Andrianarivony Henintsoa
Sauzeau Yannis

L3 Informatique
Université de Poitiers

MATHÉMATIQUES POUR L'INFO

RAPPORT Projet

Sommaire

Introduction

L’objectif de ce projet est d’utiliser les programmes des précédents TP pour une mise en pratique sur des algorithmes s'exécutant sur des graphes.
Deux algorithmes doivent être programmés. Le premier correspond à un parcours en largeur du graphe à partir d’un sommet. Ce parcours permet de connaître le chemin le plus court entre un sommet donné et les autres sommets qu’il relie.
Grâce à ce premier algorithme nous pouvons aisément en programmer un second qui va calculer les différentes composantes connexes d’un graphe donné.

Les classes utilisées

Les classes GraphSimple et GraphSimpleIO

Ces deux classes s'occupent de la modélisation d’un graphe avec des méthodes utiles à leur manipulation. Pour être plus précis, la classe GraphSimple instancie un graphe et GraphSimpleIO orchestre les entrées et les sorties.

Les classes Color et BreadthFirstSearch

La classe Color est toute simple puisqu’elle énumère trois couleurs qui représentent l’état d’un sommet lors du parcours en largeur. Un parcours que tâche d’accomplir la classe BreatdhFirstSearch (nom anglais du parcours en largeur).

La classe Connectivity

Cette classe s’occupe du deuxième algorithme, elle permet donc d’obtenir les composantes connexes d’un graphe.

La classe Test

Toutes les fonctions importantes au bon fonctionnement des programmes sont testées dans cette classe.

Algorithme 1

Le but de ce premier exercice est de construire une classe BreadthFirstSearch qui va permettre d'implémenter notre algorithme de parcours en largeur. Pour cela, il va nous falloir stocker le graphe ainsi que des données concernant chacun de ses sommets que sont sa couleur, son parent, et sa distance du sommet racine.

On crée donc un attribut *graph* de classe GraphSimple et trois attributs *color*, *parent* et *distance* qui sont respectivement des tableaux de Color, d'entier et d'entier (dont les indices correspondent aux sommets du graphe). Notre classe ainsi constituée va intégrer un constructeur prenant comme paramètre un graphe, le copier dans l'attribut *graph* et instancier les tableaux des trois autres attributs.

Il faut tout d'abord créer le type énuméré Color qui sera composé des couleurs vert, orange et rouge.

Ensuite, on définit les méthodes pour affecter (setters) et accéder (getters) aux trois attributs que sont les tableaux. Nous avons donc:

- ***void setColor(int x, Color c)*** : affecte la couleur c au sommet x
- ***Color getColor(int x)*** : retourne la couleur du sommet x
- ***void setDistance(int x, int d)*** : affecte la distance d au sommet x
- ***int getDistance(int x)*** : retourne la distance à la racine du sommet x
- ***void setParent(int x, int p)*** : affecte le parent p au sommet x
- ***int getParent(int x)*** : retourne le parent du sommet x

Une fois cela fait, on crée une méthode ***setVertex(int x, Color color, int distance, int parent)*** intégrant les trois setters précédent pour pouvoir définir directement un sommet x avec les données passé en paramètre.

Nous avons également ajouté une méthode ***print()*** permettant d'afficher l'état de chaque de sommet du graphe (couleur, distance et parent) sur le flot de sortie pour les tests.

Concernant l'implantation de l'algorithme de parcours en largeur, on définit dans un premier temps une méthode ***initializeColor()*** pour initialiser la couleur de chaque sommet en vert.

Dans un second temps, on définit une méthode auxiliaire ***rootBFS(int r)*** qui va effectuer un parcours en largeur à partir du sommet racine r passé en paramètre. Cette méthode est construite en retranscrivant le pseudo code donné par le sujet (cf. commentaires de la méthode dans *BreadthFirstSearch.java*). Ces deux dernières méthodes permettent de construire la méthode ***rootBFSFirst(int r)*** initialisant chaque sommet (couleur en vert, distance et parent à 0) et effectuant un parcours en largeur pour le sommet r.

Enfin, comme la méthode précédente ne permet pas un parcours complet lorsque le graphe possède plusieurs composantes connexes (réponse à la question f), on définit la méthode ***completeBFS()***. Celle-ci réutilise la méthode *rootBFSFirst(1)* pour initialiser tous les sommets et lancer un parcours en largeur sur le premier sommet. Ensuite, elle cherche parmi les sommets du graphe ceux qui n'ont pas été traités et lance un parcours en largeur sur ceux-ci via la méthode *rootBFS*.

Algorithme 2

Le but de cet exercice est de construire une classe Connectivity qui va permettre d'implémenter notre algorithme de récupération des composantes connexes. Pour cela, il va nous falloir stocker une instance de la classe BreadthFirstSearch du graphe dont on veut connaître sa connexité, ainsi qu'un tableau qui va stocker les composantes connexes.

On crée donc un attribut ***graphBFS*** de classe BreadthFirstSearch et un attribut ***cc*** qui est un tableau d'entier (dont les indices correspondent aux sommets du graphe). Notre classe ainsi constituée va intégrer un constructeur prenant comme unique paramètre un graphe.

Deux méthodes appartiennent à la classe Connectivity :

- ***public boolean isConnected()***
- ***public int[] getConnectedComponent()***

La première méthode *isConnected()* renvoie *true* si le graphe est connexe ou *false* sinon, c'est-à-dire dès qu'un sommet est vert après avoir fait un parcours en largeur à partir du sommet 1.

L’algorithme de récupération des composantes connexes est établi par la deuxième méthode *getConnectedComponent()*. Le fonctionnement de cette méthode est le suivant :

- si le graphe est connexe, alors on remplit le tableau *cc* de 1
- sinon tant qu’il reste des cases de *cc* à 0, on refait un parcours en largeur à partir du sommet actuel et si un sommet est rouge et non encore initialisé dans *cc*, on initialise sa valeur avec celle du sommet actuel. On incrémente le sommet actuel.
- on retourne *cc*.