DST Informatique pour tous
PSI et MP
Durée : 2 heures

15 février

La clarté de la présentation et la qualité de la rédaction font partie de l'évaluation. On veillera à utiliser des noms explicites pour les variables et les fonctions introduites. Le candidat pourra clarifier les programmes par l'ajout de commentaires judicieux si nécéssaire.

Dans tout le sujet on supposera avoir importé la bibliothèque `numpy`, sous la forme :

```
import numpy as np
```

Partie I

La Fédération mondiale du jeu de dames, regroupant soixante-dix-sept pays, tient à jour le classement international et accorde les titres de maître et grand maître international. Elle organise également les championnats mondiaux. La base de données de l'association contient des informations administratives sur les concurrents. Pour simplifier le problème, on considère trois tables : JOUEURS, PARTICIPATION et TOURNOIS.

La table JOUEURS contient les attributs suivants :

— id_J : identifiant d'un individu (entier), clé primaire ;
— nom_J : nom du joueur (chaîne de caractères) ;
— prénom_J : prénom du joueur (chaîne de caractères) ;
— adresse_J : adresse du joueur (chaîne de caractères) ;
— email_J : (chaîne de caractères) ;
— naissance_J : année de naissance (entier)
— nationalité_J
— Points_J : points total du joueur (entier).
— Classement_J (entier) :

La table TOURNOIS contient les attributs suivants :

— id_T : identifiant (entier), clé primaire ;
— nom_du_tournois_T : donnée (caratère) ;
— date_T : date du tournoi (entier) ;
— Lieu_T : lieu et adresse du tournoi
— Pays_T
— vainqueur_T : nom du vainqueur

La table PARTICIPATION contient les attributs suivants :

— id_P : identifiant (entier) de la participation, clé primaire ;
— tournoi_P : identifiant du tournoi (entier id_T) ;
— joueur_P : identifiant du joueur (entier id_J) ;
— classement_P : description de l'état du patient (chaîne de caractères).

JOUEUR	TOURNOI	PARTICIPATION
id_J	id_T	id_P
nom_J	nom_du_tournoi_T	tournoi_P
prenom_J	date_T	joueur_P
adresse_J	Lieu_T	classement_P
email_J	vainqueur_T	
naissance_J	Pays_T	
point_J		
Classement_J		
nationalité_J		

1. Écrire une requête SQL permettant d'extraire le nombre de participants aux tournois en France en 2020.
2. Écrire une requête SQL permettant d'extraire les noms des vainqueurs des tournois en Angleterre par année.
3. Écrire une requête SQL d'extraire le nombre de tournois par pays en 2023.

Partie II : Initialisation

Les premiers programmes de jeu de dames sur 100 cases furent écrits dans le milieu des années 1970.

Parallèlement à l'amélioration des logiciels de jeu de dames, des travaux visent à déterminer l'issue, gagnante, nulle ou perdante, des positions possibles des pièces sur le damier. En 2010, la base de données de toutes les positions à neuf pièces et moins, propres aux fins de partie, a été construite. La résolution complète du jeu de dames international (le jeu à 100 cases), c'est-à-dire le fait de connaître l'issue de toutes les positions possibles n'est, comme aux échecs, pas encore à portée de la technologie.

Le jeu de dames international se joue sur un damier carré divisé en 100 cases égales, alternativement claires et foncées. Le jeu se joue sur les cases foncées du damier. Il y a donc 50 cases actives. La plus longue diagonale, joignant deux coins du damier et comprenant 10 cases foncées, se dénomme la grande diagonale. Le damier doit être placé de sorte que la première case de gauche, pour chaque joueur, soit une case foncée. Le jeu de dames international se joue avec 20 pions blancs (clairs) et 20 pions noirs (foncés). Avant de débuter une partie, les 20 pions noirs et les 20 pions blancs sont disposés sur les 4 premières rangées de chaque joueur.

Le plateau sera représenté par une matrice carrée de type `array` de taille $2n \times 2n$, les cases vide seront symbolisées par l'entier 0, celles occupées par les pions du joueur 1 par l'entier 1 et celles du joueurs 2 par l'entier 2.

