TP N°06 :
PARCOURS DE GRAPHE : ECHELLE ET SERPENTS

OBJECTIFS DU TP

- Utiliser les piles et les files à partir de listes (et du module collections.deque)
- Notion de parcours d'un graphe

I. CONSTRUCTION DU PLATEAU DE JEU

On représente le plateau de jeu sous la forme d'un graphe orienté G. Les sommets du graphe G correspondent aux n cases du plateau, numérotées de 1 à n. Afin de respecter les notations habituelles, les sommets sont numérotés de 0 à n - 1 (le sommet k correspond donc à la case k + 1). Les arcs (i, j) correspondent aux déplacements autorisés depuis un sommet i (case i + 1) vers un sommet j (case j + 1) : par exemple, sur le plateau de jeu fourni en exemple, il est possible de se déplacer depuis la case n°4 (sommet n°3) vers la case n°5 (sommet n°4) en obtenant le chiffre 1 lors du lancer de dé – l'arc (3, 4) existe donc. On note G_{nu} le graphe obtenu en ignorant la présence des serpents et des échelles.

On considère tout d'abord un plateau constitué de n cases, dépourvu de serpents et d'échelles. Les joueurs jouent avec un dé à $d < n$ faces, numérotées de 1 à d.

Q1. Donner l'ensemble des sommets accessibles à partir du sommet k en fonction du nombre n de cases constituant le plateau et du nombre d de faces du dé utilisé.

En notant S_k l'ensemble des sommets accessibles depuis le sommet k. Les sommets étant numérotés de 0 à n-1 pour un plateau dont les cases sont numérotées de 1 à n :

$$S_k = \{k + i \,/\, k + i \leq n - 1\}_{i \in [\![1,d]\!]}$$

Q2. Écrire une fonction `plateau_nu` prenant comme arguments deux entiers n (représentant le nombre n de cases du plateau) et d (représentant le nombre d de faces du dé utilisé), et retournant le graphe orienté G_{nu} sous la forme d'une liste d'adjacence.

```
def plateau_nu (n:int,d:int)-> [[int]]:
    """
    Renvoie la liste d'adjacence du graphe G_nu correspondant au plateau de n cases avec un dé à d faces

    entrée : n :entier correspondant au nombre de cases sur le plateau
    d : entier correspondant au nombre de faces sur le dé utilisé pour le jeu

    sortie : liste de listes d'entiers : liste d'adjacence du graphe G_nu correspondant au graphe représentant le plateau de jeu de n cases et les cases accessibles avec un dé à d faces.
    ===
```

```
    jeux de tests
    ===
    >>> plateau_nu(3,2)
    [[1, 2], [2], []]

    >>> plateau_nu(4,1)
    [[1], [2], [3], []]

    """
    G_nu=[[] for _ in range (n)]
    for k in range (n+1) :
        for i in range (1,d+1):
            if k+i<n :
                G_nu[k].append(k+i)
    return(G_nu)
```

Q3. Proposer une suite d'instructions permettant d'affecter aux variables `S_exemple` et `E_exemple` des listes représentant, respectivement, les serpents et les échelles du plateau de jeu donné en exemple ci-avant.

```
S_exemple = [[17,13],[52,29],[88,18],[57,40],[62,22],[95,51],[97,79]]
E_exemple = [[8,30],[3,21],[28,84],[80,100],[90,91],[75,86],[58,77]]
```

Q4. Proposer une suite d'instructions permettant de vérifier que la tête de chaque serpent se situe bien plus « haut » que sa queue. On pourra éventuellement afficher un message explicite.

```
serpent_OK = True # par défaut
for s in S :
    if s[0]<s[1]:
        serpent_OK = False
print('Serpents OK ? :' + str(serpent_OK))
```

Q5. Proposer une suite d'instructions permettant de vérifier que le pied de chaque échelle se situe bien plus « bas » que son haut.

```
echelle_OK = True
for e in E :
    if e[0]>e[1]:
        echelle_OK=False
print("Echelles OK ? :"+str(echelle_OK))
```

Q6. Proposer une suite d'instructions permettant de vérifier que les extrémités de deux objets (serpents et/ou échelles) ne coïncident pas.

```
extremite_OK = True
# on place tous les éléments dans une même liste et on vérifie qu'il n'y a pas de doublons dans cette liste
extremites =[]
for ext in S+E :
    extremites.append(ext[0])
    extremites.append(ext[1])

for i in range (len(extremites)) :
    for j in range (i+1,len(extremites)) :

        if extremites[i] == extremites[j] :
            extremite_OK = False

print("extrémités OK ?"+str(extremite_OK))
```

Q7. Expliquer comment la présence d'un serpent ou d'une échelle modifie le graphe correspondant au plateau nu et sa liste d'adjacence.

