Université de Poitiers

Rapport du projet de Mathématiques pour l'informatique

Algorithmes de Parcours en largeur et de Détermination des composantes connexes d'un graphe simple

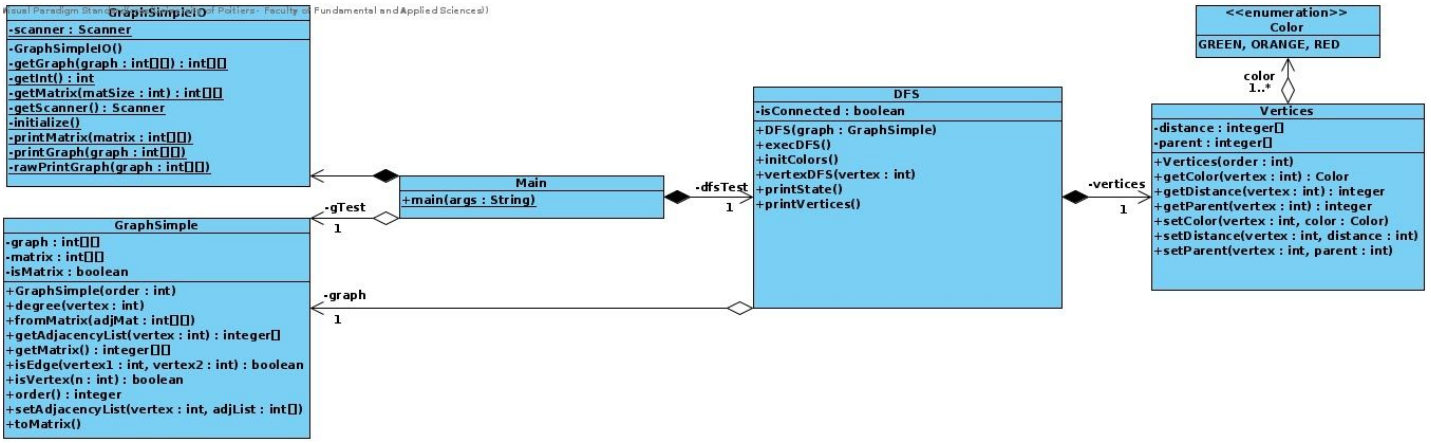
Projet de 3e année de Licence Informatique, 2020-202

Auteurs: Amandine Fradet, Florian Legendre

Table des matières:

Exercice 1 : L'algorithme de parcours en largeur

Le diagramme de classe



Explications

Color

Type énuméré servant à encoder les trois informations suivantes: non traité (couleur verte ou "Green"), découvert (couleur orange ou "Orange"), traité (couleur rouge ou "Red"). Un sommet découvert est un sommet qui était le voisin d'un sommet en cours de traitement mais qui pourrait avoir d'autres voisins non découverts. Il doit être traité pour s'assurer du fait qu'il n'a pas d'autres voisins non découverts.

DFS

DFS signifie en anglais "Depth First Search", ie. parcours en largeur. On conçoit ici un algorithme comme un objet logique pouvant être manipulé comme n'importe quel autre objet qui serait réel. Nous avons par exemple la méthode printState() qui permet d'afficher l'état d'un graphe sur lequel est exécuté l'algorithme.

L'objet instanciable DFS permet donc d'exécuter l'algorithme de parcours en largeur sur un graphe passé en paramètre lors de la construction de l'objet. Nous pourrions ajouter une méthode "setGraph()" permettant ainsi à cet objet de s'exécuter sur d'autres graphes mais l'usage que nous avons ici permet de simplement laisser ce constructeur.

La méthode vertexDFS() exécute l'algorithme de parcours en largeur sur le sommet passé en paramètre. Cependant, si le graphe n'est pas connexe certains sommets n'auront pas été explorés. La méthode execDFS() permet de pallier ce problème en relançant la méthode vertexDFS() dès qu'elle rencontre un sommet qui n'a pas été exploré par un premier passage.

L'attribut "isConnected" est un attribut utilisé par l'exercice 2 de ce TP3. Il permet de déterminer si un graphe est connexe ou non. En effet, si on doit relancer une seconde fois la méthode vertexDFS() alors c'est que le premier passage n'a pas suffit à traiter tous les sommets du graphe et donc il n'était pas connexe. La valeur du booléen "isConnected" dépend donc de la méthode execDFS().

GraphSimpleIO

La classe GraphSimpleIO gère les lectures de graphes dans des fichiers et la génération de graphes à partir de cette lecture. Elle n'est pas instantiable et toutes les méthodes sont statiques. La classe est donc conçue comme une trousse à outils sur les lectures/affichages de graphes simples pour l'ensemble du projet. Elle est notamment utilisée dans la classe Main et la classe GraphSimple.

