のクローンペア p に対して, 常に $p.f_1 < p.f_2$ が成り立つものとする. この順序関係が成り立たないクローンペアについては条件を満たすようにコード片 1 とコード片 2 の入れ替えを行う.

ステップ 1 によりすべてのコードクローン検出結果を正規化し, 統一された形式で扱うことが可能になった.

3.2 ステップ 2: クローンペアマッピング

ステップ 2 では複数の検出結果間でクローンペア同士の対応付け (クローンペアマッピング) を行う. 対応するクローンペアの判定にはクローンペアを構成するコード片の完全一致ではなく, *good* 値と *ok* 値に基づいて行う [3]. *good* 値は 2 つのクローンペアを構成するコード片同士の重複度を表すメトリクスである. また, *ok* 値は 2 つのクローンペアを構成するコード片同士の包含度を表すメトリクスである.

コードクローン検出ツールの空行や改行の扱いの違いにより, 同じ部分を指しているクローンペアであってもコード片の範囲がずれることがある. *good* 値と *ok* 値に基づいてマッピングを行うことで, 範囲がずれているコード片で構成されたクローンペア同士も対応していると判定することが可能となる.

以降, コード片 f に対し, $lines(f)$ が f に含まれるソースコード行の集合を表し, $|lines(f)|$ が $lines(f)$ の要素数, つまり f の行数を表す (図 5).

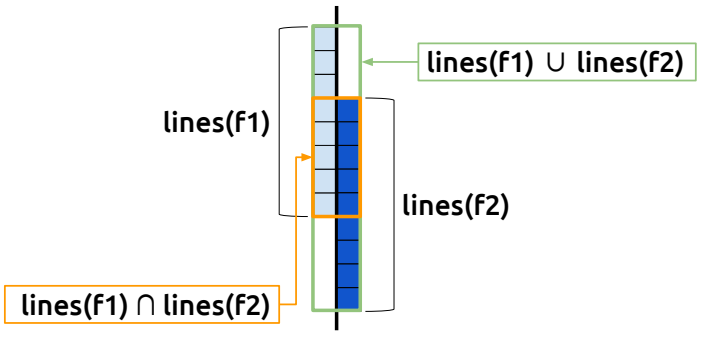

図 5: $lines(f_1)$ と $lines(f_2)$ の例

まず, *good* 値と *ok* 値を求めるために 2 つの関数 *overlap* と *contained* を定義する.
任意のコード片 f_1, f_2 に対して,

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

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

次に *good* 値と *ok* 値の定義について述べる.
任意のクローンペア p_1, p_2 に対して,

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

$$ok(p_1, p_2) = min(max(contained(p_1.f_1, p_2.f_1), contained(p_2.f_1, p_1.f_1)), max(contained(p_1.f_2, p_2.f_2), contained(p_2.f_2, p_1.f_2)))$$

これらをもとに 2 つのコードクローン検出結果 r_1, r_2 において, r_1 から r_2 へのクローンペアマッピングをアルゴリズム 1 のように定義する. アルゴリズム 1 は r_1 の各クローンペアに対して r_2 のクローンペアをそれぞれ最大 1 つ対応させる.

Algorithm 1 r_1 から r_2 へのクローンペアマッピング

```
procedure uni_map(result r1, result r2)
    for each clone_pair p2 ∈ r2.clone_pairs do
        for each clone_pair p1 ∈ r1.clone_pairs do
            ok_max ← ok(p1, corresponding_clone_pair_of[p1])
            good_max ← good(p1, corresponding_clone_pair_of[p1])
            ok ← ok(p1, p2)
            good ← good(p1, p2)
            if better(good, good_max, ok, ok_max) then
                corresponding_clone_pair_of[p1] ← p2
            end if
        end for
    end for
    return corresponding_clone_pair_of
end procedure
```

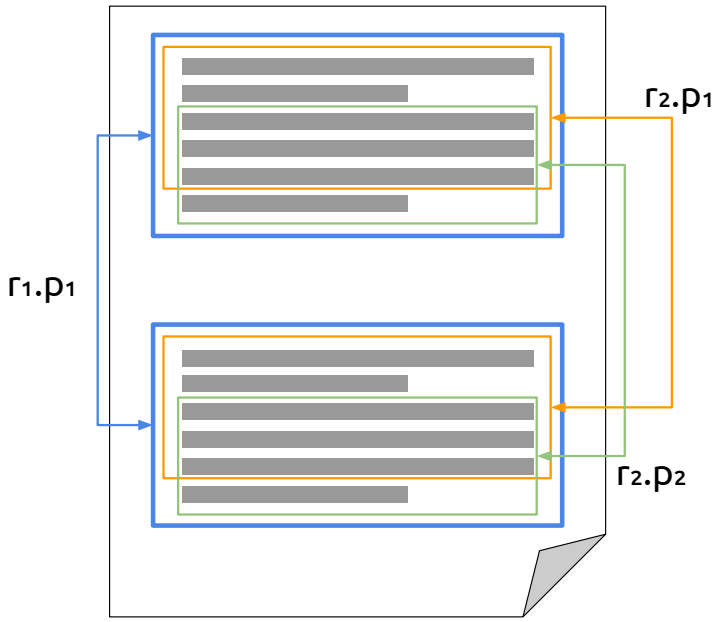

図 6: 2 つに分割されて検出されたクローンペアの例

アルゴリズム 1 中の関数 *better* は以下の条件を満たす時に真となる.

$$\begin{aligned} better(good, good\_max, ok, ok\_max) =& (good \geq p \wedge good > good\_max) \vee \\ & (good = good\_max \wedge ok > ok\_max) \vee \\ & (ok \geq p \wedge ok\_max < p) \end{aligned}$$

p とはあらかじめ与えられる閾値であり, 本研究ではこの閾値として論文 [3] で用いられている値 0.7 を用いた.

uni_map では r_2 のクローンペア 1 つが r_1 の複数のクローンペアと対応することはあるが, 逆に r_2 の複数のクローンペアが r_1 のクローンペア 1 つと対応することはない. そのため, 図 6 のように r_1 では 1 つとして検出されたクローンペアが r_2 でいくつかに分割されて検出された場合は, それぞれに対して完全にマッピングを行うことができない.

そこで **uni_map** を複数回用いて, r_1 と r_2 のクローンペアマッピングをアルゴリズム 2 のように定義する.

map 中の関数 *merge* は 2 つの **uni_map** の戻り値を受け取り, 両方のクローンペアの対応関係の和集合を返す. r_1 から r_2 へ, そして r_2 から r_1 へのクローンペアマッピングを行い両方の結果を統合することで, 図 6 のような場合でもより正確なマッピングを行うことができる.

Algorithm 2 r_1 と r_2 のクローンペアマッピング

```
procedure map(result r1, result r2)
    mapping ← merge(uni_map(r1, r2), uni_map(r2, r1))
    return mapping
end procedure
```

3.3 ステップ 3: コードクローン検出結果に対する操作

ステップ 3 は検出結果に対する操作を行う. 提案手法を実装したツールにはいくつのかの操作が用意されている. そのうちクローンペアマッピング情報を利用した操作として, 2 つのコードクローン検出結果 r_1 と r_2 に対して, 図 7 のように 6 つの領域を設定し, それぞれの領域を求めるアルゴリズムについて述べる.

図 7 で設定された 6 つの領域の詳細は, それぞれ以下の通りである.

領域 R_1

検出結果 r_1 に含まれるクローンペアの集合.

領域 R_2

検出結果 r_2 に含まれるクローンペアの集合.

領域 Ⅰ

r_1 に含まれるクローンペアのうち, r_2 に含まれるクローンペアと対応関係が存在しないものの集合.

領域 Ⅱ

r_1 に含まれるクローンペアのうち, r_2 に含まれるクローンペアと対応関係が存在するものの集合.

領域 Ⅲ

r_2 に含まれるクローンペアのうち, r_1 に含まれるクローンペアと対応関係が存在するものの集合.

領域 Ⅳ

r_2 に含まれるクローンペアのうち, r_1 に含まれるクローンペアと対応関係が存在しないものの集合.

