TD : JEU DE MORPION

Dans la grille de taille $n \times n$ du jeu de morpion, les cases sont numérotées en parcourant les lignes, comme ci-dessous pour $n = 3$.

$$\begin{bmatrix} 0 & 1 & 2 \\ 3 & 4 & 5 \\ 6 & 7 & 8 \end{bmatrix}$$

Chaque joueur coche tour à tour une case numérotée i avec un symbole S (O pour J_0, X pour J_1), dans le but de remplir entièrement une ligne, une colonne ou une diagonale. Il a ainsi $2n + 2$ possibilités de réussir.

Pour qu'un ordinateur puisse vérifier si un joueur a gagné, il doit connaître toutes les « rangées » valides (lignes, colonnes et diagonales).

I) IMPLÉMENTATION DU JEU EN PYTHON

1. Écrire une fonction Python `rangees(n)` qui renvoie la liste de toutes les rangées gagnantes possibles. Chaque rangée sera elle-même représentée par la liste (ou le tuple) des numéros de ses cases.

Tester :
```
>>> rangees(3)
[[0, 1, 2], [3, 4, 5], [6, 7, 8],
 [0, 3, 6], [1, 4, 7], [2, 5, 8],
 [0, 4, 8], [2, 4, 6]]
```

On représente chaque coup joué sous la forme d'une chaîne de caractères. Par exemple la chaîne : `'XO..O....'` représente la grille ci-contre : $\begin{bmatrix} X & O & \\ & O & \\ & & \end{bmatrix}$

2. Écrire une fonction `partie_finie(n,c)` qui reçoit la taille `n` et la chaîne `c` de la partie, et qui renvoie un tuple d'informations :

- `(True,0)` si la partie est terminée et que J_0 a gagné.
- `(True,1)` si la partie est terminée et que J_1 a gagné.
- `(True,None)` si la partie est terminée (grille pleine) mais qu'il y a match nul.
- `(False,None)` si la partie n'est pas encore terminée.

Différentes chaînes gagnantes ont été définies au début de votre fichier source afin de tester votre fonction :

Vérifier :
```
>>> partie_finie(3,c_victoire_J0_ligne)
(True, 0)
>>> partie_finie(3,c_victoire_J1_colonne)
(True, 1)
>>> partie_finie(3,c_victoire_J0_diag)
(True, 0)
>>> partie_finie(3,c_match_nul)
(True, None)
>>> partie_finie(3,c_en_cours)
(False, None)
```

II) GRAPHE DU JEU

Pour créer le graphe du jeu, il faut explorer toutes les parties possibles depuis une grille donnée et les enregistrer dans un dictionnaire. On représente un état du jeu par un doublet `(s,J)`, où `s` est une chaîne de caractères et `J` un entier valant 0 ou 1. Par exemple le doublet ci-dessous :

```
('XO..O....',0)
```

Représente la grille ci-contre : $\begin{bmatrix} X & O & \\ & O & \\ & & \end{bmatrix}$ et signifie que c'est le joueur J_0 qui va jouer.

On définit les variables globales :

```
joueurs = {'O' : 0, 'X' : 1}
symboles = {0 : 'O', 1 : 'X'}
etat_initial = ('.'*9, 0)       #grille vide et J0 commence
```

3. Écrire la fonction `suivants(etat)` recevant une position et renvoyant la liste des positions suivantes possibles.
Indication : pour créer la chaîne de caractères obtenue à partir d'une chaîne de caractères s en remplaçant le caractère d'indice `i` par un caractère c on peut utiliser les tranches et la concaténation : $s\_new = \underbrace{s[:i]}_{\text{tranche de s de l'indice 0 à i−1}} + \underbrace{c}_{\text{caractère = chaine de longueur 1}} + \underbrace{s[i+1:]}_{\text{tranche de s à partir de l'indice i+1}}$

