Lapins et renards

Remarque 1

L'énoncé comporte une imprécision sur le positionnement des renards. Il ne précise pas si pour un renard horizontal (resp. vertical), sa queue est à gauche ou à droite de sa tête (resp. au dessus ou en dessous). Dans ce corrigé, j'ai choisi de positionner systématiquement la queue à gauche (resp. en dessous).

1 Lapins et renards

1. On peut résoudre le plateau de l'énoncé en 11 coups :
   (a) Avancer le renard
   (b) Saut du lapin à droite au dessus du renard, puis vers le bas au dessus de champignon
   (c) Reculer le renard
   (d) Saut du lapin à gauche au dessus du renard et du champignon
   (e) Avancer le renard
   (f) Saut à droite du lapin au dessus du champignon, puis vers le haut au dessus du renard, puis à droite au dessus du champignon
   (g) Avancer le renard
   (h) Saut du lapin à droite au dessus du renard et du champignon pour atterir dans le terrier.
2. Il suffit de tester si tous les lapins sont situés sur l'un des 5 terriers ce qui s'écrit par exemple de la manière suivante :

```
trous = [0,4,12,20,24]

def gagne(plateau):
    for x in plateau[0]:
        if x not in trous:
            return False
    return True
```

3. Pour simplifier, on utilise deux fonctions de conversions, l'une qui à un numéro de case associe le couple de ses coordonnées, l'autre étant sa réciproque.

```
def int_to_coords(n):
    return (n//5,n % 5)

def coords_to_int(i,j):
    return 5*i+j
```

Les deux fonctions demandées peuvent alors s'écrire de la manière suivante :

```
def plateau_to_grille(plateau):
    M=[[0 for i in range(5)] for j in range(5)]
    for x in plateau[0]:
        (i,j)=int_to_coords(x)
        M[i][j]=1
    for x in plateau[2]:
        (i,j)=int_to_coords(x)
        M[i][j]=2
    for x in plateau[1]:
        (i,j)=int_to_coords(x[0])
        M[i][j]=3
        if x[1]==0:
            M[i][j-1]=3
        else:
            M[i+1][j]=3
    return M
```

```
def print_grille(M):
    for i in range(5):
        print(M[i])

def print_plateau(L):
    print_grille(plateau_to_grille(L))
```

2 Coups possibles

4. Pour connaître les coups possibles d'un renard, il suffit de savoir si les deux cases situées devant et derrière lui sont libres. Il y a deux cas à traiter suivant que le renard est horizontal ou vertical. De plus, avant de vérifier si une case est libre, il convient bien entendu de s'assurer qu'elle existe.

```
def coups_du_renard(grille,renard):
    (i,j)=int_to_coords(renard[0])
    L2=[]
    if renard[1]==0:
        if j+1<=4 and grille[i][j+1]==0: # deplacement à droite
            L2.append((coords_to_int(i,j+1),0))
        if j>=2 and grille[i][j-2]==0: # deplacement a gauche
            L2.append((coords_to_int(i,j-2),0))
    if renard[1]==1:
        if i+2<=4 and grille[i+2][j]==0: # deplacement vers le bas
            L2.append((coords_to_int(i+1,j),1))
        if i>=1 and grille[i-1][j]==0: # deplacement vers le haut
            L2.append((coords_to_int(i-1,j),1))
    return L2
```

5. Pour connaître les coups possibles d'un lapin, c'est un peu plus compliqué. Il y a 4 cas à traiter puisque le lapin peut faire un saut dans 4 directions différentes à priori. Pour sauter par exemple à droite, on vérifie déjà que la case à droite du lapin existe et qu'elle est occupée. Ensuite, on consulte les cases à droite du lapin jusqu'à tomber sur une case vide ou déborder de la grille. Si à ce moment on a trouvé une case vide, il s'agit d'un déplacement possible du lapin. Dans tous les autres cas, le lapin ne peut partir à droite. Les 3 autres directions se traitent de manière similaire.

```
def coups_du_lapin(grille,lapin):
    (i,j)=int_to_coords(lapin)
    L2=[]
    if j<4 and grille[i][j+1]!=0: # saut vers la droite
        a=j+1
        while a<=4 and grille[i][a]!=0:
            a+=1
        if a<=4:
            L2.append(coords_to_int(i,a))
    if j>0 and grille[i][j-1]!=0: # saut vers la gauche
        a=j-1
        while a>=0 and grille[i][a]!=0:
            a-=1
        if a>=0:
            L2.append(coords_to_int(i,a))
    if i<4 and grille[i+1][j]!=0: # saut vers le haut
        a=i+1
        while a<=4 and grille[a][j]!=0:
            a+=1
        if a<=4:
            L2.append(coords_to_int(a,j))
    if i>0 and grille[i-1][j]!=0: # saut vers le bas
        a=i-1
        while a>=0 and grille[a][j]!=0:
            a-=1
        if a>=0:
            L2.append(coords_to_int(a,j))
    return L2
```

