卒 業 研 究 報 告 書

題　目　　画像処理技術を用いた類似ソースコード片の
マッチング手法精度に関する比較調査

指導教員　　水野 修　教授

崔 恩瀞　助教

京都工芸繊維大学 工芸科学部 情報工学課程

学生番号　　19122047

氏　　名　　眞東　優太

令和5年2月13日提出

画像処理技術を用いた類似ソースコード片のマッチング手法精度に関する比較調査

令和 5 年 2 月 13 日　　19122047　眞東　優太

概　要

ソースコード中に存在する構文的、意味的に一致、または類似したソースコード片は類似ソースコード片と呼ばれ、一般にソフトウェアの保守を難しくする原因の一つと言われている。その保守を容易にするために、類似ソースコード片の検出を自動的に行うツールの開発が行われてきた。しかし、近年ではブレイクスルーが見られていない。従来のソースコードを構文的に解釈した後に類似したソースコード片を探したり、文字列として類似するものを検索するのではなく、画像としてソースコードを扱い画像処理技術によって類似ソースコード片を検出することを考えた。本論文では、ソースコードを画像として扱った際に類似ソースコード片を検出するにあたって、類似画像検出技術の性能を比較することにした。

調査の結果、類似画像検出技術を用いて、同一ソースコード片を検出することができ、また類似したソースコード片を検出することができた。しかし、ソフトウェアの保守に繋がる類似ソースコード片の検出を行えてはいない結果となった。そこで、検出アルゴリズムの改良、またはソースコード画像に意味のある色彩情報（シンタックスハイライトなど）を付加するなどの情報量の追加を行うことで、検出精度を高め、ソフトウェアの保守に繋がる類似ソースコード片の検出を行えるのではないかと考えている。

目　次

1. 緒言

ソースコード片をコピーし、ペーストし、ステートメントの追加、削除、変更などの調整を行ってコードを再利用することは、ソフトウェア開発では一般的な方法である。これにより、ソースコードリポジトリ内に多くの同一または類似ソースコード片が生成される。類似ソースコード片は、ソフトウェアエンジニアリングの分野における重要な研究内容であり、ソフトウェアの保守、進化、品質、再利用と承認、盗作防止など、多くの分野で具体化されている。実証研究によると、ソフトウェアシステムには 20% から 30% の類似ソースコード片が含まれている可能性があり [1] [2] [3] [4] [5]、場合によっては最大 50% になることもある [6] 。多数の類似ソースコード片は、ソフトウェアシステムに悪影響を及ぼす [7]。たとえば、意図しない動作を含むソースコード片を繰り返し使用すると、エラーが伝播し、ソフトウェアシステムの信頼性が低下する可能性がある。なので、ソフトウェアシステムでの類似ソースコード片の検出、識別、および表示は、開発者によるソフトウェアの保守を容易にする。この問題を解決するために、多くの優れた類似ソースコード片検出方法がある [4] [8] [9] [10] [11] [12] 。しかし、これらの手法は、厳密なルール定義や複雑な中間処理など、独自のルールに従ってソースコードを字句 (token) や AST(Abstract Syntax Tree) [13] 、PDG(Program Dependence Graph) に変換する必要がある。本論文では、中間処理などが少ない類似ソースコード片の検出のために、類似ソースコード片検出に使用できる類似画像検出技術を調査した。

本論文の調査方法は、まず複数の類似画像検出技術を用いて同一ソースコード片の検出性能を比較した。次に、同一ソースコードの検出性能が高かった類似画像検出技術の要因を考察した上で、類似ソースコード片の検出性能の調査を行った。

以降は本論文の章構成を示す。まず、第 2 章では、本論文の調査で必要となる背景知識についての説明を行う。第 3 章では、本論文の関連研究について説明を行う。第 4 章では、本論文に利用するデータセットや調査手法、及び調査結果の説明を行う。第 5 章では、その調査結果について各類似画像検索技術の特徴を踏まえて考察を行う。第 6 章では、今後の課題について説明し、最後の第 7 章では、結言を示す。

2. 研究背景

2.1 類似ソースコード片

2.1.1 類似ソースコード片

類似ソースコード片とは、ソースコード中に存在する互いに一致、または類似したものを指す [14] 。多くの場合、一部のソースコードをコピー、及びペーストすることによって生成される。ソフトウェアの保守を困難にする要因の 1 つとして類似コード片が挙げられる。例えば、あるコピー元のソースコード片に修正点が見つかった場合、ペースト先のソースコード片にも修正点が含まれる可能性がある。そのため、すべてのペースト先を把握し、それぞれに対して修正を行うか検討しなければならない。

2.1.2 類似コードタイプ

類似コードタイプには、基本的に 2 種類があります。プログラムテキストの類似性に基づいて類似している場合もあれば、テキストが類似していなくても機能が類似している場合もあります。

Bellon [8] は、プログラムテキストの類似性に基づいて類似している類似ソースコードにおいて、それぞれの違いの度合に基づき、それらを以下の三つに分類している。

Type1 類似ソースコード　空白やタブの有無、括弧の位置などのコーディングスタイルを除いて、完全に一致する類似ソースコード

Type2 類似ソースコード　変数名や関数名などのユーザ定義名、また変数の型 などの一部の予約語のみが異なる類似ソースコード

Type3 類似ソースコード　type-2 における変更に加えて、文の挿入や削除、変更が行われた類似ソースコード

上記の類似タイプは多くの場合、ソースコード片をコピーして別の場所に貼り付けた結果である。

ソースコード片の機能が同一または類似している場合、それらをセマンティッククローン、 またはType4類似ソースコードと呼ぶ [15] [16] 。

上記のタイプのレベルは増加するごとに、より検出が複雑となる。プログラムの構築とソフトウェアの設計について多くの背景知識を持っていたとしても、特にタイプ4類似ソースコードの検出は困難となる。

2.1.3 従来の検出ツール (NiCad)

類似ソースコード片、及び類似ソースコードと呼ばれるものを検出するツールは複数存在しており、多くの検出手法が提唱されている。検出するにあたって、ソースコードを文字、字句 (token)、行、文、関数、手続き、クラス定義などの粒度で分割を行い、独自の方法で類似度が高いものを検出している。Nicad は Roy と Cordy が開発した類似ソースコード検出ツールである [9] 。Nicad はその検出の単位として、ブロック単位または関数単位のいずれかを選択できる。Nicad は、すべてのブロックや関数のペアについて LCS（Longest CommonSubsequence）アルゴリズムを用いることで、一致するブロックまたは関数を特定し、それらを類似ソースコードとして検出する。

2.2 マッチング手法

本節ではマッチング手法について、説明する。

2.2.1 特徴点マッチング

特徴ベースマッチングとは、2 つの画像間で一致する特徴点を抽出し、その一致特徴点の近傍領域を使って照合する手法である。特徴点検出、特徴量記述、マッチングの三ステップが行われる。特徴点検出では画像中から角や線の交点、エッジなどの固有な点を特徴点として特徴点の座標を検出する。次に、特徴量記述では検出した特徴点を中心とした近傍領域を含めた画素情報を用いて、特徴点の固有性をベクトルなどで表現した値を特徴量として算出する。最後に、マッチングでは 2 つの画像同士の特徴量を比較する。この際、特徴量の距離が近いものを類似度が高いものとするが、特徴量の表現方法によって距離が変わるため類似度が高いと判別できるものにも差が生まれる。

