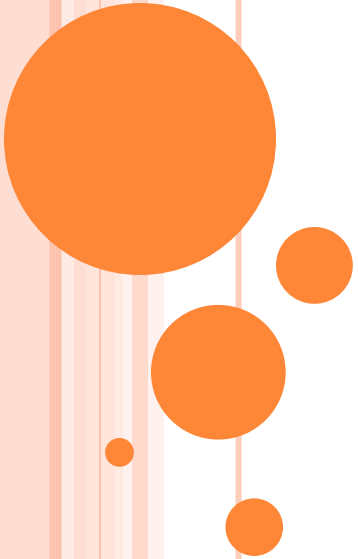


グラフの探索
Javaでの実装

二つの探索手法

- 深さ優先探索：DFS (Depth-First Search)
- 幅優先探索：BFS (Breadth-First Search)
- 共通部分
  - 元のグラフを指定して、極大木を得る

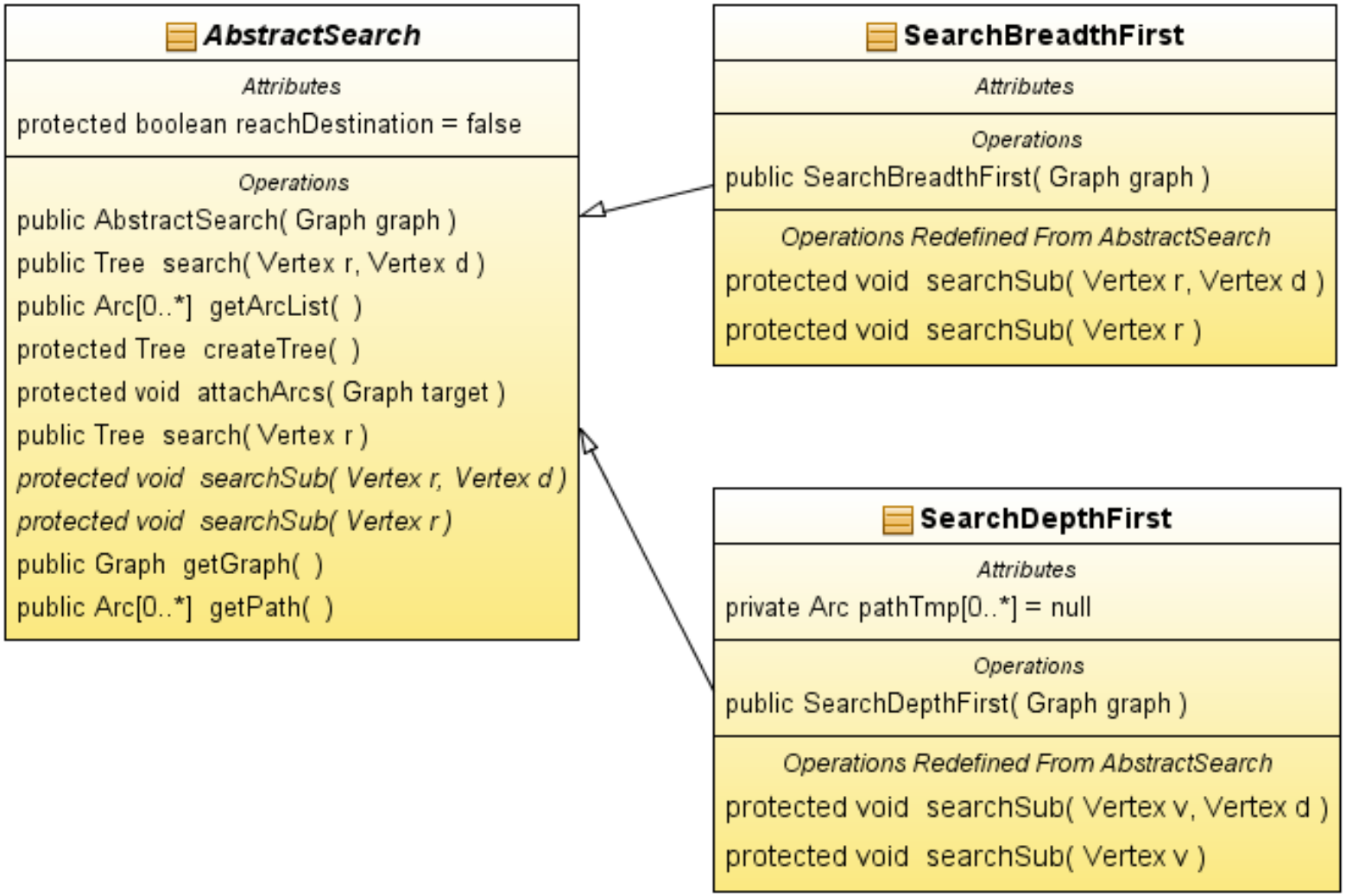
探索アルゴリズムの利用の観点から

- 利用する側からみると
  - 取り替えられる部品
  - どちらの方法が良いかはグラフに依存
  - 操作性が同じでなければ
  - 共通のクラスの派生で作ると便利
- 共通化を考える
  - 操作性の共通化
  - 共通のメソッド

Abstract classを使った共通化

- 共通のデータ構造
- 共通のメソッド
- 一部のメソッドの実装方法が異なる

- メソッド定義に修飾子abstract
  - 実装が定義されていない
  - 派生クラスで実装しなければならない
- クラス定義に修飾子abstract
  - このクラスのインスタンスが生成できないことを表す


AbstractSearch
Attributes
protected boolean reachDestination = false
Operations
public AbstractSearch(Graph graph)
public Tree search(Vertex r, Vertex d)
public Arc[0..*] getArcList( )
protected Tree createTree( )
protected void attachArcs(Graph target)
public Tree search(Vertex r)
protected void searchSub(Vertex r, Vertex d)
protected void searchSub(Vertex r)
public Graph getGraph( )
public Arc[0..*] getPath( )
SearchBreadthFirst
Attributes
Operations
public SearchBreadthFirst(Graph graph)
Operations Redefined From AbstractSearch
protected void searchSub(Vertex r, Vertex d)
protected void searchSub(Vertex r)
SearchDepthFirst
Attributes