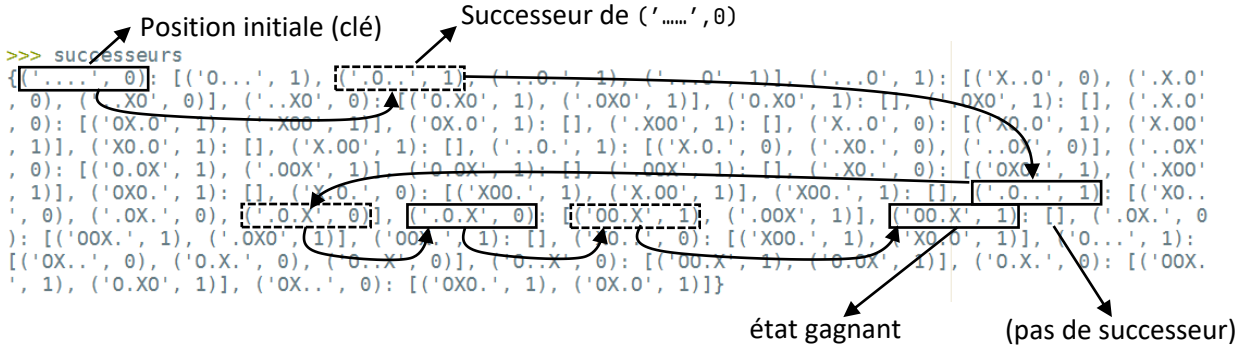
Vérifier :
```
>>> suivants(etat_initial)
[('O.......', 1), ('.O......', 1), ('..O.....', 1),
 ('...O....', 1), ('....O....', 1), ('.....O...', 1),
 ('......O..', 1), ('.......O.', 1), ('........O', 1)]

>>> suivants(('OX.XOO.X.',0))
[('OXOXOO.X.', 1), ('OX.XOOOX.', 1), ('OX.XOO.XO', 1)]
```

Le graphe du jeu sera codé par :

- un dictionnaire `successeurs` dont les clés sont les états ; la valeur associée à un état est la liste des états suivants possibles (c'est donc une liste vide si l'état est un état final) ;
- une liste `gagnants0` des états finaux gagnants pour le joueur J_0 ;
- une liste `gagnants1` des états finaux gagnants pour le joueur J_1;
- une liste `nuls` des états finaux de match nul.

Voici un exemple du dictionnaire `successeurs` représentant le graphe du jeu pour n=2. Les flèches représentent une partie où J0 commence et gagne en plaçant 'OO' sur la rangée du haut de la grille :



4. En utilisant les fonctions `suivants(etat)` et `partie_finie(n,c)`, écrire une fonction `jeu_OXO(n,i)` qui renvoie le dictionnaire `successeurs` ainsi que les listes `gagnants0`, `gagnants1` et `nuls` lorsque le joueur `i` commence la partie. Vérifier que le nombre total de sommets est de 5478 pour n=3.

Vérifier :
```
>>> successeurs,gagnants0,gagnants1,nuls=jeu_OXO(3,0)
>>> len(successeurs)
5478
>>> len(gagnants0)
626
>>> len(gagnants1)
316
>>> len(nuls)
16
```

III) CONSTRUCTION DES ATTRACTEURS

5. Écrire une fonction `inverse(dico)` :

- qui reçoit un dictionnaire `dico` dont les clés sont les éléments d'un ensemble fini E (quelconque) et la valeur associée à une clé c une liste (éventuellement vide) d'éléments c' de E ;
- et qui renvoie un dictionnaire dont les clés sont éléments de E et tel que la valeur associée à une clé $c' \in E$ est la liste (éventuellement vide) des clés c telles que c' est dans la liste `dico[c]`.

Vérifier pour n = 2 :

```
>>> successeurs, gagnants0, gagnants1, nuls = jeu_OXO(2,0)

>>> inverse(successeurs)
{('.........', 0): [], ('O...', 1): [('....', 0)], ('.O..', 1): [('....', 0)], ('..O.', 1): [('....',
0)], ('...O', 1): [('....', 0)], ('X..O', 0): [('...O', 1)], ('.X.O', 0): [('...O', 1)], ('..XO', 0):
[('...O', 1)], ('O.XO', 1): [('..XO', 0), ('O.X.', 0)], ('.OXO', 1): [('..XO', 0), ('.OX.', 0)], ('OX.
O', 1): [('.X.O', 0), ('OX..', 0)], ('.XOO', 1): [('.X.O', 0), ('.XO.', 0)], ('XO.O', 1): [('X..O', 0)
, ('XO..', 0)], ('X.OO', 1): [('X..O', 0), ('X.O.', 0)], ('X.O.', 0): [('..O.', 1)], ('.XO.', 0): [('.
.O.', 1)], ('..OX', 0): [('..O.', 1)], ('O.OX', 1): [('..OX', 0), ('O..X', 0)], ('.OOX', 1): [('..OX',
0), ('.O.X', 0)], ('OXO.', 1): [('.XO.', 0), ('OX..', 0)], ('XOO.', 1): [('X.O.', 0), ('XO..', 0)], ('
XO..', 0): [('.O..', 1)], ('.OX.', 0): [('.O..', 1)], ('.O.X', 0): [('.O..', 1)], ('OO.X', 1): [('.O.X
', 0), ('O..X', 0)], ('OOX.', 1): [('.OX.', 0), ('O.X.', 0)], ('OX..', 0): [('O...', 1)], ('O.X.', 0):
[('O...', 1)], ('O..X', 0): [('O...', 1)]}
```

