Correction Devoir suveillé N°2

Problème 1 : Abalone

Abalone est un jeu de stratégie combinatoire abstrait où s'affrontent deux joueurs. Le jeu, d'origine française, a remporté le Gobelet d'Or au Concours de créateurs de jeux de Boulogne-Billancourt en 1988 sous le nom de Sumito. Dès l'année suivante, le jeu fut édité et s'est vendu à 4,5 millions d'exemplaires, dans plus de 30 pays, vingt ans plus tard.

Principe général

Un joueur joue avec des billes blanches, l'autre avec des billes noires. Le but du jeu est d'être le premier à faire sortir 6 billes adverses du plan de jeu en les poussant avec ses propres billes.

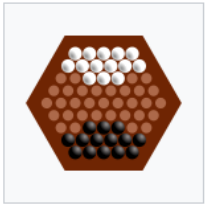
Matériel et mise en place

- Le tablier de jeu est un hexagone percé de 61 cercles supportant les billes. Le tablier est parfois appelé hexagone ou plateau.
- Chaque joueur dispose de 14 billes qui sont placées au départ selon la position indiquée dans les règles.

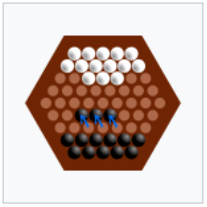
Déroulement de la partie

Le joueur possédant les billes noires commence. À tour de rôle les joueurs déplacent 1, 2 ou 3 billes de sa couleur d'un mouvement vers des cases voisines. Le déplacement peut se faire en ligne ou latéralement.

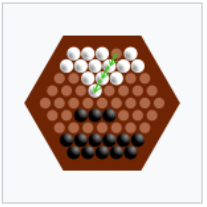
Cependant, pour simplifier l'implémentation, on supposera que seuls **les déplacements en ligne sont autorisés**.



Position de départ classique dans le jeu standard.



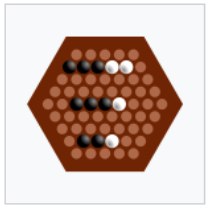
Le joueur avec les billes noires commence. Il décide de déplacer latéralement un ensemble de trois billes contiguës.



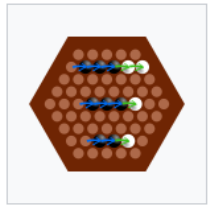
Blanc répond par un mouvement en ligne, toujours de trois billes. Il aurait pu choisir de n'en déplacer que deux ou même une seule

Déplacement des billes de l'adversaire

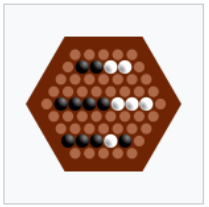
Pour pouvoir pousser les billes de son adversaire, le joueur doit se trouver en position de sumito, c'est-à-dire en supériorité numérique : la ligne doit contenir successivement plus de billes de sa couleur que de la couleur adverse . Une ligne de 3 billes adverses ou plus ne peut jamais être poussée par l'adversaire.



Les trois *sumitos* possibles : 3 contre 2, 3 contre 1 et 2 contre 1.

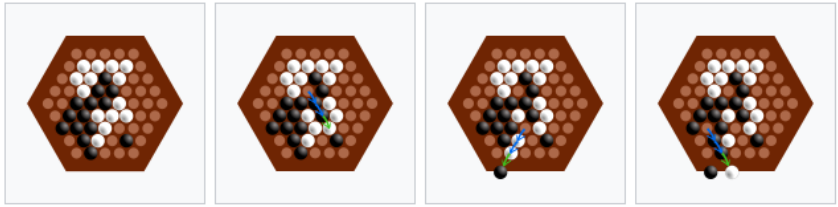


Position après la poussée dans les trois cas.



Dans ces trois cas, Noir ne peut pas pousser.
1) 2 contre 2 : force insuffisante
2) Il est impossible de pousser 3 billes, même avec 4 !
3) La bille noire à droite de la blanche empêche Noir d'effectuer une poussée.

