

I Algorithmes sur les graphes

1. Changer de représentation

On peut implémenter un graphe par son dictionnaire d'adjacence ou sa matrice d'adjacence par exemple. Pour un graphe (non pondéré) de n sommets, la matrice d'adjacence (que l'on représentera par une liste de n listes de longueurs n) est la matrice pour laquelle $m_{i,j} = 1$ si une arête joint le sommet i et le sommet j (dans ce sens) et $m_{i,j} = 0$ si une telle arête n'existe pas.

On supposera que les sommets du graphe sont les entiers de 0 à $n - 1$.

1. Écrire une fonction `matrice(d:dict)->list` qui prend en argument le dictionnaire d'adjacence du graphe et renvoie sa matrice d'adjacence.
2. Écrire une fonction `Dico(M:list)->dict` qui prend en argument la matrice d'adjacence d'un graphe et renvoie son dictionnaire d'adjacence.

2. Orienté ou non

1. Écrire une fonction `orienté(d:dict)->bool` qui prend en argument le dictionnaire d'adjacence d'un graphe et qui renvoie `True` ou `False` selon que le graphe est orienté ou non.
2. Évaluer la complexité de cette fonction en fonction du nombre n de sommets du graphe.

3. Composante fortement connexe

Une composante fortement connexe d'un graphe est un ensemble (maximal pour l'inclusion) de sommets du graphe pour lesquels il est possible de trouver un chemin sur le graphe permettant de relier tous les couples de sommets, dans un sens et dans l'autre (cette notion est bien sûr plus intéressante sur un graphe orienté).

1. Écrire une fonction `accessibles(i:int,d:dict)->list` qui prend en argument le numéro i d'un sommet et le dictionnaire d'adjacence du graphe et qui renvoie la liste des sommets que l'on peut rejoindre depuis i (pas forcément en une seule étape).
2. Écrire une fonction `transpose(d:dict)->dict` qui prend en argument le dictionnaire d'adjacence d'un graphe et qui renvoie le dictionnaire d'adjacence du graphe transposé : il s'agit du graphe ayant les mêmes sommets mais dont les arêtes sont inversées (si une arête va de i à j sur le graphe initial, une arête va de j à i sur le graphe transposé).
3. À l'aide des deux fonctions précédentes, écrire une fonction `CFC(d:dict)->list` qui prend en argument le dictionnaire d'adjacence d'un graphe et qui renvoie la liste de ses composantes fortement connexes ; chaque composante fortement connexe sera elle-même représentée par la liste de ses sommets. *La CFC qui contient i est l'intersection de la liste des sommets accessibles depuis i et de la liste des sommets depuis lesquels on peut atteindre i , donc accessibles depuis i sur le graphe transposé.*

4. Détecter un cycle

Un graphe possède un cycle à partir d'un sommet i s'il est possible de partir du sommet i et, en parcourant le graphe, de revenir au sommet i .

1. Cas d'un graphe orienté : dans un graphe orienté, si une arête peut être parcourue dans les deux sens, on considère que l'on obtient un cycle (qui ne comporte que deux sommets).
En utilisant un parcours en profondeur, écrire une fonction `cycle(i:int,d:dict)->bool` qui prend en argument le numéro d'un sommet i et le dictionnaire d'adjacence du graphe et qui renvoie `True` ou `False` selon qu'il existe un cycle à partir du sommet i .
2. Cas d'un graphe non orienté : dans un graphe non orienté, un cycle ne doit pas emprunter deux fois la même arête (il comporte donc au moins trois sommets).

Modifier votre code précédent pour qu'il détecte la présence d'un cycle depuis le sommet i . On pourra introduire un dictionnaire qui permet de mémoriser le sommet précédent dans la parcours.