Correction du DS4

(EPITA PSI 2024)

1. On crée un plateau vide puis on ajoute les 4 premiers jetons dans la position décrite :

```
def plateau_depart() :
    p = [[0 for j in range(8)] for i in range(8)]
    p[3][3],p[4][4],p[3][4],p[4][3] = -1,-1,1,1
    return p
```

2. J_2 peut jouer en $\boxed{(4,f),\ (6,e),\ (3,c),\ (2,e) \text{ ou } (5,c)}$

3. Il suffit d'ajouter les valeurs des case du plateau `p` :

```
def score(p) :
    s = 0
    for l in p :
        for k in l :
            s += k
    return s
```

4. On teste parmi les 8 cases possibles lesquelles sont dans la plateau :

```
def cases_voisines(x,y) :
    L = []
    for i in [-1,0,1] :
        for j in [-1,0,1] :
            if (i,j) != (0,0) and 0<=x+i<8 and 0<=y+j<8 :
                L.append((x+i,y+j))
    return L
```

5. Pour chaque case `(i,j)` de `l`, on teste si la valeur de `p[i][j]` est nulle, ce qui correspond à une case sans jeton :

```
def cases_libres(p,l) :
    L = []
    for c in l :
        if p[c[0]][c[1]] == 0 :
            L.append(c)
    return L
```

6. Lorsqu'on pose un jeton en (i,j), on le pose obligatoirement sur une des cases frontière du plateau précédent (donc on en enlève une) et les nouvelles cases frontières qui s'ajoutent sont parmi les 8 cases voisines de (i,j) (donc on en ajoute au plus 8) mais parmi les cases voisines de (i,j), une au moins est déjà occupée car on ne peut poser un jeton qu'à côté d'une autre jeton du plateau ; ainsi, une des cases voisines de (i,j) ne sera pas une case frontière du nouveau plateau. Au final, on ajoute au plus $-1+8-1=$ $\boxed{\text{6 nouvelles cases frontière}}$

7. On commence par retirer de `l` la case `c` (puisqu'on ne peut jouer que sur une case frontière) puis, pour chaque case voisine de `c`, on vérifie si elle est libre et si elle n'était pas encore dans la liste pour éviter les doublons. Cette dernière partie pose problème car le sujet interdit d'utiliser `in` pour vérifier si la case appartient déjà à la liste que l'on constitue ; pour contourner ce problème, on introduit un dictionnaire dont les clefs sont les cases et la valeur associée est `True` ou `False` selon qu'on a déjà ajouté la case dans la liste.

```
def cases_frontieres(p,l,c) :
    dico = {(i,j):False for i in range(8) for j in range(8)}
    L =[]
    for k in l :
        if k != c :
            L.append(k)
            dico[k] = True
    for d in cases_voisines(c[0],c[1]) :
        if p[d[0]][d[1]] == 0 and not dico[d] :
            L.append(d)
            dico[d] = True
    return L
```

8. On se contente d'ajouter les coordonnées de la case et du vecteur déplacement :

```
def deplacer(c,d) :
    return (c[0]+d[0],c[1]+d[1])
```

9. Pour qu'une capture soit possible, il faut que la case suivante dans la direction `d` soit occupée par un jeton de l'autre joueur, puis qu'on finisse par trouver une case occupée par un jeton du joueur actif, sans rencontrer de case vide (et en faisant attention à ne pas sortir du plateau) :

```
def est_direction_possible(p,c,d,j) :
    h,k = deplacer(c,d)
    possible = (0<=h<8 and 0<=k<8 and p[h][k]==-j)
    while 0<=h<8 and 0<=k<8 and p[h][k]==-j :
        h,k = deplacer((h,k),d)
    if 0<=h<8 and 0<=k<8 and p[h][h]==j :
        return possible
    else :
        return False
```

10. Pour chacune des 8 directions, on teste si une capture est possible avec la fonction précédente :

```
def directions_possibles(p,c,j) :
    L = []
    for dx in [-1,0,1] :
        for dy in [-1,0,1] :
            if (dx,dy) != (0,0) and est_direction_possible(p,c,(dx,dy),j) :
                L.append(d)
    return d
```