(1) SIFT

D.G.Lowe が SIFT（Scale Invariant Feature Transform）と呼ばれる、特徴検出・記述アルゴリズムを 2004 年に発表した [17]。SIFT は、Laplacian-of-Gaussian の近似である DoG（Difference-of-Gaussians）演算子を用いて特徴抽出を行うものである。入力画像をガウシアンフィルタを用いて平滑化し、段階的に平滑化のレベルを大きくして複数の平滑化画像を作る。その平滑化画像の中から平滑化段階の隣合うもの同士の Difference-of-Gaussians(DoG) 画像を作成する。ここで DoG 画像は複数できており、特徴点の候補を見つけるためには、段階的に並べられた DoG 画像 3 枚が要求される。3 枚中の中段階の画像の注目画素と、その周りの 8 近傍に、下段階、上段階の画像を対象に注目画素と同じ位置の画素を含めた周り 8 近傍で 1 枚の画像につき 9 画素分で計 18 画素を加えた計 26 画素とを比較し、注目画素が極値であった場合に、特徴点候補としている。ここから、エッジ上の点であるか、コントラストが 低い領域の点であるかを基準としてふるいにかけ、残ったものを特徴点としている。検出された各特徴点の周囲に 16 × 16 の領域を抽出し、さらにその領域をサブブロックに分割し、合計 128 のビン値をレンダリングする記述方法である。SIFT は

画像の回転、スケール、および限定的なアフィン変換、照明変化に対して頑健であるが、計算コストが高いのが主な欠点である。式 (2.1) は、（異なるスケールで計算された）2 つのガウス分布の差分と画像 ”I(x,y)” の畳み込みを示す。

$$D(x, y, \sigma) = (G(x, y, k\sigma) - G(x, y, \sigma)) * I(x, y) \tag{2.1}$$

ここで、G はガウス関数である。

(2) AKAZE

P.F.Alcantarilla らは 2013 年に AKAZE（Accelerated-KAZE）アルゴリズムを発表した [18] 。これは KAZE と同様に非線形拡散フィルタリングに基づいているが、その非線形スケール空間は FED（Fast Explicit Diffusion）という計算効率の高いフレームワークを用いて構築されている。AKAZE は、ヘシアン行列の行列式に基づく検出器である。オブジェクト境界を保持するようにぼかすことのできる非線形空間におけるフィルタリングで作成した段階的な画像に対して、注目画素の検出器の応答値が 8 近傍中の最大値であり、指定された閾値よりも高ければ特徴点候補としている。さらに、下段階、上段階の近傍とも比較して最大値 であれば特徴点としている。AKAZE の記述子は MLDB（Modified Local Difference Binary）アルゴリズムに基づいており、これも非常に効率的である。AKAZE の特徴は、オブジェクトのスケール変化、回転、照明変化、Blur にロバスト性をもっている。式 (2.2) は、標準的な非線形拡散の式である。

$$\frac{\partial L}{\partial t} = \mathrm{div}(c(x, y, t), \nabla L) \tag{2.2}$$

ここで、c は対象画像の構造に (2.2) 式を対応させるための関数、div は発散、L は画像輝度である。

(3) ORB

Rublee らは 2011 年に Oriented FAST and Rotated BRIEF(ORB) を発表した [19] 。ORB アルゴリズムは、修正 FAST(Features from Accelerated Segment Test) 検出と方

向正規化 BRIEF(Binary Robust Independent Elementary Features) 記述法のブレンドである。特徴量記述子である BRIEF は、回転不変性の欠如が欠点としてある。これを解決するために、オリエンテーション推定に勾配を用いず、パッチの画素値に関する 0、1 次モーメントから重心を算出し、特徴点中心と重心を結ぶ直線の傾きをオリエンテーション α として用いる。バイナリコードを生成する際に、画素値を比較する 2 点のペアの座標位置を α だけ回転することにより回転不変性を得ている。FAST のコーナーはスケールピラミッドの各層で検出され、検出された点の角度はハリスコーナースコアを用いて評価され、最高品質の点がフィルタリングされる。以上のことより ORB 記述子は、スケール、回転、限定的なアフィン変換に対して不変である。

(4) FLANN

FLANN とは Fast Library for Approximate Nearest Neighbors(近似的再近傍法のための高速ライブラリ) の略で、高速な近似的最近傍探索を行うためのライブラリであり、大規模データや高次元データに対する高速な最近傍探索のために最適化されたアルゴリズムを提供している [20] 。マッチングアルゴリズ ムの中で最も計算コストの高い部分は、高次元ベクトルに最も近い一致をテンプレートすることである。これを解決するために、Randomized KD Tree と k-means Tree の二つの木構造からデータセットの構造や目的の検索精度に応じて最適な方のアルゴリズムが自動的に選択される。これに対して、BruteForce と呼ばれる総当たりで最初の画像にある一つの特徴点の特徴量記述子を計算し、二枚目の画像中の全特徴点の特徴量と何らかの距離計算に基づいてマッチングを行う方法がある。この BruteForce よりも、FLANN を使用した方が大規模データにおいては、高速に動作する。

2.2.2 テンプレートマッチング

テンプレート・マッチング法とは検索したい画像（テンプレート）を検索先の画像上を移動させながら、テンプレートと検索先の画像との距離を特定の手法で算出し、最大の一致度の位置を最も類似した部分とする方法である。テンプレートは大きい方がノイズに強くなるため、ソースコード片の一部分のみが変更されて生成された類似ソースコード片に対しても検出しやすくすることができる。しかし、テンプレートの大きさに比例して、計算処理が増加するため、処理時間が遅くなってしまう。

本論文では、距離の定義として以下の六つの距離を使用した。

- SQDIFF:sum of square difference

$$R(x,y) = \sum_{x',y'} \left(T\left(x',y'\right) - I\left(x+x',y+y'\right)\right)^2 \tag{2.3}$$

- SQDIFF NORMED:normalized square difference

$$R(x,y) = \frac{\sum_{x',y'} \left(T\left(x',y'\right) - I\left(x+x',y+y'\right)\right)^2}{\sqrt{\sum_{x',y'} T\left(x',y'\right)^2 \cdot \sum_{x',y'} I\left(x+x',y+y'\right)^2}} \tag{2.4}$$

- CCORR:cross correlation

$$R(x,y) = \sum_{x',y'} \left(T\left(x',y'\right) \cdot I\left(x+x',y+y'\right)\right) \tag{2.5}$$

- CCORR NORMED:normalized cross correlation

$$R(x,y) = \frac{\sum_{x',y'} \left(T\left(x',y'\right) \cdot I\left(x+x',y+y'\right)\right)}{\sqrt{\sum_{x',y'} T\left(x',y'\right)^2 \cdot \sum_{x',y'} I\left(x+x',y+y'\right)^2}} \tag{2.6}$$

- CCOEFF:correlation coefficient

$$R(x,y) = \sum_{x',y'} \left(T'\left(x',y'\right) \cdot I'\left(x+x',y+y'\right)\right) \tag{2.7}$$

where

$$T'(x',y') = T(x',y') - \frac{\sum_{x'',y''}\left(T\left(x'',y''\right)\right)}{(w \cdot h)} \tag{2.8}$$

$$I'\left(x+x',y+y'\right) = I\left(x+x',y+y'\right) - \frac{I\left(x'',y''\right)}{w \cdot h} \tag{2.9}$$

- CCOEFF NORMED:normalized correlation coefficient

$$R(x,y) = \frac{\sum_{x',y'}\left(T\prime\left(x',y'\right) \cdot I\prime\left(x+x',y+y'\right)\right)}{\sqrt{\sum_{x',y'} T\prime\left(x',y'\right)^2 \cdot \sum_{x',y'} I\prime\left(x+x',y+y'\right)^2}} \tag{2.10}$$

SQDIFF は輝度値の差の二乗の合計を求めているため、値が小さいほど類似しているということになる。CCORR は、輝度値の相関を求めているため、値が大きいほど類似しているということになる。CCOEFF は、輝度値の平均を引 いてから相関を求めている。これにより、テンプレート画像と探索画像の明るさに左右されにくくなる。相関を求めているので、これも値が大きいほど類似しているということになる