On définit quatre nouvelles variables globales :

```
successeurs, gagnants0, gagnants1, nuls = jeu_OXO()
predecesseurs = inverse(successeurs)
```

On rappelle que l'attracteur de l'ensemble cible F pour le joueur J_i est l'ensemble des sommets à partir desquels le joueur J_i est certain d'atteindre un sommet de F, quels que soient les coups joués par son adversaire. C'est donc l'ensemble des positions gagnantes pour le joueur J_i.

Il se construit de la manière suivante : au départ, tous les sommets de F sont déjà gagnants pour J_i, puisqu'une partie située en un tel sommet a déjà atteint l'objectif. Ensuite, pour chaque sommet s appartenant à l'attracteur, on examine ses prédécesseurs p :

- Si le prédécesseur p est contrôlé par le joueur J_i alors on ajoute ce prédécesseur à l'attracteur ;
- Si en revanche le prédécesseur p est contrôlé par l'adversaire, alors il ne peut être ajouté à l'attracteur que si tous ses successeurs y appartiennent.

Remarque : Dans le cours, nous avons étudié un algorithme basé sur l'utilisation des degrés de chaque sommet du graphe pour décompter au fur et à mesure de l'analyse les successeurs des prédécesseurs contrôlés par l'adversaire qui entrent dans l'attracteur. Cette méthode a été mise en œuvre dans le TD sur le jeu de Nim.

Dans ce TD, on n'utilisera pas cette méthode : dans le cas où un prédécesseur est contrôlé par l'adversaire, on vérifiera si l'ensemble de ses successeurs entrent bien dans l'attracteur par itération avant de l'ajouter dans l'attracteur.

6. La fonction `attracteur(joueur)` doit renvoyer la liste `A` des états qui sont dans l'attracteur du joueur choisi. Compléter son script Python-ci-dessous :

```
def attracteur(joueur) :
    if joueur == 0 :
        A = gagnants0[:]
    else :
        A = gagnants1[:]

    def aux(s) :
        ....

    for s in A :
        aux(s)
    return A
```

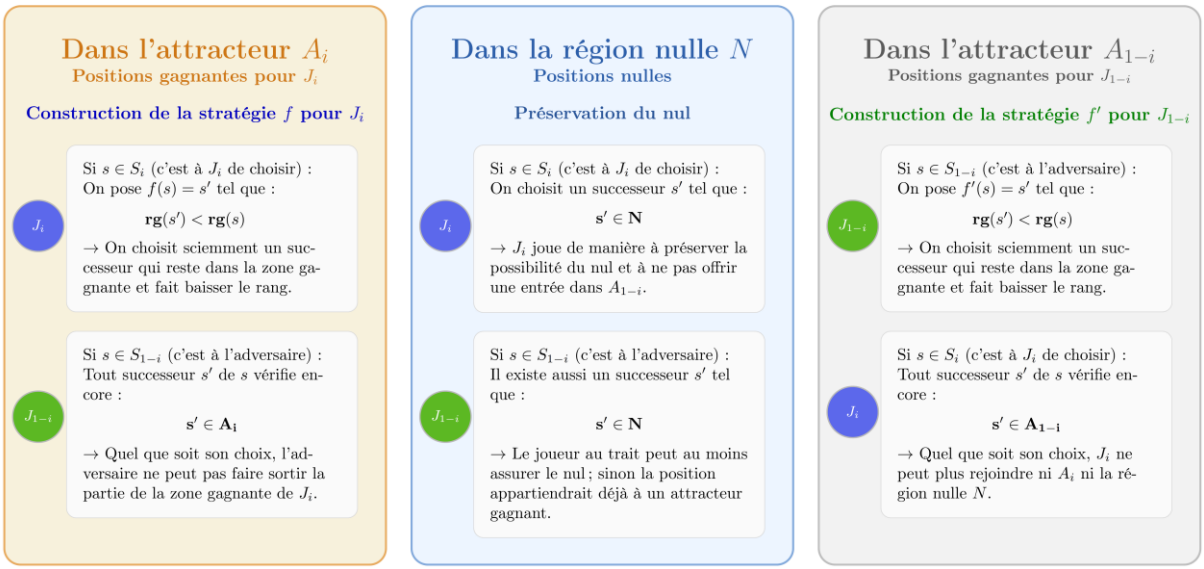
Vérifier :

```
>>> A0 = attracteur(0)
>>> len(A0)
2936
```

```
>>> A1 = attracteur(1)
>>> len(A1)
1474
```