領域 Ⅰ と領域 Ⅳ の定義は r_1 と r_2 の順序のみが異なる. そこで, 操作 **mapped**(r_a, r_b) のアルゴリズムを, 検出結果 r_a に含まれるクローンペアのうち, 検出結果 r_b に含まれるクローンペアと対応関係が存在しないものの集合を戻り値として返すように定義する. これに

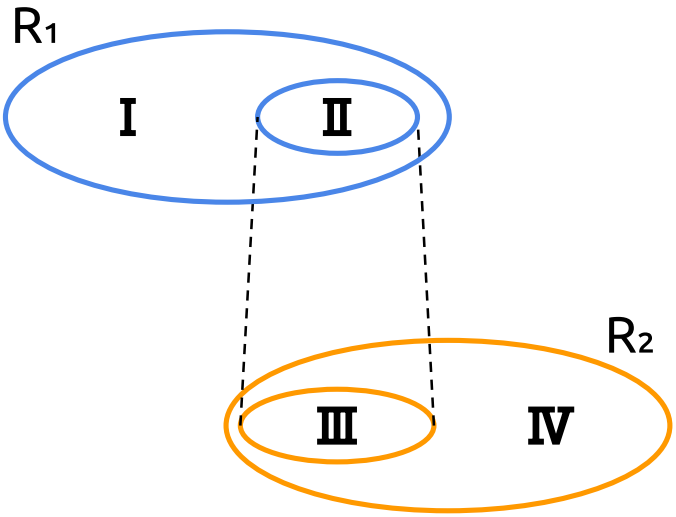

図 7: 検出結果 r_1 と r_2 に対する 6 つの領域の設定

より, 領域 I を **mapped**(r_1, r_2) で, 領域 IV を **mapped**(r_2, r_1) で求めることができる. 操作 **mapped** のアルゴリズムをアルゴリズム 3 に示す.

Algorithm 3 操作 **mapped** のアルゴリズム

```
procedure mapped(result r_a, result r_b)
    mapping ← map(r_a, r_b)
    for each clone_pair p ∈ r_a.clone_pairs do
        if mapping.has_corresponding_clone_pair_of(p) then
            mapped_set.add(p)
        end if
    end for
    return mapped_set
end procedure
```

同様に, 領域 II と領域 III の定義は r_1 と r_2 の順序のみが異なる. そこで, 操作 **unmapped**(r_a, r_b) のアルゴリズムを, 検出結果 r_a に含まれるクローンペアのうち, 検出結果 r_b に含まれるクローンペアと対応関係が存在するものの集合を戻り値として返すように定義する. これにより, 領域 II を **unmapped**(r_1, r_2) で, 領域 III を **unmapped**(r_2, r_1) で求めることができる. 操作 **unmapped** のアルゴリズムをアルゴリズム 4 に示す.

3.4 ステップ 4: コードクローン検出結果の可視化

ステップ 4 としてコードクローン検出結果の可視化を行う.

提案手法の散布図を図 8 に示す. 提案手法の散布図は左上を原点として, 横軸・縦軸共に

Algorithm 4 操作 **unmapped** のアルゴリズム

```
procedure unmapped(result r_a, result r_b)
    mapping ← map(r_a, r_b)
    for each clone_pair p ∈ r_a.clone_pairs do
        if not mapping.has_corresponding_clone_pair_of(p) then
            unmapped_set.add(p)
        end if
    end for
    return unmapped_set
end procedure
```

ファイル ID 列を表し, ファイル ID の最大値を $w-1$ とすると 1 辺の長さは w となる. 散布図上の座標 (x, y) は $0 \leq x, y \leq w-1$ を満たす. また, $(0, 0)$ と $(w-1, w-1)$ を結ぶ直線で線対称である.

提案手法で出力される散布図は GemX の散布図を参考にしているが, 2 つの点で異なる.

1 つ目として, GemX の散布図はソースコード中のトークン列の一致・不一致を表示しているのに対し, 提案手法ではファイル単位で表示を行っていることが挙げられる. GemX はトークン列単位で点を描画しているため, ファイル内のどの部分がコードクローンとして検出されているのか散布図上で確認することができる. 一方で, 大規模なプロジェクトでは大量に点を描画する必要があるため描画コストが高くなってしまう. また点が密集する部分では視認性が低下するといった問題もある. 提案手法ではファイル単位で表示を行うことで, コードクローン位置を把握することはできないが, 描画コストを下げ, ある程度点が密集した部分でも視認性を確保した.

2 つ目として, GemX の点は白と黒の 2 色表示であるが, 提案手法では散布図の点の保持する値を定義し, その値に応じて彩色されて描画されることが挙げられる. 色付きで描画することで他の点との違いを強調することができる.

提案手法では, クローンペア数とクローンペアのマッチング率の 2 種類の可視化を行い, 各散布図を出力する. それぞれの可視化の詳細について第 3.4.1 項と第 3.4.2 項で述べる.

3.4.1 クローンペア数の可視化

クローンペア数の可視化はコードクローン検出結果ごとに行われる. 点 (x, y) に対して

$$(p.f_1.file\_id = x \wedge p.f_2.file\_id = y) \vee$$
$$(p.f_1.file\_id = y \wedge p.f_2.file\_id = x)$$

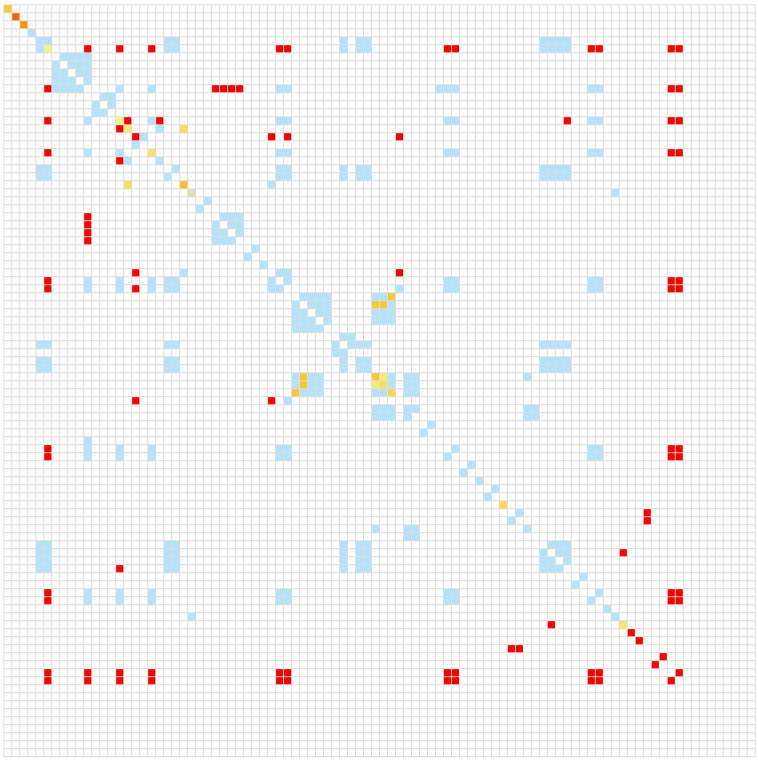

図 8: 提案手法の散布図の例

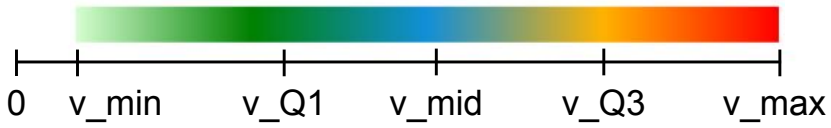

図 9: クローンペア数の配色の例

を満たすような検出結果 r に含まれるクローンペア p の集合を $r.clone\_pairs(x, y)$ とする. また, その集合の要素数を $|r.clone\_pairs(x, y)|$ とすると点 (x, y) の値 $v(x, y)$ は

$$v(x, y) = |r.clone\_pairs(x, y)|$$

で定義される.

クローンペア数の可視化はこの $v(x, y)$ の値に応じて点を彩色することで行われる. すべての $x \leq y$ を満たす点に対して, 最大値を v_{max}, 第 3 四分位数を v_{Q3}, 中央値を v_{mid}, 第 1 四分位数を v_{Q1}, 最小値を v_{min} としたときの配色の例を図 9 に示す.

3.4.2 クローンペアのマッチング率の可視化

マッチしたクローンペアの割合の可視化は指定された複数のコードクローン検出結果をもとに行われる.

はじめに, 2 つのコードクローン検出結果 r_1, r_2 が指定された場合の定義について述べる.

$p_{r_1} \in r_1.clone\_pairs(x,y)$ と $p_{r_2} \in r_2.clone\_pairs(x,y)$ の対応関係が $\mathbf{map}(r_1, r_2)$ に存在するとき, 集合 $\{p_{r_1}, p_{r_2}\}$ をマッチングセットと呼ぶ. そしてすべてのマッチングセットの和集合を $m(x,y)$ とする.

次に $N(x,y)$ を

$$N(x,y) = r_1.clone\_pairs(x,y) \cup r_2.clone\_pairs(x,y)$$

と定義する.

$m(x,y)$ と $N(x,y)$ を用いて点 (x,y) の値 $v(x,y)$ を以下のように定義する.

$$v(x,y) = \frac{|m(x,y)|}{|N(x,y)|}$$

これが点 (x,y) におけるクローンペアのマッチング率である.