Dans la suite, une configuration est un plateau de jeu représenter par une matrice carré composée de 0 de 1 et de 2 souvent notée `T`.

Pour simplifier le jeu, nous ne tiendrons pas compte de la dame. Dans la plus part des exemples du sujet, nous nous limiterons à un plateau de 36 cases et deux fois 12 pions.

1. Programmer la fonction `initialisation(n,p)` prenant comme paramètres les entiers n et p et renvoyant la matrice de type `array` de taille $2n \times 2n$, représentant le plateau initial en plaçant les pn pions du joueurs 1 sur p rangées et à l'opposée celles du joueurs 2 su p rangées.

   Par exemple dans le cas d'un plateau de 6×6 on obtient :

```
initialisation(3,2)
Out[1]:
array([[0., 1., 0., 1., 0., 1.],
       [1., 0., 1., 0., 1., 0.],
       [0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0.],
       [0., 2., 0., 2., 0., 2.],
       [2., 0., 2., 0., 2., 0.]])
```

   Et dans le cas du jeu officiel de 100 cases, le plateau initial attendu est donc :

```
    initialisation(5,4)
Out[2]:
array([[0., 1., 0., 1., 0., 1., 0., 1., 0., 1.],
       [1., 0., 1., 0., 1., 0., 1., 0., 1., 0.],
       [0., 1., 0., 1., 0., 1., 0., 1., 0., 1.],
       [1., 0., 1., 0., 1., 0., 1., 0., 1., 0.],
       [0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
       [0., 2., 0., 2., 0., 2., 0., 2., 0., 2.],
       [2., 0., 2., 0., 2., 0., 2., 0., 2., 0.],
       [0., 2., 0., 2., 0., 2., 0., 2., 0., 2.],
```

```
       [2., 0., 2., 0., 2., 0., 2., 0., 2., 0.]])
```

2. Programmer une fonction `test(T,u)` prenant comme paramètres une configuration de jeu `T` de type `array`, un `tuple u` de couple d'entier `(i,j)` et renvoyant `True` si les coordonnées (i, j) du tuple sont bien sur le plateau.
3. Programmer une fonction `score(T)`, prenant en paramètres une configuration `T` de type `array` et renvoyant le tuple (J_1, J_2) où J_i est le nombre de pions du joueurs i sur le plateau de configuration `T`.
4. Programmer la fonction `pion(T,J)` prenant comme paramètres la configuration `T`, l'entier `J` du numéro du joueurs et renvoyant la liste de uplets des coordonnées des cases occupées par les pions du joueurs J sur le plateau `T`.
5. On considère qu'un joueur a perdu la partie lorsqu'il ne lui reste plus aucune pièce en jeu, ou bien, si c'est à lui de jouer, et que toutes ses pièces sont bloquées, c'est-à-dire dans l'impossibilité de prendre ou de se déplacer.

   Dans un premier temps on considèrera que le joueur J gagne si il n'y a plus de pion du joueur adverse sur le plateau. Programmer la fonction `victoire(T,J)`, renvoyant `True` si la configuration `T` donne le joueur J gagnant et `False` sinon.

Partie III : Régles du jeu et deplacements

Il existe deux types de pièces : les pions et les dames. Le premier coup est toujours joué par les blancs. Les adversaires jouent un coup chacun à tour de rôle avec leurs pièces. Un pion se déplace obligatoirement vers l'avant, en diagonale, d'une case sur une case libre de la rangée suivante.

