目 次

第1章　はじめに

プログラミング教育の現場において学習者の習熟度を計る一般的な方法は、学習者の書いたプログラムのソースコードから挙動を読み解き, その出来栄えから計る方法である. しかし, この評価方法では評価対象となるプログラムのソースコードを１つ１つ読み解き, 模範となるプログラムのソースコードとの違いを見つけることになるため, 評価を行う際, 相当の時間と労力を費やすことになる. このことから, 学習者の作成したプログラムと模範となるプログラムの近似性を容易に把握できるシステムが求められている.[1]

一般的に, プログラム間の違いを把握するために, diff と呼ばれる処理が使用される. これは, 行単位でテキストファイル間の文字列の違いを取得するものであり, ソースコードの変更行を確認するために使われている.

しかし, プログラムの近似性を推し量る場合, ソースコード間の単純な文字列比較のみで行うことはできない. これは, プログラムが構造を持つ要素の集合であり, また同時に意味も表現しているため, 文字ベースの編集距離などで近似性を定めるのは困難なためである. このため, プログラムの近似性を推し量る場合, ソースコードの文字列比較ではなく, プログラムの構造などを考慮に入れて近似性を推し量る方法が必要になる. しかし, プログラムの構造を考慮した上で近似性を推し量る場合, プログラムを構成する複数個の情報に対し比較を行わなければならず, 人間がこの細かい比較を数値や文字で一度に把握するのは困難である.

これに対し, 複数個の情報の比較を容易にするものとして可視化が有効であると考えた. これは, 可視化が複雑な情報や大量の情報, 単純な比較では見えない情報を人間が理解しやすいものとしてグラフ化することであり, 複雑な構造をしているプログラムの持つデータを可視化することで人間がプログラムの構造を理解し, その比較を容易にさせるためである.

これらのことから, プログラム内の注目すべき複数個の情報に対して可視化を行い, そのグラフからプログラム間の比較をすることでプログラム間の近似度を推し量ることができると考えた.

本研究では, Java 言語で書かれたプログラムを対象に, プログラムの実際の動作情報を持つメソッドに着目し, メソッドの持つ要素からプログラム間の近似度を可視化させる手法を提案する.

第2章　可視化の対象

今回可視化の対象として選んだのは, Java 言語で書かれたコンパイル済みバイナリデータである. なぜかというと, Java のバイナリデータの仕様が Java 仮想マシンの仕様 [2] により定型化されており, 主要なデータの取得が容易なためである.

この章では, Java のバイナリデータの構造体を説明し, その後, 今回の近似度可視化で扱うデータについて述べる.

2.1　バイナリデータの構造体

Java のバイナリデータは, 単一の ClassFile 構造体から成り立っており, この ClassFile 構造体はいくつかの値, 及び構造体から成り立っている. ここでは, この ClassFile 構造体, 及びメソッドの実装に係る部分を説明する.

2.1.1　ClassFile 構造体

ClassFile 構造体は以下の値, 及び構造体をメンバーに持つ.

1. magic
   Java のバイナリデータであることを識別するためのマジックナンバー.
2. miner_version
   バイナリデータのフォーマットのマイナーバージョン番号.
3. major_version
   バイナリデータのフォーマットのメジャーバージョン番号.
4. constant_pool_count
   constant_pool[] のサイズ.
5. constant_pool
   他の構造体から参照されるさまざまな文字列定数, クラス名やインターフェース名, フィールド名, その他の定義を行うための構造体テーブル.
6. access_flags
   アクセス許可, 及びクラスやインターフェースにおける属性を記述するためのマスク・フラグ.

7. this_class
   constant_pool[] に対するインデックスであり, クラス, インターフェースを表現した構造体が参照される.
8. super_class
   親クラスがない場合はゼロである. それ以外の場合,constant_pool[] に対するインデックスであり, クラス, インターフェースを表現した構造体が参照される.
9. interfaces_count
   interfaces 配列のサイズ.
10. interfaces[]
    constant_pool[] に対するインデックスであり, インターフェースを表現した構造体が参照される.
11. fields_count
    fields[] のサイズ.
12. fields[]
    クラス, インターフェースで宣言されているフィールド情報が格納された構造体テーブル. この時, 継承する親のフィールドは含まれない.
13. methods_count
    methods[] のサイズ.
14. methods[]
    クラス, インターフェースで宣言されているメソッド情報が格納された構造体テーブル. この時, 継承する親のメソッドは含まれない.
15. attributes_count
    attributes[] のサイズ.
16. attributes[]
    ClassFile 構造体が持つアトリビュートが格納されている構造体テーブル.