3. 関連研究

Image-Based Clone Code Detection and Visualization

Wang と Liu はソースコードの画像から類似ソースコードを検出する方法として ICCV(Image-Based Clone Code Detection and Visualization) を提案した [11]。これは、最初に、ソースコードからコメントや空白などの非機能コードを除去し、コードの機能断片を抽出する。次に、前処理されたソースコードにハイライトを付加し、画像化する。この際、画像の切り抜き、塗り潰し、リサイズを行うことで画像に検出を行うための最適化を行っている。最後に、標準化されたソースコード画像を Jaccard 距離と知覚的ハッシュアルゴリズムで検出し、類似ソースコード情報を得て結果を返すものである。検出された類似ソースコードを視覚化して、データ内の重要な情報を強調表示し、開発者が類似ソースコード片データを分析するのを容易にすることができる。Wang と Liu は、ICCV と NiCad の 2 種類の類似ソースコード検出方法が 6 つのソフトウェアに対して検出を行った。その結果、テストした 6 つのソフトウェアの中で、NiCad と ICCV の両方がほぼ 100% という高い精度で類似ソースコードを検出できたことを示した。再現率と精度のそれぞれの平均値によると、ICCV を使用して検出された類似ソースコードの再現率は、NiCad を使用して検出されたものよりも約 5%高かった。人間の検査によって検出された類似ソースコード片情報によると、ICCV の精度は NiCad の精度に匹敵した。ICCV は実際の類似ソースコードフラグメントをいくつか見逃すが、ICCV は NiCad で検出できないフラグメントを検出できることを示した。

4. 実験

4.1 実験概要

4.1.1 実験目的

本実験の目的は、中間処理などが少ない類似ソースコード片の検出技術の調査である。そこで，本実験では類似ソースコード片検出に使用できる類似画像検出技術を調査するアプローチを取る。複数の類似画像検出技術を用いて同一ソースコード片の検出性能を比較し、性能が高いものを調査した。

4.1.2 調査対象のデータセット

本論文ではデータセットには、BigCloneBench [21] を使用した。BigCloneBench とは、 IJaDataset に含まれているデータセットの一つで、25,000 のオープンソース Java シ ステムを含むビッグデータソフトウェアリポジトリである、IJaDataset2.0 内の 800 万の検証済み類似ソースコードのコレクションである。

このベンチマークには、以上の主要な 4 つのタイプのプロジェクト内類似ソースコードとプロジェクト間類似ソースコードの両方が含まれているため、類似ソースコードの種類ごと、および類似ソースコード片類似ソースコードの構文上の類似性の全範囲にわたってツールの再現率を評価することができる。類似ソースコード検出ツールとして評価実験で頻繁に用いられているため、本論文でも採用した。

本論文では、BigCloneBench は Java で書かれたソースコードで構成されているため、コメント部分、import 文を削除した。

4.1.3 実験手順

本論文はソースコードが一枚の画像として二枚与えられた状態で、両者における類似ソースコード片を検出することを目的とする。実験方法は以下の手順で進めた。この手順を図 4.1 で示す。

手順 (1) 画像から文字部分を検出する。

手順 (2) 検出された文字部分の座標を元に、行単位で画像を抽出する。

手順 (3) それぞれの類似比較法で類似する行、または類似する部分を検出する。

(1) 手順 1

手順 1 では、画像処理技術の一つである、OCR(Optical Character Recognision) を使用して画像から文字部分を検出する。今回は python で tesseract [22] を使用した。

tesseract は、

(1) 連結成分分析で画像から成分の輪郭を取得する

(2) 成分の輪郭を文字間隔の種類によって行、単語の粒度まで段階的に分割

(3) 二つのパスによって、単語を認識する

(4) 認識した単語の画像上の左上と右下の座標を出力する

このようにして、出力された座標を元に画像に加工を行ったのが図 4.2 である。

(2) 手順 2

手順 2 では、検出された文字部分の座標を元に、行単位で画像を抽出する。(図 4.3,4.4) 手順 1 の tesseract では各単語の認識した四角が可視化されていた。この時、四角の左上と右下の座標がわかるため、左端の単語の x 座標に中心を通る単語を同じ行として右端を探索し、その間を一つの行として抽出している。

