Informatique — MPSI/PCSI

Correction du TP n° 11

Exercice 11.1

1. Pour ajouter un sommer à un graphe, la seule précaution à prendre est de ne pas l'écraser s'il existe déjà, notamment sa liste de sommets adjacents. Le code suivant convient.

```
def sommet(G,s):
    if s not in G:
        G[s]={"adj":[]}
```

2. Pour initialiser un graphe avec tous ses sommets contenus dans une séquence `S`, il suffit de d'appeler pour chaque sommet `s` de `S` la fonction `sommet`. Le code suivant convient.

```
def sommets(S):
    G={}
    for s in S:
        sommet(G,s)
    return(G)
```

Notant qu'il est possible de définir un dictionnaire par compréhension, on peut le simplifier comme :

```
def sommets(S):
    return({s:{"adj":[]} for s in S})
```

3. Pour définir une arête (s,t), il suffit d'ajouter le sommet t aux sommets adjacents de s et le sommet s aux sommets adjacents de t. La seule précaution à prendre est de ne pas ajouter un sommet s'il est déjà dans la liste des sommets adjacents. En tenant compte de la réciprocité, il vient le code suivant.

```
def arete(G,s,t):
    if t not in G[s]["adj"]:
        G[s]["adj"].append(t)
    if s not in G[t]["adj"]:
        G[t]["adj"].append(s)
```

ou, pour éviter toute répétition et donc tout « copier-coller » avec risque d'erreur sur l'interversion des variables `s` et `t` :

```
def arete(G,s,t):
    for s,t in (s,t),(t,s):
        if t not in G[s]["adj"]:
            G[s]["adj"].append(t)
```

4. Pour compléter le dictionnaire avec les sommets adjacents correspondant à une liste d'arêtes donnée `A`, il suffit d'appeler la fonction `arete` avec chaque couple `(s,t)` de `A`. La code suivant convient.

```
def aretes(G,A):
    for s,t in A:
        arete(G,s,t)
```

5. On peut évaluer le degré d'un sommet en comptant le nombre de fois qu'il intervient comme sommet adjacent. En faisant un parcours des dictionnaires associés à chaque sommet, il vient le code suivant.

```
def degre(G,s):
    d=0
    for v in G.values():
        if s in v["adj"]:
            d+=1
    return(d)
```

6. Pour ajouter le degré de chaque sommet, il serait possible d'utiliser la fonction `degre` mais la complexité temporelle serait alors, en $O(n^3)$, avec n le nombre de sommets, ce qui est absurde. Il est donc nécessaire de réduire le nombre de parcours. En se limitant au parcours de tous les sommets adjacents (`for s in v["adj"]`) de chaque sommet (`for v in G.values()`), on a une complexité en $O(n^2)$. Reste alors le moyen de calculer simultanément tous les degrés. En notant qu'il est possible de modifier une valeur d'un dictionnaire, on initialise pour chaque sommet la valeur associée à `"d"` à 0 puis on l'incrémente à chaque fois que le sommet est trouvé dans la liste des sommets adjacents. C'est ce que fait le code suivant.

```
def degres(G):
    for s in G:
        G[s]["d"]=0
    for v in G.values():
        for s in v["adj"]:
            G[s]["d"]+=1
```

Exercice 11.2

1. Pour initialiser le dictionnaire de parcours, on commence par créer un copie profonde du dictionnaire du graphe afin de ne pas lier les dictionnaires de chaque sommet (d'où le `G[s].copy()`). On parcourt ensuite les dictionnaires de chaque sommet pour ajouter une étiquette `"vu"`, avec une valeur initialisée à 0. Le code suivant convient.

```
def init_parcours(G):
    # copie « profonde » du dictionnaire
    P = {s:G[s].copy() for s in G}
    # initialisation des sommets comme non marqués
    for s in P:
        P[s]["vu"]=0
    return(P)
```

2. On traduit simplement et directement l'algorithme donné en cours. On ajoute seulement un affichage en fin de boucle pour voir l'état de la file (le `print(F)`). Le code suivant convient.

```
def exploration_composante(G,r):
    G[r]["vu"]=1
    E = [r]
    F = deque([r])
    while len(F)!=0:
        v = F.popleft()
        for s in G[v]["adj"]:
            if G[s]["vu"]==0:
                G[s]["vu"]=1
                E.append(s)
                F.append(s)
        print(F)
    return(E)
```