また,

$$v_{avg} = \frac{|\bigcup_{y=0}^{w-1} \bigcup_{x=0}^{w-1} m(x,y)|}{|\bigcup_{y=0}^{w-1} \bigcup_{x=0}^{w-1} N(x,y)|}$$

を r_1 と r_2 における平均マッチング率と呼ぶ.

図 10 を例に 2 つのコードクローン検出結果間の点 (x,y) におけるマッチング率を求める. 図中の対応するクローンペア間には黒線をひいている. 上記のマッチングセットの定義より, 図 10 に存在するマッチングセットは, $\{r_1.p_1, r_2.p_1\}$, $\{r_1.p_2, r_2.p_1\}$, $\{r_1.p_3, r_2.p_2\}$, $\{r_1.p_3, r_2.p_3\}$, $\{r_1.p_4, r_2.p_5\}$ の 5 つである. これらマッチングセットの和集合を求めることで

$$m(x,y) = \{r_1.p_1, r_1.p_2, r_1.p_3, r_1.p_4, r_2.p_1, r_2.p_2, r_2.p_3, r_2.p_5\}$$

となる. また, 図より

$$\begin{aligned} N(x,y) &= r_1.clone\_pairs(x,y) \cup r_2.clone\_pairs(x,y) \\ &= \{r_1.p_1, r_1.p_2, r_1.p_3, r_1.p_4, r_2.p_1, r_2.p_2, r_2.p_3, r_2.p_4, r_2.p_5\} \end{aligned}$$

となる. よって, 点 (x,y) でのマッチング率は,

$$\begin{aligned} v(x,y) &= \frac{|m(x,y)|}{|N(x,y)|} \\ &= \frac{8}{9} \end{aligned}$$

となる.

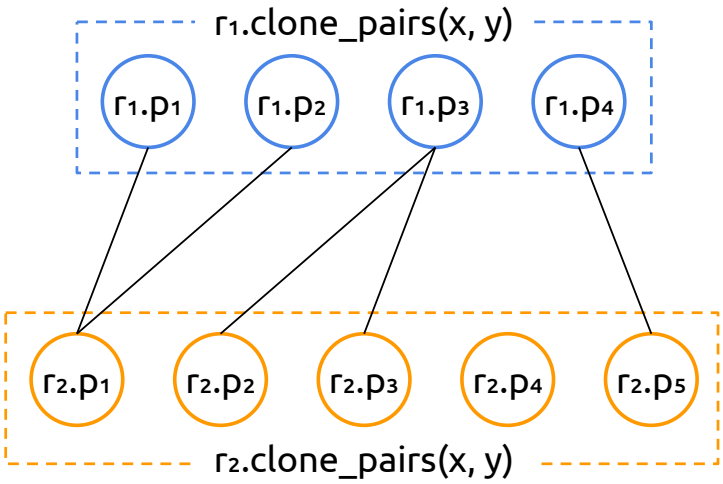

図 10: 2 つのコードクローン検出結果の例

次に, 上記の $m(x,y)$ と $N(x,y)$ の定義を拡張して, 任意の n 個のコードクローン検出結果 r_1, r_2, $\cdots$, r_n が指定された場合の定義について述べる.

$1 \leq i \leq n$ に対して $p_{r_i} \in r_i.clone\_pairs(x,y)$ を満たすようなある集合 $\{p_{r_1}, p_{r_2}, \cdots, p_{r_n}\}$ に含まれるすべての 2 つのクローンペア間に対応関係が存在しているとき, この集合をマッチングセットと呼ぶ. そして, すべてのマッチングセットの和集合を $m(x,y)$ とする.

次に $N(x,y)$ を

$$N(x,y) = \bigcup_{i=1}^{n} r_i.clone\_pairs(x,y)$$

と定義する.

点 (x,y) におけるクローンペアのマッチング率はコードクローン検出結果が 2 つの場合と同様に $m(x,y)$ と $N(x,y)$ を用いて

$$v(x,y) = \frac{|m(x,y)|}{|N(x,y)|}$$

と定義される.

また, $r_1, r_2, \cdots, r_n$ における平均マッチング率も同様に

$$v_{avg} = \frac{|\bigcup_{y=0}^{w-1} \bigcup_{x=0}^{w-1} m(x,y)|}{|\bigcup_{y=0}^{w-1} \bigcup_{x=0}^{w-1} N(x,y)|}$$

と定義する.

図 11 を例に 3 つのコードクローン検出結果間の点 (x,y) におけるマッチング率を求める. 図中の対応するクローンペア間には黒線をひいている. 上記のマッチングセットの定義

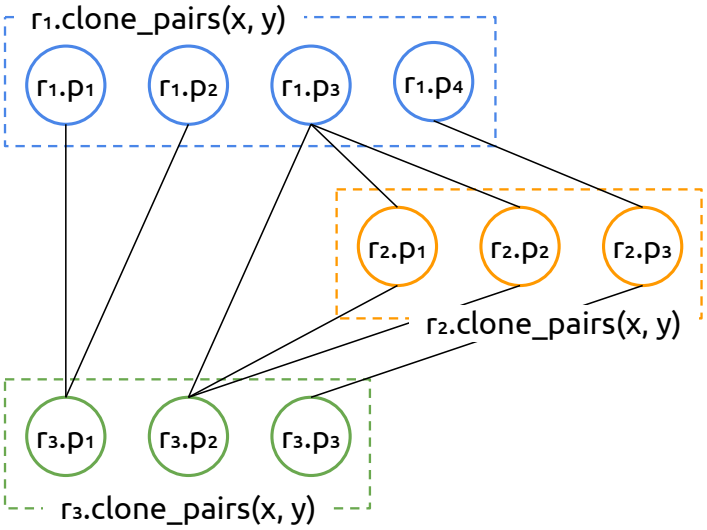

図 11: 3 つのコードクローン検出結果の例

より, 図 11 に存在するマッチングセットは, $\{r_1.p_3, r_2.p_1, r_3.p_2\}$ と $\{r_1.p_3, r_2.p_2, r_3.p_2\}$ の 2 つである. $\{r_1.p_4, r_2.p_3, r_3.p_3\}$ は $r_1.p_4$, $r_3.p_3$ 間に対応関係が存在しないため, マッチングセットではない.

これらのマッチングセットの和集合を求めることで

$$m(x, y) = \{r_1.p_3, r_2.p_1, r_2.p_2, r_3.p_2\}$$

となる. また, 図より

$$\begin{aligned} N(x, y) &= \bigcup_{i=1}^{3} r_i.clone\_pairs(x, y) \\ &= r_1.clone\_pairs(x, y) \cup r_2.clone\_pairs(x, y) \cup r_3.clone\_pairs(x, y) \\ &= \{r_1.p_1, r_1.p_2, r_1.p_3, r_1.p_4, r_2.p_1, r_2.p_2, r_2.p_3, r_3.p_1, r_3.p_2, r_3.p_3\} \end{aligned}$$

となる. よって, 点 (x, y) でのマッチング率は

$$\begin{aligned} v(x, y) &= \frac{|m(x, y)|}{|N(x, y)|} \\ &= \frac{2}{5} \end{aligned}$$

となる.

クローンペアのマッチング率の可視化はこの $v(x, y)$ の値に応じて点を彩色することで行われる. 配色の例を図 12 に示す. 0 % を赤色として, 99 % に近づくにつれて黄色へと変化

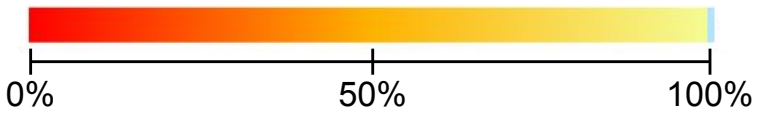

図 12: マッチング率の配色の例

する. $v(x,y)$ が 100 %, つまりすべてのクローンペアに対応関係が存在するときのみ水色で彩色される. また, $r.clone\_pairs(x,y) = 0$ のときは $v(x,y)$ の値に関わらず, 常に白色で彩色される.

3.5 データ構造

第 3.4 節で述べた散布図は, 1 辺の長さを w として $(0,0)$ と $(w-1, w-1)$ を結ぶ直線で線対称である. つまり, 散布図上の点 (x,y) が持つ値 $v(x,y)$ は $v(y,x)$ と等しい. よって, $x \leq y$ となるような点の値のみ保持しておくことでメモリ消費を半分程度に抑えることができる.

実装したツールでは散布図の値を一次元配列として保持し, データアクセス時に二次元座標 (x,y) から対応する一次元座標 $k_{(x,y)}$ へ変換する. 図 13 に 4×4 の散布図と一次元配列の対応の例を示す. 太枠で囲まれた点は $x \leq y$ となる部分である.