しかし、tesseract が 単語と認識するなかに文字列を定義する、「”」がある。これは先述の手法において、 以下のソースコードを読み取った際に、

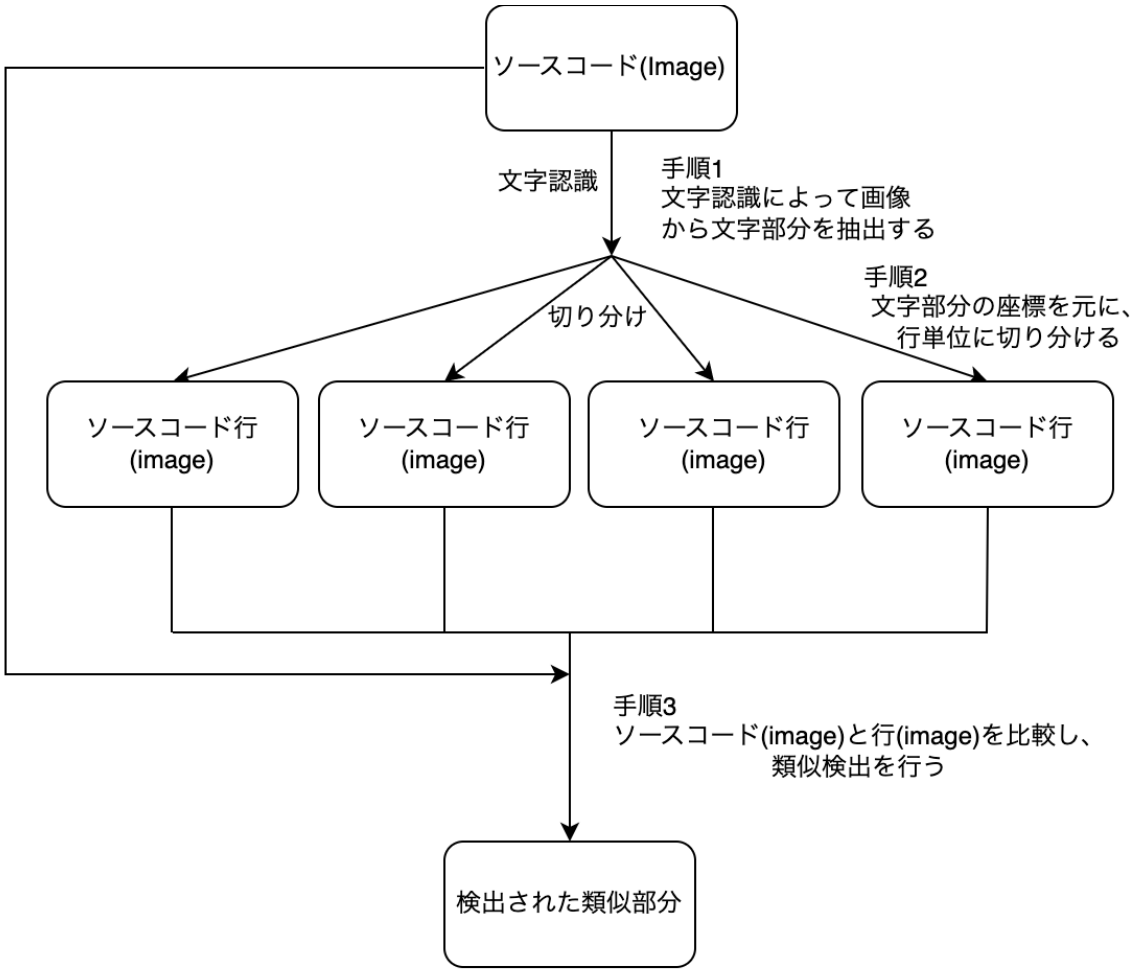
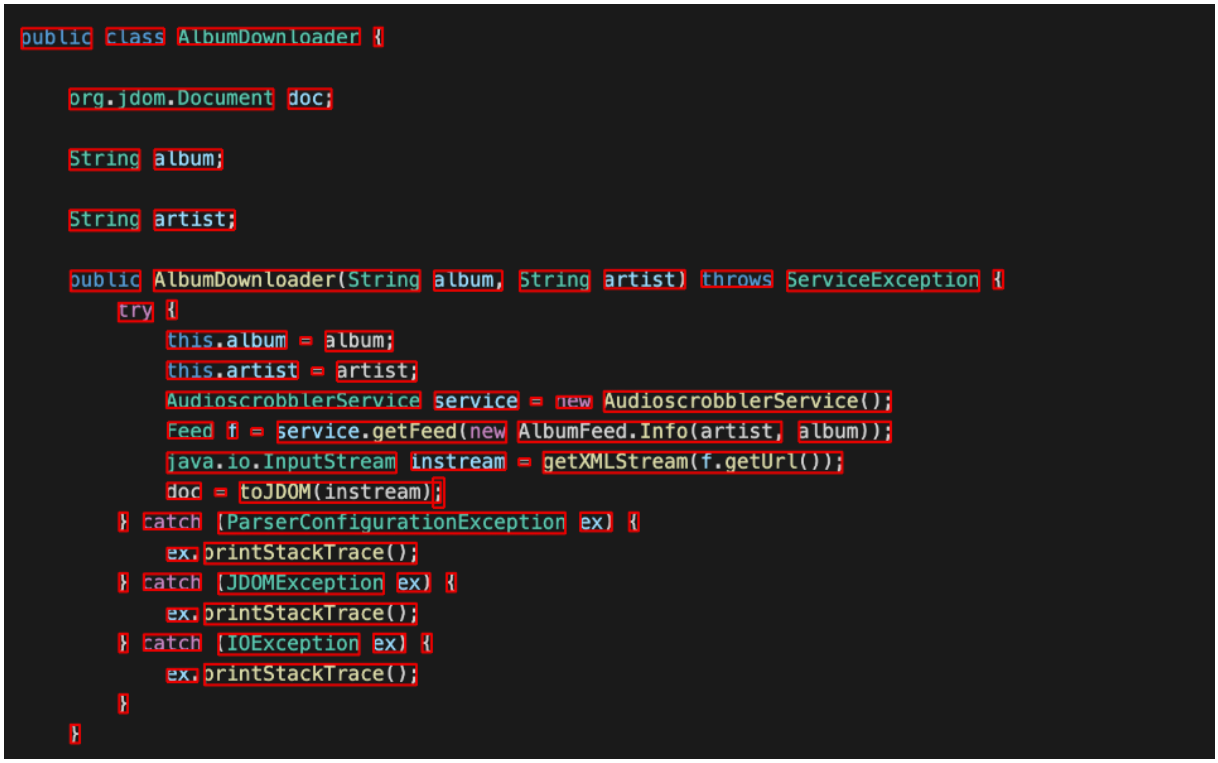


図 4.1　実験における概要図



```
public class AlbumDownloader {

    org.jdom.Document doc;

    String album;

    String artist;

    public AlbumDownloader(String album, String artist) throws ServiceException {
        try {
            this.album = album;
            this.artist = artist;
            AudioscrobblerService service = new AudioscrobblerService();
            Feed f = service.getFeed(new AlbumFeed.Info(artist, album));
            java.io.InputStream instream = getXMLStream(f.getUrl());
            doc = toJDOM(instream);
        } catch (ParserConfigurationException ex) {
            ex.printStackTrace();
        } catch (JDOMException ex) {
            ex.printStackTrace();
        } catch (IOException ex) {
            ex.printStackTrace();
        }
    }
```

図 4.2　文字認識を行っている図

```
public AlbumDownloader(String album, String artist) throws ServiceException {
```

図 4.3　行ごとに抽出した図-1

```
try {
```

図 4.4　行ごとに抽出した図-2

```
unit = "Unit data: [" + item_title + " - " + item_pubDate + " - " + item_description + " - " + item_link + "]";
```

図 4.5　”を含むソースコードを文字認識した図

```
unit = "Unit data: [" + item_title +
```

図 4.6　途中で切れてしまう図

```
unit = "Unit data: [" + item_title + " - " + item_pubDate + " - " + item_description + " - " + item_link + "]";
```

図 4.7　正常に抽出された図

```
        unit = "Unit data:[" + item_title + " - " +
           item_pubDate + " - " + item_description + " - " +
           item_link + "]";
```

図 4.5 のように、四角が上方に偏ってしまうことがある。このような場合、同じ行として認識できない場合があり、行の抽出を行うと途中で切れてしまう画像と正常に抽出される画像が生成された。(図 4.6、4.7)

本論文の対象として図 4.6 は不適切なため目視で除外した。

(3) 手順 3

手順 3 では、それぞれの比較方法で類似したソースコード片を検出する。今回は、特徴点マッチングとテンプレートマッチングを使用した。

特徴点マッチング 特徴ベースマッチングとは、2 つの画像間で一致する特徴点を抽出し、その一致特徴点の近傍領域を使って照合する手法である。

特徴点検出として以下の三つの手法を使用した。

- AKAZE
- ORB
- SIFT

次に、マッチングとして以下の二つの手法を使用した。

- BriteForce
- FLANN

テンプレートマッチング テンプレート・マッチング法とは検索したい画像 (テンプレート) を検索先の像上を移動させながら、テンプレートと検索先の画像との距離を特定の手法で算出し、最大の一致度の位置を最も類似した部分とする方法である。今回は 距離の定義として以下の六つの距離を使用した。

- SQDIFF:sum of square difference
- SQDIFF NORMED:normalized square difference
- CCORR:cross correlation
- CCORR NORMED:normalized cross correlation
- CCOEFF:correlation coefficient
- CCOEFF NORMED:normalized correlation coefficient

4.2 実験結果

4.2.1 特徴点マッチング

(1) 特徴点検出

まずは、ソースコードの行ごとの画像に対して、特徴点検出を行った結果を示す。

今回は、下のソースコード片から検出する例を示す。

```
java.io.InputStream instream = getXMLStream(f.getUrl());
```

画像は、手順 2 で抽出を行った結果、高さが 25 ～ 30pixel となった。

この時、AKZE と ORB では一部の画像で特徴抽出することができなかった。 (図 4.9) 一方で、SIFT では特徴抽出することができた。(図 4.10)

ここで、画像を拡大することで特徴点の検出が出来た。(図 4.11、4.12) 画像の拡大倍率ごとの特徴点の数を表 4.1 に示す。

(2) マッチング

次に、各ツールで検出した特徴点のマッチング結果について示す。今回は、特定の閾値を元に一定の類似度よりも高い特徴点のペアを類似したものと判定し、そのペアが最も多かった行をそのソースコードにおける最も類似した行と判定することにした。

ソースコード (図 4.8) から特徴点検出で使用した一行を抽出した際に、元のソースコード画像のどこに存在しているかについて検証を行う。マッチング手法として、BruteForce と FLANN を用いて検証した。表 4.2、4.3 に示す。

以上の結果より、SIFT で検出した特徴点が認識率が高いことがわかった。これを元に、100 ファイルから任意のソースコード片を取り出し、検出を行った結果、認識率が 73%となった。

