File - ContractionAlgorithm.java

```java
package gre.lab1.groupD;

import gre.lab1.graph.*;

import java.util.LinkedList;
import java.util.List;

/**                                                    graphe réduit
 * L'algorithme de contraction permet d'avoir un graphe des composantes fortement connexe, et permet aussi
 * pour chaque CFC (SCC en anglais) d'avoir leurs sommets composants ainsi que leurs successeurs.
 *
 * @author Edwin Haeffner
 * @author Arthur Junod
 */
public class ContractionAlgorithm implements GenericAlgorithm<GraphCondensation> {

  private final SccAlgorithm sccAlgo; //L'agorithme à utiliser pour trouver les scc

  /**
   * Constructeur
   * @param sccAlgo l'algorithme à utiliser pour trouver les composantes fortement connexes
   */
  public ContractionAlgorithm(SccAlgorithm sccAlgo){
    this.sccAlgo = sccAlgo;
  }

  /**
   * Crée l'objet GraphCondensation
   * @param graph Le graphe orienté à traiter.
   * @return un GraphCondensation
   */
  @Override
  public GraphCondensation compute(DirectedGraph graph) {
    GraphScc scc = sccAlgo.compute(graph);
    DirectedGraph condensation = new DirectedGraph(scc.count()); //count = nombre de composantes fortement connexes
    List<List<Integer>> mapping = new LinkedList<>(); //Donne les informations de chaque composante fortement connexe
                                                       La taille est connue => ArrayList !!

    //On ajoute chaque composante fortement connexe au mapping ~~
    for (int i = 0; i < scc.count(); i++) {
      mapping.add(new LinkedList<>());
    }

    //Peuple chaque scc de leurs composantes respectives  Pas très clair
    for (int i = 0; i < graph.getNVertices(); i++) {
      int sccIndex = scc.componentOf(i);
      mapping.get(sccIndex-1).add(i);
    }

    //Pour chaque sommet du graphe initial
    for (int i = 0; i < graph.getNVertices(); i++) {
      int sccIndexI = scc.componentOf(i) - 1; // On récupère l'indice de la scc du sommet 'i'
      //Pour chaque successeur de 'i'
      for (int j : graph.getSuccessorList(i)) {
        int sccIndexJ = scc.componentOf(j) - 1; //On récupère l'indice de la scc du sommet 'j' ( successeur de 'i')
        // Si 'i' et 'j' appartiennent à des scc différentes, on ajoute un arc entre les deux scc
        // On vérifie aussi si l'arc n'est pas déjà existant pour éviter les doublons
        if(sccIndexI != sccIndexJ && !condensation.getSuccessorList(sccIndexI).contains(sccIndexJ)){
          condensation.addEdge(sccIndexI, sccIndexJ);
        }
      }
    }
```

Contains est vraiment à éviter, cela nécessite de parcourir à chaque fois la collection. Lent en pratique !

File - ContractionAlgorithm.java

```
62
63      return new GraphCondensation(graph,condensation, mapping);
64    }
65 }
```

File - Main.java

```java
package gre.lab1.groupD;

import gre.lab1.graph.DirectedGraph;
import gre.lab1.graph.DirectedGraphReader;
import gre.lab1.graph.GraphCondensation;

import java.io.IOException;
import java.util.List;

/**
 * Ce programme permet de trouver les composantes fortement connexes grâce à l'algorithme de Tarjan et
 * d'ensuite les analyser en tant que chaîne de Markov grâce à l'algorithme de contraction qui nous permet d'avoir
 * un graphe de ces composantes fortement connexes.
 * Les graphes sont passés sous forme de fichier .txt avec une structure définie à l'avance (cf. "format données"
 * dans le pdf du labo).
 *
 * @author Edwin Haeffner
 * @author Arthur Junod
 */
public class Main {

  final static String FILE_TO_READ = "data/chaine1.txt";

  public static void main(String[] args) throws IOException {

    DirectedGraph graph = DirectedGraphReader.fromFile(FILE_TO_READ);

    //On initialise un algorithme de contraction avec l'algorithme de Tarjan
    ContractionAlgorithm ca = new ContractionAlgorithm(new TarjanAlgorithm());

    printGraphCondensationResult(ca.compute(graph));
  }

  /**
   * Permet d'afficher les informations liées à la chaine de Markov donnée par l'objet GraphCondensation
   * @param gc le graphe à afficher
   */
  static void printGraphCondensationResult(GraphCondensation gc){
    System.out.println("Nombre de composantes fortement connexes : " + gc.condensation().getNVertices());

    for (int i = 0; i < gc.mapping().size(); i++) {
      System.out.println("Composante " + (i+1) + " :");
      //Si la list de successeurs est vide, alors la composante est persistante
      System.out.println("  - Statut : " + (gc.condensation().getSuccessorList(i).isEmpty() ? "persistante" : "transitoire"));
      System.out.println("  - Sommets : " + printArrayPlusOne(gc.mapping().get(i)));
      if (!gc.condensation().getSuccessorList(i).isEmpty()) {
        System.out.println("  - Successeurs : " + printArrayPlusOne(gc.condensation().getSuccessorList(i)));
      }
    }
  }

  /**
   * Permet d'avoir un affichage qui commence à 1 au lieu de 0 pour pouvoir lire les informations plus facilement
   * @param l Une liste de 'int' qui représente les index de chacun des sommets
   * @return Un String du graphe
   */
  static String printArrayPlusOne(List<Integer> l){
    return l.stream().map(n -> n + 1).toList().toString();
```

format, vous n'imprimez rien (détail)