private Arc pathTmp[0..*] = null
Operations
public SearchDepthFirst(Graph graph)
Operations Redefined From AbstractSearch
protected void searchSub(Vertex v, Vertex d)
protected void searchSub(Vertex v)

実装時の注意

- 元のグラフを壊さない
- 結果をtreeとして得る
  - 頂点の集合をコピーする
  - 探索に使用した弧の一覧を作成する
  - 探索に使用した弧のコピーを追加する
- 終点を指定した場合
  - 始点から終点への経路のみを生成する

AbstractSearchクラスのフィールド

```
/** 探索対象となるgraph */
  protected Graph graph;
/** 探索した頂点のリスト */
  protected List<Vertex> listOfVertex = null;
/** 終点を指定した場合、その終点に到達したか否か */
  protected boolean reachDestination = false;
/** 探索に使用した孤 */
  protected List<Arc>  arcList= null;
/** 始点 */
  protected Vertex r = null;
/** 終点 */
  protected Vertex d = null;
/** 始点から終点への道 */
  protected List<Arc> path=null;
```

AbstractSearchクラスの主要メソッド

public AbstractSearch(Graph graph)	コンストラクタ
public Tree search(Vertex r)	探索実行（終点指定なし）
public Tree search(Vertex r, Vertex d)	探索実行（終点指定）
protected Tree createTree()	探索結果から木を得る
protected void attachArcs(Graph target)	使用した弧をtargetに追加

- 以下は、各派生クラスで実装する

protected abstract void searchSub(Vertex r)	探索の実体
protected abstract void searchSub(Vertex r, Vertex d)	
Protected abstract void createPath(List<Arc> aList)	道の生成