次に、閾値を変化させることで類似していると判定する類似度を変化させ、SIFT におけるマッチングと正しい行へのマッチング数の変化を確認した。表 4.4、4.5 に示す。

```
import ch.rolandschaer.ascrblr.scrobbler.AudioscrobblerService;
import ch.rolandschaer.ascrblr.data.*;
import ch.rolandschaer.ascrblr.util.ServiceException;
import java.util.*;
import java.net.*;
import java.io.*;
import javax.xml.parsers.ParserConfigurationException;
import org.jdom.JDOMException;

public class AlbumDownloader {

    org.jdom.Document doc;

    String album;

    String artist;

    public AlbumDownloader(String album, String artist) throws ServiceException {
        try {
            this.album = album;
            this.artist = artist;
            AudioscrobblerService service = new AudioscrobblerService();
            Feed f = service.getFeed(new AlbumFeed.Info(artist, album));
            java.io.InputStream instream = getXMLStream(f.getUrl());
            doc = toJDOM(instream);
        } catch (ParserConfigurationException ex) {
            ex.printStackTrace();
        } catch (JDOMException ex) {
            ex.printStackTrace();
        } catch (IOException ex) {
            ex.printStackTrace();
        }
    }

    public boolean saveArt(String path) {
        try {
            File file = AlbumHelper.songbirdToJavaFile(path, artist + " - " + album, "jpg");
            org.jdom.Element e = doc.getRootElement();
            org.jdom.Namespace ns = e.getNamespace();
            e = e.getChild("coverart", ns).getChild("large", ns);
            System.out.println("Attempting download");
            return download(new URL(e.getText()), file);
        } catch (Exception e) {
            e.printStackTrace();
            return false;
        }
    }

    public boolean download(URL url, File file) {
        OutputStream out = null;
        URLConnection conn = null;
        InputStream in = null;
        try {
            out = new BufferedOutputStream(new FileOutputStream(file));
            conn = url.openConnection();
            in = conn.getInputStream();
            byte[] buffer = new byte[4096];
            int numRead;
            long numWritten = 0;
            while ((numRead = in.read(buffer)) != -1) {
                out.write(buffer, 0, numRead);
                numWritten += numRead;
            }
        } catch (Exception e) {
            System.out.println(e);
            return false;
        } finally {
            try {
                if (in != null) {
                    in.close();
                }
                if (out != null) {
                    out.close();
                }
            } catch (IOException ioe) {
                return false;
            }
        }
        return true;
    }

    public ArrayList<String> getTracksForAlbum() {
        AudioscrobblerService service = new AudioscrobblerService();
        try {
            org.jdom.Element e = doc.getRootElement();
            org.jdom.Namespace ns = e.getNamespace();
            List<org.jdom.Element> elementList = e.getChild("tracks", ns).getChildren("track");
            ArrayList<String> trackList = new ArrayList<String>();
            for (org.jdom.Element element : elementList) {
                trackList.add(element.getAttributeValue("title"));
            }
            return trackList;
        } catch (Exception e) {
            e.printStackTrace();
            return new ArrayList<String>();
        }
    }

    public static org.jdom.Document toJDOM(java.io.InputStream in) throws org.jdom.JDOMException,
java.io.IOException {
        org.jdom.input.SAXBuilder builder = new org.jdom.input.SAXBuilder();
        return builder.build(in);
    }

    public static java.io.InputStream getXMLStream(String url) throws java.io.IOException {
        java.net.URL u = new java.net.URL(url);
        java.net.URLConnection conn = u.openConnection();
        return conn.getInputStream();
    }
}
```

図 4.8　探索元のソースコード画像

図 4.9　特徴抽出することができなかった図

図 4.10　SIFT で特徴抽出することができた図

図 4.11　拡大した写真を AKAZE で検出した図

図 4.12　拡大した写真を ORB で検出した図

表 4.1　各ツールが検出した特徴点の数

拡大倍率	AKAZE	ORB	SIFT
1	0	0	339
5	506	318	643
10	1428	461	621
30	1834	95	621

表 4.2　BruteForce でマッチングを行った結果

特徴点	マッチング数	正しい行へのマッチング数	判定結果
SIFT	79	76	認識
AKAZE(1 倍)	0	0	非認識
AKAZE(5 倍)	28	0	誤認識
AKAZE(10 倍)	72	1	誤認識
AKAZE(30 倍)	70	0	誤認識
ORB(1 倍)	0	0	非認識
ORB(5 倍)	0	0	非認識
ORB(10 倍)	0	0	非認識
ORB(30 倍)	0	0	非認識

表 4.3　FLANN でマッチングを行った結果

特徴点	マッチング数	正しい行へのマッチング数	判定結果
SIFT	79	76	認識
AKAZE(1 倍)	0	0	非認識
AKAZE(5 倍)	9	0	誤認識
AKAZE(10 倍)	18	1	誤認識
AKAZE(30 倍)	29	0	誤認識
ORB(1 倍)	0	0	非認識
ORB(5 倍)	18	2	誤認識
ORB(10 倍)	24	3	誤認識
ORB(30 倍)	4	0	誤認識

表 4.4　SIFT で特徴点を検出し BruteForce でマッチングを行った結果

閾値	マッチング数	正しい行へのマッチング数	判定結果
0.15	40	40	認識
0.3	77	74	認識
0.5	105	102	認識
0.7	131	124	認識
0.9	180	160	認識

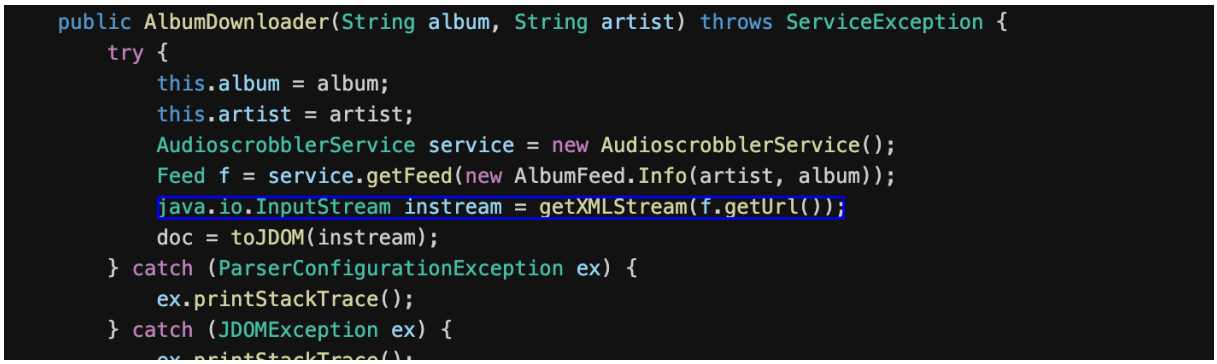
表 4.5　SIFT で特徴点を検出し FLANN でマッチングを行った結果

閾値	マッチング数	正しい行へのマッチング数	判定結果
0.15	40	40	認識
0.3	77	74	認識
0.5	105	102	認識
0.7	131	124	認識
0.9	180	160	認識

4.2.2 テンプレートマッチング

特徴点マッチングと同様に、ソースコードの行ごとの画像に対して、テンプレートマッチングを行った結果を示す。テンプレート画像、検索対象の画像は、特徴点マッチングと同じものを使用した。6 つの距離を使用して検出を行った結果を以下に示す。