11. Pour chaque case de `l`, on vérifie si au moins une direction permet de jouer :

```
def cases_jouables(p,l,j) :
    L = []
    for c in l :
        if len(directions_possibles(p,c,j))!=0 :
            L.append(c)
    return L
```

12. On commence par déterminer le joueur actif : le joueur adverse est celui correspondant à la couleur du premier jeton rencontré (*Il aurait sans doute été plus simple de rajouter cette valeur en paramètre de la fonction*). Puis, on retourne tous les jetons rencontré dans la direction `d`, jusqu'à retrouver un jeton de la couleur du joueur actif :

```
def capturer(p,c,d) :
    h,k = deplacer(c,d)
    j = -p[h][k]
    while p[h][k]==-j :
        p[h][k] = j
        h,k = deplacer((h,k),d)
```

13. Copie, élément par élément :

```
def copier_plateau(p) :
    P = [[p[i][j] for j in range(8)] for i in range(8)]
    return P
```

14. On commence donc par copier le plateau puis on pose un jeton sur la case `c` et on capture les jetons, en vérifiant toutes les directions possibles puisque la pose d'un jeton peut entraîner des captures dans plusieurs directions :

```
def poser_jeton(p,c,j) :
    P = copier_plateau(p)
    h,k = c
    P[h][k] = j
    for d in directions_possibles(p,c,j) :
        capturer(P,c,d)
    return P
```

15. Pour examiner n tours, il faut en moyenne examiner 7^n plateaux, ce qui nécessite $7^n \times 2 \times 10^{-4}$ secondes donc au maximum on doit avoir $7^n \times 2 \times 10^{-4} \leqslant 4 \times 10^{17}$ donc $n \leqslant \dfrac{\ln(2) + 21\ln(10)}{\ln(7)} \in [25, 26]$ donc au maximum 25 tours.

16. Comme la fonction `poser_jeton` ne modifie pas le plateau initial, pas de problème. Le coup de valeur maximale dépend du joueur actif : si c'est J_2, on cherche le coup de valeur minimale en fait (obtenu en faisant le produit par `j` dans le code qui suit) :

```
def ia_naive(p,l,f,j) :
    val = -10001*j
    for c in l :
        next = f(poser_jeton(p,c,j))
        if j*next > j*val :
            val = next
            case = c
    return (c,val)
```

17. Comme on suppose qu'il y a toujours un coup jouable, le cas de base ($n = 1$) correspond à la réponse de `ia_naive`

```
def ia_minmax(p,l,f,j,n) :
    if n == 1 :
        return ia_naive(p,l,f,j)
    else :
        cases = cases_jouables(p,l,j)
        if j == 1 :
            V = -10001
            for c in cases :
                P = poser_jeton(p,c,j)
                L = cases_frontieres(p,l,c)
                c,val = ia_minmax(P,L,f,-1,n-1)
                if val > V :
                    next,V = c,val
            return next,V
        else :
            V = 10001
            for c in cases :
                P = poser_jeton(p,c,j)
                L = cases_frontieres(p,l,c)
                c,val = ia_minmax(P,L,f,1,n-1)
                if val < V :
                    next,V = c, val
            return next,V
```

18. Les modifications à apporter concernent les cas où le joueur actif ne peut pas jouer. Deux cas sont à envisager :

- soit $n \geqslant 2$, on n'a pas encore exploré toute la « profondeur » de le recherche donc le joueur passe simplement son tour et on recommence un trou avec le même joueur, en passant le booléen `tp` à la valeur `True`.
- soit $n = 1$ et on est à la « profondeur » de recherche. On distingue à nouveau deux cas :
  - soit le booléen `tp` contient la valeur `True`, ce qui signifie que le joueur actif joue pour une deuxième fois consécutivement et donc que l'autre joueur n'a pas non plus pu jouer dans cette position, les deux joueurs sont bloqués et la partie est finie. On évalue le score du plateau et on décrète un joueur vainqueur (ou éventuellement match nul) ; on renvoie cependant toujours un couple pour que les appels récursifs précédents puissent fonctionner (pendant la phase de remontée)
  - soit le booléen `tp` contient la valeur `False`, ce qui signifie que le joueur actif est bloqué mais joue pour la première fois donc on doit vérifier si l'autre joueur peut continuer à joueur. On relance donc une dernière fois la fonction avec l'autre joueur (sans changer la valeur de n car la fonction bloquerait avec $n = 0$) mais en basculant le booléen `tp` sur `True`