Lorsqu'il y a une échelle entre une case p et h, tous les arcs partant d'un sommet s et arrivant en p sont remplacés par des arcs de s à h. Dans la liste d'adjacence du graphe, les occurrences du sommets p dans les sous-listes sont remplacées par h.

De même, lorsqu'il y a un serpent entre un sommet t et q, tous les arcs partant d'un sommet s et arrivant en q sont remplacés par un arc de s à q. Dans la liste d'adjacence du graphe, les occurrences du sommets t dans les sous-listes sont remplacées par q.

Q8. Écrire une fonction `plateau` prenant comme arguments :

- un entier n (représentant le nombre n de cases du plateau) ;
- un entier d (représentant le nombre d de faces du dé utilisé) ;
- une liste S (représentant la position des serpents, selon la convention précisée précédemment);
- une liste E (représentant la position des échelles, selon la convention précisée précédemment)

et retournant le graphe orienté G sous forme d'une liste d'adjacence. On pourra utiliser la fonction `plateau_nu` définie précédemment.

```
def plateau (n:int, d:int, S:[[int]],E:[[int]]):
    """
    fonction qui renvoie la liste d'adjacence du graphe G représentant un plateau de jeu avec n cases, des serpents, des échelles et un dé à d faces
```

```
    entrées :
    n :entier correspodnant au nombre de cases sur le plateau
    d : entier correspondant aux nombre de faces sur le dé utilisé pour le jeu
    S : liste de liste correspondant aux serpents du plateau, chaque sous-liste contient deux entiers correspondant aux cases de départ et d'arrivée du serpent
    E: liste de listes correspondant aux échelles du plateau, chaque sous-liste contient deux entiers correspondant aux cases de départ et d'arrivée de l'échelle

    sortie : Liste de listes d'entiers correspondant à la liste d'adjacence du graphe G décrivant le plateau

    ===
    jeux de tests
    ===
    >>> plateau(8,2,[],[[3,6]])
    [[1, 5], [5, 3], [3, 4], [4, 5], [5, 6], [6, 7], [7], []]
    """
    G= plateau_nu(n,d)

    for case in G :
        # ajout des serpents
        for s in S :
            for num_case in range(len(case)):

                if s[0]==case[num_case]+1 :

                        case[num_case] = s[1]-1
        # ajout des échelles
        for e in E :

            for num_case in range(len(case)) :

                if e[0]== case[num_case] +1 :

                    case[num_case] = e[1]-1

    return(G)
```

II. EXPLORATION DU PLATEAU DE JEU

On souhaite déterminer le chemin le plus court (en nombre de coups) entre la case de départ (case n°1) et la case d'arrivée (case n°100). Pour ce faire, on réalise le ***parcours en largeur*** du graphe G. Afin de simuler l'utilisation de files d'attente à l'aide de listes, on commence par créer quelques fonctions utilitaires.

Q9. Écrire une fonction `est_vide` prenant comme argument une liste F (représentant la file d'attente F), et retournant VRAI si la liste est vide et FAUX sinon.

```
def est_vide(F:list)-> bool :
    """
    permet de déterminer si une file F est vide ou non

    entrée : liste F représentant une pile
    sortie : True si la liste est vide, False sinon
    ===
    jeux de tests
    ===
    >>> est_vide([])
    True

    >>> est_vide([1,2,3])
    False
    """
    return(F==[])
```

Q10. Écrire une fonction `enfiler` prenant comme arguments une liste `F` (représentant la file d'attente F) et un objet `x`, et retournant une copie de la file `F` au bout de laquelle l'objet `x` a été enfilé.

```
def enfiler (F:list,x)-> list :
    """
    renvoie une copie de la liste F à laquelle un élément x a été enfilé (ajouté
en tête

    entrée : liste F et un élément x
    sortie une liste (copie de F) à laquelle x est enfilé
    ===
    jeux de tests
    ===
    >>> enfiler([5,4],'bouh')
```

```
    ['bouh', 5, 4]
    >>> enfiler([2,3],1)
    [1, 2, 3]
    """
    return([x]+F[:])
```