深さ優先探索

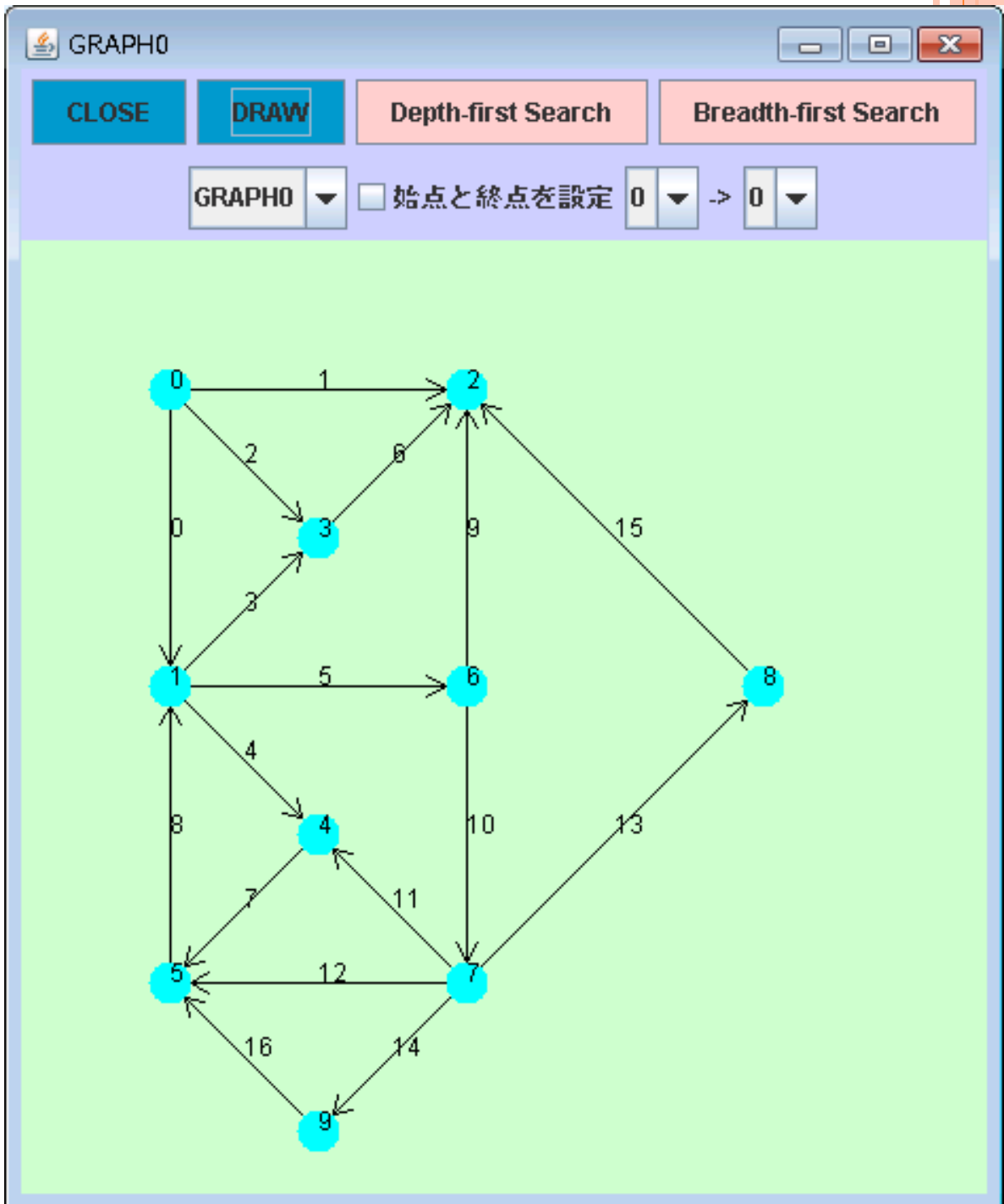
```
protected void searchSub(Vertex v) {
        List<Arc> aList = graph.getArcs(v);
        if (aList == null) {
            return;
        }
        for (Arc a : aList) {//頂点v から出ている弧
            //弧の先の頂点
            Vertex to = graph.getTerminal(a, v);
            if (!listOfVertex.contains(to)) {//弧の先の頂点はまだtreeに無い
                //頂点を追加
                listOfVertex.add(to);
                /** 探索に使用した弧を保存 */
                arcList.add(a);
                searchSub(to);
            }
        }
    }
```