File - Main.java

```
  }

}


```

File - TarjanAlgorithm.java

```java
package gre.lab1.groupD;

import gre.lab1.graph.DirectedGraph;
import gre.lab1.graph.GraphScc;
import gre.lab1.graph.SccAlgorithm;

import java.util.Stack; Ok mais Deque (avec une implémentation appropriée) est à privilégier, Stack est une relique.

/**
 * Classe servant à trouver toutes les composantes fortement connexe d'un graphe.
 * Cette classe implémente l'algorithme de Tarjan -> Plus d'info : <a href="https://fr.wikipedia.org/wiki/Algorithme_de_Tarjan">Algorithme de Tarjan - Wikipedia</a> Un peu cavalier... u_u
 * @author Edwin Haeffner
 * @author Arthur Junod
 */
public final class TarjanAlgorithm implements SccAlgorithm {

  /**
   * Structure de données qui nous permet de manipuler plus facilement les différentes données utiles
   * à l'algorithme de Tarjan.
   *
   * @author Edwin Haeffner
   * @author Arthur Junod
   */
  private static class DataTarjan {
    public Stack<Integer> stack = new Stack<>();  // Pile qui nous permet de garder en mémoire les sommets restant à classer
    public int numDfsnum = 0;                     // Variable à incrémenter pour générer l'ordre de visite de chaque sommet
    public int numCompos = 0;                     // Variable à incrémenter pour générer le numéro de chaque composante fortement connexe
    public int nbVertices;                        // Nombre de sommets

    // Chaque tableau est de la taille du nombre de sommets, de ce fait, on peut accéder à la donnée relative au sommet i avec tab[i]
    public int[] dfsnum;  // Tableau contenant l'ordre de visite de chaque sommet
    public int[] scc;     // Tableau contenant l'appartenance de chaque sommet à une composante fortement connexe
    public int[] low;     // Tableau contenant le plus vieil ancêtre accessible de chaque sommet

    /**
     * Constructeur permettant de préparer les tableaux et récupérer le nombre de sommets
     * @param graph Le graphe sur lequel nous allons appliquer l'algorithme de Tarjan
     *
     */
    public DataTarjan(DirectedGraph graph){
      nbVertices  = graph.getNVertices();
      dfsnum      = new int[nbVertices];
      scc         = new int[nbVertices];
      low         = new int[nbVertices];
    }
  }

          A mettre dans DataTarjan (ou passer en paramètre de sccProcedure)
  private DirectedGraph graph; // Le graphe sur lequel on applique Tarjan

  /**
   * Applique l'algorithme de Tarjan sur un graphe et permet d'en récupérer le résultat
   *
   * @param graph Le graphe sur lequel on applique Tarjan
   * @return Un nouveau graphe qui a les informations sur les différentes composantes fortement connexes du graphe donné
   */
  @Override
  public GraphScc compute(DirectedGraph graph) {

```

File - TarjanAlgorithm.java

```java
    this.graph = graph;

    DataTarjan data = new DataTarjan(graph);

    //Effectue la procédure de découverte des composantes fortements connexe sur chaque sommet
    for(int i = 0; i < data.nbVertices; ++i){
      // Vérifie que le sommet n'a pas été déjà traité par l'appel récursif de sccProcedure()
      if(data.scc[i] == 0){
        sccProcedure(i,data);
      }
    }


    return new GraphScc(graph, data.numCompos, data.scc);
  }

  /**
   * Procédure récursive pour trouver les composantes fortement connexes
   * @param v le numéro du sommet sur lequel on travaille
   * @param d les informations du graphe
   */
  private void sccProcedure(int v, DataTarjan d){
    d.dfsnum[v] = ++d.numDfsnum; //On incrémente l'ordre de passage et on l'assigne au sommet 'v'
    d.low[v]    = d.numDfsnum;   // Au début le plus vieil ancêtre accessible d'un sommet ne peut être que lui-même

    //On rajoute le sommet sur la pile
    d.stack.push(v);

    //Pour chaque sommet successeur à 'v'
    for(int successor : graph.getSuccessorList(v)){
      if(d.dfsnum[successor] == 0) sccProcedure(successor, d); //Si le sommet n'a pas été encore visité, nous y allons récursivement
      // Mettre à jour la valeur "low" de v si successor à un ancêtre accessible qui est plus vieux (si son low est plus petit)
      if(d.scc[successor] == 0) d.low[v] = Math.min(d.low[successor], d.low[v]);
    }
    /*
    Si on n'a pas trouvé d'ancêtre accessible plus vieux que soi-même dans les successeurs,
    on crée une composante fortement connexe pour ce sommet.
    */
    if(d.low[v] == d.dfsnum[v]){
      d.numCompos++; //Commence à numéroter les composantes à partir de 1.
      int top;
      do{
        top = d.stack.pop(); // On récupère chaque ancêtre grâce à la pile
        d.scc[top] = d.numCompos; // On donne à chaque ancêtre récupéré le numéro de la composante
      } while (top != v); // jusqu'à arriver au sommet 'v'
    }
  }
}
```

Constante

"avec ce sommet et tous ses successeurs dans l'arbre DFS" ?

Mérite d'être mentionné plus tôt (par exemple à la déclaration), idem l'ordre de pasage.