## PYTHON. **Distance à un sommet et accessibilité dans un graphe**

Un graphe $G$ est ici codé en PYTHON par le dictionnaire des listes d'adjacence : autrement dit, l'ensemble des clés est l'ensemble $S$ des $n$ sommets de $G$ et et pour tout $x \in S$, la valeur est la liste des $y \in S$ tels que $x \to y$ dans $G$ (c'est-à-dire $y$ successeur de $x$).

**La distance $d(x)$ de $s$ à $x$ est la longueur (nombre d'arêtes) minimale des chemins reliant $s$ à $x$.**

En particulier, $s$ est le seul sommet dont la distance depuis $s$ vaut 0. On note $m$ le nombre d'arêtes.

On veut écrire une fonction de coût $O(n+m)$ qui étant donné un sommet source $s$ renvoie le dictionnaire $D$ des distances de $s$ à tout sommet $x \in S$ : le dictionnaire $D$ est composé des couples $(x, d(x))$, où la clé $x$ décrit l'ensemble des sommets accessibles depuis $s$ dans $G$. Les sommets non accessibles depuis $s$ ne sont pas des clés du dictionnaire.

*Remarque* : Le problème de plus court-chemin dans un graphe valué est plus général consiste à déterminer le coût minimal d'un chemin reliant $s$ à $x$ (le coût d'un chemin est défini comme la somme des valeurs de ses arêtes). Ce problème est plus compliqué.

### Partie I. Parcours en largeur : construction récursive par niveaux

**1)** Écrire une fonction auxiliaire récursive `aux(G,k,D,L)` qui étant donnés

- un graphe codé par le dictionnaire `G`
- un entier $k \geq 1$
- un dictionnaire `D` dont les clés sont les sommets dont la distance de $s$ à $x$ est $< k$
- la liste `L` des sommets $x$ de $G$ dont la distance depuis $s$ vaut exactement $k-1$

effectue les opérations suivantes :

- ajoute à `D` les couples $(x,k)$ correspondant aux sommets $x$ dont la distance depuis $s$ vaut $k$
- renvoie la liste des sommets $x$ de $G$ dont la distance depuis $s$ vaut exactement $k$.

*Remarque* : Le dictionnaire `D` associé à $k=0$ est le dictionnaire `{s:0}`.

**2)** En déduire la fonction `distances(G,s)` qui renvoie le dictionnaire `D`.

**3)** Montrer que la complexité est en $O(n+m)$, où $n$ est le nombre de sommets et $m$ le nombre d'arêtes.

### Partie II. Parcours en largeur : construction par une file d'attente

On suppose qu'une structure de file est implémentée à l'aide des fonctions suivantes :

`creerFile(n)` : crée une file vide de capacité $n$

`estVide(F)` : renvoie `True` si la file `F` est vide

`defiler(F)` : supprime l'élément en début de file

`enfiler(x,F)` : ajoute un élément en fin de file

On procède de la façon suivante :

- on initialise le dictionnaire $D$ par {`s : 0`}

- on ajoute $s$ à une file $F$ initialement vide

- tant que la file $F$ n'est pas vide, on effectue les opérations suivantes :

  - on défile un élément $x$ de la file $F$

  - pour tout successeur $y$ de $x$ non encore été visité (c'est-à-dire qui n'est pas dans le dictionnaire, on ajoute $y$ à la file $F$ et $D$ avec la valeur convenable (à déterminer par le lecteur)

**4)** Écrire une fonction `distances2(G,s)` qui renvoie le dictionnaire `D`.

**5)** (★) Justifier la terminaison de l'algorithme. Donner sans justification des invariants de boucle concernant les propriétés de la distance des sommets appartenant au dictionnaire et à la file.

**Partie III. Accessibilité depuis un sommet**

On veut déterminer la liste `L` des sommets accessibles depuis un sommet fixé $s$

(*remarque* : lorsque le graphe n'est pas orienté, il s'agit de la composante connexe de $s$).

Les algorithmes des deux parties précédentes permettent de l'obtenir facilement, en considérant la liste des clés du dicionnaire : `L = D.keys()`.

On va utiliser un parcours en profondeur soit à l'aide d'une fonction auxiliaire récursive soit à l'aide d'une pile.

On construit un dictionnaire de marquage V dont les clés sont les sommets déjà visités..

**6) En utilisant une fonction auxiliaire récursive**

On utilise une fonction auxiliaire `traite(x)` qui effectue les opérations suivantes :

- si $x$ est déjà dans le dictionnaire `V`, on ne fait rien

- sinon, on ajoute $x$ au dictionnaire `V` en prenant une valeur arbitraire, par exemple `V[x] = 1`

- on effectue des appels récursifs `traite(y)` de tous les successeurs de $x$ dans le graphe

(*remarque* : On peut se contenter de traiter les successeurs qui ne sont pas déjà dans `V`).

Écrire une fonction `acces(G,s)` qui renvoie la liste $L$ des sommets accessibles depuis $s$.

**7) En utilisant une pile**

On procède comme suit :

- initialement, le dictionnaire `V` est `{s:1}` et la pile `P` contient uniquement s

Tant que la pile `P` n'est pas vide :

- on dépile un élément $x$ de la pile `P`

- on ajoute à la pile tous les successeurs de $x$

(*remarque* : on peut se contenter d'ajouter uniquement les successeurs qui ne sont pas déjà dans `V`).