6. Dans un premier temps, on a besoin d'une fonction qui réalise une copie d'un plateau. Attention aux copies de listes, une syntaxe `L1=L` ne fonctionne pas. On crée donc de nouvelles listes, qui contiennent les mêmes éléments.

```
def copie(plateau):
    return [[x for x in plateau[0]],
            [y for y in plateau[1]],
            [z for z in plateau[2]]]
```

Ensuite, on calcule les déplacements possibles de tous les lapins, et ceux de tous les renards. Pour chacun d'entre eux, on crée une copie du tableau actuel, dans lequel on modifie la position de l'animal en question.

```
def coups_possibles(plateau):
    L2=[]
    grille=plateau_to_grille(plateau)
    for i in range(len(plateau[0])):
        A=coups_du_lapin(grille,plateau[0][i])
        for l in A:
            new_plateau=copie(plateau)
            new_plateau[0][i]=l
            L2.append(new_plateau)
    for i in range(len(plateau[1])):
        B=coups_du_renard(grille,plateau[1][i])
        for r in B:
            new_plateau=copie(plateau)
            new_plateau[1][i]=r
            L2.append(new_plateau)
    return L2
```

3 Resolution

8. Pour faire simple, je me suis contenté d'utiliser deux listes :

   - La première `Deja_vus` recense la liste de tous les plateaux que l'on peut atteindre à partir du plateau initial.
   - La seconde `L` est la liste de tous les plateaux encore en cours d'exploration.

   Initialement, ces deux listes sont réduites au plateau initial. A chaque étape, on extrait le premier élément de liste `L`. On calcule tous les plateaux que l'on peut atteindre en un coup à partir de `L`. Parmi tous ces plateaux, ceux que l'on a jamais déjà rencontré sont ajoutés à la fois à `Deja_vus` et à `L`.

```
def resolution(plateau):
    L=[plateau]
    Deja_vus=[plateau]
    while L!=[]:
        p=L.pop(0)
        if gagne(p):
            return p
        else:
            L2=coups_possibles(p)
            for y in L2:
                if y not in Deja_vus:
                    Deja_vus.append(y)
                    L.append(y)
    print("Pas de solutions")
```

Dans cet algorithme, les appels à `coups_possibles` ne sont pas très couteux, car pour un même plateau, il y a au plus 4 coups possibles par lapin, et 2 coups possibles par renard, et on les trouve avec de l'ordre de quelques dizaines d'opérations élémentaires. En revanche, les vérifications `y not in Deja_vus` sont de plus en plus couteuses au fur et à mesure que la taille de la liste `Deja_vus` augmente. Dès lors qu'un plateau nécessite plus d'une vingtaine de coups pour gagner, la taille de la liste atteint très vite de l'ordre de 10000 voire 100000 plateaux différents, et l'algorithme prend un temps conséquent pour trouver la solution.

4 Bonus (reconstitution de la solution)

L'algorithme précédent se contente de dire quel est l'état du plateau lorsque l'on a gagné, mais ne précise pas comment y arriver. Pour palier ce défaut, on peut utiliser la technique suivante. On attribue un numéro à chaque plateau une fois qu'il est découvert. De plus, on lui associe à l'aide d'un tableau `Peres` le numéro du plateau précédent, qui a permis de l'atteindre en un coup.

```
def calcule_coups_solution(plateau):
    L=[(plateau,0)]
    Deja_vus=[plateau]
    Peres=[-1]
    rang=0
    while L!=[]:
        p=L.pop(0)
        if not gagne(p[0]):
            L2=coups_possibles(p[0])
            for y in L2:
                if y not in Deja_vus:
                    Deja_vus.append(y)
                    Peres.append(p[1])
                    rang+=1
                    print(rang)
                    L.append((y,rang))
        else:
            return Deja_vus,Peres,p
    print("Pas de solutions")
```

Cela permet une fois le plateau gagnant retrouvé de remonter d'indice en indice jusqu'au plateau initial, et donc de reconstituer le chemin jusqu'à la solution. Voici une implémentation de cette méthode.

```
def reconstruit_solution(plateau):
    G,P,p=calcule_coups_solution(plateau)
    L2=[]; i = p[1]
    while i>=0:
        L2.append(G[i])
        i=P[i]
    n=len(L2)
    for i in range(n-1,-1,-1):
        print_grille(plateau_to_grille(L2[i]))
        print(" ")
```