Exemple de milieu de partie



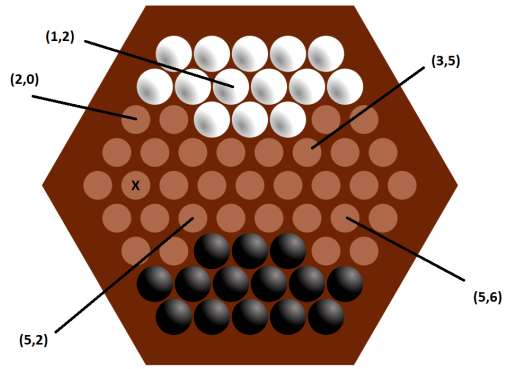
Implémentation en python

Le but de cette partie sera de créer l'arbre du jeu sur une profondeur fixée.

Par la suite, on fera référence au case du jeu à l'aide d'un couple-position (i, j) tel que :

- i représente le numéro de la ligne, en commençant par la ligne du haut, numérotée 0.
- j représente le numéro de l'élément sur une ligne fixée, en commençant par celui le plus à gauche, numéroté 0.

Voici quelques exemples :



1 Quel est le couple-position de la case référencée par une croix X ?

Correction : (4,1)

Chaque état de la partie, sera représenté par une **étiquette**, c'est-à-dire une chaine de caractères telle que :

- les billes blanches seront représentées par le caractère `'B'`
- les billes noires seront représentées par le caractère `'N'`
- l'absence de bille sur la case est représentée par `'X'`

La chaine de caractères parcourt chaque position de gauche à droite en commençant par la ligne la plus haute et en descendant. Le caractère `'|'` représente un retour à la ligne.

Par exemple, la position initiale (celle de l'image précédente) est représentée par :

```
initiale='BBBBB|BBBBBB|XXBBBXX|XXXXXXXX|XXXXXXXXX|XXXXXXXX|XXNNNXX|NNNNNN|NNNNN'
```

2 Donner la chaine de caractères donnant l'état de la partie pour l'image la plus à gauche de la section "Exemple du milieu de partie".

Correction : "XXXXX|XBBBBX|XBBNBXX|XXBNNXXX|XXNNNBXXX|XNNBBBXX|NNNBXXX|XNBXNX|XNXXX"

3 Écrire une fonction `etiquette_en_liste(etiquette:str)->list` qui prend en argument une étiquette et renvoie une liste de liste `L` telle que `L[i][j]` soit le caractère représentant la position (i, j). Par exemple, l'appel `etiquette_en_liste(initiale)` renvoie :

```
[['B', 'B', 'B', 'B', 'B'], ['B', 'B', 'B', 'B', 'B', 'B'], ['X', 'X', 'B', 'B',
 'B', 'X', 'X'], ['X', 'X', 'X', 'X', 'X', 'X', 'X', 'X'], ['X', 'X', 'X', 'X', 'X', 'X',
 'X', 'X', 'X', 'X', 'X'], ['X', 'X', 'N', 'N', 'N', 'X', 'X'], ['N', 'N', 'N',
 'N', 'N', 'N'], ['N', 'N', 'N', 'N', 'N']]
```

Correction :

```
def etiquette_en_liste(etiquette:str)->list:
    liste_fille=[]
    liste_mere=[]
    for k in etiquette:
        if k!='|':
            liste_fille.append(k)
        else:
            liste_mere.append(liste_fille)
            liste_fille=[]
    liste_mere.append(liste_fille)
    return liste_mere
```

4 Écrire une fonction `liste_en_etiquette(L:list)->str` qui effectue l'opération inverse.

