

DS3 PC* : Jeux

Démonstration.

□

Consignes

Dans ce devoir :

- Toutes les manipulations de slicing sont autorisées.
- Les réponses sont à écrire sur le bulletin réponse. On remettra les pages dans l'ordre sans agrafe sans en omettre une seule (même si elles sont vides) et on indiquera bien son nom et prénom sur TOUTES LES PAGES.

Description du jeu d'Othello

Othello (aussi connu sous le nom Reversi) est un jeu de société combinatoire abstrait opposant deux joueurs.

Il se joue sur un tablier unicolore de 64 cases, 8 sur 8, appelé *othellier*. Les joueurs disposent de 64 pions bicolores, noirs d'un côté et blancs de l'autre. En début de partie, quatre pions sont déjà placés au centre de l'othellier : deux noirs, en e4 et d5, et deux blancs, en d4 et e5. Chaque joueur, noir et blanc, pose l'un après l'autre un pion de sa couleur sur l'othellier selon des règles précises. Le jeu s'arrête quand les deux joueurs ne peuvent plus poser de pion. On compte alors le nombre de pions. Le joueur ayant le plus grand nombre de pions de sa couleur sur l'othellier a gagné.

Noir commence toujours la partie. Puis les joueurs jouent à tour de rôle, chacun étant tenu de capturer des pions adverses lors de son mouvement. Si un joueur ne peut pas capturer de pion(s) adverse(s), il est forcé de passer son tour. Si aucun des deux joueurs ne peut jouer, ou si l'othellier ne comporte plus de case vide, la partie s'arrête. Le gagnant en fin de partie est celui qui possède le plus de pions.

La capture de pions survient lorsqu'un joueur place un de ses pions à l'extrémité d'un alignement de pions adverses contigus et dont l'autre extrémité est déjà occupée par un de ses propres pions. Les alignements considérés peuvent être une colonne, une ligne, ou une diagonale. Si le pion nouvellement placé vient fermer plusieurs alignements, il capture tous les pions adverses des lignes ainsi fermées. La capture se traduit par le retournement des pions capturés. Ces retournements n'entraînent pas d'effet de capture en cascade : seul le pion nouvellement posé est pris en compte.

Au début, Noir peut jouer en c4,d3,e6 ou f5. Dans l'exemple de la figure 2, il joue en d3 puis Blanc joue en c3.

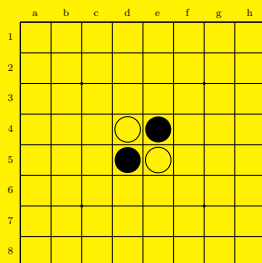


FIGURE 1 – Début

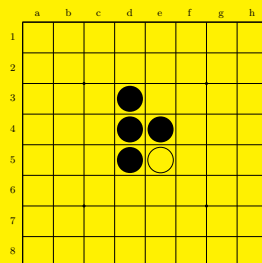


FIGURE 2 – Noir joue en d3

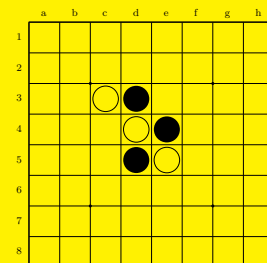


FIGURE 3 – Blanc joue en c3

Dans la situation de la figure 4, les noirs prennent d'abord une ligne puis les blancs récupèrent simultanément une ligne et une colonne.

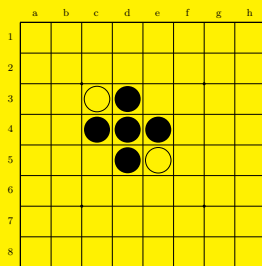


FIGURE 4 – Noir joue en c4

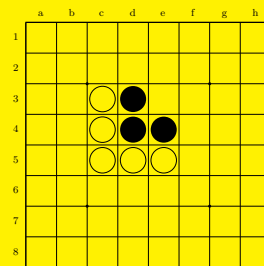


FIGURE 5 – Blanc joue en c5

L'objectif de ce devoir est d'implémenter l'algorithme *min-max* avec heuristique et profondeur maximale pour le jeu Othello.