*Remarque* : Les sommets de `V` sont les éléments déjà visités ; le dictionnaire joue ainsi un rôle de marquage des sommets traités. Un sommet peut être ajouté plusieurs fois à la pile (autant qu'il a de prédécesseurs). C'est sa première occurrence de sortie (et non d'entrée) dans la pile qui correspond à sa position dans le parcours au profondeur.

Écrire une fonction `acces2(G,s)` qui renvoie la liste $L$ des sommets accessibles depuis $s$.

**8)** On souhaite désormais construire un dictionnaire `V` de sorte que pour tout sommet accessible $x$ depuis $s$, on puisse construire rapidement un chemin reliant $s$ à $x$.

Il suffit d'attribuer à tout sommet $y$ distinct de $s$ son père, c'est-à-dire le sommet $x$ par lequel $y$ a été visité (comme successeur de $x$). Par convention, on prend `V[s] = s`.

a) En adaptant les fonction du 7) ou du 8), écrire une fonction `peres(G,s)` qui renvoie le dicionnaire des pères `V`. On utilisera des couples de type (*fils,père*) dans la pile ou dans comme arguments de la fonction auxiliaire récursive.

b) Écrire une fonction `chemin(V,x)` qui étant donné le dictionnaire des pères `V` et un sommet accessible $x$ (c'est-à-dire $x$ est une clé de `V`) renvoie une liste codant un chemin reliant $s$ à $x$ (la liste des sommets du chemin).

**Corrigé**

**1)**

```
01      def aux(G,k,D,L) :
02          L_new = []
03          for x in L :
04              for y in G[x] :
05                  if not(y in D) :
06                      D[y] = k ; L_new.append(y)
07          return L_new
```

*Remarque* : Il est essentiel de ne pas modifier `D[y]` si $y$ est déjà dans le dictionnaire, c'est-à-dire ssi $y$ est à une distance de $s$ qui est strictement inférieure à $k$.

**2)**

```
08      def distances(G,s) :
09          D = {s:0} ; L = [0] ; k = 0
10          while L :
11              L = aux(G,k,D,L) ; k = k+1
                  ### on continue tant que la liste L n'est pas vide
12          return D
```

**3)** Chaque sommet $x$ apparaît au plus une fois dans la liste $L$. Donc chaque arête est visité une seule fois.
Le test et les instructions des lignes 05 et 06 ont une complexité $O(1)$.
Le test `while L` de la ligne 10 a lui aussi une complexité $O(1)$.
La boucle est itérée au plus $n$ fois (car à chaque étape, on traite au moins un nouveau sommet).
Donc la complexité est en $O(n+m)$.

**4)**

```
def distances2(G,s) :
    D = {s:0} ; F = creerFile(len(G)) ; enfiler(s,F)
    while not(estVide(F)) :
        x = defiler(F)
```

```
            for y in G[x] :
                if not(y in D) :
                    enfiler(y,F) ; D[y] = D[x] + 1
        return D
```

**5)** Chaque sommet n'est ajouté à la file au plus une fois et un sommet est supprimé à chaque étape. Donc l'algorithme termine (en au plus $n$ étapes, où $n$ est le nombre de sommets).

*Invariants de boucle* :

- Si on note $x_0, ..., x_{p-1}$ les éléments de $F$, on a $d(x_0) \leq d(x_1) \leq ... \leq d(x_{p-1}) \leq 1 + d(x_0)$
- tout sommet $x$ présent dans le dictionnaire est passé par la file et s'il est sorti de la file, on a $d(x) \leq d(x_0)$

De ce fait, les sommets $x$ sont sortis de la file par valeur croissante de $d(x)$.

**6)**

```
def acces(G,s) :
    def traite(x) :
        if not (x in V) :
            V[x] = 1
            for y in G[x] : traite(y)
    V = {} ; traite(s)
    return V.keys()
```

**7)**

```
def acces2(G,s) :
    P = [s] ; V = {}
    while P :
        x = P.pop() ;
        if not (x in V) :
            V[x] = 1
            for y in G[x] :  pile.append(y)
    return V.keys()
```

**8)** a) Avec une pile :

```
def acces2(G,s) :
    P = [(s,s)] ; V = {}
    while P :
        (x,z) = P.pop()    ### x est un successeur de y
        if not (x in V) :    ### on séléctionne la première arête * → x
            V[x] = z
            for y in G[x] :  pile.append((y,x))
    return V.keys()
```

En récursif :

```
def acces(G,s) :
    def traite(x,z) :     # on traite x en tant que fils de z
        if not (x in V) :
            V[x] = z
            for y in G[x] : traite(y,x)
    V = {} ; traite(s,s)
```

```
        return V.keys()
```

**8)** b)

```
    def chemin(V,x) :
        L = [x] ; y = x
        while V[y] != y :      ###   cela revient à y != s
            y = V[y] ; L.append(y)
        return reversed(L)
```

avec

```
    def reversed(L) :
        return([L[k] for k in range(len(L)-1,-1,-1)])
```