1. Programmer la fonction `deplacer(T,u:tuple,v:tuple)`, prenant comme paramètres la configuration `T`, les tuples de couples d'entiers, `u` et `v` modifiant `T` en déplaçant le pion de la cases de coordonnées u sur la case de coordonnées v.
2. On dispose de la fonction suivante :

```
def option(T,c:tuple):
    assert test(T,c)
    m=len(T)
    D=[]
    i,j=c
    J=T[i,j]
    if J==1:
        if i<m-1 and j>0 and j<m-1:
            D=[(i+1,j-1),(i+1,j+1)]
        elif i<m-1 and j==0:
            D=[(i+1,j+1)]
        elif i<m-1 and j==m-1:
            D=[(i+1,j-1)]
    if J==2:
        if i>0 and j>0 and j<m-1:
            D=[(i-1,j-1),(i-1,j+1)]
        elif i>0 and j==0:
            D=[(i-1,j+1)]
        elif i>0 and j==m-1:
            D=[(i-1,j-1)]
    return D
```

   Programmer une fonction `deplacement(T,c:tuple)` prenant comme paramètres la configuration `T` et le tuple de couples d'entiers,`c`, appelant la fonction `option()` et renvoyant la liste des déplacements possibles (sans prendre), c'est à dire la listes des tuples, correspondants aux coordonnées des cases accessible pour le pions de coordonnées `c`.
3. La prise des pièces adverse est obligatoire et s'effectue aussi bien en avant qu'en arrière. Lorsqu'un pion se trouve en présence, diagonalement, d'une pièce adverse derrière laquelle se trouve une case libre, il doit obligatoirement sauter par-dessus cette pièce et occuper la case libre. Cette pièce adverse est alors enlevée du damier. Cette opération complète est la prise par un pion.

   Compléter le script de la fonction suivante, renvoyant le dictionnaire `D2` des prises possible. Le dictionnaire `D2` a pour clés les coordonnées des pions à prendre (en une prise) et d'unique valeur associée le tuple des coordonnées de la case ou le pion doit se déplacer après la prise :

```
def prendre(T,u):
    i,j=u
    J=T[i,j]

    D={(i+1,j-1):(i+2,j-2),(i+1,j+1):(i+2,j+2),
               (i-1,j-1):(i-2,j-2),(i-1,j+1):(i-2,j+2)}

    D2={}
    for u in D:
    ........................
    ........................
    ........................
    return D2
```

Pour illuster le fonctionement de la fonction, par exemple l'exécution du script suivant doit renvoyer :

```
T
Out[2]:
array([[0., 1., 0., 1., 0., 1.],
       [0., 0., 1., 0., 1., 0.],
       [0., 0., 0., 0., 0., 0.],
       [0., 0., 1., 0., 0., 0.],
       [0., 2., 0., 2., 0., 2.],
       [2., 0., 2., 0., 2., 0.]])

prendre(T,(4,1))
Out[3]: {(3, 2): (2, 3)}
prendre(T,(4,3))


T
Out[4]:
array([[0., 1., 0., 1., 0., 1.],
       [0., 0., 1., 0., 1., 0.],
       [0., 0., 0., 0., 0., 0.],
       [0., 0., 1., 0., 1., 0.],
       [0., 2., 0., 2., 0., 2.],
       [2., 0., 2., 0., 2., 0.]])



prendre(T,(4,3))
Out[5]: {(3, 2): (2, 1), (3, 4): (2, 5)}
```