2.1.2 CodeAttribute 構造体

ClassFile 構造体の中で実際にメソッドの定義が行われているのは, methods[] に格納されている構造体 method_info が持つ構造体 Code_attribute である. Code_attribute には, Java 仮想マシン命令, 及びその補助的な情報が保持されており, 以下の値, 構造体をメンバーに持つ.

1. attribute_name_index
   constant_pool へのインデックスであり, 文字列 *"Code"* が参照される.
2. attribute_length
   該当 Code_attribute のサイズ情報.
3. max_stack
   このメソッド実行時におけるオペランド・スタックの最大深さ.

4. max_locals
   メソッド起動時に割り当てられるローカル変数の数.
5. code_length
   code[] のサイズ.
6. code[]
   メソッドを実装している Java 仮想マシン・コードの実際のバイト列.
7. exception_table_length
   exception_table[] のサイズ.
8. exception_table
   code[] 中の例外処理に対する情報を持つ構造体を格納する構造体テーブル.
9. attributes_count
   attributes[] のサイズ.
10. attributes[]
    Code_attribute 構造体が持つアトリビュートが格納されている構造体テーブル.

2.1.3 マシン・コード

Code_attribute の持つ code[] は, 実行する操作を指定するオペコード, 及びその後に続くゼロ個以上のオペランドから成る可変長バイト列である. この時, オペコード毎に後に続くオペランドの数が異なるため, code[] 中の任意の位置におけるバイトがオペコードか, それに続くオペランドかを判定するためには, オペコードとそれに続くオペランドの数の組み合わせをすべて把握し, バイト列の先頭から判定を行う必要がある. 本研究では, Java 仮想マシン仕様について書かれた Web サイト [3] からこの組み合わせを取得し判定を行う.

2.2 可視化で扱うデータ

可視化するのは, メソッド毎における以下の要素である.

1. メソッドを実装している Java 仮想マシン・コードの実際のバイト列.
2. メソッドの呼び出し関係.
3. メソッド実行時におけるオペランド・スタックの最大深さ.
4. メソッド起動時に割り当てられるローカル変数の数.

これらは, プログラムの実際の挙動に大きく寄与する要素であり, アルゴリズムの違いも反映している. 以下に各要素の詳細を述べる.

2.2.1 メソッドを実装しているJava仮想マシン・コードの実際のバイト列

Code_attributeの持つcode[]がこれに当たる．これは実行する操作を指定するオペコード, 及びその後に続くゼロ個以上のオペランドから成る可変長バイト列である．

2.2.2 メソッドの呼び出し関係

バイト列中における, 関数の呼び出し命令である *"invokeinterface", "invokespecial", "invokestatic", "invokevirtual"* に対して, そのオペランドを *constant_pool[]* のインデックスとし, *constant_pool[]* から呼び出すメソッド名, 及びそのメソッドを持つクラス名を取得する．

2.2.3 メソッド実行時におけるオペランド・スタックの最大深さ

Code_attribute の持つ *max_stack* これに当たる．

2.2.4 メソッド起動時に割り当てられるローカル変数の数

Code_attribute の持つ *max_locals* がこれに当たる．この時, メソッド起動時に渡されるパラメータ用の引数も含まれる．

第3章　可視化手法

ここでは, プログラムの持つデータに対し, どのような手法を用いて近似度を可視化 *[5]* しているかについて述べる.

3.1　グラフノードの位置関係によるメソッドの近似度可視化手法

メソッド間での命令列の違い, 及びループ構造の違いをメソッド間の近似度とし, この近似度の可視化を行う. 以下その手法について述べる.

各メソッドに対し, メソッドの持つバイト列からオペコードのみを取り出した命令バイト列を生成する. この命令バイト列に対し, 各メソッド間でレーベンシュタイン距離 *[6]* を計算しメソッド間の命令列の違いとする.