IV) CONSTRUCTION D'UNE STRATÉGIE PAR ATTRACTEURS

On souhaite construire une stratégie à partir de la connaissance des attracteurs et de la région nulle en utilisant une stratégie suivant la règle de priorité $A_i \succ N \succ A_{1-i}$: on cherche d'abord un coup qui mène dans sa région gagnante ; à défaut, on choisit un coup qui préserve le nul ; et ce n'est qu'en dernier ressort que l'on subit une position perdante.



On définit une nouvelle variable globale :

`attracteurs = { 0 : attracteur(0), 1 : attracteur(1), None : region_nulle}`

… où `region_nulle` est définie par : $\text{region\_nulle} = S \setminus (A_i \cup A_{1-i})$

7. Écrire une fonction `strategie1(etat)` qui renvoie un tuple `(position,type)` où `position` est la position optimale à jouer parmi les successeurs possibles (on la choisira au hasard), et `type` est une chaîne de caractère définissant le type de position :
- "position_gagnante" : la position renvoyée appartient à l'attracteur qui joue le coup ;
- "position_perdante" : la position renvoyée appartient à l'attracteur de l'adversaire ;
- "position_nulle" : la position renvoyée est issue de la région nulle.

La fonction `JeuxAvecStrategie1(n)` (voir le fichier python source du TD) est donnée ci-dessous. Elle permet de simuler une partie à partir d'un état initial avec la stratégie implémentée à la question 7 :

```
def JeuxAvecStrategie1(etat, n):
    fin_partie = False

    if etat in attracteurs[0]:
        partie = [("position_initiale_gagnante_J0 / J"
                   +str(etat[1])+" commence")]
    elif etat in attracteurs[1]:
        partie = [("position_initiale_gagnante_J1 / J"
                   +str(etat[1])+" commence")]
    else:
        partie = [("position_initiale_nulle / J"
                   +str(etat[1])+" commence")]

    etat_ = etat
    while fin_partie == False:
        etat, type = strategie1(etat_)
        partie.append((etat_, type))

        # Vérifie si c'est un état final
        fin_partie, joueur_gagnant = partie_finie(n,etat[0])
        etat_ = etat

    partie.append((etat_, "fin partie"))

    return partie, joueur_gagnant
```

Vérifier que votre stratégie fonctionne correctement :

- État initial grille vide, J0 commence :

```
>>> JeuxAvecStrategie1(etat_initial,3)
(['position_initiale_nulle / J0 commence',
(('.........', 0), 'position_nulle'),
(('...O.....', 1), 'position_nulle'),
(('...O..X..', 0), 'position_nulle'),
(('...O..XO.', 1), 'position_nulle'),
(('.X.O..XO.', 0), 'position_nulle'),
(('OX.O..XO.', 1), 'position_nulle'),
(('OX.O..XOX', 0), 'position_nulle'),
(('OXOO..XOX', 1), 'position_nulle'),
(('OXOOX.XOX', 0), 'position_nulle'),
(('OXOOXOXOX', 1), 'fin partie')], None)
```

- État initial dans l'attracteur A0, J0 commence :

```
>>> JeuxAvecStrategie1(('..XO.....', 0),3)
(['position_initiale_gagnante_J0 / J0 commence',
(('..XO.....', 0), 'position_gagnante'),
(('O.XO.....', 1), 'position_perdante'),
(('O.XO....X', 0), 'position_gagnante'),
(('O.XO.O..X', 1), 'position_perdante'),
(('O.XO.OX.X', 0), 'position_gagnante'),
(('O.XOOOX.X', 1), 'fin partie')], 0)
```

- État initial dans l'attracteur A1, J1 commence :

```
>>> JeuxAvecStrategie1(('X..O...O.', 1),3)
(['position_initiale_gagnante_J1 / J1 commence',
(('X..O...O.', 1), 'position_gagnante'),
(('X.XO...O.', 0), 'position_perdante'),
(('X.XO...OO', 1), 'position_gagnante'),
(('XXXO...OO', 0), 'fin partie')], 1)
```