Correction :

```
def liste_en_etiquette(L:list)->str:
    texte=''
    for l in L:
        for k in l:
            texte=texte+k
        texte=texte+'|'
    return texte
```

Chaque ligne du plateau ne comporte pas le même nombre de cases (5 pour la ligne 0, puis 6 pour la ligne 1 et ainsi de suite). On remarque que la ligne i comporte en fait $9 - |4 - i|$ cases.

5 Écrire une fonction `taille(i:int)->int` qui renvoie le nombre de cases de la ligne i, si i est bien un entier représentant une des lignes du plateau. Sinon, la fonction renverra `None` en affichant un message d'erreur.

Correction :

```
def taille(i):
    if i>=0 and i<=8 and type(i)==int:
        return 9-abs(4-i)
    else:
        print("Attention la variable n'est pas un indice représentant une des cases du plateau")
        return None
```

6 Écrire une fonction `verif_case(c:tuple)->bool` qui prend en argument une variable `c=(i,j)` et qui vérifie si (i, j) est un couple-position du plateau. Elle renvoie `True` si c'est bien le cas, `False` sinon.

Correction :

```
def verif_case(c:tuple)->bool:
    i,j=c
    if 0<=i<=8 and 0<=j<=taille(i)-1:
        return True
    return False
```

On va commencer à implémenter les mouvements possibles, tout d'abord, remarquons qu'à une position donnée, il y a jusqu'à 6 mouvements possibles :

- en haut à gauche qu'on codera par `'HG'`
- en haut à droite qu'on codera par `'HD'`
- en bas à gauche qu'on codera par `'BG'`
- en bas à droite qu'on codera par `'BD'`
- à gauche qu'on codera par `'CG'`
- à droite qu'on codera par `'CD'`

7 Créer une fonction `mouvement(couple:tuple,direction:str)->tuple` qui prend en arguments la variable `couple=(i,j)` et un mouvement donné par `direction` codé comme précisé ci-dessus (par exemple `'HG'`) et qui renvoie le couple (i',j') qui représente le couple-position une fois le mouvement `direction` exécuté depuis la case (i,j). Si le mouvement n'envoie pas sur une case du tableau, la fonction renvoie `None`.

Correction :

```
def mouvement(couple:tuple,direction:str)->tuple:
    i,j=couple
    if direction=='HG':
        if 0<=i<=4:
            res=(i-1,j-1)
        else:
            res=(i-1,j)
    if direction=='HD':
        if 0<=i<=4:
            res=(i-1,j)
        else:
            res=(i-1,j-1)
    if direction=='CG':
        res=(i,j-1)
    if direction=='CD':
        res=(i,j+1)
    if direction=='BG':
        if 0<=i<=3:
            res=(i+1,j)
        else:
            res=(i+1,j-1)
    if direction=='BD':
        if 0<=i<=3:
            res=(i+1,j+1)
        else:
            res=(i+1,j)
    if verif_case(res):
        return res
    else :
        return None
```

8 Écrire une fonction `nombre_caractere(chaine:str,caractere:str)->int` qui prend en arguments une chaine de caractères `chaine` et un caractère `caractere` et qui renvoie le nombre de fois qu'apparait le caractère dans la chaine.

Correction :

```
def nombre_caractères(chaine,caractère):
    nb=0
    for k in chaine:
        if k==caractère:
            nb=nb+1
    return nb
```

9 Écrire une fonction python `autre_couleur(couleur:str)->str` qui prend en arguments un caractère `couleur` qui représente une des deux couleurs (`'B'` pour blanc et `'N'` pour noir) et qui renvoie le caractère représentant la seconde couleur. La fonction renverra `None` sinon.

Correction :

```
def autre_couleur(couleur):
    if couleur=="B":
        return "N"
    elif couleur=="N" :
        return "B"
    else:
        return None
```

10 Écrire une fonction `ligne_admissible(n:int,m:int)->bool` qui prend en arguments deux entiers et qui renvoie `True` ou `False` selon si la ligne constituée de n boules consécutifs de la couleur du joueur puis de m boules consécutifs de la couleur de l'adversaire a le droit (selon les règles) d'être déplacée. Par exemple (3,0) renvoie True car une ligne de 3 boules de la couleur du joueur puis d'aucune boule de la couleur de l'adversaire a le droit d'être déplacée.