変換式は以下の式で定義される.

$x \leq y$ の時

$$\begin{aligned} k_{(x,y)} &= x + \sum_{i=0}^{y-1}(i+1) \\ &= x + \frac{y(y+1)}{2} \end{aligned}$$

$x > y$ の時は x と y の値を入れ替えてこの式に代入すれば良い.

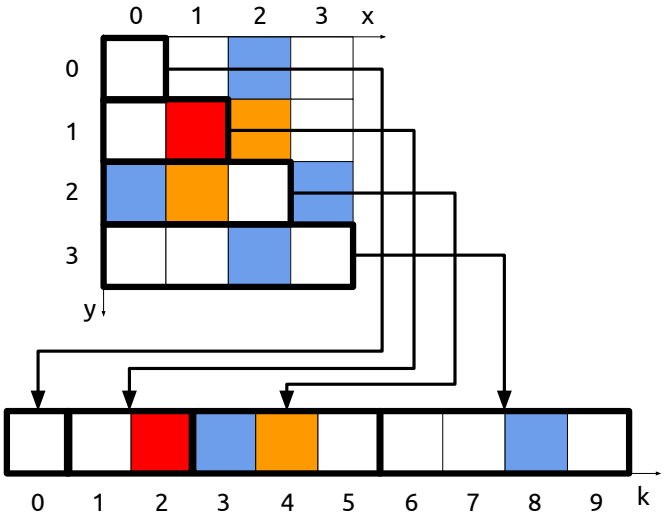

図 13: 4×4 の散布図と一次元配列の対応

4 評価実験

本章では, 3 つのソフトウェア Apache-Ant, fastjson および Joda-Time と 3 つのコードクローン検出ツール CCFinderX, CCVolti および NiCAD を対象とした 2 つの評価実験について述べる.

評価実験 1 では, 異なる 2 つのコードクローン検出ツールの, それぞれのデフォルトパラメータ設定で得られた検出結果の比較を行った. 評価実験 2 では, 同一コードクローン検出ツールの, 異なる 2 つのパラメータ設定で得られた検出結果の比較を行った.

これらの評価実験から, 提案手法によりコードクローン検出結果の比較が容易に行えることを確認した.

対象としたソフトウェアの詳細とコードクローン検出ツールの詳細をそれぞれ表 6, 表 7 に示す. 表 6 中の LOC はソフトウェアごとの全ソースコードの物理行数を表し, 論理 LOC は LOC から空行とコメントのみの行を除いた行数である.

表 6: 対象のソフトウェア

略称	ソフトウェア名	言語	ファイル数	LOC	論理 LOC
ant	Apache-Ant 1.10.3	Java	887	226,237	109,831
fastjson	fastjson 1.2.55	Java	172	51,268	39,458
joda	Joda-Time 2.10.1	Java	166	70,898	28,790

表 7: 対象のコードクローン検出ツール

ツール名	検出手法	検出可能なコードクローンタイプ
CCFinderX	トークンベース	1, 2
CCVolti	トークンベース	1, 2, 3, 4
NiCAD	テキストベース	1, 2, 3

4.1 評価実験 1: 異なる 2 つのコードクローン検出ツールでの検出結果の比較

評価実験 1 では, 異なる 2 つのコードクローン検出ツールの, それぞれのデフォルトパラメータ設定で得られた検出結果の比較を行う.

実験手順としては, まず CCFinderX, CCVolti および NiCAD にデフォルトパラメータを設定した. 次に, すべてのソフトウェアに対してそれぞれコードクローン検出を行った. そして, 得られた検出結果から 2 つずつを選択し本ツールを用いて比較した.

各ツールのデフォルトパラメータを以下に示す.

- CCFinderX

  Minimum Clone Length (デフォルト値: 50)
  コードクローンとして検出するコード片に含まれるトークン数の最小値.

  Minimum TKS (デフォルト値: 12)
  コードクローンとして検出するコード片に含まれるトークンの種類の最小値.

  Shaper Level (デフォルト値: 2-Soft Shaper)
  使用するブロックシェイパ [20] の種類.

  P-match Application (デフォルト値: use)
  P-match[21] を使用するか否か.

  - not use : P-match を使用しない.
  - use : P-match を使用する.

  Prescreening Application (デフォルト値: not use)
  多くのコードクローンを含むソースファイルを対象外とするか否か.

  - not use : ソースファイルを対象外としない.
  - use : ソースファイルを対象外とする.

- CCVolti

  sim (デフォルト値: 0.9)
  検出するコードクローンの類似度の最小値.

  size (デフォルト値: 50)
  コードクローンとして検出するメソッドに含まれるトークン数の最小値.

  sizeb (デフォルト値: size と同じ)
  コードクローンとして検出するコードブロックに含まれるトークン数の最小値.

- NiCAD

  granularity（デフォルト値: functions)
  コードクローン検出を行う粒度.
  - blocks：コードブロック単位で検出する.
  - functions：関数単位で検出する.

  threshold（デフォルト値: 0.3)
  コードクローンとして検出するコード片の非類似度の閾値.

  minsize（デフォルト値: 10)
  コードクローンとして検出するコード片の行数の最小値.

  maxsize（デフォルト値: 2500)
  コードクローンとして検出するコード片の行数の最大値.

  transform（デフォルト値: none)
  コードクローン検出時にソースコード文字列を変形させる.
  - none：ソースコード文字列を変形させない.
  - none 以外：指定された定義ファイルを用いて, ソースコード文字列を変形させる.

  rename（デフォルト値: blind)
  ソースコード文字列中の識別子を別の識別子に置き換える際の置換方法.
  - none：識別子を置換しない.
  - blind：すべての識別子を ”X” で置換する.
  - consistent：n を整数として, 一貫性を保ったまま識別子を ”Xn” で置換する.

  filter（デフォルト値: none)
  指定された非終端記号に対応するソースコードを含むコードクローンを除外する.
  - none：文法を無視しない.
  - none 以外：指定された非終端記号に対応するソースコードを含むコードクローンを除外する.

  abstract（デフォルト値: none)
  指定された非終端記号に対応するソースコードをすべてある 1 つの識別子に置換する.
  - none：ソースコードを置換しない.

- none 以外：指定された非終端記号に対応するソースコードをすべてある 1 つの識別子に置き換える.

normalize (デフォルト値: none)

コードクローン検出時にソースコード文字列を正規化する.

- none：ソースコード文字列を正規化しない.
- none 以外：指定された定義ファイルを用いて, ソースコード文字列を正規化する.

選択した 2 つのコードクローン検出ツールを T_1, T_2 とし, それぞれの検出結果を r_1, r_2 とする. そして, 検出結果に対して, 図 14 のように 6 つの領域を設定する.

それぞれの領域の詳細は以下の通りである.

領域 R_1

コードクローン検出ツール T_1 で検出されたクローンペアの集合.

領域 R_2

コードクローン検出ツール T_2 で検出されたクローンペアの集合.

領域 Ⅰ

T_1 で検出されたクローンペアのうち, T_2 で検出されたクローンペアと対応関係が存在しないものの集合. $\mathbf{unmapped}(r_1, r_2)$ により得られる.

領域 Ⅱ

T_1 で検出されたクローンペアのうち, T_2 で検出されたクローンペアと対応関係が存在するものの集合. $\mathbf{mapped}(r_1, r_2)$ により得られる.

領域 Ⅲ

T_2 で検出されたクローンペアのうち, T_1 で検出されたクローンペアと対応関係が存在するものの集合. $\mathbf{mapped}(r_2, r_1)$ により得られる.

領域 Ⅳ

T_2 で検出されたクローンペアのうち, T_1 で検出されたクローンペアと対応関係が存在しないものの集合. $\mathbf{unmapped}(r_2, r_1)$ により得られる.