Remarque. Chez les joueurs d'Othello, il est d'usage de représenter les cases de l'othellier en donnant le numéro de colonne (une lettre) suivi du numéro de ligne (un chiffre) comme dans d3. Mais nous ne respectons pas cette habitude : nous indiquons les cases par un tuple d'entiers (ligne, colonne) et nous commençons à compter à partir de zéro et non un. Ainsi la case d3 est représentée par ses coordonnées (2,3).

1 Prise en main

On se donne les constantes suivantes :

```
1 B="B"#black
2 W="W"#white
3 N=8#nb de lignes
```

Une grille du jeu est représentée par une chaîne de N^2 caractères, `B` désignant une case occupée par un pion noir, `W` une case occupée par un pion blanc et `.` une case libre. Pour remplir cette chaîne de caractère, on met bout à bout toutes les lignes de l'othellier.

Par exemple, voici comment est représentée la grille de départ de la figure 1 :

```
1 " .....WB.....BW....."
```

Il y a bien 3 lignes vides (3×8 fois le symbole `.`) puis encore 3 cases vides (3 fois le symbole `.`) avant le premier `W`.

Il est intéressant de représenter les grilles par des chaînes de caractères plutôt que des listes de listes (matrices) parce que les chaînes de caractères sont immutables et peuvent donc être stockées dans un historique de la partie sans risque de modification accidentelle.

On dispose des fonctions `list2string(l:list[str])>str` et `string2list(s:str)>list[str]`. La première transforme une liste de caractères en chaîne de caractères et la seconde fait l'inverse. On ne demande pas d'écrire ces fonctions.

```
1 s=" .....WB.....BW....."
2 print(string2list(s[25:40])) #pas de place pour s en entier
3 print(list2string(string2list(s)))
```

```
[ '.', '.', 'W', 'B', '.', '.', '.', '.', '.', '.', 'B', 'W', '.', '.', '.']
.....WB.....BW.....
```

Q.1 Écrire la fonction `pos2coords (x:int)->tuple[int,int]` qui prend en paramètre une *position* (un nombre entre 0 et $N^2 - 1$) et renvoie un couple de *coordonnées*. Par exemple, le premier `B` de la grille initiale est sur la case e4 de l'othellier, ce qui correspond à la position 28 et au tuple de coordonnées (3,4).

```
1 pos2coords(10) ,pos2coords(28)
```

```
((1, 2), (3, 4))
```

Q.2 Écrire la fonction `coords2pos (i:int,j:int)->int` qui réalise l'opération inverse de la précédente.

```
1 coords2pos(1,2)
```

```
10
```

Q.3 Écrire la fonction `get (g:str,i:int,j:int)->str: return g[coords2pos(i,j)]` qui prend en paramètres une grille `g` représentée par une chaîne de caractères, un numéro de ligne `i` et de colonne `j` et renvoie le pion trouvé à cette coordonnée.

```
1 g=" .....WB.....BW....."
2 get(g,3,4)
```

```
'B'
```

Q.4 Écrire la fonction `set_str(g:str,i:int,j:int,c:str)` qui met la valeur `c` aux coordonnées (i,j) . Cette opération renvoie donc une nouvelle chaîne de caractère représentant une nouvelle situation de jeu. Cette fonction outil sera utilisée pour fabriquer des grilles tests.

```
1 g=" .....WB.....BW....."
2 print(set_str(g,3,4,W))
```

```
.....WW.....BW.....
```

Q.5 Écrire la fonction `other(J)` qui prend en paramètre un joueur (blanc ou noir) et renvoie l'autre joueur (noir ou blanc).

```
1 other(W),other(B)
```

```
('B', 'W')
```

2 États du jeu

Un *état* du jeu est un tuple (couleur,grille) dont le premier membre indique qui doit jouer (blanc ou noir) et le second est une chaîne de caractères représentant l'othellier.