- État initial dans l'attracteur A0, J1 commence :

```
>>> JeuxAvecStrategie1(('..O.OX...', 1),3)
(['position_initiale_gagnante_J0 / J1 commence',
(('..O.OX...', 1), 'position_perdante'),
(('X.O.OX...', 0), 'position_gagnante'),
(('X.O.OX.O.', 1), 'position_perdante'),
(('X.OXOX.O.', 0), 'position_gagnante'),
(('XOOXOX.O.', 1), 'fin partie')], 0)
```

- État initial dans l'attracteur A1, J0 commence :

```
>>> JeuxAvecStrategie1(('.O...OX.X', 0),3)
(['position_initiale_gagnante_J1 / J0 commence',
(('.O...OX.X', 0), 'position_perdante'),
(('.O...OXOX', 1), 'position_gagnante'),
(('.O..XOXOX', 0), 'position_perdante'),
(('.O.OXOXOX', 1), 'position_gagnante'),
(('XO.OXOXOX', 0), 'fin partie')], 1)
```

8. Après avoir simulé plusieurs parties à partir de l'état initial, conjecturer sur l'existence d'une stratégie gagnante pour l'un des deux joueurs.

V) ALGORITHME MINIMAX

L'algorithme minimax attribue à chaque position une cote de la manière suivante, **du point de vue de J_0** :

- Si la position est un état final dans lequel J_0 gagne, la cote est +1 ;
- Si la position est un état final dans lequel J_1 gagne, la cote est −1 ;
- Si la position est un état final de match nul la cote est 0 ;
- Si la position a des successeurs et que c'est au tour de J_0 de jouer, la cote est le maximum de la cote de toutes les positions suivantes possibles ;
- Si la position a des successeurs et que c'est au tour de J_1 de jouer, la cote est le minimum de la cote de toutes les positions suivantes possibles.

9. Écrire une fonction récursive avec mémoïsation `minimax(etat)` qui renvoie la cote de la position selon cet algorithme. Vérifier que 2936 positions ont la cote +1, 1474 ont la cote −1 et 1068 la cote 0.

10. Quelle est la cote de l'état initial ? Existe-t-il une stratégie gagnante pour l'un des deux joueurs ?

11. Écrire la fonction `strategieMinMax(etat)` renvoyant le 3-uplet `(position,type,cote)` où `position` est la position parmi les successeurs possibles de cote maximale si le joueur ayant la main est J_0 et minimale si c'est J_1. Comme avant, `type` est une chaîne de caractère définissant le type de position et cote est la valeur de la `cote` associée à la position renvoyée.

- "position_MAX" : la position renvoyée est de cote MAX ;
- "position_MIN" : la position renvoyée est de cote MIN ;

Vérifier que l'application de la stratégie donne :

- sur l'état initial (région nulle - J0 joue) → (('O.......', 1), 'position_MAX', 0)
- sur un état de A0 (J0 joue) → (('OO.OXX...', 1), 'position_MAX', 1)
- sur un état de A1 (J1 joue) → (('OOXOOXX.X', 0), 'position_MIN', -1)
- sur un état de A1 (J0 joue) → (('OOXO.OX.X', 1), 'position_MAX', -1)
- sur un état de A0 (J1 joue) → (('OOXX.....', 0), 'position_MIN', 1)

Si vous le souhaitez, vous pouvez tester la stratégie `stategieMinMax` dans une partie en utilisant la fonction `JeuxAvecStrategieMiniMax` :

- État initial grille vide, J0 commence :

```
>>> JeuxAvecStrategieMiniMax(etat_initial,3)
(['position_initiale_nulle / J0 commence',
(('.........', 0), 'position_MAX', 0),
(('O........', 1), 'position_MIN', 0),
(('O...X....', 0), 'position_MAX', 0),
(('OO..X....', 1), 'position_MIN', 0),
(('OOX.X....', 0), 'position_MAX', 0),
(('OOX.X.O..', 1), 'position_MIN', 0),
(('OOXXX.O..', 0), 'position_MAX', 0),
(('OOXXXOO..', 1), 'position_MIN', 0),
(('OOXXXOOX.', 0), 'position_MAX', 0),
(('OOXXXOOXO', 1), 'fin partie')], None)
```