CCOEFF、CCOEFF_ NORMED、CCORR、SQDIFF、SQDIFF_ NORMED、CCORR_ NORMED の全てにおいて、図 4.13 のように認識できた。



```
public AlbumDownloader(String album, String artist) throws ServiceException {
    try {
        this.album = album;
        this.artist = artist;
        AudioscrobblerService service = new AudioscrobblerService();
        Feed f = service.getFeed(new AlbumFeed.Info(artist, album));
        java.io.InputStream instream = getXMLStream(f.getUrl());
        doc = toJDOM(instream);
    } catch (ParserConfigurationException ex) {
        ex.printStackTrace();
    } catch (JDOMException ex) {
```

図 4.13　テンプレートマッチングで検出した図

5. 考察

5.1 特徴点マッチング

5.1.1 特徴点抽出

図 4.1 より、画像を拡大していない段階では、SIFT が最も検出数が多かったが、画像を拡大していくにつれて、AKAZE が比例して検出数が増えていった。

特徴点とは、画像中から角や線の交点、エッジなどの固有な点を対象としているため、画像を一定の水準よりも拡大にするにつれて、文字が潰れてしまうことより特徴点の検出数が減少してしまうと考えられたが、AKAZE は非線形拡散フィルタリングに基づいていることが要因と考えられた。

5.1.2 マッチング

図 4.2、4.3、4.4、4.5 より SIFT においては、テンプレート画像の倍率を変えることなく認識率が高いことがわかった。100 ファイルから任意のソースコード片を取り出し検出を行った際に 73%になってしまったが、要因として、

- 文法による予約語のみの文や宣言文、代入文などが誤検出しやすくなった
- タイプ 1 の類似ソースコードを検出した

などが考えられた。

結果としては、SIFT で特徴点検出を行った際に、誤検出とされるものがタイプ 1、タイプ 2 の類似ソースコードであることがわかった。

また、閾値の変化によって、検出されるマッチングの変化が実験でわかった。ここで、

```
m.distance < n * n.distance
```

特定の特徴量に対して、m は最も近い特徴量であり distance は特定の特徴量との距離、n は二番目に近い特徴量であり distance は特定の特徴量との距離、n は閾値である。この条件式が真であるときに、特定の特徴量と m は類似した特徴量として判定している。このことより、閾値が低いものほどマッチングの精度が高い実験結果と一致する。

最後に、同一ソースコードではなく類似ソースコードを検索する実験を行った。(図 5.1、5.2、5.3、5.4)

この結果、図 5.1 には

```
public class RSS_parser_work {
```

図 5.2 には

```
import java.io.InputStream
```

が、それぞれ検索結果となった。

類似しているソースコード片ではあるが、類似ソースコード片におけるタイプとはどれにも属さない行だと判断できた。しかし、類似していると判定できる結果を複数用意することができれば、類似ソースコード片の検出においては、精度や適合率は大きく下がってしまうが、再現率は高くなるのではないかと考えられる。

5.2 テンプレートマッチング

第 5 章 2 節の実験において、すべての距離において同一ソースコード片を認識することができた。しかし、テンプレートマッチングはテンプレートの画像と検索先の画像の両方でサイズが同じでなければ検索精度が低くなってしまうため、特徴点マッチングのようにスケール、回転には頑丈ではないという欠点がある。同一ソースコード片ではなく類似ソースコード片を検索する場合、変数名の違いなどだけでも文字数が変化してしまい検索できない可能性があると考え、追加実験を行った。

この結果、CCORR を除く、すべての距離による検出において検索ができた。(図 5.1、5.2、5.5、5.6、5.7)

図 5.1、5.5 においては予約語による同一文字列の影響が大きかったため、検出に成功していた。しかし、図 5.2、5.6、図 5.7 においては、InputStream や get、Url、”)) の文字列による検索結果が現れたと思われるが、テンプレート画像は一行全体であるのに対して、検索結果は行の一部分であることが多いため、類似ソースコード片におけるタイプとはどれにも属さない行だと判断できた。しかし、類似していると判定できる結果を複数用意することができれば、類似ソースコード片の検出においては、精度や適合率は大きく下がってしまうが、再現率は高くなるのではないかと考えられる。