Correction :

```
def ligne_admissible(n,m):
    if (n,m) in [(1,0),(2,0),(3,0),(3,2),(3,1),(2,1)]:
        return True
    else:
        return False
```

11 Écrire une fonction `coup_admissible(liste_tableau:list,position:tuple,direction:str,couleur:str)` qui prend en arguments la liste de listes `liste_tableau` obtenue à l'aide de l'étiquette et de la fonction `etiquette_en_liste`, un 2-uplet `position`, une chaine de caractère `direction` codant le mouvement et d'un caractère `couleur` et qui renvoie `True` si le joueur lié à `couleur` peut déplacer la bille en position `(i,j)` dans la direction `direction`, `False` sinon.

Correction :

```
def coup_admissible(liste_tableau,position,direction,couleur):
    couleur_adv=autre_couleur(couleur)
    ligne=couleur
    if liste_tableau[position[0]][position[1]]!=couleur:
        return False
    position=mouvement(position,direction)
    while position!=None and liste_tableau[position[0]][position[1]]==couleur:
        position=mouvement(position,direction)
        ligne+=couleur
    while position!=None and liste_tableau[position[0]][position[1]]==couleur_adv:
        position=mouvement(position,direction)
        ligne+=couleur_adv
    if position!=None and liste_tableau[position[0]][position[1]]==couleur:
        return False
    else:
        if ligne_admissible(nombre_caractères(ligne,couleur),nombre_caractères(ligne,couleur_adv)):
            return True
        else :
            return False
```

12 Écrire une fonction `positions_billes(liste_tableau:list,couleur:str)->list` qui prend des arguments expliqués à la question précédente et renvoie la liste des couples (i, j) où se trouvent les billes de la couleur donnée par la variable `couleur`.

Correction :

```
def positions_billes(liste_tableau:list,couleur:str)->list:
    L=[]
    for i in range(9):
        for j in range(taille(i)):
            if liste_tableau[i][j]==couleur:
                L.append((i,j))
    return L
```

13 Écrire une fonction `nouvel_etat_tableau(liste_tableau:list,position:tuple,direction:str)->list` qui renvoie une copie de la liste `liste_tableau` mis à jour après que le joueur est joué son coup à la position et à la direction données par les deux variables correspondantes. On pourra utiliser la fonction `copy.deepcopy(L)` pour copier une liste `L`. On ne vérifiera pas forcément que le coup est admissible à l'intérieur de cette fonction (on suppose qu'on a déjà vérifié cela au préalable dans le script).

Correction :

```
def nouvel_etat_tableau(liste_tableau:list,position:tuple,direction:str)->list:
    liste_tableau2=copy.deepcopy(liste_tableau)
    couleur=liste_tableau[position[0]][position[1]]
    if couleur not in ['B','N']:
        return None
    couleur_adv=autre_couleur(couleur)
    liste_tableau2[position[0]][position[1]]='X'
    position=mouvement(position,direction)
    while position!=None and liste_tableau2[position[0]][position[1]]==couleur:
        position=mouvement(position,direction)
    if position!=None :
        liste_tableau2[position[0]][position[1]]=couleur
        position=mouvement(position,direction)
    if position!=None:
        bille=liste_tableau2[position[0]][position[1]]
        while position!=None and liste_tableau2[position[0]][position[1]]==couleur_adv:
            position=mouvement(position,direction)
        if position!=None and bille==couleur_adv :
            liste_tableau2[position[0]][position[1]]=couleur_adv
    return liste_tableau2
```

14 Écrire une fonction `nouvelles_etiquettes(etiquette:str,couleur:str)->list` qui renvoie la liste de toutes les étiquettes de tous les états de la partie possibles après que le joueur de la couleur `couleur` ait joué. Sachant que la variable `etiquette` est l'étiquette de l'état de la partie avant que celui-ci n'ait joué.