また, バイト列から条件分岐命令である *"if_comp<cond>"*, *"if_icomp<cond>"*, *"if<cond>"*, *"ifnonnull"*, *"ifnull"* に対して, その命令バイトからの移動量を表すオペランドの値がマイナスになる場合をループとし, 命令の位置, 及び移動先の位置からループの包含関係を求める. この包含関係において最も上位にあるループに対し, ループの重なり数 N, ループの移動量 V とし以下の計算によりループ構造の持つ重み M を定める.

$$M = (2N - 1) \times V$$

各メソッド間でこのループ構造の重みの総和同士の差を計算しループ構造の違いとする.

上記の *2* つの値の和をメソッド間の距離とし, メソッドを行・列とする対称行列を作成する. この対称行列に対し主座標分析 *[4]* を用いて *2* 次元座標上に射影する. そして, 得られた座標値をもちいて各メソッドを画面上にグラフノードとして描画する.

実際のこの手法の使用例が図 *3.1* であり, グラフが表現している情報は図 *3.2* である.

図 3.1 グラフノードプロット

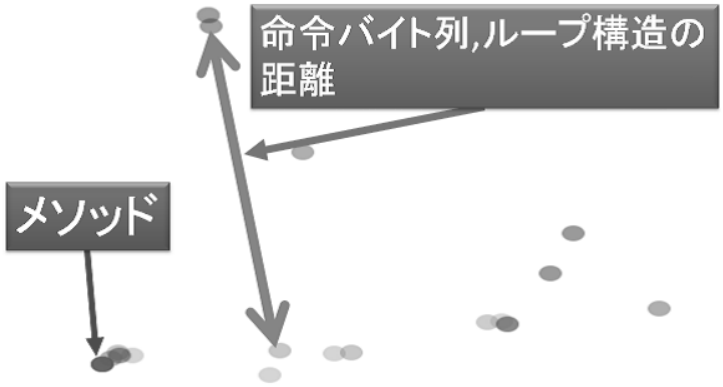

図 3.2 ノードが表現している情報

3.2　線によるメソッド間の処理の流れ可視化手法

メソッド間の呼び出し関係の可視化を行う．以下その手法について述べる．

各メソッド間において，メソッド呼び出し関係を呼び出し元から呼び出し先へのノード間矢印付き線として表現する．

実際のこの手法の使用例が図 *3.3* であり，グラフが表現している情報は図 *3.4* である．

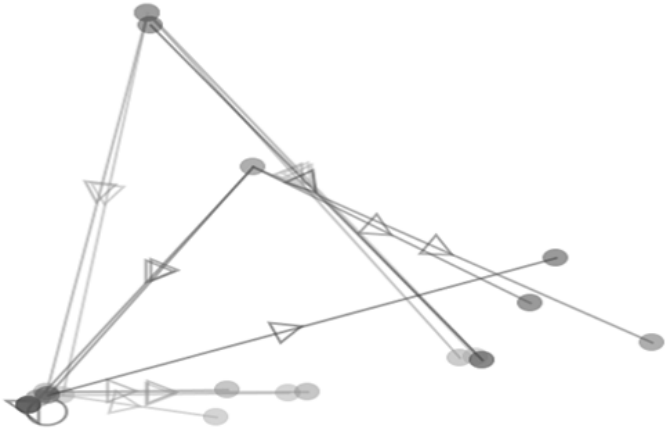

図 3.3 矢印付き線の描画

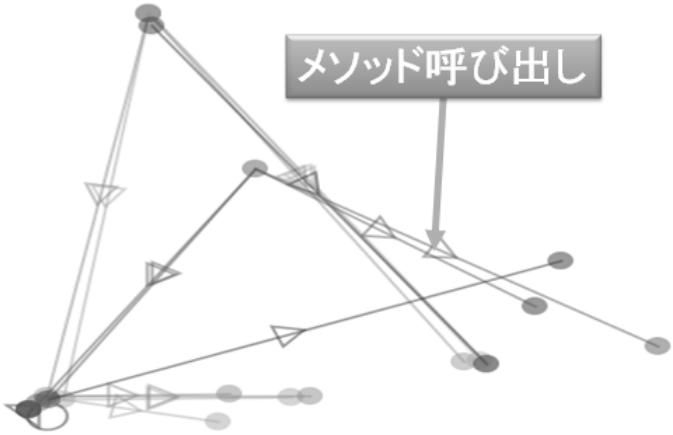

図 3.4 矢印付き線が表現している情報

3.3　点列によるメソッド内の処理の流れ可視化手法

メソッドの処理の流れの可視化を行う. 以下その手法について述べる.

