Devoir Surveillé n° 1 : corrigé

MP Lycée Camille Jullian

22 octobre 2021

I. Modélisation de graphes et algorithmes élémentaires.

Question 1

Pour le premier graphe proposé, $l = [5, [[0,1],[0,2],[0,3],[1,2],[2,3]]]$ (il y a bien sûr plein d'autres possibilités équivalentes pour la liste des arêtes). Pour le deuxième graphe,
$l = [5, [[0,1],[1,2],[2,3],[3,4],[3,1],[4,2]]]$ convient.

Question 2

Puisque l'emploi de la fonction len est autorisé, on aurait tort de se priver, il faut juste faire attention à bien renvoyer la longueur de la liste des arêtes :

```
def taille(g) :
    return len(g[1])
```

Question 3

On va simplement parcourir la liste des arêtes en incrémentant un compteur à chaque fois qu'on croise une arête dont le premier sommet ou le deuxième est le sommet i. On précède ce parcours d'un test pour vérifier que le numéro du sommet est inférieur ou égal à $n-1$.

```
def degre(g,i) :
    if i>g[0]-1 :
        return('Numéro de sommet invalide')
    a=0
    for j in g[1] :
        if j[0]==i or j[1]==i :
            a+=1
    return a
```

Question 4

On va utiliser le même type de structure que pour la question précédente, en stockant cette fois dans une liste les numéros des sommets voisins (on est obligés de séparer plus clairement le cas où c'est le premier élément de l'arête qui est égal à i, du cas où il s'agit du deuxième élément).

```
def voisins(g,i) :
    if i>g[0]-1 :
        return('Numéro de sommet invalide')
    l=[]
    for j in g[1] :
        if j[0]==i :
            l.append(j[1])
        elif j[1]==i :
            l.append(j[0])
    return l
```

On effectue un parcours complet de la liste d'arêtes, avec un nombre fixe d'opérations à chaque étape (deux tests au maximum, et deux append au maximum), donc la complexité de notre programme est en $O(m)$, où m est la longueur de la liste d'arêtes.

Question 5

Le plus simple est de programmer un calcul classique de maximum en testant les degrés des sommets l'un après l'autre (on va bien sûr réutiliser la fonction programmée ci-dessus).

```
def degremax(g) :
    m=0
    for i in range(g[0]) :
        a=degre(g,i)
        if a>m :
            m=a
    return m
```

Niveau complexité, on effectue une boucle de n étapes (une pour chaque sommet) donc chacune nécessaite $O(m)$ calculs pour calculer le degré (cf plus haut, l'introduction de la variable a permet d'éviter de calculer une deuxième fois le degré si on a besoin de remplacer la valeur de m), soit une complexité en $O(nm)$. Ce n'est pas franchement optimal : on pourrait par exemple créer une liste à n éléments contenant initialement des 0, parcourir une seule fois la liste d'arêtes en incrémentant les valeurs de la liste correspondant aux deux extrémités de chaque arête, puis calculer l'indice du maximum de cette liste. On aurait ainsi un $O(m)$ pour le parcours de la liste d'arêtes, puis un $O(n)$ pour la recherche du max dans la liste, soit au total un $O(n+m)$, ce qui est nettement meilleur.

Question 6

Encore un programme où on va se contenter de parcourir la liste des arêtes et tester une condition. On s'arrêtera à l'aide d'un return dès qu'on croise l'arête recherchée. Si on atteint la fin de la liste d'arêtes, c'est que notre arête n'est pas dans le graphe, il est temps de renvoyer False.

```
def lien(g,i,j) :
    if i>g[0]-1 or j>g[0]-1 :
        return('Numéro de sommet invalide')
    for k in g[1] :
        if k==[i,j] or k=[j,i] :
            return True
    return False
```

Question 7

On va bien sûr exploiter le programme précédent pour savoir s'il y a besoind d'ajouter, puis le cas échéant, simplement mettre à jour la liste d'arêtes.

```
def creelien(g,i,j) :
    if i>g[0]-1 or j>g[0]-1 :
        return('Numéro de sommet invalide')
    if not lien(g,i,j) :
        g[1].append([i,j])
```

La complexité est la même que celle de la fonction lien (les quelques opérations ajoutées sont en coût constant), qui nécessite lui-même un parcours de la liste d'arêtes, donc à nouveau du $O(m)$. Notons tout de même qu'il s'agit d'une complexité dans le pire cas ici, puisqu'on peut très bien avoir un programme qui renvoie le résultat dès qu'il croise la première arête (si c'est la bonne !). Difficile d'estimer la complexité moyenne, car cela dépend en fait de la probabilité de trouver l'arête dans la liste (quand on ne la trouve pas, on parcourt systématiquement tout la liste).