- État initial dans l'attracteur A0, J0 commence :

```
>>> JeuxAvecStrategieMiniMax(('..XO.....', 0),3)
(['position_initiale_gagnante_J0 / J0 commence',
(('..XO.....', 0), 'position_MAX', 1),
(('O.XO.....', 1), 'position_MIN', 1),
(('OXXO.....', 0), 'position_MAX', 1),
(('OXXOO....', 1), 'position_MIN', 1),
(('OXXOOX...', 0), 'position_MAX', 1),
(('OXXOOXO..', 1), 'fin partie')], 0)
```

- État initial dans l'attracteur A1, J1 commence :

```
>>> JeuxAvecStrategieMiniMax(('X..O...O.', 1),3)
(['position_initiale_gagnante_J1 / J1 commence',
(('X..O...O.', 1), 'position_MIN', -1),
(('X.XO...O.', 0), 'position_MAX', -1),
(('XOXO...O.', 1), 'position_MIN', -1),
(('XOXOX..O.', 0), 'position_MAX', -1),
(('XOXOXO.O.', 1), 'position_MIN', -1),
(('XOXOXOXO.', 0), 'fin partie')], 1)
```

- État initial dans l'attracteur A0, J1 commence :

```
>>> JeuxAvecStrategieMiniMax(('..O.OX...', 1),3)
(['position_initiale_gagnante_J0 / J1 commence',
(('..O.OX...', 1), 'position_MIN', 1),
(('X.O.OX...', 0), 'position_MAX', 1),
(('XOO.OX...', 1), 'position_MIN', 1),
(('XOOXOX...', 0), 'position_MAX', 1),
(('XOOXOXO..', 1), 'fin partie')], 0)
```

- État initial dans l'attracteur A1, J0 commence :

```
>>> JeuxAvecStrategieMiniMax(('.O...OX.X', 0),3)
(['position_initiale_gagnante_J1 / J0 commence',
(('.O...OX.X', 0), 'position_MAX', -1),
(('OO...OX.X', 1), 'position_MIN', -1),
(('OOX..OX.X', 0), 'position_MAX', -1),
(('OOXO.OX.X', 1), 'position_MIN', -1),
(('OOXOXOX.X', 0), 'fin partie')], 1)
```

VI) STRATÉGIE HEURISTIQUE

Une idée de fonction heuristique pour avoir une estimation de la cote d'une position selon un algorithme minimax sans calculer la cote de tous les états possibles est la suivante :

$$h(x,i) = \begin{cases} 2n+3 & si\ la\ position\ est\ gagnante\ pour\ J_i \\ -(2n+3) & si\ la\ position\ est\ gagnante\ pour\ J_{1-i} \\ g(x,i) - g(x,1-i) & sinon \end{cases}$$

… où $g(x,i)$ (resp. $g(x,1-i)$) est le nombre de possibilités de gain pour J_i (resp. J_{1-i}) pour la grille x (le nombre de rangées où ne figure aucun symbole de son adversaire).

Par exemple : pour la grille $\begin{bmatrix} \mathrm{X} & \mathrm{O} \\ & \mathrm{O} \\ & \end{bmatrix}$, $g(x,0) = 5$ et $g(x,1) = 3$. On a donc $h(x,0) = 2$.

12. Écrire une fonction `compte_lcd(grille,joueur,n)` qui renvoie le nombre de ligne, colonne ou diagonales que le joueur pourrait remplir. Vérifier avec la grille initiale et l'exemple donné précédemment.

Remarque : Vous n'avez plus le droit d'utiliser les variables `successeurs`, `gagnants0`, `gagnants1`, `nuls` et autres variables / dictionnaires associés par la suite. En effet, la raison d'être d'utiliser une heuristique est d'éviter d'explorer complètement le graphe du jeu. Vous pouvez par contre encore utiliser la fonction `suivants(etat)`.

13. En déduire une fonction `heuristique(etat,i,n)` qui renvoie la cote d'une position calculée par la méthode exposée ci-dessus pour le joueur J_i.

```
Vérifier :
      >>> heuristique(etat_initial,0,3)       # État non gagnant
      0
      >>> heuristique((('XO..O....'),0),0,3)  # État non gagnant
      2
      >>> heuristique((('XO..O....'),0),1,3)  # État non gagnant
      -2
      >>> heuristique(('OXXOOXO..', 1),0,3)   # État gagnant pour J0
      9
      >>> heuristique(('OXXOOXO..', 1),1,3)   # État gagnant pour J0
      -9
```

On envisage une stratégie MiniMax basée sur cette heuristique, **du point de vue de $\mathbf{J_0}$**, avec une profondeur maximale d'exploration de l'arbre. L'algorithme attribue à chaque position une cote de la manière suivante :

- Si la position est un état final (feuille) ou que la profondeur maximale est atteinte, la cote est retournée par `heuristique()` du point de vue de J_0 ;
- Si la position a des successeurs et que c'est au tour de J_0 de jouer, la cote est le maximum de la cote de toutes les positions suivantes possibles ;
- Si la position a des successeurs et que c'est au tour de J_1 de jouer, la cote est le minimum de la cote de toutes les positions suivantes possibles.