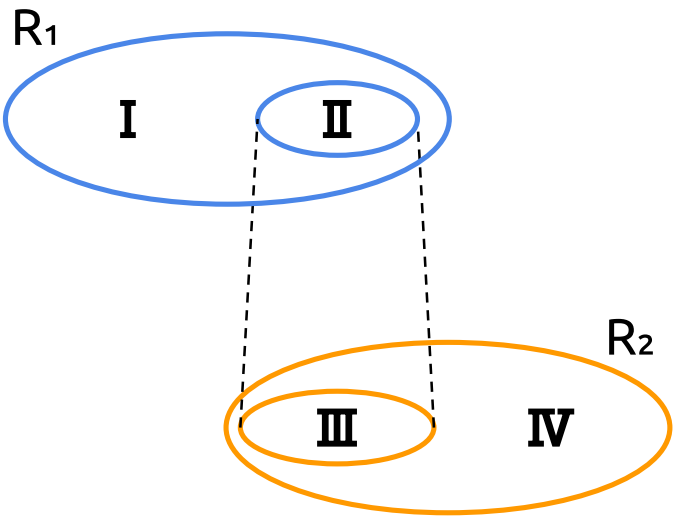

図 14: コードクローン検出結果と領域の設定

4.1.1 実験結果

まず, ソフトウェアごとの各領域のクローンペア数を表 8, 9, 10 に示す. 表中の m_{avg} は T_1 と T_2 から得られたコードクローン検出結果の平均マッチング率である. また, m_1, m_2 の定義は以下の通りである.

$$m_1 = \frac{|\ \mathrm{II}\ |}{|R_1|}$$
$$m_2 = \frac{|\ \mathrm{III}\ |}{|R_2|}$$

第 3.2 節で述べたように, クローンペアマッピングはある 1 つのクローンペアに対して複数のクローンペアが対応する可能性がある. そのため領域 II と領域 III に含まれるクローンペア数は異なる. 一方で, 領域 II と領域 III に含まれるクローンペア数が近いほどクローンペア同士が 1 対 1 で対応していることになるため, 2 つのコードクローン検出結果間で同じ粒度のクローンペアが多く検出されていることになる.

表 8: *ant* における各領域のクローンペア数

T_1	T_2	R_1	R_2	I	II	III	IV	m_1	m_2	m_{avg}
CCFinderX	CCVolti	1,397	547	1,165	232	298	249	0.17	0.54	0.27
CCFinderX	NiCAD	1,397	497	1,127	270	230	267	0.19	0.46	0.26
CCVolti	NiCAD	547	497	283	264	163	334	0.48	0.33	0.41

表 9: *fastjson* における各領域のクローンペア数

T_1	T_2	R_1	R_2	Ⅰ	Ⅱ	Ⅲ	Ⅳ	m_1	m_2	m_{avg}
CCFinderX	CCVolti	1,533	834	779	754	350	484	0.49	0.42	0.47
CCFinderX	NiCAD	1,533	136	1,249	284	96	40	0.19	0.71	0.23
CCVolti	NiCAD	834	136	654	180	108	28	0.22	0.79	0.30

表 10: *joda* における各領域のクローンペア数

T_1	T_2	R_1	R_2	Ⅰ	Ⅱ	Ⅲ	Ⅳ	m_1	m_2	m_{avg}
CCFinderX	CCVolti	2,634	671	2,220	414	504	167	0.16	0.75	0.28
CCFinderX	NiCAD	2,634	338	2,506	128	132	206	0.05	0.39	0.09
CCVolti	NiCAD	671	338	557	114	112	226	0.17	0.33	0.22

表 10 において, (T_1, T_2) が (CCFinderX, NiCAD) と (CCVolti, NiCAD) のときでは, 領域 Ⅱ と領域 Ⅲ に含まれるクローンペア数が近い値となっている. しかし, それ以外のソフトウェアとコードクローン検出ツールの組み合わせではある程度の差が存在し, 検出されるクローンペアの粒度はコードクローン検出ツールの影響を受けている.

m_{avg} は, ソフトウェアや T_1 と T_2 の組み合わせによって程度が異なるもののおおよそ 0.2 から 0.3 の範囲であり, 異なるコードクローン検出ツール間で同じ部分を指すクローンペアの割合は低い.

次に, *fastjson* に対し CCFinderX を適用して得られた検出結果を r_{CCFX}, NiCAD を適用して得られた検出結果を r_{NCD} として, これらのマッチング率の散布図を図 15 に示す. 表 9 から, *fastjson* における, CCFinderX と NiCAD の組み合わせでの平均マッチング率は 0.23 である. そのため, 図 15 でも多くの点が赤色, つまりマッチング率がほぼ 0 に近い点として彩色されている.

ここで, r_{CCFX} と r_{NCD} の領域 Ⅰ において, クローンペア数の可視化を行った結果を図 16 に示す. 図 16 中の, 青で囲まれた赤で彩色された点, つまり最もコードクローン数が多い点の座標は $(x, y) = (26, 26)$ であり, その数は $v(26, 26) = 540$ 個であった. 表 9 から領域 Ⅰ のクローンペア数は 1,127 個であるため, $r_{CCFX}.clone\_pairs(26, 26)$ がその半数近くを占めている.

$r_{CCFX}.clone\_pairs(26, 26)$ に含まれるクローンペアの例を図 17 に示す. ファイル ID 26 に対応するファイル名は `JSONLexerBase.java` であり, JSON ファイルの字句解析が実装されていた. 字句解析には定形処理が多く, if 文や case 文の連続する処理も多い. CCFinderX では図 17 のように制御文の繰り返しによるコードクローンを多数検出していた. しかし, こ

のようなコードクローンの多くは開発者にとって興味がないものであることが経験的に明らかになっている [23] ため, `JSONLexerBase.java` に含まれるコードクローンのほとんどは無視できる.

よって, 提案手法により, r_{CCFX} は図 17 のような繰り返しを含む無意味なコードクローンを多く含んでいることが確認できた.

4.1.2 実行時間と最大使用メモリ量

各ソフトウェアにおいて, コードクローン検出結果を読み込みはじめてから各領域のクローンペア数を求め, 結果を出力し終えるまでの実行時間を計測した. また, その間の最大使用メモリ量も計測した. 計測を 5 回行い, その平均を表 11 に示す.

表 11: ソフトウェアごとの本ツールの実行時間と最大使用メモリ量

ソフトウェア名	実行時間 (s)	最大使用メモリ量 (MB)
ant	42	31
fastjson	22	26
joda	19	18

4.2 評価実験 2: 異なる 2 つのパラメータ設定での検出結果の比較

評価実験 2 では, 同一コードクローン検出ツールの, 異なる 2 つのパラメータ設定で得られた検出結果の比較を行う.

実験手順としては, まず NiCAD にあらかじめ定めた 3 つのパラメータを設定した. 次に, すべてのソフトウェアに対してそれぞれコードクローン検出を行った. そして, 得られたコードクローン検出結果から 2 つずつ選択し, 本ツールを用いて比較した.

3 種類のパラメータを設定した NiCAD をそれぞれ N_1, N_2, N_3 とし, その詳細を表 12 に示す. N_1 にはデフォルトパラメータを設定し, N_2 には N_1 から granularity のみ変更したパラメータを設定した. また, N_3 には N_1 から granularity, threshold, rename および abstract の 4 つを変更したパラメータを設定した.

表 12: NiCAD に適用したパラメータの詳細

パラメータ名	N_1	N_2	N_3
granularity	functions	blocks	blocks
threshold	0.3	0.3	0.1
minsize	10	10	10
maxsize	2,500	2,500	2,500
transform	none	none	none
rename	blind	blind	none
filter	none	none	none
abstract	none	none	expression
normalize	none	none	none

4.2.1 実験結果

まず, ソフトウェアごとの各領域のクローンペア数を表 13, 14, 15 に示す.

第 4.1 節で行った実験では, 領域 Ⅱ と領域 Ⅲ のそれぞれに含まれるクローンペア数に平均で 90 個の差がある. しかし, 本節で行った実験ではその差は平均で 4 個であった. よって, N_1, N_2 および N_3 のパラメータの違いは, コードクローン検出ツールの違いと比較してクローンペアの粒度に与える影響は小さい.

また, 各ソフトウェアの N_1 と N_2 を比較した場合において, m_1 が約 1.00 となっている. これは N_1 を適用して得られたクローンペアのほとんどすべてが N_2 でも検出されていることを示しており, R_1 と R_2 がおおよそ $R_1 \subset R_2$ の関係にあることがわかる.

m_{avg} は 5 つのソフトウェアとパラメータの組み合わせで 0.5 以上だが, それ以外では 0.25 を下回っている. また, m_{avg} が低下するパラメータの組み合わせに規則性はなく, ソフトウェアごとに傾向が異なっている.

次に, *joda* に対し N_1 を適用して得られた検出結果を r_{N_1}, N_2 を適用して得られた検出結果を r_{N_2} とし, これらのマッチング率の散布図を図 18 に示す. 表 15 から, *joda* におけ

表 13: *ant* における各領域のクローンペア数

T_1	T_2	R_1	R_2	Ⅰ	Ⅱ	Ⅲ	Ⅳ	m_1	m_2	m_{avg}
N_1	N_2	497	1,115	5	492	492	623	0.99	0.44	0.61
N_1	N_3	497	2,186	254	243	243	1,943	0.49	0.11	0.18
N_2	N_3	1,115	2,186	718	397	401	1,785	0.36	0.18	0.24

表 14: *fastjson* における各領域のクローンペア数

T_1	T_2	R_1	R_2	Ⅰ	Ⅱ	Ⅲ	Ⅳ	m_1	m_2	m_{avg}
N_1	N_2	136	1,038	0	136	136	902	1.00	0.13	0.23
N_1	N_3	136	1,523	30	106	138	1,385	0.78	0.09	0.15
N_2	N_3	1,038	1,523	236	802	802	651	0.77	0.53	0.63

表 15: *joda* における各領域のクローンペア数

T_1	T_2	R_1	R_2	Ⅰ	Ⅱ	Ⅲ	Ⅳ	m_1	m_2	m_{avg}
N_1	N_2	338	448	1	337	337	111	1.00	0.75	0.86
N_1	N_3	338	728	59	279	279	449	0.83	0.38	0.52
N_2	N_3	448	728	132	316	316	412	0.71	0.43	0.54

る, N_1 と N_2 の組み合わせでの平均マッチング率は 0.86 である. そのため, 図 15 と比較して水色の点, つまりマッチング率が 1.00 である点が大幅に増加している.