Q.6 Écrire la fonction `init()->tuple[str,str]` qui renvoi l'état initial.

```
1 init()
```

```
('B', ' .....WB.....BW..... ')
```

Q.7 Écrire la fonction `sort(c1:tuple[int,int],c2:tuple[int,int])->tuple[tuple[int,int],tuple[int,int]]` qui prend en paramètres deux tuples de coordonnées et renvoie d'abord le plus petit puis le plus grand.

```
1 print(sort((1,3),(0,4)))
2 print(sort((1,3),(1,2)))
3 print(sort((1,3),(2,0)))
```

```
((0, 4), (1, 3))
((1, 2), (1, 3))
((1, 3), (2, 0))
```

Q.8 On veut déterminer si toutes les cases situées *strictement* entre deux cases c_1 et c_2 données par leurs coordonnées ont la même valeur v . La droite qui joint c_1 à c_2 doit posséder une direction valide (horizontale, verticale ou à 45 degrés par rapport à l'horizontale) et c_1 et c_2 ne doivent pas être adjacentes. La valeur des cases c_1 et c_2 n'est pas prise en compte.

Pour le code partiel de la fonction

`line (g:str,c1:tuple[int,int],c2:tuple[int,int],v:str)->bool` ci-dessous, écrire sur le bulletin réponse les lignes L10, L11, L20, L22, L23, L28, L29, L30, L31 EN INDIQUANT LE NUMÉRO DE LIGNE À CHAQUE FOIS.

```

1  def _row_ok(g,c1,c2,v): # c1<=c2; c1 et c2 sur une mme ligne
2      if abs(c1[1]-c2[1])<=1: return False
3      for j in range(c1[1]+1,c2[1]):
4          if get(g,c1[0],j)!=v: return False
5      return True
6
7  def _col_ok(g,c1,c2,v):
8      if abs(c1[0]-c2[0])<=1: return False
9      for i in range(c1[0]+1,c2[0]):
10         ??
11     return ??
12
13 def _diag_ok(g,c1,c2,v):
14     if abs(c1[1]-c2[1])<=1 and abs(c1[0]-c2[0])<=1: return False
15     for k in range(1,c2[1]-c1[1]):
16         if get(g,c1[0]+k,c1[1]+k)!=v: return False
17     return True
18
19 def _anti_diag_ok(g,c1,c2,v):
20     if ??: return False
21     for k in range(1,c1[1]-c2[1]):
22         ??
23     return ??
24
25 def line (g:str,c1:tuple[int,int],c2:tuple[int,int],v:str)->bool:
26     c1,c2=sort(c1,c2)
27     if c1[0]==c2[0]: return _row_ok(g,c1,c2,v)
28     if c1[1]==c2[1]: return ??
29     if c2[0]-c1[0] == c2[1]-c1[1] : return ??
30     if ?? : return ??
31     return ??

```

Exemple d'appel :

```

1 l = ['.']*N**2
2 g = list2string(l)
3 g=set_str(g,5,1,W);g=set_str(g,4,2,B)
4 g=set_str(g,3,3,B);g=set_str(g,2,4,B);g=set_str(g,1,5,W)
5 print(line(g,(5,1),(1,5),B))
6 g=set_str(g,2,4,['.'])
7 print(line(g,(5,1),(1,5),B))
8 print(line(g,(5,1),(4,2),B))

```

True
False
False

Démonstration. Puisque $c_1 \leq c_2$ (du fait de l'appel à `sort`) :

- Pour les lignes, on emprunte la direction (1,0) dans la boucle `for` à partir de `c1` ;
- Pour les colonnes, on emprunte la direction (0,1) dans la boucle `for` à partir de `c1` ;
- Pour la diagonale, on emprunte la direction (1,1) dans la boucle `for` à partir de `c1` ;

- Pour l'anti-diagonale, on emprunte la direction $(1, -1)$ dans la boucle `for` à partir de `c1` ou la direction $(-1, 1)$ à partir de `c2` ;