Cela revient à définir récursivement la valeur minimax tronquée à profondeur p par :

$$V_p(s) = \begin{cases} h(s, J_0) & si\ Succ(s) = \emptyset\ ou\ p = 0 \\ \max\limits_{s' \in Succ(s)} V_{p-1}(s') & si\ s \in S_0 \\ \min\limits_{s' \in Succ(s)} V_{p-1}(s') & si\ s \in S_1 \end{cases}$$

La profondeur p représente le nombre de coups explorés à partir de la position courante. À chaque coup joué, ce budget diminue de 1. Si c'est à J_0 de jouer, alors $p = 1$ permet de voir un coup de J_0, $p = 2$ permet de voir un coup de J_0 puis une réponse de J_1, et $p = 3$ permet de voir à nouveau un coup de J_0.

14. Écrire une fonction récursive `minimax_heur(etat,p,n)` qui renvoie la cote de la position selon cet algorithme.

Vous pouvez tester avec les exemples suivants :

- Tester les cas terminaux : la fonction doit renvoyer la même valeur extrême ou nulle du point de vue de J_0, quelle que soit la profondeur :

```
>>> minimax_heur(('OOOXX....', 1), 0, 3)   # J0 gagne
9
>>> minimax_heur(('OOOXX....', 1), 4, 3)   # J0 gagne
9
>>> minimax_heur(('XXXOO....', 0), 4, 3)   # J1 gagne
-9
>>> minimax_heur(('XXXOO....', 0), 0, 3)   # J1 gagne
-9
```

- Tester les coups gagnants immédiats : une position où J_0 peut gagner d'un seul coup, avec $p = 1$, doit donner une victoire certaine pour J_0 :

```
>>> minimax_heur(('OO.XX....', 0), 1, 3)   # J0 gagne en 1 coup
9
>>> minimax_heur(('OO.XX....', 0), 4, 3)   # J0 gagne en 1 coup
9
```

- Coup immédiat où J_1 peut gagner d'un seul coup, avec $p = 1$, doit donner une victoire certaine pour J_1 :

```
>>> minimax_heur(('XX.OO.O..', 1), 1, 3)
-9
```

- Coup où J_0 gagne en deux coups (il faut donc $p = 3$) :

```
>>> minimax_heur(('O...X...O', 0), 1, 3)
-2
>>> minimax_heur(('O...X...O', 0), 2, 3)
0
>>> minimax_heur(('O...X...O', 0), 3, 3)
9
```

- Coup dans l'attracteur de J_0 mais où J_0 ne peut pas gagner en un seul coup :

```
>>> minimax_heur(('..XO.....', 0), 1, 3)
0
>>> minimax_heur(('..XO.....', 0), 4, 3)
1
>>> minimax_heur(('..XO.....', 0), 5, 3)
9
```

- Coup dans l'attracteur de J_1 mais où J_1 ne peut pas gagner en un seul coup :

```
>>> minimax_heur(('XOXO...O.', 1), 1, 3)
-2
>>> minimax_heur(('XOXO...O.', 1), 4, 3)
-9
```

- Test global sur l'état initial : Si on explore toute la profondeur depuis la grille vide, on doit obtenir :

```
>>> minimax_heur(etat_initial, 9, 3)
0
```

Une fois la fonction `minimax_heur(etat,p,n)` construite, nous allons l'utiliser pour définir une stratégie heuristique paramétrée par la profondeur p, dont on cherchera ensuite à évaluer l'efficacité en la comparant à la vraie valeur minimax.

Pour choisir un coup depuis un état, on évalue chacun de ses successeurs immédiats en explorant encore $p-1$ coups, puis on conserve ceux dont la cote est optimale.