GraphSimple

Cette classe modélise l'objet mathématique "graphe simple". Un graphe pouvant être représenté comme un tableau de listes d'adjacences (cf. l'attribut "graph" dans le diagramme) ou comme une matrice d'adjacence (cf. l'attribut "matrix" dans le diagramme) notre classe dispose des deux attributs. L'attribut booléen "isMatrix" sert à savoir si la représentation par matrice d'adjacence existe ou non. Un objet graphSimple a toujours au minimum une représentation par tableau de listes d'adjacence. L'ordre passé en paramètre du constructeur permet d'initialiser les différents attributs qui sont des tableaux .

Main

La classe qui permet de tester les fonctions du projet (notamment l'algorithme de parcours en largeur et de détermination des composantes connexes).

Dans un terminal de commande (type Bash) on peut alors utiliser le programme comme suit:

- Pour lancer l'algorithme DFS en lisant un fichier contenant un tableau de listes d'adjacence: java Main < chemin/vers/graphes/graph-000.alists
- Pour lancer l'algorithme DFS en lisant un fichier contenant une matrice d'adjacence; java Main -m < chemin/vers/graphes/graph-000.amatrix

Vertices

Classe instanciable servant de structure de données. Elle contient, pour chaque sommet du graphe, des informations nécessaires à l'exécution de l'algorithme de parcours en largeur.

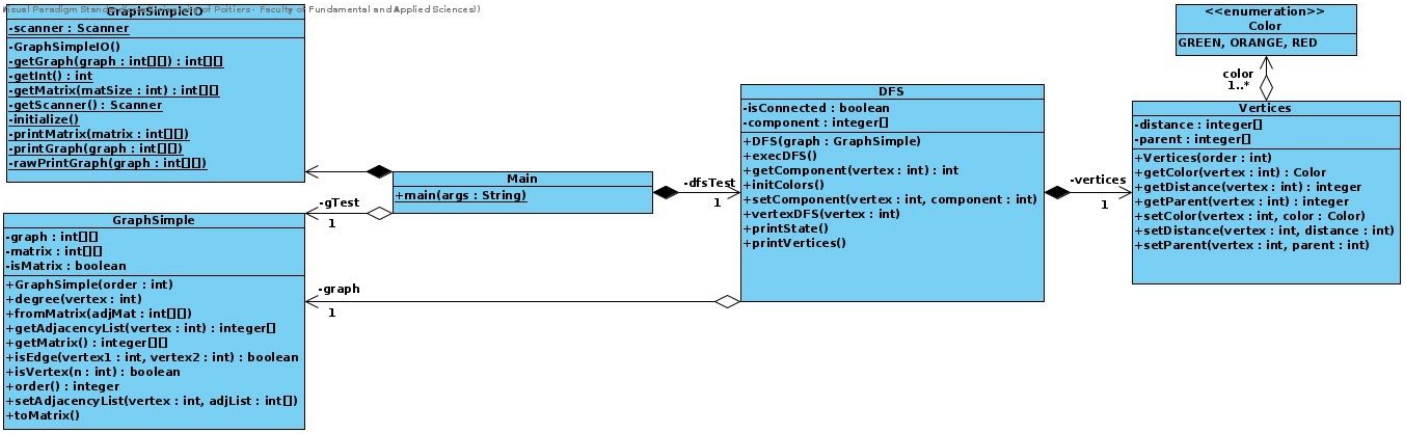
Ces informations sont:

- La couleur de chaque sommet (ie. non traité / découvert / traité)
- Le parent permettant de remonter d'un pas vers la racine
- La distance du sommet à la racine

Chacune de ces informations ont un accesseur/mutateur associé.

Exercice 3: Détermination des composantes connexes par l'algorithme de parcours en largeur

Le diagramme de classe



Explications

L'implémentation d'une solution pour déterminer les composantes connexes tient complètement dans la classe DFS et se base, ici, sur l'algorithme de parcours en largeur.

En effet, on ajoute un attribut "component" qui est un tableau d'entiers. Chaque indice du tableau correspond à un sommet du graphe. La valeur contenue à cet indice indique, pour chaque sommet du graphe, un représentant de la composante connexe à laquelle il appartient.

Par exemple, l'indice 4 (le sommet 5 donc) peut contenir la valeur 2. Cela signifie que le sommet 5 appartient à la même composante connexe[1] que le sommet 2.

On remarque que lors d'un passage par la méthode vertexDFS(), tous les sommets sont traités (donc "Red") si on peut tous les rejoindre depuis la racine choisie. Sinon certains restent verts. À tous les sommets traités lors d'un passage par vertexDFS() on peut donc associer la valeur de la racine qui est alors le représentant de sa composante connexe.

Quand on refait un passage par vertexDFS(), on repart alors d'une nouvelle racine et les sommets qui seront traités par ce nouveau passage appartiennent à la composante connexe de cette nouvelle racine. On peut donc leur associer la valeur de cette nouvelle racine pour désigner leur composante connexe. Etc.

[1] On se base ici sur le fait qu'une composante connexe est une classe d'équivalence, un sommet est alors un représentant de cette classe