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.

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 ; on supposera que les sommets du graphe sont les entiers de 0 à $n-1$

2. Orienté ou non

1. Écrire une fonction `oriente(d:dict)->bool` qui pend 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. 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 (sans faire marche arrière bien sûr)
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`.

4. Composante fortement connexe

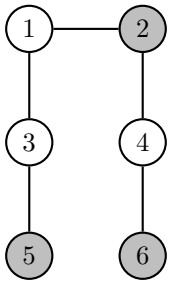
Une composante fortement connexe d'un graphe est un ensemble (maximal pour l'inclusion) de sommets du graphe pour lesquels il est possible de trouver un chemin sur le graphe permettant de relier tous les couples de sommets, dans un sens et dans l'autre (cette notion est bien sûr plus intéressante sur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

1. Écrire une fonction `accessibles(i:int,d:dict)->list` qui pend 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`.
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.

II Chasse au trésor sur un graphe

On suppose disposer d'un graphe non orienté connexe (toutes les arêtes peuvent être parcourues dans les deux sens et pour tout couple de sommets, il existe au moins un chemin permettant de les joindre) représenté par son dictionnaire d'adjacence. On suppose également avoir placé des pièces sur certains sommets du graphe : un dictionnaire `pieces` possède comme clés les sommets du graphe et la valeur associé est `True` si des pièces sont placées sur le sommet, `False` sinon.
On souhaite trouver un chemin permettant de récupérer toutes les pièces du graphe, en essayant de minimiser autant que possible le chemin parcouru. Pour cela on adopte une stratégie gloutonne qui consiste à aller chercher en premier les pièces placées sur le sommet le plus proche de notre position actuelle.

1. Appliquer la stratégie décrite sur le graphe suivant, en partant du sommet 1 ; obtient-on un chemin de longueur minimale ?
   Les pièces sont placées sur les sommets grisés.



2. On commence par écrire une fonction `suivant(s:int,G:dict,pieces:dict)->list` qui prend en argument le numéro d'un sommet `s`, le dictionnaire `G` du graphe et le dictionnaire `pieces` et qui renvoie la liste des sommets par lesquels il faut passer pour récupérer la pièces suivante.

a) Pourquoi le parcours en largeur est-il plus adapté que le parcours en profondeur pour cette démarche gloutonne ?

b) Écrire la fonction `suivant(s:int,G:dict,T:dict)->int`.

3. En déduire une fonction `recolte(s:int,G:dict,pieces:dict)->list` qui renvoie le trajet (représenté par une liste de sommets) à effectuer pour ramasser toutes les pièces ; on pourra commencer par déterminer le nombre de sommets sur lesquels sont posées des pièces.