幅優先探索

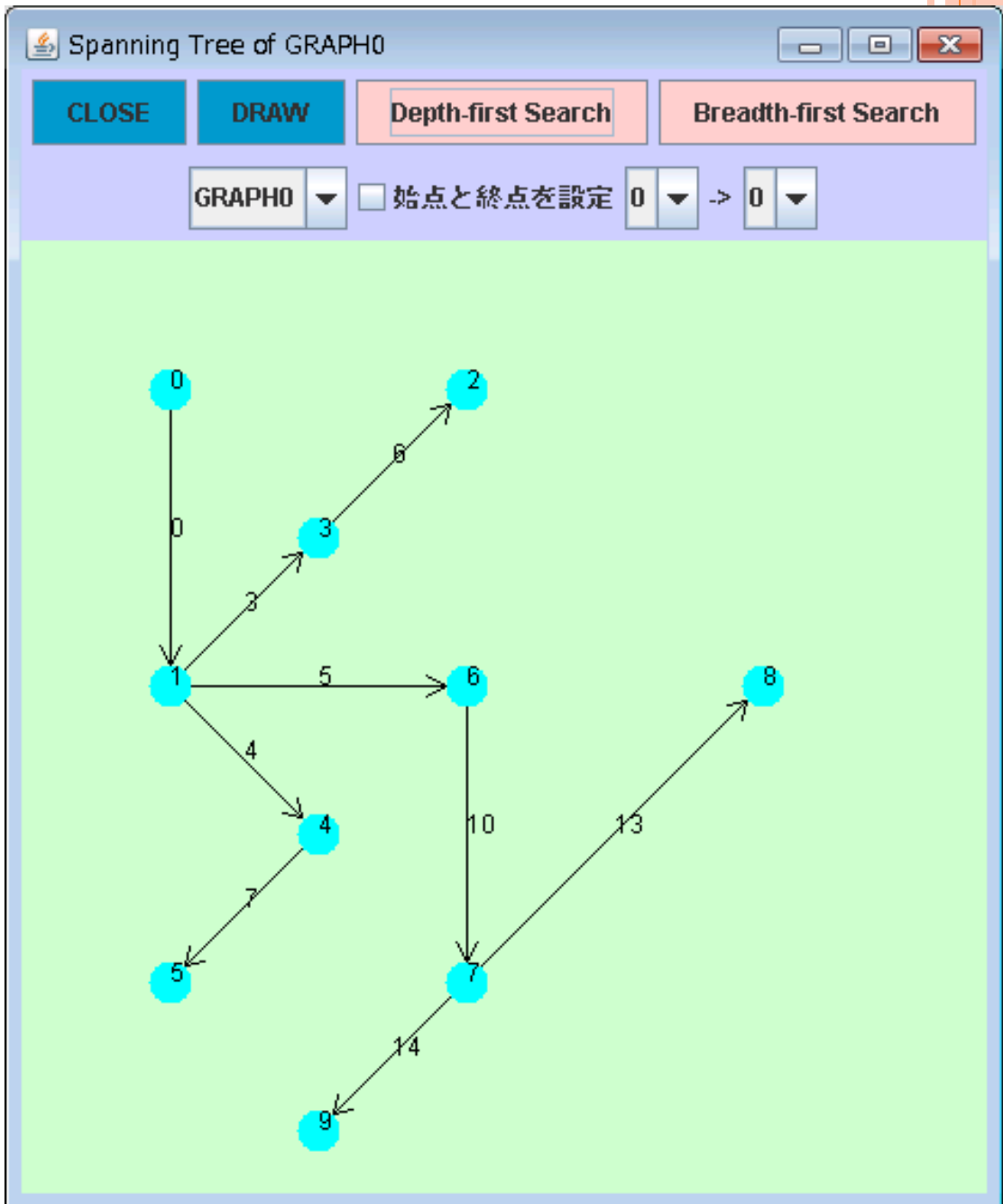
```
protected void searchSub(Vertex r) {
    //待ち行列の作成
    ConcurrentLinkedQueue<Vertex> queue
                        = new ConcurrentLinkedQueue<>();
    queue.add(r);
    while (!queue.isEmpty()) {
        Vertex v = queue.poll();
        List<Arc> aList = graph.getArcs(v);
        if (aList != null) {
            for (Arc a : aList) {
                checkArc(v, null, a, queue);
            }
        }
    }
}
```

```
protected void checkArc(
        Vertex v, Vertex d, Arc a, ConcurrentLinkedQueue<Vertex> queue) {
        Vertex to = graph.getTerminal(a, v);//vと反対側
        if (!listOfVertex.contains(to) && !queue.contains(to)) {
            addFoundVertex(to, v);
            arcList.add(a);
            if (d != null && to.equals(d)) {//目標に到達したか
                reachDestination = true;
                createPath(arcList);
                return;
            }
            queue.add(to);
        }
    }
```