バイト列に対し, 各命令を 1 プロットとし, 順に上から下へ並べたサマリを作成する. 条件分岐はプロットの左右の位置関係, ループ構造はジャンプの元と先を結ぶ線として表現する.

実際のこの手法の使用例が図 *3.5* であり, グラフが表現している情報は図 *3.6* である.

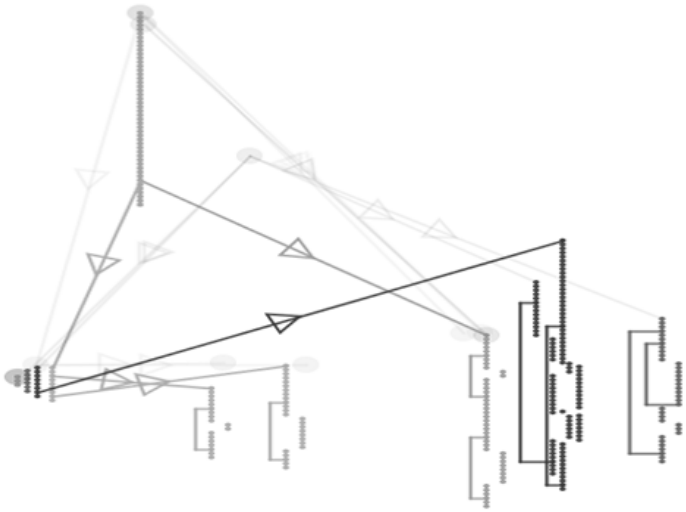

図 3.5 点列の描画

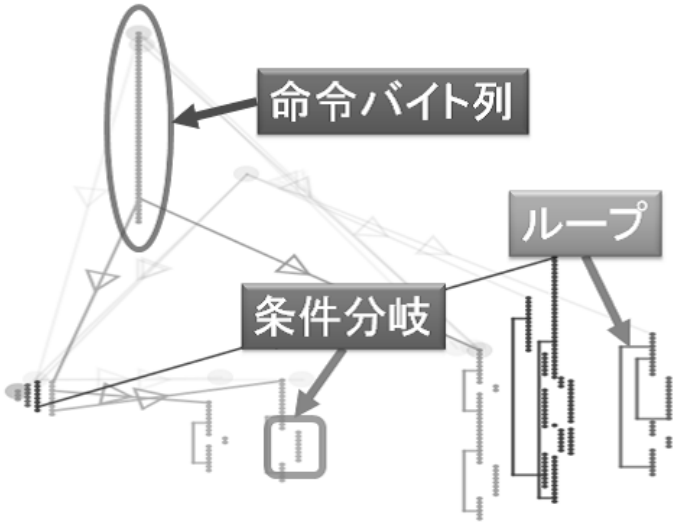

図 3.6 点列が表現している情報

3.4 円の大きさによるメソッドの使用するメモリ領域の大きさ可視化手法

メソッドの使用するメモリ領域サイズの可視化を行う. 以下その手法について述べる.

メソッドの実行中におけるオペランド・スタックの最大の深さとメソッド起動時に割り当てられるローカル変数の数の和を計算する, メソッドノードを中心とする円を作成し, 円の大きさと上記の値を比例させる.

実際のこの手法の使用例が図 *3.7* であり, グラフが表現している情報は図 *3.8* である.