Correction :

```
def nouvelles_etiquettes(etiquette:str,couleur:str)->list:
    L=[]
    liste_tableau=etiquettage_en_liste(etiquette)
    pos_billes=positions_billes(liste_tableau,couleur)
    for pos in pos_billes:
        for direction in ['HG','HD','BG','BD','CD','CG']:
            if coup_admissible(liste_tableau,pos,direction,couleur):
                l=nouvel_etat_tableau(liste_tableau,pos,direction)
                e=liste_en_etiquette(l)
                L.append(e)
    return L
```

15 Écrire une fonction `partie_finie(etiquette)` qui renvoie True si l'état de la partie est terminal, False sinon.

Correction :

```
def partie_finie(etiquette:str)->bool:
    n=nombre_caractères(etiquette,'B')
    m= nombre_caractères(etiquette,'N')
    if n<=8 or m<=8:
        return True
    else :
        return False
```

16 Écrire `graphe(etiquette:str,couleur:str,profondeur:int)->tuple` qui prend en arguments les deux variables sus-citées et un entier `profondeur` et qui construit l'arbre du jeu dont la racine est étiquetée par `etiquette` qui représente l'état de la partie initiale, la variable `couleur` code la couleur du joueur qui contrôle la partie initialement. La fonction renverra le tuple :
`S,A,E,C,P`
tel que :

- `S=[0,...,n-1]` représente la liste des indices de chaque sommet ;
- `A` est la liste des listes `[i,j]` représentant l'arrête orienté entre `i` et `j`.
- `E` est tel que `E[k]` est l'étiquette du sommet `k`.
- `C` est tel que `C[k]` est la couleur du sommet `k`.
- `P` est tel que `P[k]` est la profondeur du sommet `k` dans l'arbre.

On pourra s'aider de ce qui suit :

- On initialise les variables à l'arbre qui contient qu'un sommet étiqueté par `etiquette` et contrôlé par le joueur lié à `couleur`
- On initialise une variable n à 0.
- tant que n est inférieur strict au nombre de sommets dans l'arbre :
  - Si la profondeur du sommet n est inférieure stricte à la variable profondeur et que le sommet n'est pas terminal :
    * On calcule la liste de toutes les étiquettes possibles à partir de l'étiquette du sommet n et de la couleur du sommet n à l'aide de la fonction `nouvelles_etiquettes`. Pour chaque étiquette se trouvant dans cette liste, on crée un sommet relié à n étiqueté par cette étiquette, contrôlé par la couleur opposée et dont la profondeur aura augmenté de 1 par rapport à celle de n.
  - On augmente n de 1.
- On retourne toutes les variables demandées.

Correction :

```
def graphe(etiquette:str,couleur:str,profondeur:int)->tuple:
    S=[0]
    A=[]
    E=[etiquette]
    C=[couleur]
    P=[0]
    n=0
    while n<len(S):
        if P[n]<profondeur and not partie_finie(E[n]):
            etiq=E[n]
            coul=C[n]
            prof=P[n]
            L=nouvelles_etiquettes(etiq,coul)
            for e in L:
                c=autre_couleur(coul)
                A.append([n,len(S)])
                S.append(len(S))
                E.append(e)
                C.append(c)
                P.append(prof+1)
        n=n+1
```



```
    return S,A,E,C,P
```

17 Après avoir fini toutes les questions précédentes à la perfection, vous pouvez proposer une fonction d'évaluation de l'état de la partie, voire, faire la fonction qui programme l'algorithme du minimax et qui permet de calculer le gain possible à partir de la racine à l'aide de votre fonction d'évaluation.