□

Q.9 On donne le code incomplet de la fonction `can_play(p:str,g:str,c:tuple[int,int])->bool` qui prend en paramètre un joueur (blanc ou noir), une grille représentée par une chaîne de caractères, et une case donnée par ses coordonnées. La fonction indique par un booléen si le joueur peut déposer un de ses pions sur la case.

```

1 def can_play(p:str,g:str,c:tuple[int,int])->bool:
2     o = other(p)#autre couleur
3     if g[coords2pos(*c)]!='.':return ??
4     for j in range(0,N):#analyse de la ligne de c
5         if j!=c[1] and line(g,c,(c[0],j),o) and get(g,c[0],j)==p:return True
6     for i in range(0,N):#analyse de col
7         ??
8         i,j=c[0]-1,c[1]-1
9         while i>=0 and j>=0:#diagonale vers le haut
10            if line(g,c,(i,j),o) and get(g,i,j)==p: return True
11            i,j=i-1,j-1
12        i,j=??
13        ??
14            ??
15            ??
16        ??
17        ??
18            ??
19            ??
20        ??
21        ??
22            ??
23            ??
24    return ??

```

Écrire les lignes L7 puis L12 à L24 sur le bulletin réponse.

Exemple d'appels :

```

1 _,g=init()
2 g=set_str(g,3,3,B)
3 g=set_str(g,2,3,B)
4 #g est dans la situation de la figure 2
5 can_play(W,g,(5,3)), can_play(W,g,(3,2)),\
6 can_play(W,g,(2,2)),can_play(B,g,(2,2))

```

(False, False, True, False)

La fonction `set_str` est pratique pour les tests. Toutefois, les chaînes de caractères n'étant pas mutables, elle engendre des coûts non négligeables si on l'utilise de façon répétée. Pour passer d'un état du jeu à un autre, on préfère transformer la grille courante (qui est donnée sous forme de `str`) en une matrice de caractères. On effectue les modifications voulues sur la matrice puis on transforme finalement cette dernière en une nouvelle chaîne de caractères.

Q.10 Écrire la fonction `str2mat(g:str)->list[list[str]]` qui transforme une grille donnée sous forme de chaîne de caractères en une matrice de caractères.

```

1 _,g = init()
2 str2mat(g)

```

```

[['.', '.', '.', '.', '.', '.', '.', '.'],
 ['.', '.', '.', '.', '.', '.', '.', '.'],
 ['.', '.', '.', '.', '.', '.', '.', '.'],
 ['.', '.', '.', 'W', 'B', '.', '.', '.'],
 ['.', '.', '.', 'B', 'W', '.', '.', '.'],
 ['.', '.', '.', '.', '.', '.', '.', '.'],
 ['.', '.', '.', '.', '.', '.', '.', '.'],
 ['.', '.', '.', '.', '.', '.', '.', '.']]

```

Q.11 Écrire la fonction `mat2str(m:list[list[str]])->str` qui réalise l'inverse de la précédente.

```

1 _,g = init()
2 m=str2mat(g)
3 mat2str(m)

```

```
'.....WB.....BW.....'
```

Q.12 Écrire la fonction

`set_mat(m:list[list[str]],c1:tuple[int,int],c2:tuple[int,int],p:str)->None` qui prend en paramètres une matrice représentant l'othellier, deux cases et un joueur. La fonction met toutes les cases strictement comprises entre c_1 et c_2 de la couleur du joueur. La matrice est modifiée au cours de cette opération. On suppose que la droite joignant les 2 cases est horizontale, verticale ou oblique à 45 degrés.

```

1 _,g=init()
2 m = str2mat(g)
3 print(can_play(B,g,(2,3)))
4 print(line(g,(2,3),(4,3),W))
5 set_mat(m,(2,3),(4,3),B)
6 m

```

```
True
True
```

```

[['.', '.', '.', '.', '.', '.', '.', '.'],
 ['.', '.', '.', '.', '.', '.', '.', '.'],
 ['.', '.', '.', '.', '.', '.', '.', '.'],
 ['.', '.', '.', 'B', 'B', '.', '.', '.'],
 ['.', '.', '.', 'B', 'W', '.', '.', '.'],
 ['.', '.', '.', '.', '.', '.', '.', '.'],
 ['.', '.', '.', '.', '.', '.', '.', '.'],
 ['.', '.', '.', '.', '.', '.', '.', '.']]