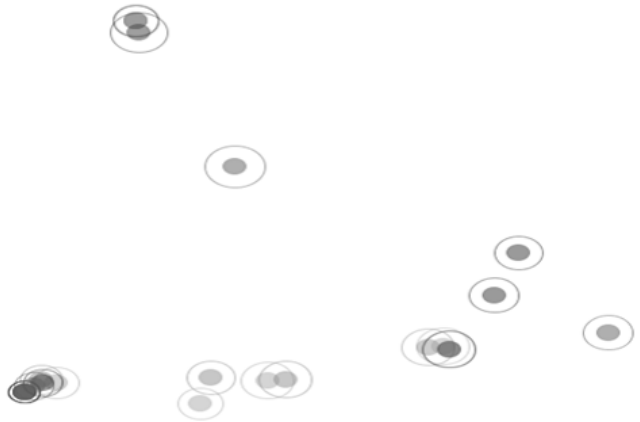

図 3.7 円の描画

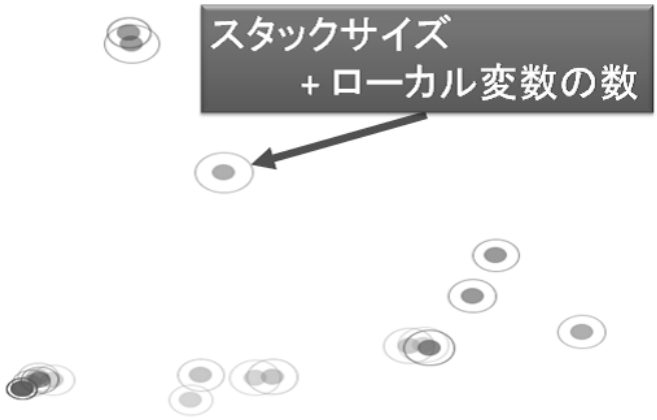

図 3.8 円が表現している情報

3.5 実行例

実際にこれらの可視化手法を用いて配列操作を行うプログラムコード図 *3.9*, 及びこのプログラムに対して宣言の前後を入れ替えたプログラム, ループ文を変更したプログラムを描画したのが図 *3.10*, 図 *3.11* である.

また, プログラムコード図 *3.9* とは異なる処理をするプログラムとして, ソートプログラム図 *3.13*, 図 *3.12* と描画したのが図 *3.14*, 図 *3.15* である.

図 *3.16* は, 可視化を行うプログラムを対象とした出力である.

```
import java.util.Arrays;
public class Filter1 {
    public static void main(String[] args){
        int[] array = new int[] { 5, 4, 6, 7, 1, 9, 8, 3, 2 };
        array = filter_1(array, 4);
        System.out.println(Arrays.toString(array));
    }
    public static int[] filter_1(int[] array, int num){
        int NUM = 0;
        for(int i=0; i<array.length; i++){
            if(array[i] <= num){
                NUM++;
            }
        }
        int[] array1 = new int[NUM];
        for(int i=array.length-1; i>-1; i--){
            if(array[i] <= num){
                array1[--NUM] = array[i];
            }
        }
        return array1;
    }
}
```