ここで, r_{N_1} と r_{N_2} の領域 Ⅰ において, クローンペア数の可視化を行った結果を図 19 に示す.

図 19 中, 青で囲まれた点の座標は $(x, y) = (23, 23)$ で, クローンペア数は $v(23, 23) = 1$ 個であった. 表 9 から領域 Ⅰ のクローンペア数は 1 個であるため, これ以外のクローンペアはすべて r_{N_2} のクローンペアと対応している. このクローンペアを図 20 に示す.

rename : blind の意味に従い, 図 20 の各コード片中の関数名や変数名といった識別子をすべて "X" に置換する. すると, 2 つのコード片の差異は, 文字列リテラルと `else` 文の有無の違いだけである. この else 文は null チェックを行う if 文に付随して置かれている. 一般的に, null チェックに付随してなんらかの処理を実行するソースコードはよく書かれるものであり, ソフトウェアの欠陥を抑制するために重要である. 図 20 のコード片 2 にも, コード片 1 のように対応する else 文が必要な可能性があり, N_1 と N_2 の検出結果を提案手法で比較することで重要なコードクローンが発見できた.

4.2.2 実行時間と最大使用メモリ量

各ソフトウェアにおいて, コードクローン検出結果を読み込みはじめてから各領域のクローンペア数を求め, 結果を出力し終えるまでの実行時間を計測した. また, その間の最大使用メモリ量も計測した. 計測を 5 回行い, その平均を表 16 に示す.