4. Modifier le programme de la fonction `victoire(T,J)`, en considérant qu'un joueur a perdu la partie lorsque soit il ne lui reste plus aucune pièce en jeu, soit, c'est à lui de jouer, et que toutes ses pièces sont bloquées, c'est-à-dire qu'il est dans l'impossibilité de prendre ou de se déplacer.
5. Lorsqu'au cours d'une prise par un pion, celui-ci se trouve à nouveau en présence, diagonalement, d'une pièce adverse derrière laquelle se trouve une case libre, il doit obligatoirement sauter par-dessus cette seconde pièce, voire d'une troisième et ainsi de suite, et occuper la case libre se trouvant derrière la dernière pièce capturée. Les pièces adverses ainsi capturées sont ensuite enlevées du damier dans l'ordre de la prise. Cette opération complète est une rafle par un pion. La prise du plus grand nombre de pièces est prioritaire, donc obligatoire. Dans ce cas, une dame compte pour une pièce, tout comme un pion. Elle ne confère nulle priorité et n'impose aucune obligation.
   Compléter les cinq lignes du script de la fonction `Prendre_opt(T,u)`, programmée en récursif, prenant comme paramètres la configuration `T`, le tuple `u` et renvoyant le couple (j, P) où j la longueur de la plus grande prise possible pour le pion de la case de coordonnées u (c-a-d le nombre maximum de pions adverse prenable par le pion en u,et 0 si pas de prise possible) et la liste des listes des couples $[v_i, u_i]$, où v_i sont les coordonnées des pions pris, u_i la case d'arrivée à la ième prise (la liste est vide si pas de prise possible). Une de ses liste sera appelé chemin de prise.

```
def Prendre_opt(T,u):
    P=[]
    S=1
    D=prendre(T,u)
    if len(D)==0:
(1)         return .......
    i,j=u
    J=T[i,j]
    for v in D:
        T2=T.copy()
        T2[v[0],v[1]]=0
        T2[i,j]=0
        T2[D[v][0],D[v][1]]=J
(2)     S1,P1=..................
          if S1+1==S:
             for u in P1:
(3)                u=.............
                   P.append(u)
            if S1+1>S:
                S=max(S1+1,S)
(4)             P=...........
                for u in P1:
(5)                  u=..............
                     P.append(u)
          return int(S),P
```

L'execution de la fonction devra, par exemple, renvoyer :

```
T
Out[5]
array([[0., 1., 0., 0., 0., 1.],
       [0., 0., 1., 0., 1., 0.],
       [0., 0., 0., 0., 0., 0.],
       [0., 0., 1., 0., 1., 0.],
       [0., 2., 0., 2., 0., 2.],
       [2., 0., 2., 0., 2., 0.]])

Prendre_opt(T,(4,3))
Out[6]:
(4,
 [[[(3, 2), (2, 1)], [(1, 2), (0, 3)], [(1, 4), (2, 5)], [(3, 4), (4, 3)]],
  [[(3, 4), (2, 5)], [(1, 4), (0, 3)], [(1, 2), (2, 1)], [(3, 2), (4, 3)]]])

 T
Out[7]:
array([[0., 1., 0., 0., 0., 0.],
       [0., 0., 0., 0., 1., 0.],
       [0., 0., 0., 1., 0., 0.],
       [0., 0., 1., 0., 0., 0.],
       [0., 1., 0., 1., 0., 2.],
       [2., 0., 2., 0., 2., 0.]])

Prendre_opt(T,(5,2))
Out[8]: (2, [[[(4, 3), (3, 4)], [(2, 3), (1, 2)]]])
```

Par exemple `[[(4, 3), (3, 4)], [(2, 3), (1, 2)]]` est un chemin de prise du pion en $(5, 2)$.

6. Programmer la fonction `prise(T,u,p)` prenant comme paramètres une configuration `T`, une liste de prise p d'un chemin de prise, de type `p`$= [[v_1, u_1], \ldots, [v_l, u_l]]$, avec u_1 la première case de saut , u_l celle d'arrivée et les v_i les pions pris, et renvoyant la nouvelle configuration après la prise.

Partie IV : Graphe bibarti

1. Programmer la fonction `degG(G)` prenant comme paramètre le dictionnaire de la liste d'adjacence d'un graphe G et renvoyant le dictionnaire des degrés des noeuds du graphe.
2. Programmer la fonction `transposeG(G)` prenant comme paramètre le dictionnaire de la liste d'adjacence d'un graphe G et renvoyant le dictionnaire du transposé du graphe c-a-d la liste des antécedants des noeuds.
3. Pour créer notre graphe biparti le noeud du graphe sera étiqueté par le couple `(T,J)`, où T est la configuration et J le joueur dont c'est le tour de jeu.

   Il n'est pas possible d'utiliser une matrice comme clé nous allons donc hacher cette matrice de configuration pour ne manipuler que des entiers.

   Programmer la fonction hachage `hachage(T)` renvoyant l'entier constitué des chiffres des entiers composants la première ligne de la matrice de gauche à droite puis de la suivante et ainsi de suite.