Q11. Écrire une fonction **defiler** prenant comme argument une liste F (représentant la file d'attente F), et retournant l'objet **x** situé en tête de la file **F** ainsi qu'une copie de la file **F** dont l'objet **x** situé en tête a été défilé.

```
def defiler (F:list)->(list,any) :
    """
    renvoie une copie de la liste F à laquelle un élément x a été défilé


    entrée : liste F
    sortie une liste (copie de F) à laquelle x est défilé et x l'élément que
l'on a retiré


    ===
    jeux de tests
    ===
    >>> defiler([1,2,34])
    [1, 2],34
    >>> defiler([])
    la liste est vide
    [], None
    """
    if est_vide(F):
        print("la liste est vide")
        return [],None
    x = F[-1]
    return(F[:len(F)-1],x)
```

En utilisant la structure de file et les fonctions précédentes, on réalise un parcours en largeur du graphe G.

Q12. Écrire une fonction **parcours_largeur** prenant comme argument une liste G (représentant le graphe orienté G sous la forme d'une liste d'adjacence), et retournant le nombre minimal de coups nécessaires pour aller de la case n°1 à la dernière case du plateau représenté par le graphe G.

```
def parcours_en_largeur (G:[[int]])->int :
    """
```

```
    Détermine le nombre minimum de coups nécessaire pour gagner au jeu des
serpents et des échelles

    Entrée : G la liste d'adjacence du graphe représentant le plateau de jeu

    sortie : un entier correspondant au nombre de coups minimal à jouer pour
aller de la case n°1 à la dernière.

    === jeux de tests ===

    >>> parcours_en_largeur(plateau_nu(10,4))
    la liste coups donne (pour indication) [0, 1, 1, 1, 1, 2, 2, 2, 2, 3]
    >>> 3
    """
    #initialisation
    n = len(G)
    coups=[-1]*n

    #positionnement sur la première case
    F=[0]
    coups[0]= 0

    while coups[-1] == -1 :
        F,j = defiler(F)

        for v in G[j] :
            if coups[v] == -1 :
                F = enfiler (F,v)
                coups[v]=coups[j]+1
    return(coups[-1])
```

Q13. En s'appuyant sur la structure de la fonction `parcours_largeur`, écrire une fonction `plus_court_chemin` prenant comme argument une liste G (représentant le graphe orienté G sous la forme d'une liste d'adjacence), et retournant un plus court chemin pour aller de la case n°1 à la dernière case du plateau représenté par le graphe G, représenté sous la forme d'une liste d'entiers représentant le numéro des cases visitées.

```
def plus_court_chemin (G:[[int]])->[int]:
    """
```

```
    Détermine le plus court chemin permettant de gagner au jeu des serpents et des échelles
    entrée : G ([[int]]) : liste d'adjacence du graphe représentant le plateau de jeu
    sortie : [int] représentant les case à parcourir pour gagner en le moins de coups possibles
    ===
    Jeux de tests
    ===
    plus_court_chemin(plateau_nu(10,4))
    renvoie : [1, 2, 6]
    """
    # initialisation
    n=len(G)
    coups = [-1]*n
    predecesseur = [None]*n

    #positionnement sur la première case
    F=[0]
    coups[0]=0

    while coups[-1]==-1 :
         F,j = defiler(F)

         for v in G[j] :
            if coups[v]==-1:
                coups[v] = coups[j] + 1
                predecesseur[v] = j
                F = enfiler(F,v)

    #parcours "à l'envers" de la liste des prédecesseurs et création du chemin avec une structure similaire à une file

    chemin = [predecesseur[-1],n-1]

    while chemin[0]!=0 :
        chemin = [predecesseur[chemin[0]]] + chemin
```

```
    #correction des indices car numéro de la case = numéro du sommet + 1
    chemin_corrige = [sommet+1 for sommet in chemin]


    return(chemin_corrige)
```

Q14. Quel est le plus court chemin dans le cas du plateau de jeu proposé en exemple ?

```
plateau_exemple = plateau(100,6,S_exemple,E_exemple)


message = "Un des plus courts chemins pour gagner sur le plateau donné en exemple
est "
message+= str(plus_court_chemin(plateau_exemple))
message+= "en "
message += str(parcours_en_largeur(plateau_exemple))
message += " coups"
print(message)


>>> Un des plus courts chemins pour gagner sur le plateau donné en exemple est
[1, 21, 22, 84, 89, 94, 100]en 6 coups
```

Reprendre les fonction précédentes en utilisant collections.deque permettant de manipuler des files. On pourra notamment utiliser les fonctions `F.appendleft()`.