Conversions (II-3)

La matrice d'adjacence peut-être créée sous forme de tableau `numpy` (auquel cas l'élément de la ligne i, colonne j s'obtient directement par $M[i,j]$), ou bien sous forme de liste de listes (**mais** dans ce cas la syntaxe $M[i,j]$ n'est pas admise, il faut passer par $M[i][j]$). J'utiliserai ici des tableaux `numpy`, en précisant le type `int` pour les éléments des matrices, afin d'économiser la mémoire et d'éviter l'affichage de points décimaux inutiles.
Notez que les dimensions du tableau à créer doivent être fournies sous forme de **couple**, par exemple `np.zeros((n,n))` (avec deux paires de parenthèses car `np.zeros` reçoit **un couple** pour unique paramètre) !

Partant de la liste de couples A, je peux utiliser la syntaxe compacte `for (i,j) in A` ou bien utiliser un indice k pour parcourir les $A[k]$...

Autre remarque importante : pour créer la liste des listes d'adjacence, je commence par créer une liste de n listes vides, auxquelles j'ajouterai les sommets adjacents le moment venu. Pour cela **il faut proscrire** la commande `n*[[ ]]`, qui copie n liens **vers la même liste !** Toute modification de l'une des listes se répercute alors sur toutes les autres... Préférez donc par exemple `[[ ] for k in range(n)]`, où la boucle crée bien n listes différentes. On peut aussi enchaîner des `append` à l'aide d'une boucle `for`...

Là encore la syntaxe `for (i,j) in A` est efficace : il suffit d'ajouter j à la liste d'adjacence $L[i]$ pour tout couple (i,j) de A.

```
def GtoM(n,A):
    M = np.zeros((n, n), int)
    for (i,j) in A:
        M[i,j] = 1
    return M
```

```
def GtoL(n,A):
    L = [[] for k in range(n)]
    for (i,j) in A:
        L[i].append(j)
    return L
```

Même principe pour construire les listes d'adjacence en partant de M, mais cette fois-ci j'utilise une double boucle pour parcourir les éléments de M.

Enfin version mixte pour remplir M à partir de L : j'ai besoin de l'indice i mais les valeurs de j sont les valeurs présentes dans $L[i]$, pas besoin de les numéroter !

```
def MtoL(M):
    n = len(M)
    L = [[] for k in range(n)]
    for i in range(n):
        for j in range(n):
            if M[i,j] == 1:
                L[i].append(j)
    return L
```

```
def LtoM(L):
    n = len(L)
    M = np.zeros((n,n), int)
    for i in range(n):
        for j in L[i]:
            M[i,j] = 1
    return M
```

Parcours en profondeur (III-1-c)

Presque tout est dit dans le texte, la sous-fonction `Visiter` est dans la colonne de gauche, la fin de la fonction `PP` dans celle de droite :

```
def PP(L):
    def Visiter(u):
        nonlocal instant
        c[u] = 'gris'
        d[u] = instant
        instant += 1
        for v in L[u]:
            if c[v] == 'blanc':
                Visiter(v)
        c[u] = 'noir'
        f[u] = instant
        instant += 1
```

```
    n = len(L)
    c = n*['blanc']
    d = n*[0]
    f = n*[0]
    instant = 1
    for u in range(n):
        if c[u] == 'blanc':
            Visiter(u)
    return d, f
```

Tri topologique (III-2)

Légère modification du programme précédent : j'initialise une *pile* `T` à la pile vide et j'empile sur `T` chaque sommet dès qu'il devient noir (à la fin de la fonction `Visiter`). Ainsi il n'est même plus nécessaire de mémoriser la chronologie des tableaux `d` et `f` précédents. Il paraîtrait logique de déclarer `T` comme `nonlocal`, mais ce n'est pas nécessaire car `T` est visible dans la fonction `Visiter` et que ladite fonction ne crée pas de nouvelle variable `T` lors de l'instruction `empiler`.
Je renvoie la liste des sommets dans l'ordre de traitement établi en dépilant tous les éléments.
Noter que l'on peut aussi manipuler sa propre liste, en utilisant éventuellement pour finir la méthode (hors programme) `reverse` : `T.reverse()` modifie `T` en la remplaçant par son *image miroir*.

```
from files import *

def TriTopo(L):
    def Visiter(u):
        c[u] = 'gris'
        for v in L[u]:
            if c[v] == 'blanc':
                Visiter(v)
        c[u] = 'noir'
        empiler(u,T)
    n = len(L)
    c = n*['blanc']
    T = pile_vide()
    for u in range(n):
        if c[u] == 'blanc':
            Visiter(u)
    return [depiler(T) for k in range(n)]
```

Parcours en largeur (IV-3)

Là encore, presque tout est dit dans le texte, il faut tout de même programmer (ou importer) les primitives du type "file d'attente". Pour importer les primitives du module "personnel" `files.py`, `from files import *` doit fonctionner, à condition que le fichier `files.py` soit présent dans le dossier courant et et que le script soit lancé (au moins la première fois) par `Ctrl+F5` au lieu de `F5` pour que Python définisse le dossier courant comme dossier par défaut...

```
from files import *
from numpy import inf

def PL(L, s):
    n = len(L)
    c = n*['blanc']
    d = n*[inf]
    p = n*[-1]
    F = file_vide()
    c[s] = 'gris'
    d[s] = 0
    ajouter(s, F)
    while not est_file_vide(F):
        u = suivant(F)
        for v in L[u]:
            if c[v] == 'blanc':
                c[v] = 'gris'
                ajouter(v, F)
                d[v] = d[u]+1
                p[v] = u
        c[u] = 'noir'
    return d, p
```