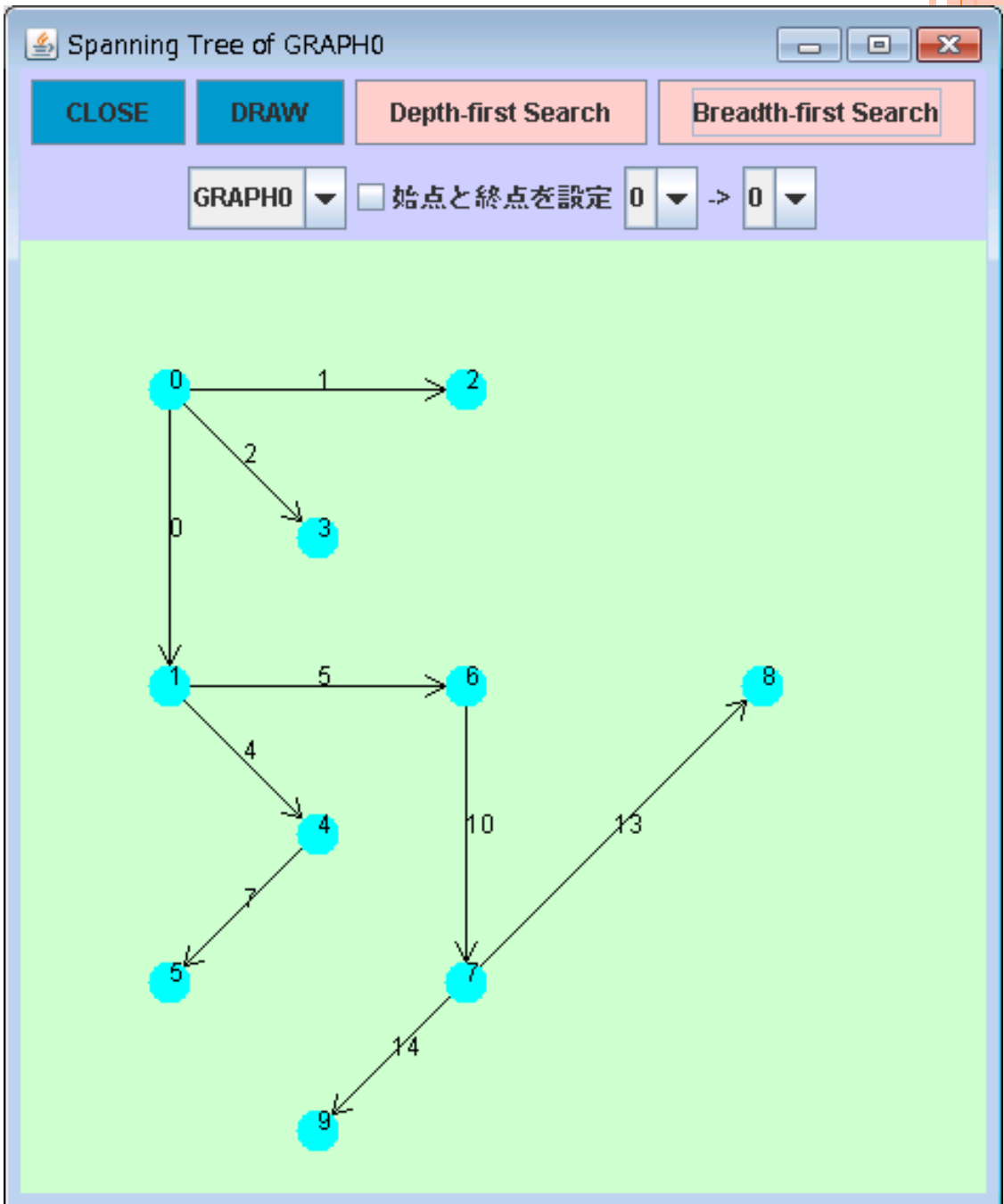
対象グラフの表示


GRAPH0
CLOSE
DRAW
Depth-first Search
Breadth-first Search
GRAPH0
始点と終点を設定
0
->
0
0
1
2
3
4
5
6
7
8
9
1
2
6
0
3
9
15
3
5
4
8
10
13
7
11
12
16
14