表 16: ソフトウェアごとの本ツールの実行時間と最大使用メモリ量

ソフトウェア名	実行時間 (s)　最大使用メモリ量 (MB)	
ant	19	31
fastjson	11	6.9
joda	8.8	7.4

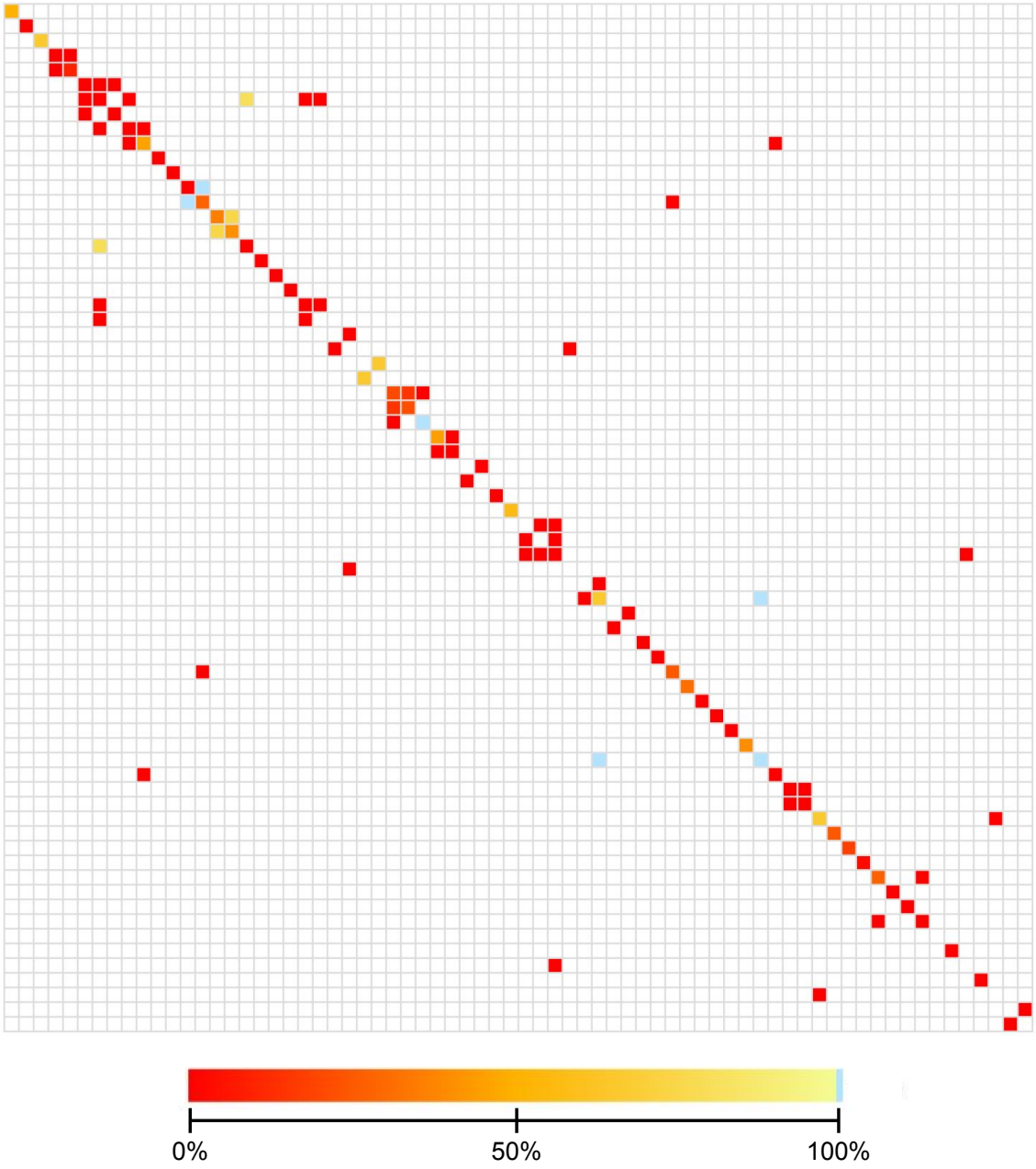

図 15: r_{CCFX} と r_{NCD} のマッチング率の可視化結果

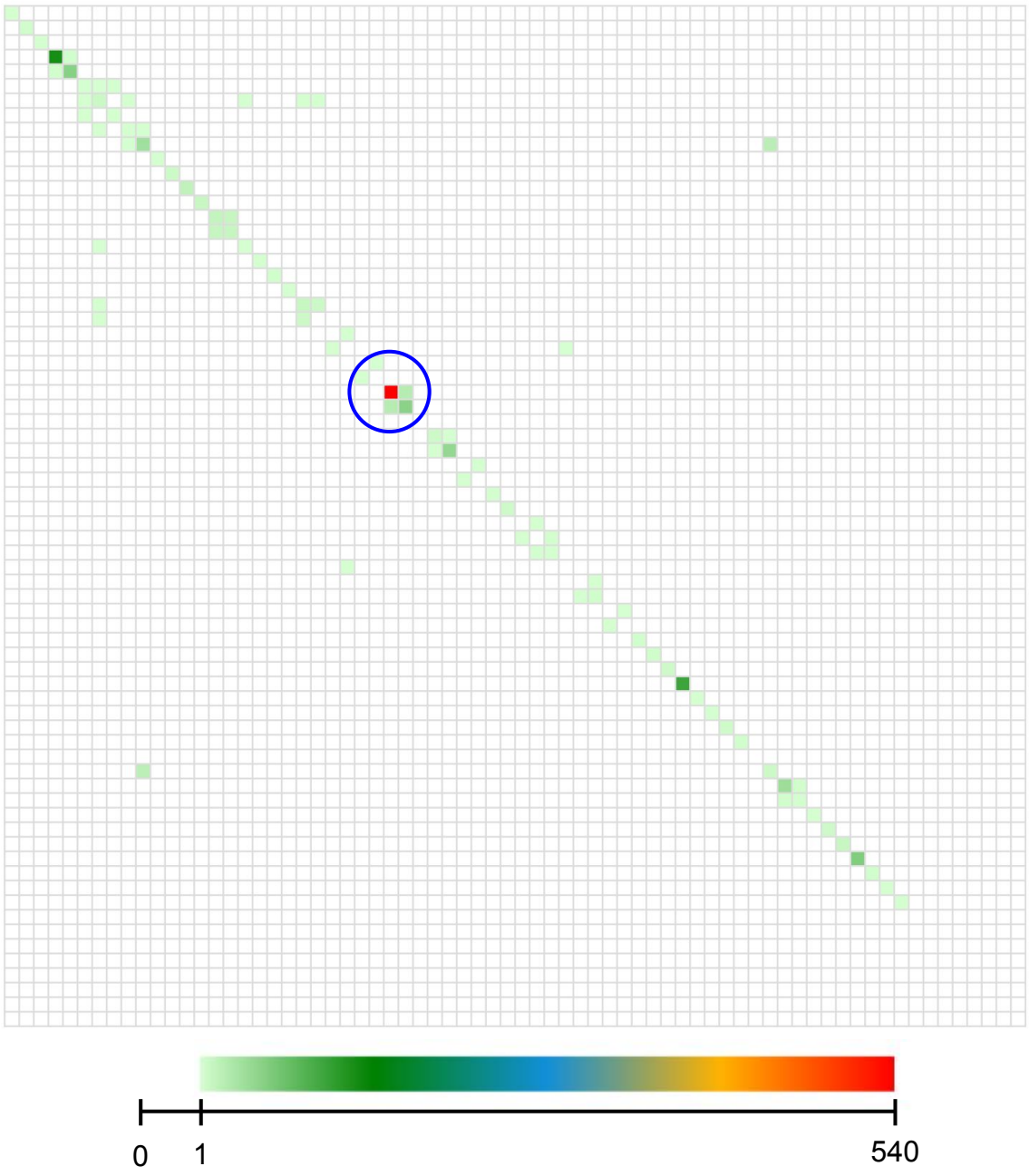

図 16: r_{CCFX} と r_{NCD} の領域 I におけるクローンペア数の可視化結果

```
if (ch == '"') {
        pos = bp;
        scanString();
        return;
}

if (ch == '[') {
        token = JSONToken.LBRACKET;
        next();
        return;
}

if (ch == '{') {
        token = JSONToken.LBRACE;
        next();
        return;
}
```

```
if (ch >= '0' && ch <= '9') {
        pos = bp;
        scanNumber();
        return;
}

if (ch == '[') {
        token = JSONToken.LBRACKET;
        next();
        return;
}

if (ch == '{') {
        token = JSONToken.LBRACE;
        next();
        return;
}
```