Plus précisément :

- Si `etat` est contrôlé par J_0, la fonction renvoie les successeurs `suiv_opt` qui maximisent `minimax_heur(suiv,p-1,n)` :
- Si `etat` est contrôlé par J_1, la fonction renvoie les successeurs `suiv_opt` qui minimisent `minimax_heur(suiv,p-1,n)` ;

$$suiv\_opt[etat] = \begin{cases} \underset{s' \in Succ(etat)}{\arg\max} \; minimax\_heur(s', p-1, n) & si\ etat \in S_0 \\ \underset{s' \in Succ(etat)}{\arg\min} \; minimax\_heur(s', p-1, n) & si\ etat \in S_1 \end{cases}$$

15. Écrire une fonction `strategieMiniMax_heur(etat,p,n)` applicable à un état non terminal `etat`, avec $p \geq 1$. Cette fonction renvoie la liste des successeurs immédiats de etat qui sont optimaux selon l'algorithme `minimax_heur`, pour une profondeur totale d'exploration égale à p.
Vous pouvez tester avec les exemples suivants :

- Coup où J_0 gagne immédiatement : la stratégie doit trouver le coup avec $p=1$
```
>>> strategieMiniMax_heur(('OO.XX....', 0),1,3)
[('OOOXX....', 1)]
```
- Coup où J_1 peut gagner immédiatement :
```
>>> strategieMiniMax_heur( ('XX.OO.O..', 1),1,3)
[('XXXOO.O..', 0)]
```
- Position dans l'attracteur de J_0, mais sans gain immédiat : cette position finit par être gagnante pour J_0 (position(s) renvoyée(s) dans l'attracteur de J_0) mais pas immédiatement.
```
>>> strategieMiniMax_heur(('..XO.....', 0),1,3)
[('..XO.O...', 1)]
>>> ('..XO.O...', 1) in attracteurs[0]
False
>>> strategieMiniMax_heur(('..XO.....', 0),3,3)
[('..XO.O...', 1)]
>>> strategieMiniMax_heur(('..XO.....', 0),5,3)
[('O.XO.....', 1)]
>>> ('O.XO.....', 1) in attracteurs[0]
True
>>> strategieMiniMax_heur(('..XO.....', 0),9,3)
[('O.XO.....', 1), ('..XOO....', 1), ('..XO....O', 1)]
```
- Position dans l'attracteur de J_1, mais sans gain immédiat :
```
Tester   >>> strategieMiniMax_heur(('O..O....X', 1),1,3)
et       >>> strategieMiniMax_heur(('O..O....X', 1),3,3)
```

On veut maintenant évaluer la qualité de cette stratégie heuristique à profondeur p. Pour cela, on parcourt un ensemble d'états accessibles depuis l'état initial. Pour chacun de ces états, on compare :

- les successeurs retenus par `strategieMiniMax_heur(etat,p,n)` ;
- l'ensemble de tous les successeurs possibles de `etat`.

On détermine ensuite, à l'aide de la fonction `minimax`, combien de ces successeurs sont gagnants, nuls ou perdants, puis on compare les répartitions obtenues.

Afin de comparer les stratégies dans un cadre identique, on se limite aux positions atteintes après 2 coups depuis l'état initial. On étudie alors, sur cet ensemble fixe de positions, la qualité réelle des coups sélectionnés par la stratégie à une profondeur p.

La fonction d'évaluation est donnée ci-dessous et est déjà codée dans votre fichier source python :

```
def EvaluerStrategieMiniMax_heur(p,n):
    # ordre des cases : [minimax = -1, minimax = 0, minimax = 1]
    # répartition des coups choisis par l'heuristique
    # entre perdants / nuls / gagnants ;
    nbre_choix = [0, 0, 0]

    # répartition de tous les coups possibles
    # entre perdants / nuls / gagnants.
    nbre_tout  = [0, 0, 0]

    # parcourt les états atteints après 2 coups depuis l'état initial
    for etat1 in suivants(etat_initial):
        for etat2 in suivants(etat1):
            # coups choisis par la stratégie heuristique à prof. = p
            choix = strategieMiniMax_heur(etat2, p, n)

            # comptage des vraies valeurs minimax des coups choisis
            # par l'heuristique
            for s in choix:
                nbre_choix[minimax(s) + 1] += 1

            # comptage des vraies valeurs minimax de tous les coups
            # possibles
            for s in suivants(etat2):
                nbre_tout[minimax(s) + 1] += 1

    print("Ordre : [J1 gagne, nul, J0 gagne] = [-1, 0, 1]")
    print("Coups choisis par la stratégie heuristique :", nbre_choix)
    print("Tous les coups possibles                   :", nbre_tout)
```

16. Exécuter la fonction `EvaluerStrategieMiniMax_heur(p,3)` pour plusieurs valeurs de p, par exemple p = 1,3,5,9. Préciser si l'augmentation de p semble améliorer la qualité des choix.