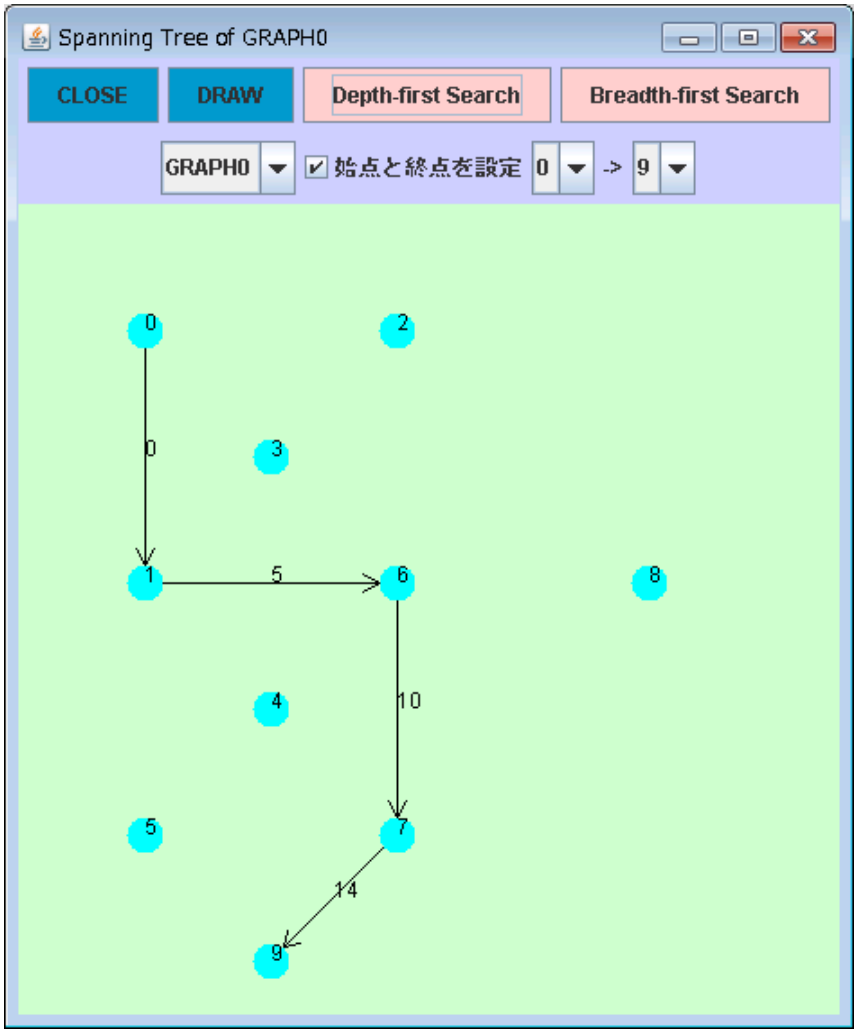
深さ優先探索の実施


Spanning Tree of GRAPH0
CLOSE
DRAW
Depth-first Search
Breadth-first Search
GRAPH0
始点と終点を設定
0
->
0
0
2
6
0
3
3
1
5
6
8
4
4
10
13
7
5
7
14
9

幅優先探索の実施


Spanning Tree of GRAPH0
CLOSE
DRAW
Depth-first Search
Breadth-first Search
GRAPH0
始点と終点を設定
0
->
0
0
1
2
3
4
5
6
7
8
9
1
2
0
5
4
7
10
13
14