図 17: $r_{CCFX}.clone\_pairs(26, 26)$ に含まれるクローンペアの例

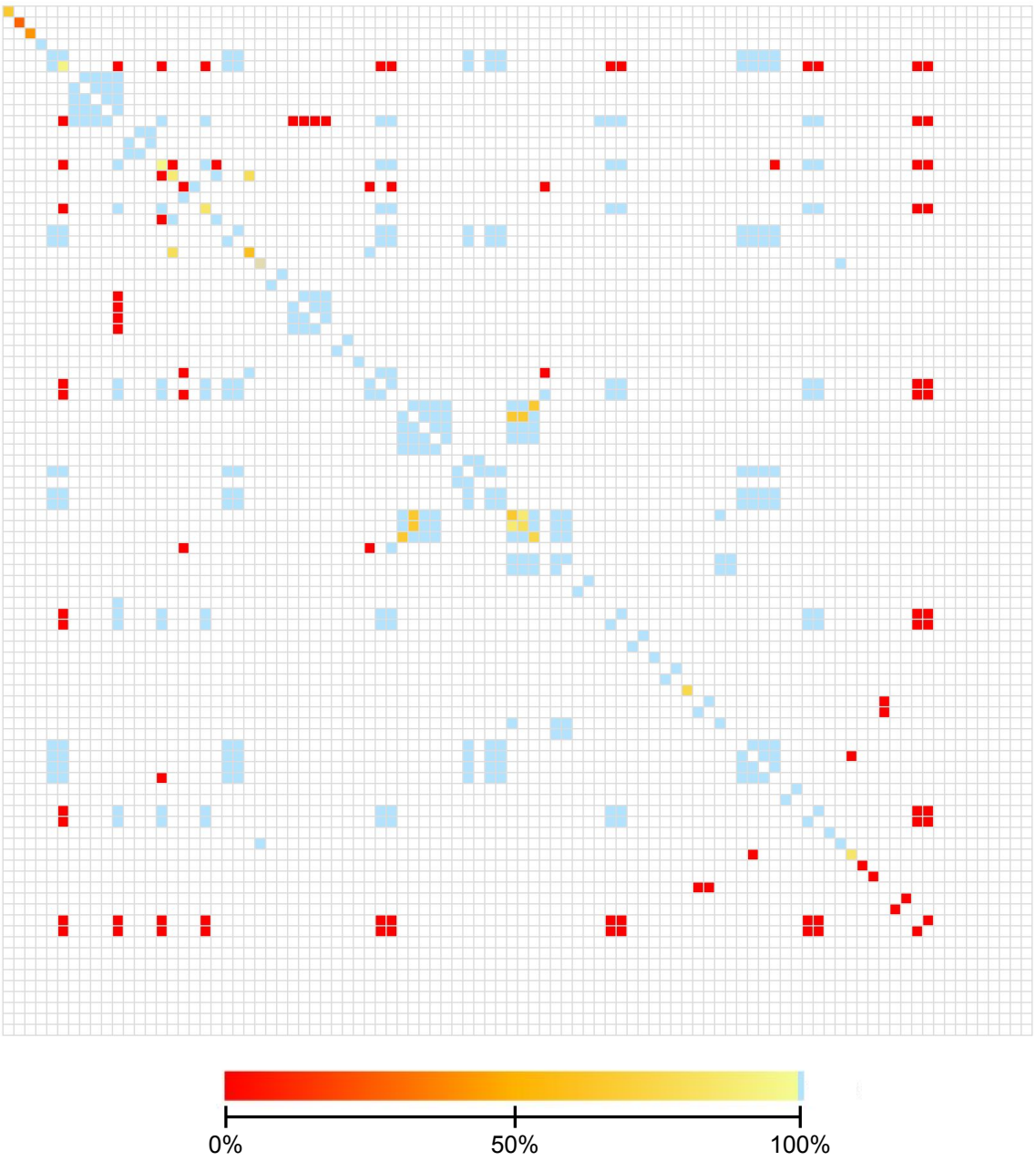

図 18: r_{N_1} と r_{N_2} のマッチング率の可視化結果

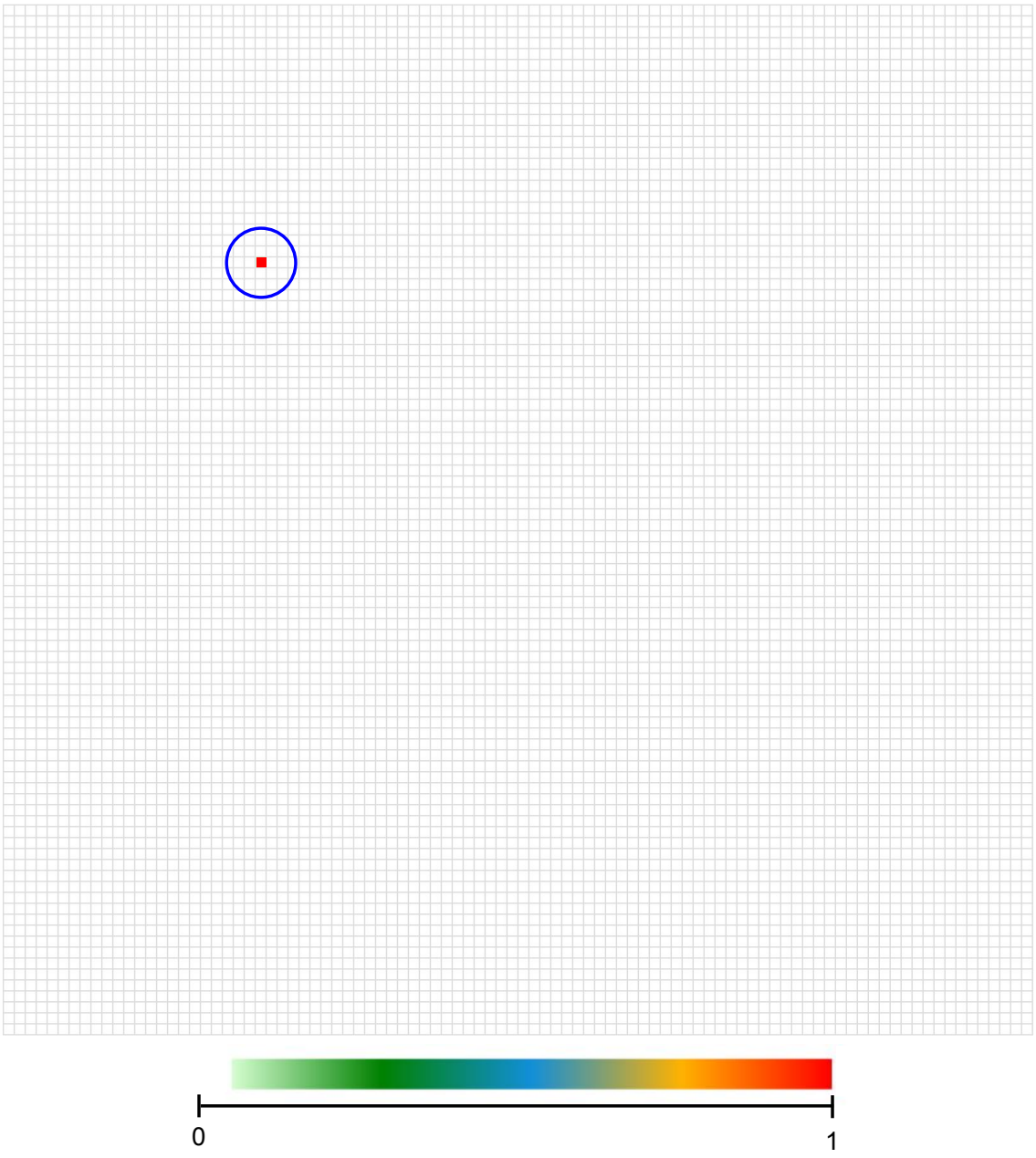

図 19: r_{N_1} と r_{N_2} の領域 I におけるクローンペア数の可視化結果

コード片 1

```
public static void setProvider(Provider provider) throws SecurityException {
    SecurityManager sm = System.getSecurityManager();
    if (sm != null) {
        sm.checkPermission(new JodaTimePermission("DateTimeZone.setProvider"));
    }
    if (provider == null) {
        provider = getDefaultProvider();
    } else {
        validateProvider(provider);
    }
    cProvider.set(provider);
}
```

コード片 2

```
public static void setNameProvider(NameProvider nameProvider) throws SecurityException {
    SecurityManager sm = System.getSecurityManager();
    if (sm != null) {
        sm.checkPermission(new JodaTimePermission("DateTimeZone.setNameProvider"));
    }
    if (nameProvider == null) {
        nameProvider = getDefaultNameProvider();
    }
    cNameProvider.set(nameProvider);
}
```

図 20: r_{N_1} と r_{N_2} の領域 I に含まれるクローンペア

5　まとめと今後の課題

本研究では, クローンペア同士の対応付けを行う手法としてクローンペアマッピングを定義した. また, 複数のコードクローン検出結果に対して 6 つの領域とマッチング率を定義し, それらを可視化する手法を提案した. そして, これらから複数コードクローン検出結果の定性的・定量的な評価を容易にする比較法を提案した.

評価実験では, 提案手法を実装したツールを用いて, 異なる 2 つのコードクローン検出ツールでの検出結果を比較した. また, 異なる 2 つのパラメータ設定での検出結果を比較した. これにより, 複数コードクローン検出結果の比較が容易に行えることを確認した.

今後の課題は 2 つある.

1 つ目の課題は, 3 つ以上のコードクローン検出結果の比較実験を行うことである.. 評価実験では 2 つのコードクローン検出結果に対し 6 つの領域を定義した. しかし, 3 つ以上のコードクローン検出結果を比較する場合, 定義される領域が増加し, それに伴い m_1 や m_2 のような値の数も増加する. そのため, 数値を理解する労力が増大し, 比較が困難になってしまうことが問題である.

2 つ目の課題は, 複数コードクローン検出結果の共通部分や差異をクローンセットの形で提示することである. 提案手法では, 複数コードクローン検出結果の共通部分や差異をクローンペアの形で提示している. しかし, 実際にコードクローンに対して開発者が集約や同時修正といった保守作業を検討するとき, 確認すべきコードクローンをまとめて比較できるため, クローンペアではなくクローンセットの形で提示するほうが利便性が高い. この機能の実現には, クローンペア同士の対応関係ではなく, クローンセット同士の対応関係に注目した比較法の開発が必要である.

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, 非常に御多忙の中, 日々研究に関する直接の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により本論文を完成させることができました. 井上 教授に厚く御礼申し上げます.

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, 本論文や発表における問題点の提示など, 多くの御助言を賜りました. 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 神田 哲也 助教には, 本論文や発表における問題点の提示など, 多くの御助言を賜りました. 神田 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 春名 修介 特任教授には, 本論文や発表における問題点の提示など, 多くの御助言を賜りました. 春名 特任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 瀬村 雄一 氏, 横井 一輝 氏には, 日々研究に関する多くの御助言を賜りました. 両氏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 藤原 裕士 氏, 本田 紘貴 氏には, 本論文の修正・添削や発表について御指導をして頂くなど, 日々研究に関する御協力を賜りました. 両氏に心より深く感謝いたします.

最後に, その他様々な御指導, 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心より深く感謝いたします.

参考文献

[1] 井上 克郎, 神谷 年洋, 楠本 真二. コードクローン検出法. コンピュータソフトウェア, Vol. 18, No. 5, pp. 47–54, 2001.

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

[3] Stefan Bellon, Rainer Koschke, Giulio Antoniol, Jens Krinke, and Ettore Merlo. Comparison and evaluation of clone detection tools. IEEE Transaction Software Engineering, Vol. 31, No. 10, pp. 804–818, 2007.

[4] Tiantian Wang, Mark Harman, Yue Jia, and Jens Krinke. Searching for better configurations: a rigorous approach to clone evaluation. In European Software Engineering Conference and the Symposium on the Foundations of Software Engineering, pp. 455–465, Aug. 2013.

[5] The Apache Software Foundation. Apache Ant. https://ant.apache.org/.

[6] Alibaba Group Holding Ltd. alibaba/fastjson: A fast JSON parser/generator for Java. https://github.com/alibaba/fastjson.

[7] Stephen Colebourne. Joda-Time. https://www.joda.org/joda-time/.

[8] 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.

[9] Chanchal K. Roy and James R. Cordy. NICAD: Accurate Detection of Near-Miss Intentional Clones Using Flexible Pretty-Printing and Code Normalization. In Proceedings of the 16th IEEE International Conference on Program Comprehension, ICPC 2008, pp. 172-181, 2008.

[10] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. CCFinder: a multilinguistic token-based code clone detection system for large scale source code. IEEE Transaction Software Engineering, Vol. 28, No. 1, pp. 654–670, 2002.

[11] 横井 一輝, 崔 恩瀞, 吉田 則裕, 井上 克郎. 情報検索技術に基づく細粒度ブロッククローン検出. コンピュータ ソフトウェア, Vol. 35, No. 4, pp. 16-36, 2018.

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

[13] 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, STOC 1998, pp. 604–613, 1998.

[14] Lingxiao Jiang, Ghassan Misherghi, Zhendong Su, and Stéphane Glondu. DECKARD: Scalable and Accurate Tree-based Detection of Code Clones. In Proceedings of the 29th International Conference on Software Engineering, ICSE 2007, pp. 96-105, 2007.

[15] PMD Open Source Project. PMD. https://pmd.github.io/.

[16] Nils Gode and Rainer Koschke. Incremental clone detection. In Proceedings of the Euromicro Conference on Software Maintenance and Reengineering, CSMR 2009, Mar. 2009.

[17] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. CCFinder: a multilinguistic token-based code clone detection system for large scale source code. Annual report of Osaka University: academic achievement, Vol. 2001, pp. 22–25, 2002.

[18] Elmar Juergens, Florian Deissenboeck, and Benjamin Hummel. CloneDetective – a workbench for clone detection research. In Proceedings of the 31st International Conference on Software Engineering, ICSE 2009, pp. 603-606, 2009.

[19] Simon Harris. Simian - Similarity Analyser. https://www.harukizaemon.com/simian/.

[20] Yoshiki Higo, Yasushi Ueda, Toshihiro Kamiya, Shinji Kusumoto and Katsuro Inoue. On software maintenance process improvement based on code clone analysis. In Proceedings of the 4th International Conference on Product Focused Software Process Improvement, pp. 185-197, Rovaniemi, Finland, Dec. 2002.

[21] Brenda S. Baker. On Finding Duplication and Near-Duplication in Large Software Systems. In Proceedings of the 2nd IEEE Working Conference on Reverse Engineering, pp. 86-95, Jul. 1995.

[22] 神谷 年洋. The Official CCFinderX Website. http://www.ccfinder.net/ccfinderx.html.

[23] Yoshiki Higo, Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. Method and Implementation for Investigating Code Clones in a Software System. Information and Software Technology, Vol. 49, No. 9-10, pp. 985–998, 2007.