```

Q.13 Écrire la fonction `play(p:str,g:str,c:tuple[int,int])->tuple[str,str]` qui prend en paramètres un joueur, un othellier représenté par une chaîne de caractères et une case représentée par ses coordonnées. La fonction s'assure que la case est jouable pour le joueur au moyen d'une assertion puis attribue la case au joueur et effectue toutes les mises à jour nécessaires. Elle retourne le nouvel état du jeu, c'est-à-dire un tuple formé de l'autre joueur et de la nouvelle configuration de la grille.

```

1 p1,g1=init()
2 print(p1)
3 c=(2,3)
4 p2,g2=play(p1,g1,c)
5 print(p2)
6 str2mat(g2)

```

```
B
W
```

```

[['.', '.', '.', '.', '.', '.', '.', '.'],
 ['.', '.', '.', '.', '.', '.', '.', '.'],
 ['.', '.', '.', 'B', '.', '.', '.', '.'],
 ['.', '.', '.', 'B', 'B', '.', '.', '.'],
 ['.', '.', '.', 'B', 'W', '.', '.', '.'],
 ['.', '.', '.', '.', '.', '.', '.', '.'],
 ['.', '.', '.', '.', '.', '.', '.', '.'],
 ['.', '.', '.', '.', '.', '.', '.', '.']]

```

- Q.14** Écrire la fonction `next_pos(p:str,g:str)->list[tuple[int,int]]` qui prend en paramètres un joueur et une grille (c.a.d un état) et renvoie la liste des cases où le joueur peut placer son pion.

```
1 p,g=init()
2 next_pos(p,g)
```

```
[(2, 3), (3, 2), (4, 5), (5, 4)]
```

AJOUT À POSTERIORI : si `p` ne peut pas jouer, la règle dit que c'est à l'autre joueur de procéder au cas où il pourrait lui-même jouer. Sinon, il n'y a pas de prochain état de jeu.

- Q.15** Écrire la fonction `next_states(p:str,g:str)->list[tuple[str,str]]` qui retourne la liste des états accessibles depuis l'état `(p,g)`.

```
1 p,g=init()
2 next_states(p,g)
```

```
[('W', '.....B.....BB.....BW.....'),
 ('W', '.....BBB.....BW.....'),
 ('W', '.....WB.....BBB.....'),
 ('W', '.....WB.....BB.....B.....')]
```

Un état *terminal* est un état à partir duquel aucun des joueurs ne peut jouer.

- Q.16** Écrire le fonction `end(g:str)->bool` qui renvoie `True` si et seulement si la grille `g` ne permet à aucun joueur de jouer.
- Q.17** Écrire la fonction `outcome(p:str,g:str)->str` qui prend en paramètres un état terminal et retourne le nom du joueur gagnant s'il en existe un et `None` en cas de match nul. Si l'état n'est pas terminal, une erreur est soulevée.

```
1 p,g=init()
2 m = str2mat(g)
3 for i in range(N):
4     for j in range(N):
5         if j>=i-2:m[i][j]=W
6         else:m[i][j]=B
7 outcome(W,mat2str(m))
```

```
'W'
```

3 Attracteur

On considère un jeu d'accessibilité $G = ((S, A), S_1, S_2)$ (non nécessairement Othello) à nombre d'états finis où S_1 (états du joueur J_1) et S_2 (états de J_2) forment une partition de l'ensemble S des états. On suppose connu F_1 , l'ensemble des états gagnants pour le joueur J_1 .

- Q.18** Donner l'algorithme qui calcule l'attracteur du joueur 1.
- Q.19** Justifier que l'algorithme termine.