Algorithme de Dijkstra (V-3)

Là aussi, presque tout est dit dans le texte, il y a juste à programmer la fonction `Extraire_min` : dans cette première version naïve (et de complexité linéaire), je parcours les sommets présents dans `E` à la recherche de `u_min`, celui (ou l'un de ceux) pour lequel l'estimation stockée dans `e` est minimale ; enfin je supprime ledit `u_min` de la liste `E`, à l'aide de la méthode `remove` (hors programme) proposée par Python pour les listes, **à ne pas confondre avec `pop` ni avec `del`** ! En effet, `E.pop(i)`, tout comme `del E[i]`, supprime **la valeur d'indice** i de la liste `E`, tandis que `E.remove(x)` supprime **la première occurrence de la valeur** x dans la liste `E`.
La sous-fonction `Extraire_min` est dans la colonne de gauche, la fin de la fonction `Dijkstra` dans celle de droite :

```
from numpy import inf

def Dijkstra(L, s):
    def Extraire_min(E):
        u_min = E[0]
        e_min = e[u_min]
        for u in E[1:]:
            if e[u] < e_min:
                e_min = e[u]
                u_min = u
        E.remove(u_min)
        return u_min
```

```
    n = len(L)
    e = n*[inf]
    p = n*[-1]
    E = list(range(n))
    e[s] = 0
    while E != []:
        u = Extraire_min(E)
        for (v,V) in L[u]:
            if e[u]+V < e[v]:
                e[v] = e[u] + V
                p[v] = u
    return e, p
```

Amélioration à l'aide d'un tas (V-4)

On peut programmer `Extraire_min` avec une complexité logarithmique en stockant **dans un tas** (cf. le thème 1) les listes de la forme `[v,e[v]]`, ordonnées selon les valeurs de la *clé* `e[v]`.

Pour cela je crée un premier tas trivial `T` avec à la racine la liste `[s,0]`, suivie de tous les autres sommets du graphe affectés de l'estimation `inf`. Rappelons que les éléments utiles du tas `T` sont ceux d'indice supérieur ou égal à 1, d'où le 0 en tête de la liste `T`, qui ne sert à rien, si ce n'est à garantir la formule simple de détermination de l'indice du père d'un élément du tas.

Il ne reste qu'à reprendre les programmes du thème 1, avec toutefois quelques variantes. En effet, pour pouvoir retrouver dans le tas en temps constant (et non pas linéaire !) le sommet `v` dont je dois diminuer l'estimation, je maintiens à jour une liste `I` des indices dans le tas : `T[I[v]]` sera ainsi la liste `[v,e[v]]`.

Je modifie donc la fonction `Extraire_min`, d'une part pour maintenir la structure de tas, d'autre part pour tenir à jour le tableau `I`.
Par ailleurs, la fonction `Entasser` du thème 1 n'est plus nécessaire (puisque la création du tas est triviale), mais j'utilise une fonction `Diminuer_cle`, qui diminue l'estimation associée à un sommet et le fait remonter dans le tas pour préserver sa structure, le tout avec une complexité logarithmique et sans oublier de tenir à jour le tableau `I` !
Le tas `T` se vide inexorablement (commme l'ensemble `E` de la version précédente), jusqu'à ce qu'il ne contienne plus que le 0 placé en tête pour les raisons exposées ci-dessus !

Le code est sur la page suivante.

```
from numpy import inf

def Diminuer_cle(t, I, v, V):
    i = I[v] #l'indice de v dans le tas t
    t[i][1] = V
    p = i//2 #le père
    while i > 1 and t[i][1] < t[p][1]:
        t[i],t[p] = t[p],t[i]
        I[t[i][0]],I[t[p][0]] = I[t[p][0]],I[t[i][0]]
        i,p = p,p//2

def Extraire_min(t, I):
    u = t[1][0]
    t[1] = t[-1]
    I[t[1][0]] = 1
    t.pop()
    n = len(t)-1
    i = 1
    while i <= n//2:
        if 2*i == n or t[2*i][1] < t[2*i+1][1]:
            f = 2*i
        else:
            f = 2*i+1
        if t[i] > t[f]:
            t[i],t[f] = t[f],t[i]
            I[t[i][0]],I[t[f][0]] = I[t[f][0]],I[t[i][0]]
            i = f
        else:
            break
    return u


def Dijkstra_tas(L, s):
    n = len(L)
    e = n*[inf]
    p = n*[-1]
    I = n*[0]
    e[s] = 0
    T = [0,[s,0]]
    I[s] = 1
    i = 2
    for u in range(n):
        if u != s:
            T.append([u,inf])
            I[u] = i;i += 1
    while T != [0]:
        u = Extraire_min(T,I)
        for (v,V) in L[u]:
            if e[u]+V < e[v]:
                e[v] = e[u]+V
                Diminuer_cle(T, I, v, e[u]+V)
                p[v] = u
    return e, p
```