図 3.9 配列操作プログラム

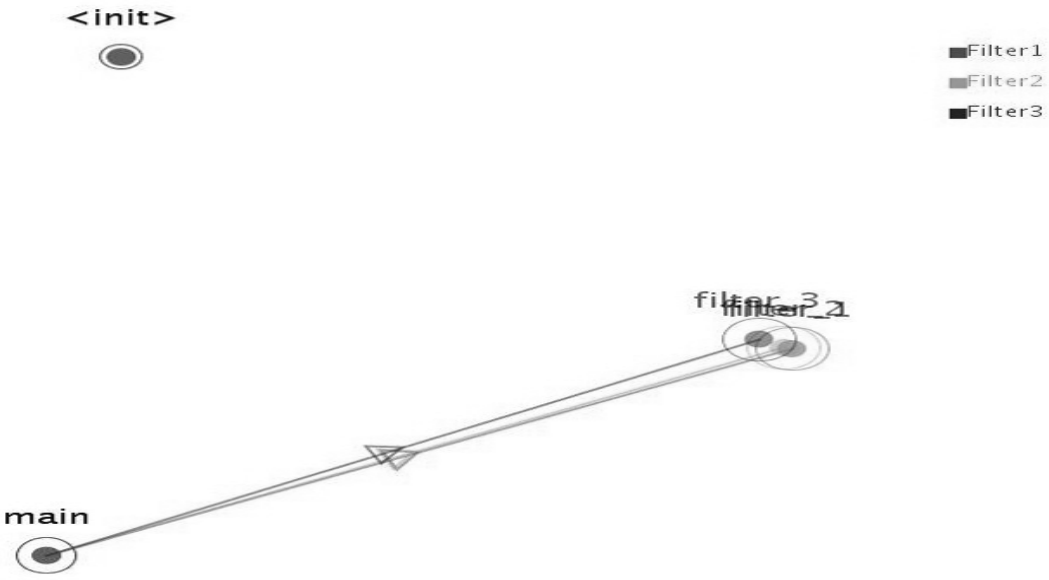

図 3.10 似ているプログラム

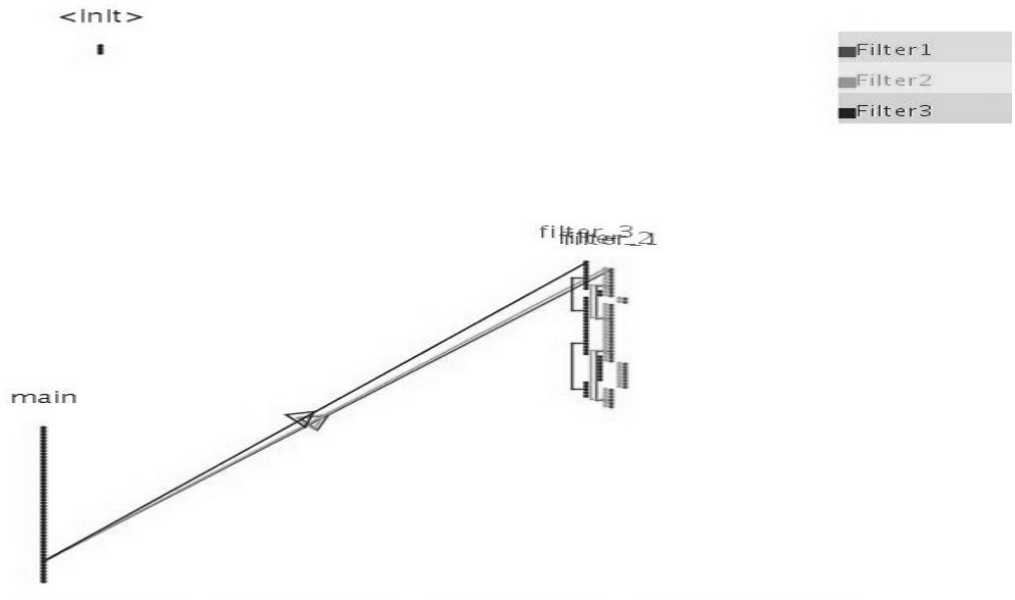

図 3.11 似ているプログラムのバイトフロー

```java
import java.util.Arrays;
public class QuickSort {
    public static void main(String[] args) {
        int[] array = new int[] { 5, 4, 6, 7, 1, 9, 8, 2, 3 };
        quickSort(array, 0, array.length - 1);
        System.out.println(Arrays.toString(array));
    }
    private static void quickSort(int[] array, int left, int right) {
        if (left <= right) {
            int pivotData = array[(left + right) / 2];
            int leftPointer = left;
            int rightPointer = right;
            while (leftPointer <= rightPointer) {
                while (array[leftPointer] < pivotData) {
                    leftPointer++;
                }
                while (array[rightPointer] > pivotData) {
                    rightPointer--;
                }
                if (leftPointer <= rightPointer) {
                    int tmp = array[leftPointer];
                    array[leftPointer] = array[rightPointer];
                    array[rightPointer] = tmp;
                    leftPointer++;
                    rightPointer--;
                }
            }
            quickSort(array, left, rightPointer);
            quickSort(array, leftPointer, right);
        }
    }
}
```

図 3.12 クイックソート

```java
import java.util.Arrays;
public class BubbleSort {
    public static void main(String[] args) {
        int[] array = new int[] { 5, 4, 6, 7, 1, 9, 8, 2, 3 };
        array = bubbleSort(array);
        System.out.println(Arrays.toString(array));
    }
    public static int[] bubbleSort(int[] array){
        for(int i = 0; i < array.length; i++){
            for(int j = 0; j < array.length - 1 - i; j++){
                if(array[j] > array[j + 1]){
                    int tmp = array[j];
                    array[j] = array[j + 1];
                    array[j + 1] = tmp;
                }
            }
        }
        return array;
    }
}
```