On revient à Othello :

- Q.20** Dans cette question seulement, on suppose que le nombre de cases $N \times N$ de la grille est arbitrairement grand. Parmi les configuration pleines (c.a.d sans case vide), combien y en a-t-il qui sont gagnants pour les noirs ? En donner un équivalent simple en fonction de 4, π et N .
On ne demande pas de distinguer entre les grilles pleines effectivement accessibles depuis la situation de départ et les autres.
- Q.21** Considérons la situation suivante : les blancs viennent de placer un pion en case c . Dans la direction ouest, il y a depuis c , exactement $k_1 + 1$ cases blanches contigües (en comptant c), au nord-ouest $k_2 + 1$, au nord $k_3 + 1$, au nord-est $k_4 + 1$, à l'est $k_5 + 1$, au sud-est $k_6 + 1$, au sud $k_7 + 1$ et au sud-ouest $k_8 + 1$ avec $\forall i, k_i \geq 0$.
Combien y a-t-il d'états dont l'état courant est le successeur en un coup ?

Démonstration. On construit la suite $(A_i(F_1))_{i \in \mathbb{N}}$ telle que :

- $A_0(F_1) = F_1$ (ensemble des sommets gagnants pour J_1 en 0 coup)
- Pour $n \in \mathbb{N}$, $A_{n+1}(F_1) = A_n(F_1) \cup B_n^1 \cup C_n^1$ où
 - $B_n^1 = \{s \in S_1 \mid \exists(s, s') \in A, s' \in A_n(F_1)\}$: ensemble des sommets de S_1 qui ont un voisin dans $A_n(F_1)$
 - $C_n^1 = \{s \in S_2 \mid \forall(s, s') \in A, s' \in A_n(F_1)\}$: ensemble des sommets de S_2 dont tous les voisins sont dans $A_n(F_1)$.
- $A_n(F_1)$ est l'ensemble des sommets gagnants en au plus n coups pour J_1 .

On calcule donc les A_i jusqu'à ce que $A_{i+1} = A_i$.

La suite A_i est croissante dans l'ensemble des états. Celui-ci étant fini, la suite est stationnaire apcr. Il est donc naturel de s'arrêter en cas d'égalité de deux états successifs. On vient de voir que cela arrive forcément en un temps fini. D'où la terminaison.

Supposons d'abord que le nombre N^2 de cases soit paire. On pose $N^2 = 2M$. Il y a au total 2^{2M} grilles pleines. Parmi celles-ci, il y a $\binom{2M}{M}$ choix de matchs nuls. Donc le nombre de grilles gagnantes pour les noirs est la moitié de $2^{2M} - \binom{2M}{M}$

Or, en utilisant Stirling, on obtient l'approximation suivante pour le nombre de matchs nuls :

$$\binom{2M}{M} = \frac{(2M)!}{M! M!} = \frac{(2M)!}{(M!)^2} \sim \frac{\left(\frac{2M}{e}\right)^{2M} \sqrt{2 \times 2\pi M}}{\left(\frac{M}{e}\right)^{2M} \sqrt{2\pi M}^2} = \frac{4^M}{\sqrt{\pi M}}$$

On a donc environ

$$\frac{1}{2} \times (2^{2M} - 4^M \times \frac{1}{\sqrt{\pi M}}) = 4^M \times \frac{1 - \frac{1}{\sqrt{\pi M}}}{2} = 2^{N^2} \times \frac{1 - \frac{\sqrt{2}}{N\sqrt{\pi}}}{2} \underset{\text{apcr}}{>} 2^{N^2-2}$$

grilles gagnantes pour les noirs.

Le cas avec N impair est plus simple puisqu'il n'y a pas de match nul pour les grilles pleines : c'est $\frac{1}{2} \times 2^{N^2} = 2^{N^2-1}$.

Remarque. Parmi le grand nombre de grilles pleines gagnées par les noirs identifiées ci-dessus, reconnaître quelles grilles sont accessibles effectivement depuis la position initiale permettrait de restreindre la taille du point de départ de l'attracteur. C'est sans doute hors de portée pour un devoir de CPGE.

