Cours LIFI - 2004

TD n°2

6 octobre 2004

1 Algorithmique des graphes

On suppose qu'on a les classes `Node`, `Edge` et `Graph` définie au TP2 et dont l'interface est rappelée ici.

```
public class Node implements Serializable {
    public int getNbEdges();
    public Edge getEdge(int i);
    public int getX();
    public int getY();
}

public class Edge implements Serializable {
    public Node getFrom();
    public Node getTo();
}

public class Graph implements Serializable      {
    public Graph();
    public Node newNode(int x,int y);
    public Edge newEdge(Node a,Node b);
    public int getNbNodes();
    public int getNbEdges();
    public Node getNode(int i);
    public Edge getEdge(int i);
}
```

On souhaite définir plusieurs algorithmes liés à ces graphes.

1.1 Détection des composantes connexes

On veut trouver combien de composantes connexes[1] il y a dans un graphe donné. Pour cela, on va attribuer à chaque sommet un numéro (en commençant à zéro). Tous les sommets d'une même composante connexe auront le même numéro et donc le nombre de composantes connexes sera donné par le plus grand numéro utilisé.

Pour simplifier, on suppose que la classe `Node` contient un champ public `n` qui est initialisé à -1 lors de l'instanciation d'un `Node`.

On va ensuite partir d'un sommet au hasard et lui associer le numéro 0 et on va se déplacer successivement sur tous ses voisins (les sommets reliés par une arête) et leur donner aussi le numéro 0. Quand on a exploré tous les voisins, si il reste des sommets

[1] une composante connexe est un ensemble maximal de noeuds tel que l'on peut aller de tout noeud de l'ensemble vers n'importe quel autre.

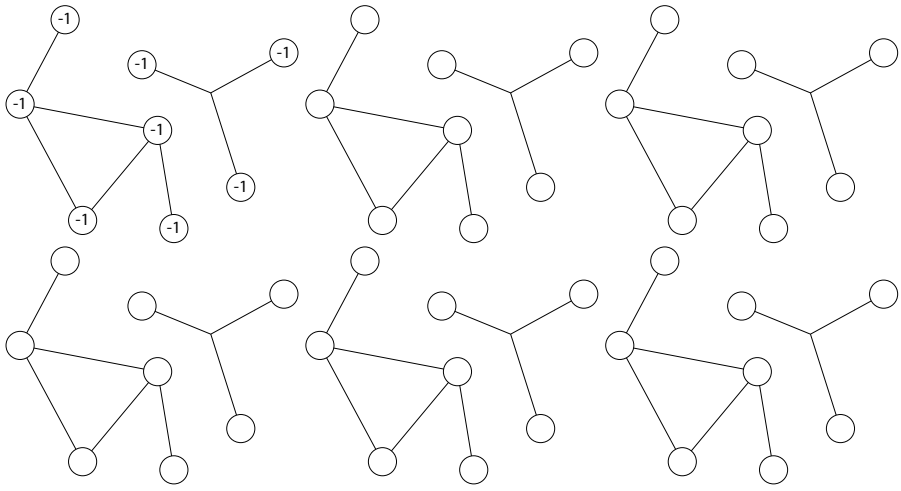

Figure 1: L'état des champs `n` au différentes étapes de la détermination des composantes connexes.

que l'on a pas visité dans le graphe, on recommence avec un de ces sommets et en utilisant le numéro 1. On recommence jusqu'à ce que l'on ai visité tous les sommets du graphe.

Question 1 (Étude d' un example). *Compléter sur la figure 1 les valeurs des champs* `n` *de chaque noeud à chaque étape de l'algorithme.*

Question 2 (Version n°1). *Écrire une méthode* `attributeId()` *de la classe* `Graph` *qui implémente l'algorithme décrit ci-dessus. On commencera par une version récursive. On utilisera un tableau de* `boolean` *de même taille que le nombre de noeuds pour savoir si un noeud a été visité ou non.*

Question 3 (Version n°2). *Faire une version non récursive. On utilisera un vecteur pour stocker la liste des sommets à visiter.*

Rajouter à la classe `Graph` un champ `Vector components` qui garde une référence vers un sommet dans chaque composante connexe.

1.2 Détection de cycles

On va maintenant utiliser une approche similaire à celle pour compter les composantes connexes pour déterminer si le graphe contient des cycles. On partira d'un sommet et on visitera ses voisins jusqu'à ce qu'on les ai tous explorés. Si en visitant ainsi les voisins, on visite deux fois un même noeud, c'est qu'il y a un cycle. On répétera cette procédure pour chaque composante connexe.

Question 4. *Définir une méthode* `hasCycles()` *de la classe* `Graph` *qui renvoie vrai ou faux selon que le graph contient des cycles ou non.*