終点を指定した深さ優先探索の実施



AbstractSearch.java

```
package graphAnalysis;

import java.util.List;
import graphLib.*;
import java.util.HashMap;

/**
 * AbstractSearch.java
 * 探索の抽象クラス
 * @author tadaki
 */
public abstract class AbstractSearch {

    /** 探索対象となるgraph */
    protected Graph graph;
    /** 探索した頂点のリスト */
    protected List<Vertex> listOfVertex = null;
    /** 終点を指定した場合、その終点に到達したか否か */
    protected boolean reachDestination = false;
    /** 探索に使用した孤 */
    protected List<Arc> arcList = null;
    /** 始点 */
    protected Vertex source = null;
    /** 終点 */
    protected Vertex destination = null;
    /** 始点から終点への道 */
    protected List<Arc> path = null;

    /** Creates a new instance of AbstractSearch */
    public AbstractSearch(Graph graph) {
        this.graph = graph;
    }

    /**
     * 探索実行
     * @param source 探索を開始する頂点
     * @param destination 探索の終点（指定しない場合はnull）
     * @return 探索結果のtree
     * 終点を指定し、かつ終点に到達できない場合はnullを返す
     */
    public Tree search(Vertex r, Vertex d) {
        if (r == null || !graph.getVertexes().contains(r)) {
            return null;
        }
```

```
        this.source = r;
        if (d != null && graph.getVertexes().contains(d)) {
            this.destination = d;
        }
        listOfVertex = Utils.createVertexList();
        listOfVertex.add(r);
        arcList = Utils.createArcList();
        /** 探索開始 */
        if (this.destination != null) {
            searchSub(r, d);
        } else {
            searchSub(r);
        }
        /** 探索結果を木に整形 */
        Tree tree = createTree();
        /** 終点が指定され、かつ終点に到達しないならば、
         * treeとしてnullを返す
         */
        if (this.destination != null && !reachDestination) {
            tree = null;
        }
        return tree;
    }

    public List<Arc> getArcList() {
        return arcList;
    }

    /**
     * 探索結果を木に整形
     * @return 探索結果の木
     */
    protected Tree createTree() {
        Tree tree = new Tree(graph, false); //結果を保存する木(頂点のみをこぴー)
        attachArcs(tree);
        tree.setName("Spanning Tree of " + graph.getName());
        //探索結果のtreeの根の設定
        tree.setRoot(tree.getMap().get(source));
        return tree;
    }

    /**
     * pathに使用されている弧を追加する
```

```
     */
    protected void attachArcs(Graph target) {
        HashMap<Vertex, Vertex> vmap = target.getMap();
        for (Arc a : arcList) {
            Vertex head = graph.getHead(a);
            Vertex tail = graph.getTail(a);
            target.addArc(vmap.get(head), vmap.get(tail), a.toString());
        }
    }

    /**
     * 探索実行（終点を指定しない）
     * @param source 探索を開始する頂点
     * @return 探索結果のtree
     */
    public Tree search(Vertex r) {
        return search(r, null);
    }

    /*
     * 探索の実体
     * @param source 探索の現在位置
     * @param destination 探索の終端
     */
    protected abstract void searchSub(Vertex r, Vertex d);

    /*
     * 探索の実体
     * @param source 探索の現在位置
     */
    protected abstract void searchSub(Vertex r);

    /**
     * 始点から終点への道の生成
     * @param aList
     */
    protected abstract void createPath(List<Arc> aList);

    /**
     * グラフを取得
     * @return
     */
    public Graph getGraph() {
        return graph;
```

```
    }

    public List<Arc> getPath() {
        return path;
    }

    public List<Vertex> getListOfVertex() {
        return listOfVertex;
    }
}
```