3. Pour explorer toute la composante connexe de A du graphe G, connu par son dictionnaire `G`, on fait simplement attention à partir du dictionnaire de parcours `init_parcours(G)` pour avoir l'étiquette `"vu"` initialisée à 0. Le code suivant exécuté dans le *shell* suffit.

```
>>> exploration_composante(init_parcours(G),"A")
deque(['B', 'G'])
deque(['G', 'C', 'D'])
deque(['C', 'D'])
deque(['D'])
deque(['F'])
deque([])
['A', 'B', 'G', 'C', 'D', 'F']
```

4. Là encore, on traduit simplement et directement l'algorithme donné en cours en faisant simplement attention à partir du dictionnaire de parcours. Le code suivant convient.

```
def parcours_largeur(G):
    P = init_parcours(G)
    E=[]
    # on explore les composantes non marquées
    for s in P:
        if P[s]["vu"]==0:
            E+=exploration_composante(P,s)
    return(E)
```

5. Le code suivant suffit et permet d'afficher dans le *shell*

```
>>> parcours_largeur(G)
deque(['B', 'G'])
deque(['G', 'C', 'D'])
deque(['C', 'D'])
deque(['D'])
deque(['F'])
deque([])
deque(['H'])
deque([])
['A', 'B', 'G', 'C', 'D', 'F', 'E', 'H']
```

6. Pour initialiser les distances de tous les sommets, et par anticipation le sommet parent, on modifie la fonction `init_parcours` comme :

```
def init_parcours(G):
    # copie « profonde » du dictionnaire
    P = {s:G[s].copy() for s in G}
    # initialisation
    for s in P:
        P[s]["vu"]=0              # étiquette marqué/vu
        P[s]["d"]=float('inf')    # distance
        P[s]["p"]=None            # parent
    return(P)
```

7. Pour définir la fonction `distance`, il s'agît d'adapter la fonction `exploration_composante` pour intégrer une mesure de distance, initialisée à 0 pour le sommet `r` puis que l'on met à jour en même temps que l'étiquette `"vu"` en rajoutant 1 par rapport à la distance du sommet défilé (ici notée `d`). Enfin, on arrête l'exploration une fois le sommet `a` atteint. La code suivant convient.

```
def distance(G,r,a):
    P = init_parcours(G)
    P[r]["vu"]=1
    P[r]["d"]=0
    F = deque([r])
    while len(F)!=0 and P[a]["vu"]==0:
        v = F.popleft()
        d = P[v]["d"]
        for s in P[v]["adj"]:
            if P[s]["vu"]==0:
                P[s]["vu"]=1
                P[s]["d"]=d+1
                F.append(s)
        # print(F,[(s,P[s]["d"]) for s in P])
    return(P[a]["d"])
```

   La ligne commentée permet d'afficher l'état de la file et de l'étiquette de distance en cours de parcours.

8. Pour définir la fonction `chemin`, il s'agît d'adapter la fonction `distance` en remplaçant la mise à jour de la distance par le sommet parent. Ainsi, une fois atteint le sommet d'arrivée `a`, il est ensuite nécessaire de remonter la liste des parents jusqu'à trouver le sommet de départ `r` de parent nul (`None`). Comme la liste des sommets parcourus est formée en sens inverse, on renvoie son inverse `C[::-1]`. Le code suivant convient.

```
def chemin(G,r,a):
    P = init_parcours(G)
    P[r]["vu"]=1
    P[r]["p"]=None
    F = deque([r])
    while len(F)!=0 and P[a]["vu"]==0:
        v = F.popleft()
        for s in P[v]["adj"]:
            if P[s]["vu"]==0:
                P[s]["vu"]=1
                P[s]["p"]=v
                F.append(s)
    C = [a]
    p = P[a]["p"]
    while p is not None:
        C.append(p)
        p = P[p]["p"]
    return(C[::-1])
```

* *
*