Pour utiliser cette fonction, il faut l'appeler en prenant comme valeur initiale `tp=False`.

```
def ia_minmax(p,l,f,j,n,tp) :
    cases = cases_jouables(p,l,j)
    if len(cases) != 0 :
        if n == 1 :
            return ia_naive(p,l,f,j)
        else :
            if j == 1 :
                V = -10001
                for c in cases :
                    P = poser_jeton(p,c,j)
                    L = cases_frontieres(p,l,c)
                    c,val = ia_minmax(P,L,f,-1,n-1,False)
                    if val > V :
                        next,V = c,val
```

```
                return next,V
            else :
                V = 10001
                for c in cases :
                    P = poser_jeton(p,c,j)
                    L = cases_frontieres(p,l,c)
                    c,val = ia_minmax(P,L,f,1,n-1,False)
                    if val < V :
                        next,V = c, val
                return next,V
    else :
        if n > 1 :
            return ia_minmax(p,l,f,-j,n-1,True)
        else :
            if tp :
                s = score(p)
                if s > 0 :
                    return None,10000
                elif s < 0 :
                    return None,-10000
                else :
                    return None,0
            else :
                return ia_minmax(p,l,f,-j,1,True)
```

19. On calcule juste la somme avec deux boucles :

```
def encode(p) :
    s = 0
    for i in range(8) :
        for j in range(8) :
            s += (1+p[i][j])*3**(8*i+j)
    return s
```

20. Il faut retrouver l'écriture en base 3 de la valeur de `e` et remplir le plateau en conséquence : on parcourt les 64 case du plateau avec une variable `n`, les valeurs de i et j sont les quotient et reste de la division euclidienne de `n` par 3.

```
def decode(e) :
    p = [[0 for j in range(8)] for i in range(8)]
    for n in range(64) :
        i,j = n//8,n%8
        p[i][j] = e%3-1
        e = e//3
    return p
```

21. $\texttt{e} \leqslant \displaystyle\sum_{i=1}^{7}\sum_{j=1}^{7} 2\times 3^{8i+j} = 2\sum_{i=1}^{7}\left(3^{8i}\times\frac{3^8-1}{2}\right) = (3^8-1)\frac{3^{8\times 8}-1}{3^8-1} = \boxed{3^{64}-1}$

22. On utilise une sous-requête pour déterminer la profondeur maximale demandée :

```
SELECT caseX,caseY FROM plateaux WHERE codePlateau=3917277946
    AND joueurActif=1 WHERE profondeur = (SELECT MAX(profondeur) FROM plateaux
    WHERE codePlateau=3917277946 AND joueurActif=1)
```

23. On joint les tables `joueurs` et `parties` avant de sélectionner la joueuse et on calcule la moyenne de ses scores sur les parties où elle jouait en tant que joueur J_1 :

```
SELECT AVG(score) FROM parties AS p JOIN joueurs AS j
    ON j.idJoueur = p.joueurs_idJ1 WHERE nomComplet='Beth Harmon'
```

24. Ici, on va utiliser `LIMIT` pour ne garder que les trois premiers résultats (on aurait aussi pu le faire pour la question **22**) : lorsque les noirs gagnent, la valeur du score est négative donc ce sont les 3 plus petits score qu'il faut sélectionner (donc on classe dans l'ordre croissant).

```
SELECT nomComplet,score FROM joueurs AS j JOIN parties AS p
    ON p.joueurs_idGagnant = j.idJoueur ORDER BY score LIMIT 3
```