A ceci s'ajoute encore une difficulté. Il existe des parties qui se terminent par une victoire sans que la grille soit pleine. L'exemple ci-dessous est tiré de Wikipedia :

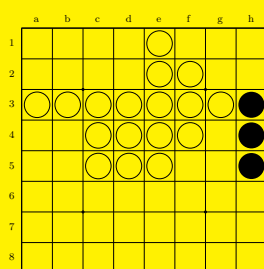


FIGURE 6 – Hassan 3 – 61 Vers-tuyft J. (European Grand Prix Ghent 2017)

Situation : les blancs viennent de placer un pion en case c. Dans la direction ouest, il y a depuis c, exactement $k_1 + 1$ cases blanches contigües (en comptant C), au nord-ouest $k_2 + 1$, au nord $k_3 + 1$, au nord-est $k_4 + 1$, à l'est $k_5 + 1$, au sud-est $k_6 + 1$, au sud $k_7 + 1$ et au sud-ouest $k_8 + 1$.

Un état précédant l'actuel possède forcément un othellier différent puisque les blancs ont placé un pion. S'il y a eu un changement à l'ouest de C, c'est que C était vide au coup d'avant, et qu'il y avait plusieurs (au moins une) cases noires contigües depuis c en direction de l'ouest suivies par une

case blanche. La case à la distance k_1 de c était forcément blanche sinon on n'aurait pas $k_1 + 1$ blancs contigus depuis C sur la direction ouest dans l'état actuel. Ainsi on se retrouve au coup d'avant avec une situation de la forme :

```
obligatoirement blanche   Noire si changement à l'ouest
|                             |
|                             |
| WWWWWWWWWWWWWWWWWWWWWBBB.
|                             |
|                             | c
|----->
| k_1 case
```

La case blanche la plus proche de C dans la direction ouest peut occuper k_1 position.

Le nombre de situations possibles est donc $\max(1, k_1)$ (ce qui inclut le cas où $k_1 = 0$).

Cette analyse est valable dans toutes les directions. Ainsi, le nombre de situations antérieures possibles si les blancs jouent en case c est $\prod_{i=1}^8 \max(1, k_i)$ avec au moins un des $k_i > 1$ (puisque les blancs viennent de placer un pion en case c).

Ainsi, à partir d'un état du jeu, pour trouver les états immédiatement antérieurs pour les blancs, il faut parcourir toutes les cases blanches et trouver pour chacune les k_1, \dots, k_8 puis contruire les $\prod_i k_i$ états antérieurs. Même chose pour les noirs. On comprend que le calcul de l'attracteur n'est pas gagné! \square

4 Algorithme min-max

Les questions Q20 et Q21 nous ont convaincu que le calcul de l'attracteur pour le jeu d'Othello est difficile. Tentons une autre approche. L'objectif est ici l'évaluation du *score min-max pour le joueur noir* dans le jeu d'Othello.

On adopte pour la question suivante le point de vue du cours : un état quelconque rapporte 1 point si les noirs disposent d'une stratégie gagnante depuis l'état courant, -1 si ce sont les blancs qui disposent d'une stratégie gagnante et 0 sinon.

Si l'état courant est terminal, alors il rapporte respectivement 1, 0 ou -1 s'il correspond à une victoire des noirs, un match nul ou une victoire des blancs.

Pour un état courant e non-terminal :

- s'il appartient aux noirs, alors son score est le maximum des scores accessibles depuis e ;
- s'il appartient aux blancs, alors son score est le minimum des scores accessibles depuis e ;

Q.22 Écrire la fonction récursive `minmax(e:tuple[str,str])>int` qui calcule le score minmax de l'état `e` du point de vue du joueur noir. On rappelle qu'un état est un tuple (joueur, chaîne de caractères représentant une grille).

L'appel `minmax(init())` prend beaucoup de temps et nous sommes contraints de l'interrompre.