図 3.13 バブルソート

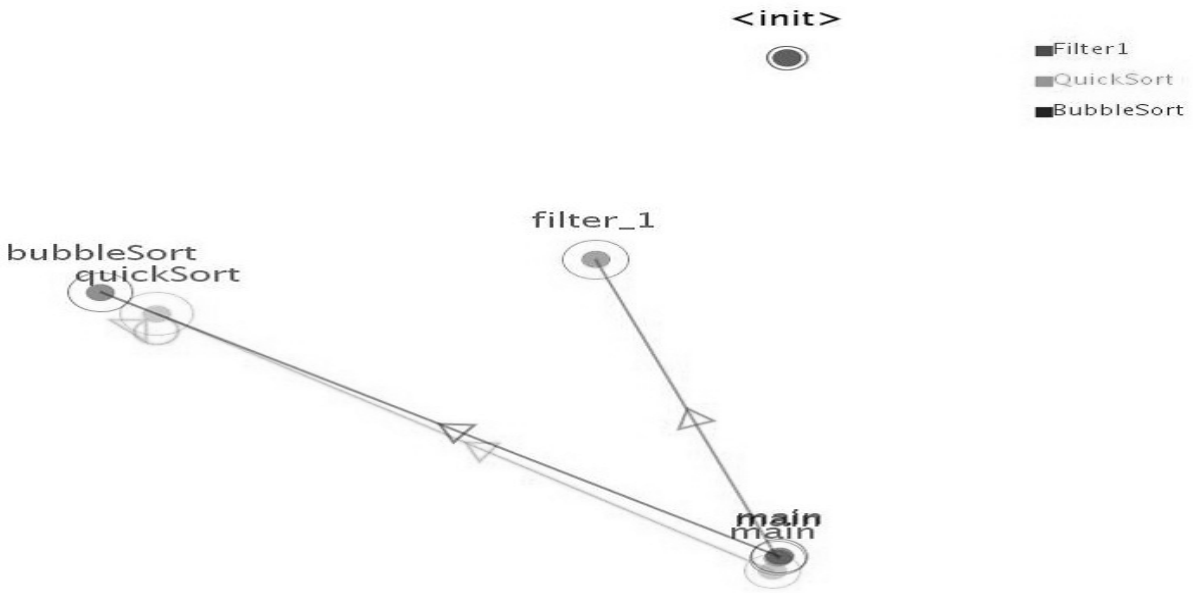

図 3.14 似ていないプログラム

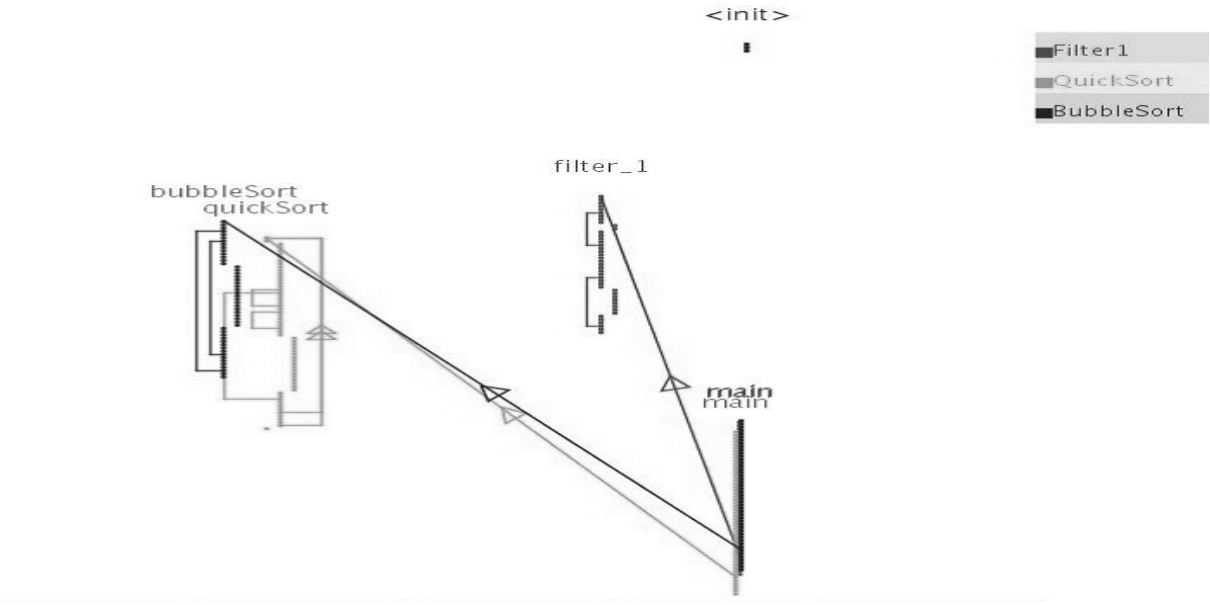

図 3.15 似ていないプログラムのバイトフロー

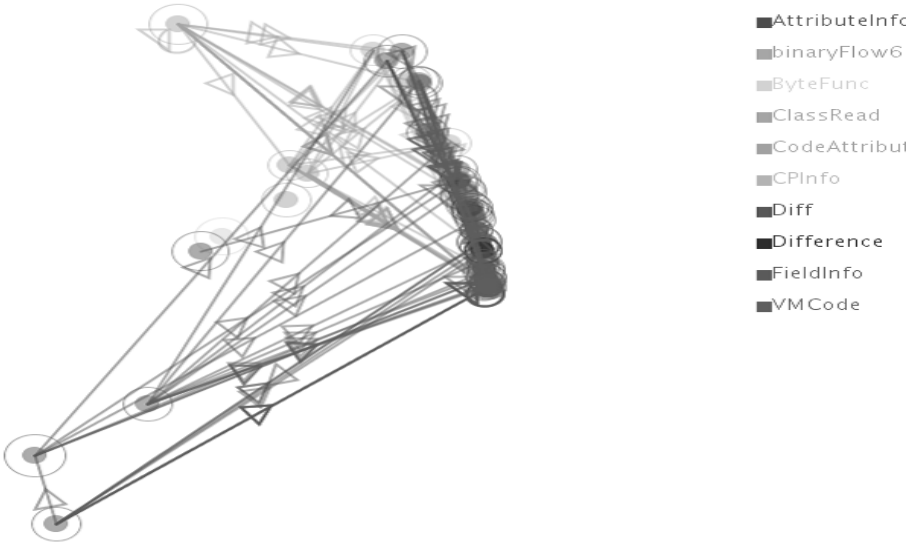

図 3.16 大規模なプログラムの可視化

第4章　考察

本研究では, プログラムのもつ関数とその呼び出し関係に着目し, その可視化の手法について考えた. この結果, プログラムの実際の挙動や実装されているアルゴリズムの比較を容易にすることができた.

また今回は, データの取得を容易にするために *Java* のコンパイル済みバイナリデータを用いたが, ソースコードもまた高度に構造化されたものであるため, 機械的にデータを取得することが可能である. このことから, ソースコードを対象とした近似度可視化も同じ手法を用いることができると考えられる.

今後の課題として, プログラムの処理が複数クラスに渡る場合のプログラム間の近似度可視化手法が挙げられる. これには, クラスの違い, クラス間の継承関係, 及び継承に対するメソッドのオーバーライドなど, より複雑な構造に対する可視化手法が必要になる.

謝辞

本研究を行うにあたり多大な助言, ご指導をいただいた指導教官の下薗真一准教授に心から感謝致します. また, いろいろとお世話になった篠原・下薗研究室の皆様に御礼申し上げます.

参考文献

[1] ドットインストール *(*添削通信コース*),* http://dotinstall.com/courses

[2] Tim Lindholm, Frank Yellin, 村上雅章*, Java* 仮想マシン仕様 *(*第 2 版*),* ピアソン・エデュケーション*, 2001*

[3] ORACLE (The Java Virtual Machine Specification), http://docs.oracle.com/javase/specs/jvms/se7/html/index.html

[4] Project CASE, http://case.f7.ems.okayama-u.ac.jp/

[5] Ben Fry, Visualizing Data, O'Reilly Media, 2007

[6] V.I.Levenstein, Binary codes capable of correcting insertions and reversals, Sov. Phys. Dokl., 10:707-10, 1966