図 5.1　テンプレートの画像 1

図 5.2　テンプレートの画像 2

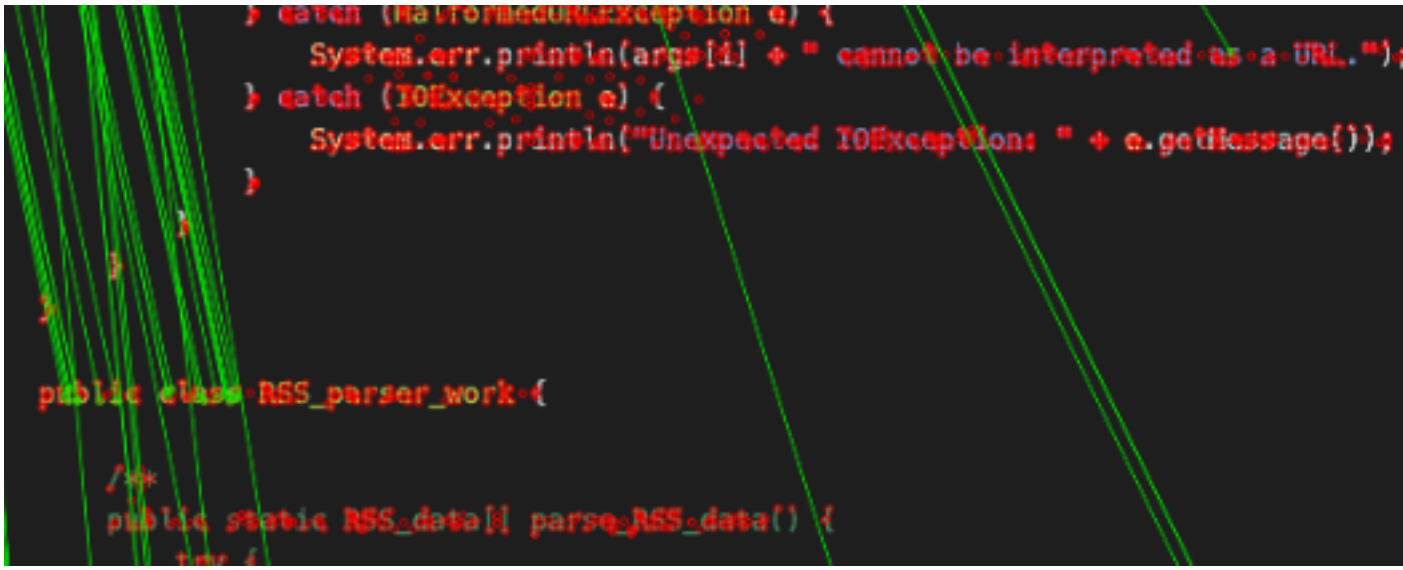


図 5.3　テンプレート画像 1 の SIFT による検索結果の図

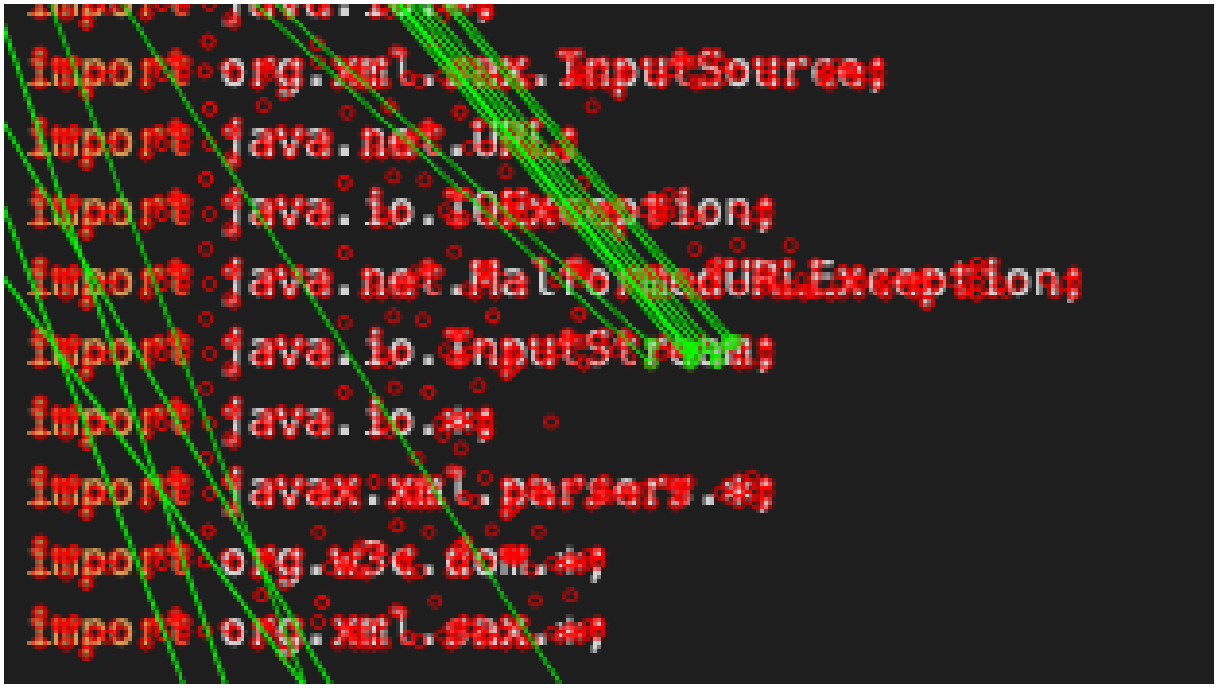


図 5.4　テンプレート画像 2 の SIFT による検索結果の図

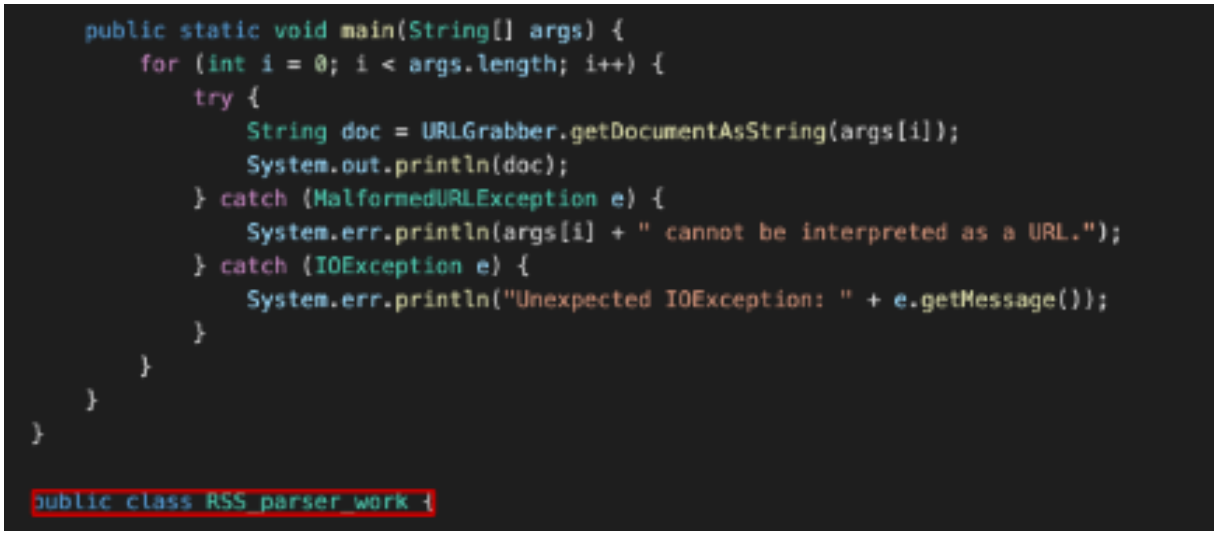
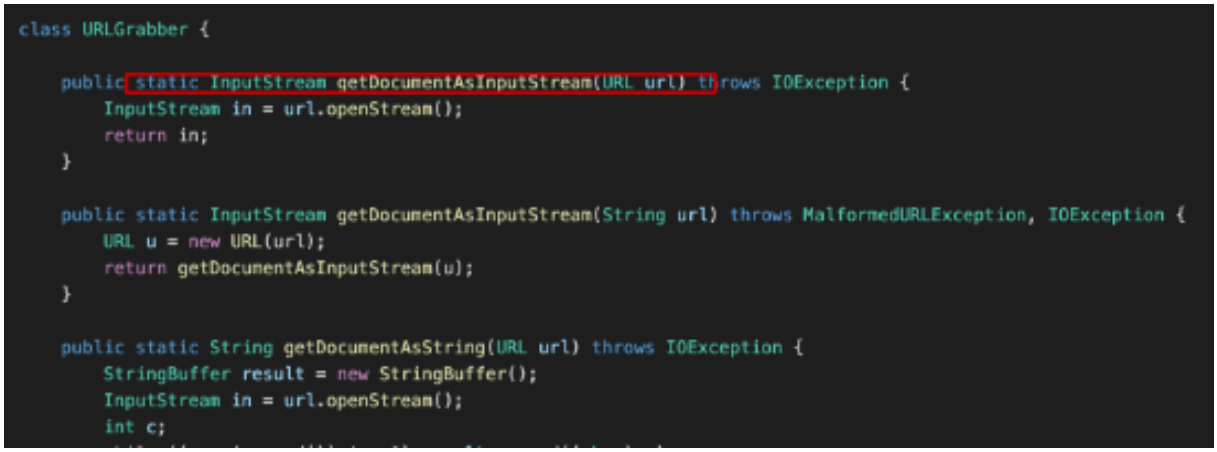


図 5.5　テンプレート画像 1 の検索結果の図

```
/**
public static RSS_data[] parse_RSS_data() {
    try {
        System.out.println("Start RSS Parser");
        DocumentBuilderFactory document_builder_factory = DocumentBuilderFactory.newInstance();
        DocumentBuilder document_builder = document_builder_factory.newDocumentBuilder();
        Document document =
                document_builder.parse(
                        new InputSource(
                                new StringReader(
                                        URLGrabber.getDocumentAsString("http://www.comune.torino.it/cgi-bin/torss/rssfeed.cgi?id=93"))));
        document.getDocumentElement().normalize();
        System.out.println("Root element : " + document.getDocumentElement().getNodeName());
        System.out.println("title : " + document.getDocumentElement().getAttribute("title"));
        System.out.println("pubDate : " + document.getDocumentElement().getAttribute("pubDate"));
        NodeList node_list = document.getElementsByTagName("item");
        System.out.println("Information of RSS item");
        int node_number = node_list.getLength();
        System.out.println("Found " + node_number + " elements.");
        RSS_data[] result       = new RSS_data[node_number];
        RSS_data rss_unit       = null;
        String title            = "0";
        String link             = "0";
        String category         = "0";
        String lat              = "0";
        String lng              = "0";
        String description      = "0";
        Date pubDate            = new Date();
        for (int i = 0; i < node_number; i++) {
                Node node = node_list.item(i);
```

図 5.6　テンプレート画像 2 の SQDIFF による検索結果の図



```
class URLGrabber {

    public static InputStream getDocumentAsInputStream(URL url) throws IOException {
        InputStream in = url.openStream();
        return in;
    }

    public static InputStream getDocumentAsInputStream(String url) throws MalformedURLException, IOException {
        URL u = new URL(url);
        return getDocumentAsInputStream(u);
    }

    public static String getDocumentAsString(URL url) throws IOException {
        StringBuffer result = new StringBuffer();
        InputStream in = url.openStream();
        int c;
```