Pour limiter le temps d'exécution, nous implantons maintenant l'algorithme min-max dans sa version avec profondeur maximum et heuristique.

On s'intéresse d'abord à l'heuristique. Pour cela, nous avons besoin de quelques fonctions de comptage :

Q.23 Écrire la fonction `compter(g:str)->tuple[int,int]` qui prend en paramètre une grille et renvoie un tuple formé du nombre de `B` et du nombre `W`.

```
1 p1,g1=init()
2 c=(2,3)
3 p2,g2=play(p1,g1,c)#situation de la figure2
4 compter(g2)
```

```
(4, 1)
```

Q.24 Écrire la fonction `bords(g:str)->tuple[int,int]` qui prend en paramètre une grille et renvoie un tuple formé du nombre de `B` et du nombre `W` situés sur les bords (coins compris).

Q.25 Écrire la fonction `coins(g:str)->tuple[int,int]` qui compte le nombre de **B** et du nombre **W** situés sur les coins.

Quelques parties d'Othello suffisent pour se convaincre qu'il est intéressant de placer ses pions sur les bords de l'othellier et encore plus sur un des 4 coins : ces cases ne pourront plus changer de couleur. L'heuristique consiste à compter un point pour chaque case **B** et -1 les cases **W**. Si une case est sur un bord, on la compte 3 fois de plus et si elle est sur un coin, on la compte 10 fois de plus.

Un **W** sur un bord mais pas sur un coin rapporte donc un score de $-1 + (-3) = -4$ et un **B** sur un coin amène un score de $1 + 3 + 10 = 14$.

Q.26 Écrire la fonction `h(g:str)->int` qui implémente cette heuristique.

Une heuristique positive signifie donc que, pour cette grille, les noirs sont plutôt avantageés. Et une valeur négative indique le contraire.

L'algorithme du min-max avec profondeur d maximum et heuristique fonctionne comme suit :

- Si un état est terminal, pas de changement par rapport à l'algorithme précédent.
- Sinon, de deux choses l'une :
 - si on est à la profondeur d , on renvoie le score attribué à l'état par l'heuristique.
 - sinon, on renvoie le maximum des scores des états accessibles depuis l'état courant si l'état courant appartient au joueur noir, et le minimum sinon.

Q.27 Écrire la fonction

`minmax_hd(e:tuple[str,str],h:Callable[[str],int],d:int)->int` qui prend en paramètre un état, une heuristique et une profondeur. La fonction implémente le calcul du score min-max avec heuristique et profondeur maximum pour le joueur noir. La profondeur est décémentée dans les appels internes : si elle vaut zéro pour un état non terminal, le calcul du score par l'heuristique est déclenché.

```
1 import time
2 t1 = time.time()
3 score = minmax_hd(init(),h,7)
4 t2 = time.time()
5 print("score={} obtenu en {}s".format(score,t2-t1))
```

```
score=5 obtenu en 43.181448459625244s
```

Après exploration de l'arbre de décision jusqu'à la profondeur 7, on déduit que la situation semble légèrement favorable aux noirs.

Remarque. On le voit, même en fixant une profondeur maximale d'exploration raisonnable (7 ici), l'algorithme min-max prend beaucoup de temps à s'exécuter. Pour faire mieux, on peut décider de ne pas explorer certaines branches de l'arbre lorsqu'on a un moyen de savoir qu'elles n'amélioreront pas notre connaissance du score final.

Dans le calcul d'un minimum (resp. maximum), si on sait par avance que la valeur de ce minimum sera inférieure (resp. supérieure) à celle du maximum (resp. minimum) calculé au niveau supérieur, on peut s'abstenir d'explorer les descendants de l'état courant.

C'est le principe de l'algorithme du *min-max avec élagage alpha-beta* qui gère deux paramètres α, β supplémentaires par rapport au précédent. La valeur α borne le minimum du score en cours de calcul (il ne pourra pas descendre en dessous de α) et β borne le maximum (le score ne pourra pas monter au-dessus de β). Mais cet algorithme est hors-programme en ITC.