II. Partitions d'un ensemble.

Question 8

Dans le cas d'une partition en singletons, le tableau parent contient tout simplement les entiers de 0 à $n-1$ puisque chaque élément est seul dans son sous-groupe donc est nécessairement son propre parent.

```
def creepartitionsingletons(n) :
    return [i for i in range(n)]
```

Question 9

Intuitivement, il s'agit de « remonter » à partir de i jusqu'au représentant du sous-ensemble, en calculant le parent de i, puis le parent de son parent etc, jusqu'à aboutir à un élément qui soit son propre parent (et qui sera donc le représentant du sous-ensemble). Dans le programme qui suit, on recopie volontairement la valeur de i dans une variable auxiliaire pour avoir un programme qui renvoie le représentant sans modifier la valeur de la variable i (ce sera essentiel pour que le programme de la question suivante fonctionne correctement).

```
def representant(parent,i) :
    j=i
    while parent[j] !=j :
        j=parent[j]
    return j
```

Question 10

C'est une question très facile à faire en utilisant le programme de la question 9, il suffit d'appliquer l'algorithme proposé. On va programmer une procédure plutôt qu'une fonction ici, il n'y a rien d'intelligent à renvoyer (imprécision de l'énoncé).

```
def fusion(parent,i,j) :
    p=representant(parent,i)
    q=representant(parent,j)
    parent[p]=q
```

Question 11

Cette question nécessite un peu plus de réflexion que les précédentes. L'idée la plus simple est de créer une liste pour chaque représentant, puis de parcourir l'ensemble des entiers de 0 à $n-1$, et d'ajouter chacun de ses entiers à la liste de son représentant. Pour simplifier l'écriture du programme, on va créer n listes initialement vides, et ajouter chaque entier à la liste de son représentant (en calculant ce dernier à l'aide de la fonction écrite plus haut). À la fin de cette étape, on aura beaucoup de listes qui seront restées vides (toutes celles qui ne correspondent pas à un représentant), on se contentera d'afficher la liste des listes non vides parmi celles obtenues.

```
def partition(parent) :
    n=len(parent)
    l=[[] for i in range(n)]
    for i in range(n) :
        j=representant(parent,i)
        l[j].append(i)
    return [k for k in l if k !=[]]
```

Ce n'était pas demandé par l'énoncé mais on peut remarquer que cet algorithme est loin d'être optimal dans la mesure où on va fréquemment remonter plusieurs fois la même chaine de parentalité pour des éléments appartenant au même sous-ensemble. Ainsi, si 1 a pour parent 3, qui lui-même

a pour parent 7 (représentant du sous-ensemble), quand on va calculer representant(parent,1) à la ligne 5 du programme, on va passer par l'entier 3 qu'on ferait mieux d'insérer tout de suite dans la même liste que 1 plutôt que de refaire un peu plus tard le calcul redondant de representant(parent,3). Il existe des solutions pour éviter ce problème, mais qui compliquent assez nettement l'écriture du programme. Bien sûr, on a également une complexité spatiale largement améliorable puisqu'on a créé beaucoup plus de listes que nécessaire en début de programme.

III. Recherche d'une coupe minimale.

Question 12

On va évidemment suivre l'algorithme proposé par l'énoncé, mais qui n'est pas si facile que ça à mettre en oeuvre. Pour gérer au mieux les arêtes marquées, on va créer une variable m initialement égale à la taille de g (nombre d'arêtes disponibles), et qu'on diminuera à chaque fois qu'on marque une arête, pour qu'elle représente toujours le nombre d'arêtes non marquées disponibles. De plus, à chaque fois qu'on marquera une arête, on l'échangera dans la liste des arêtes de g avec l'arête placée en position m dans la liste, de façon à ce que les m arêtes non encore marquées soient toujours les m premières de la liste. On exploitera bien sûr les programmes précédemment écrits, notamment le programme partition de la question 11 qui nous permettra de savoir si notre partition contient ou non plus de deux sous-ensembles.

```
from random import randint
def coupeminrandom(g) :
    n=g[0]
    P=creepartitionsingletons(n)
    m=taille(g)
    while m>0 and len(partition(P))>2 :
        p=randint(0,m-1)
        i,j=g[1][p]
        fusion(P,i,j)
        m=m-1
        g[1][p],g[1][m]=g[1][m],g[1][p]
    a=partition(P)
    if len(a)==2 :
        return a
    l=[]
    for i in range(len(a)-1) :
        l=l+a[i]
    return [l,a[-1]]
```

Question 13

Cette dernière question n'est en fait pas très compliquée si on analyse bien ce qu'il y a à faire. Même pas besoin d'utiliser le programme de la question 11, la donnée du tableau parent est largement suffisante. Il suffit en effet de compter les arêtes $[i, j]$ du graphe pour lesquelles les deux entiers i et j n'appartiennent pas au même sous-ensemble, c'est-à-dire tout simplement pour lesquelles i et j n'ont pas le même représentant dans le tableau parent.

```
def lienscoupe(g,parent) :
    n=0
    for a in g[1] :
    i,j=a
    if representant(parent,i) !=representant(parent,j) :
        n+=1
    return n
```