図 5.7　テンプレート画像 2 の SQDIFF、CCORR 以外による検索結果の図

6. 今後の課題

図 4.1、4.2、4.3 よりテンプレート画像を拡大して SIFT で特徴点検出を行いマッチングを行った結果、検出した特徴点の数は減少し、BruteForce と FLANN の両方でマッチング数と正しい行へのマッチング数は減少した。これは、SIFT は画像の回転、スケール、および限定的なアフィン変換に対して頑健である [17] ということに反する結果となった。これの原因を探ることによって、認識率の精度向上につながると考える。

また、図 4.4、4.5 より閾値が小さいほどでマッチングの精度が高くなることがわかったが、ここで閾値が高い時に検出したマッチングから閾値が低い時に検出したマッチングを除去することで同一ではなく類似したソースコード片の検出につながると考える。

そして、ソースコードにはシンタックスハイライトによる色付けを行うことができる。これは構文情報に基づいて色付けされるもので、色彩的に類似しているソースコード片は類似ソースコード片である可能性があると考えられる。このことから、認識率の高かった特徴点マッチングにおける SIFT や、テンプレートマッチングにおける CCOEFF、CCOEFF_NORMED、CCORR_NORMED、SQDIFF、SQDIFF_NORMED の検出結果に色情報による検出結果を組み合わせることで、さらに類似ソースコード片の検出率が上がると考えられる。

7. 結言

本論文では画像処理技術と類似画像の検索に着目し、類似ソースコード片の検索の性能について調査した。調査対象は Java の ソースコードであり、特徴点マッチングとテンプレートマッチングの性能である。

調査の結果、ソースコードから任意のソースコード片を検出することにおいては、特徴点検出では、SIFT が優れ、マッチングにおいてはどちらもマッチング率には差が生じたが、正しい行へのマッチング率においては有意な差は生じなかった。テンプレートマッチングでは、提示した六つの距離においてはすべてで検出することができたが、類似したソースコード片を検出することにおいては、距離によって差が出た。

類似ソースコード片を検出するにあたっては、閾値の設定に左右されることが明白ではあるが、適度な値を設定することで、同一のみではなく類似ソースコードのタイプ 1、及びタイプ 2 が検出できるのではないかと思われる。

謝辞

本研究を行うにあたり、研究課題の設定や研究に対する姿勢、本報告書の作成に至るまで、全ての面で丁寧なご指導を頂きました、本学情報工学・人間科学系水野修教授ならびに崔 恩瀞助教に厚く御礼申し上げます。本報告書執筆にあたり貴重な助言を多数頂きました、本学情報工学専攻中森陸斗先輩、渡邉紘矢先輩をはじめとする、ソフトウェア工学研究室の皆さん、学生生活を通じて著者の支えとなった家族や友人に深く感謝致します。

参考文献

[1] W.-K. Chen, B. Li, and R. Gupta, “Code Compaction of Matching Single-Entry Multiple-Exit Regions,” in Static Analysis, ed. R. Cousot, pp.401–417, Lecture Notes in Computer Science, Springer, Berlin, Heidelberg, 2003.

[2] M. Kim, V. Sazawal, D. Notkin, and G. Murphy, “An empirical study of code clone genealogies,” SIGSOFT Softw. Eng. Notes, vol.30, pp.187–196, Sept. 2005.

[3] J.-F. Patenaude, E. Merlo, M. Dagenais, and B. Lague, “Extending software quality assessment techniques to Java systems,” Proceedings Seventh International Workshop on Program Comprehension, pp.49–56, May 1999.

[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, vol.28, no.7, pp.654–670, July 2002.

[5] H. Sajnani, V. Saini, J. Svajlenko, C.K. Roy, and C.V. Lopes, “SourcererCC: Scaling Code Clone Detection to Big-Code,” 2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE), pp.1157–1168, May 2016.

[6] M. Rieger, S. Ducasse, and M. Lanza, “Insights into system-wide code duplication,” 11th Working Conference on Reverse Engineering, pp.100–109, Jan. 2004.

[7] C. Roy and J. Cordy, “A Survey on Software Clone Detection Research,” ●●, pp.●●–●●, 2007.

[8] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. Merlo, “Comparison and Evaluation of Clone Detection Tools,” IEEE Transactions on Software Engineering, vol.33, no.9, pp.577–591, Sept. 2007.

[9] C.K. Roy and J.R. Cordy, “NICAD: Accurate Detection of Near-Miss Intentional Clones Using Flexible Pretty-Printing and Code Normalization,” 2008 16th IEEE International Conference on Program Comprehension, pp.172–181, June 2008.

[10] L. Jiang, G. Misherghi, Z. Su, and S. Glondu, “DECKARD: Scalable and Accurate Tree-Based Detection of Code Clones,” 29th International Conference on Software

Engineering (ICSE'07), pp.96–105, May 2007.

[11] Y. Wang and D. Liu, "Image-Based Clone Code Detection and Visualization," 2019 International Conference on Artificial Intelligence and Advanced Manufacturing (AIAM), pp.168–175, Oct. 2019.

[12] M. Hammad, Ö. Babur, H.A. Basit, and M. Van Den Brand, "Clone-Seeker: Effective Code Clone Search Using Annotations," IEEE Access, vol.10, pp.11696–11713, 2022.

[13] I.D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier, "Clone detection using abstract syntax trees," Proceedings. International Conference on Software Maintenance (Cat. No. 98CB36272), pp.368–377, Jan. 1998.

[14] 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，vol.J91-D，no.6，pp.1465–1481，2008.

[15] D. Rattan, R. Bhatia, and M. Singh, "Software clone detection: A systematic review," Information and Software Technology, vol.55, no.7, pp.1165–1199, July 2013.

[16] H. Min and Z. Li Ping, "Survey on Software Clone Detection Research," Proceedings of the 2019 3rd International Conference on Management Engineering, Software Engineering and Service Sciences, pp.9–16, ICMSS 2019, Association for Computing Machinery, New York, NY, USA, 1 月 12, 2019.

[17] D.G. Lowe, "Distinctive Image Features from Scale-Invariant Keypoints," International Journal of Computer Vision, vol.60, no.2, pp.91–110, Nov. 2004.

[18] P. Alcantarilla, J. Nuevo, and A. Bartoli, "Fast Explicit Diffusion for Accelerated Features in Nonlinear Scale Spaces," Procedings of the British Machine Vision Conference 2013, pp.13.1–13.11, British Machine Vision Association, Bristol, 2013.

[19] E. Rublee, V. Rabaud, K. Konolige, and G. Bradski, "ORB: An efficient alternative to SIFT or SURF," 2011 International Conference on Computer Vision, pp.2564–2571, Jan. 2011.

[20] M. Marius and L. D, "FAST APPROXIMATE NEAREST NEIGHBORS WITH AUTOMATIC ALGORITHM CONFIGURATION:," Proceedings of the Fourth International Conference on Computer Vision Theory and Applications, pp.331–340,

SciTePress - Science and and Technology Publications, Lisboa, Portugal, 2009.

[21] J. Svajlenko, J.F. Islam, I. Keivanloo, C.K. Roy, and M.M. Mia, "Towards a Big Data Curated Benchmark of Inter-project Code Clones," 2014 IEEE International Conference on Software Maintenance and Evolution, pp.476–480, Sept. 2014.

[22] R. Smith, "An Overview of the Tesseract OCR Engine," Ninth International Conference on Document Analysis and Recognition (ICDAR 2007), vol.2, pp.629–633, Sept. 2007.