   Ainsi après exécutions la fonction de hachage renvoie par exemple :

```
hachage(np.array([[1., 0., 1., 0., 1., 0.],
       [0., 1., 0., 1., 0., 1.],
       [0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0.],
       [2., 0., 2., 0., 2., 0.],
       [0., 2., 0., 2., 0., 2.]]))
Out[2]: 10101001010100000000000202020020202
hachage(np.array([[0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0.],
       [2., 0., 2., 0., 2., 0.],
       [0., 2., 0., 2., 0., 2.]]))
Out[3]: 202020020202
```

4. Programmer la fonction `plateau(h,m)` prenant comme paramètres l'entier h du hachage d'une configuration `T`, l'entier m de la taille initiale du plateau et renvoyant la matrice de la configuration initiale `T`. On prendra garde au fait que le chiffre de gauche du nombre h n'est pas forcément l'entier de la première case du plateau.
5. Pour constituer le graphe biparti les noeuds correspondants aux configurations de jeu `T`, seront étiquetés par le couple d'entiers `(h,J)` où `h` est l'entier correspondant à la configuration de jeu `T` après hachage soit `h=hachage(T)` et `J` le numéro du joueur dont c'est le tour de jeu.

   On dispose de la fonction suivante :

```
def config(T,u):
    l=0
    J=T[u[0],u[1]]
    l,P=Prendre_opt(T,u)
    C=[]
    if l>0:

        for p in P:
            T1=prise(T,p,u)
            C.append((hachage(T1),int(3-J)))
            return C,l
    D=deplacement(T,u)

    for v in D:
        T1=T.copy()
        deplacer(T1,u,v)
        C.append((hachage(T1),int(3-J)))
    return C,l
```

Que renvoi les scripts suivants :

(a)
```
T
Out[10]:
array([[0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0.],
       [1., 1., 0., 1., 0., 0.],
       [2., 0., 2., 0., 2., 0.]])


config(T,(5,0))
Out[11]
.....................
```

(b)
```
config(T,(5,2))
Out[12]
.....................
```

6. Programmer la fonction `configuration(T,J)` prenant comme paramètres la configuration `T`, le numéro d'un joueur J, appelant les fonctions `pion(),victoire()` et `config()` et renvoyant la liste D des noeuds des configurations possible après que le joueur J est joué, c-a-d la liste des couples d'entiers $(h_i, 1)$ (respectivement $(h_i, 2)$) si $J = 2$ (respectivement si $J = 1$) et les entiers h_i des différentes configurations après hachage.
7. Programmer la fonction `graph(T,J)` renvoyant le dictionnaire G de la liste d'adjacence en complétant le programme, adaptant le programme de recherche en profondeur suivant :

```
def graph(T,J):
    m=len(T)
    if victoire(T,J):
        return {}

    L=pion(T,J)
    if len(L)==0:
        return {}

    G={}
    attente=[]
    D=configuration(T,J)
    attente+=D
    G[(hachage(T),J)]=D
    while len(attente)>0 and len(G)<10000:
        ........................
       ........................
        ........................
        ........................
         ........................


         ..................
    return G
```

8. Comme nous l'avons vu, la constitution du graphe contenant toutes les configurations reste très théorique. On peut imaginer se limiter à un mini plateau de taille 6×6 et de 12 pions chacun. Comme le nombres des configurations restent importants, on imposera une condition d'arrêt sur la taille de G dans le programme `grap()`.

   Programmer la fonction `cond_victoire(G,J,m)` prenant comme paramètres la liste d'adjacence G, le numéro de joueur, l'entier m de la dimension du plateau et renvoyant la liste des noeuds de G proposant une configuration victorieuse pour le joueur J, c-a-d renvoyant l'ensemble des noeuds de la condition d'ateignabilité.
9. Programmer `attracteur(G,W)` renvoyant l'attracteur du graphe G, en terminant le programme suivant :

```
def attracteur(G,W):
    A=[]
    tG=transpose(G)
    degG=degG(G)

    def parcour(u):
        if u not in A:
            A.append(u)
        if u in tG:
            ....................:
                ...................
                ......................:
                    parcour(v)
    for u in W:
        parcour(u)
    return A
```