SearchDepthFirst.java

```
package graphAnalysis;

import java.util.List;
import graphLib.*;

/**
 * 深さ優先探索
 * SearchDepthFirst.java
 * @author tadaki
 */
public class SearchDepthFirst extends AbstractSearch {

    /** 探索途中の道の弧 */
    private List<Arc> pathTmp = null;

    /**
     * Creates a new instance of SearchDepthFirst
     * @param graph 探索対象のgraph
     */
    public SearchDepthFirst(Graph graph) {
        super(graph);
        pathTmp = Utils.createArcList();
    }

    /*
     * 探索の実体
     * @param source 探索の現在位置
     * @param destination 探索の終端
     */
    @Override
    protected void searchSub(Vertex v, Vertex d) {
        if (graph.getArcs(v) == null) {
            return;
        }
        for (Arc a : graph.getArcs(v)) {//頂点v から出ている弧
            //弧の先の頂点
            Vertex to = graph.getTerminal(a, v);
            if (!listOfVertex.contains(to)) {//弧の先の頂点はまだtreeに無
い
                //頂点を追加
                listOfVertex.add(to);
                /** 使用した孤を一時的に保存 */
                pathTmp.add(a);
                if (to.equals(d)) {//目標に到達したか
```

```
                    reachDestination = true;
                    /** 目標までの弧を保存 */
                    for (Arc aa : pathTmp) {
                        arcList.add(aa);
                    }
                    createPath(arcList);
                    return;
                }
                searchSub(to, d);
                pathTmp.remove(a);
            }
        }
    }

    /*
     * 探索の実体
     * @param source 探索の現在位置
     */
    @Override
    protected void searchSub(Vertex v) {
        List<Arc> aList = graph.getArcs(v);
        if (aList == null) {
            return;
        }
        for (Arc a : aList) {//頂点v から出ている弧
            //弧の先の頂点
            Vertex to = graph.getTerminal(a, v);
            if (!listOfVertex.contains(to)) {//弧の先の頂点はまだtreeに無
い
                //頂点を追加
                listOfVertex.add(to);
                /** 探索に使用した弧を保存 */
                arcList.add(a);
                searchSub(to);
            }
        }
    }

    @Override
    protected void createPath(List<Arc> aList) {
        if(destination==null)return;
        path = Utils.createArcList();
        for(Arc a:aList)path.add(a);
    }
```

```
 }
```

SearchBreadthFirst.java

```
package graphAnalysis;

import java.util.concurrent.ConcurrentLinkedQueue;
import graphLib.*;
import java.util.List;

/**
 * 幅優先探索
 * SearchBreadthFirst.java
 * @author tadaki
 */
public class SearchBreadthFirst extends AbstractSearch {

    /**
     * Creates a new instance of SearchDepthFirst
     * @param graph 探索対象のgraph
     */
    public SearchBreadthFirst(Graph graph) {
        super(graph);
    }

    /*
     * 探索の実体
     * @param source 探索の現在位置
     * @param destination 探索の終端
     */
    @Override
    protected void searchSub(Vertex r, Vertex d) {
        //待ち行列の作成
        ConcurrentLinkedQueue<Vertex> queue = new
ConcurrentLinkedQueue<>();
        queue.add(r);
        while (!queue.isEmpty()) {
            Vertex v = queue.poll();
            List<Arc> aList = graph.getArcs(v);
            if (aList != null) {
                for (Arc a : aList) {
                    checkArc(v, d, a, queue);
                }
            }
        }
    }

    /*
```

```
     * 探索の実体
     * @param source 探索の現在位置
     */
    @Override
    protected void searchSub(Vertex r) {
        //待ち行列の作成
        ConcurrentLinkedQueue<Vertex> queue = new
ConcurrentLinkedQueue<>();
        queue.add(r);
        while (!queue.isEmpty()) {
            Vertex v = queue.poll();
            List<Arc> aList = graph.getArcs(v);
            if (aList != null) {
                for (Arc a : aList) {
                    checkArc(v, null, a, queue);
                }
            }
        }
    }

    protected void checkArc(
            Vertex v, Vertex d, Arc a, ConcurrentLinkedQueue<Vertex>
queue) {
        Vertex to = graph.getTerminal(a, v);//vと反対側
        if (!listOfVertex.contains(to) && !queue.contains(to)) {
            addFoundVertex(to, v);
            arcList.add(a);
            if (d != null && to.equals(d)) {//目標に到達したか
                reachDestination = true;
                createPath(arcList);
                return;
            }
            queue.add(to);
        }
    }

    protected void addFoundVertex(Vertex to, Vertex from) {
        listOfVertex.add(to);
    }

    @Override
    protected void createPath(List<Arc> aList) {
        if (destination == null) {
            return;
```

```
            }
            path = Utils.createArcList();
            Vertex v = destination;
            if (graph.isDirected()) {
                while (!v.equals(source)) {
                    for (Arc a : aList) {
                        if (graph.getTail(a).equals(v)) {
                            v = graph.getHead(a);
                            path.add(a);
                            break;
                        }
                    }
                }
            } else {
                while (!v.equals(source)) {
                    for (Arc a : aList) {
                        if (graph.getTail(a).equals(v)
                                || graph.getHead(a).equals(v)) {
                            Vertex w = graph.getTerminal(a, v);
                            v = w;
                            path.add(a);
                            break;
                        }
                    }
                }

            }
        }
    }
```