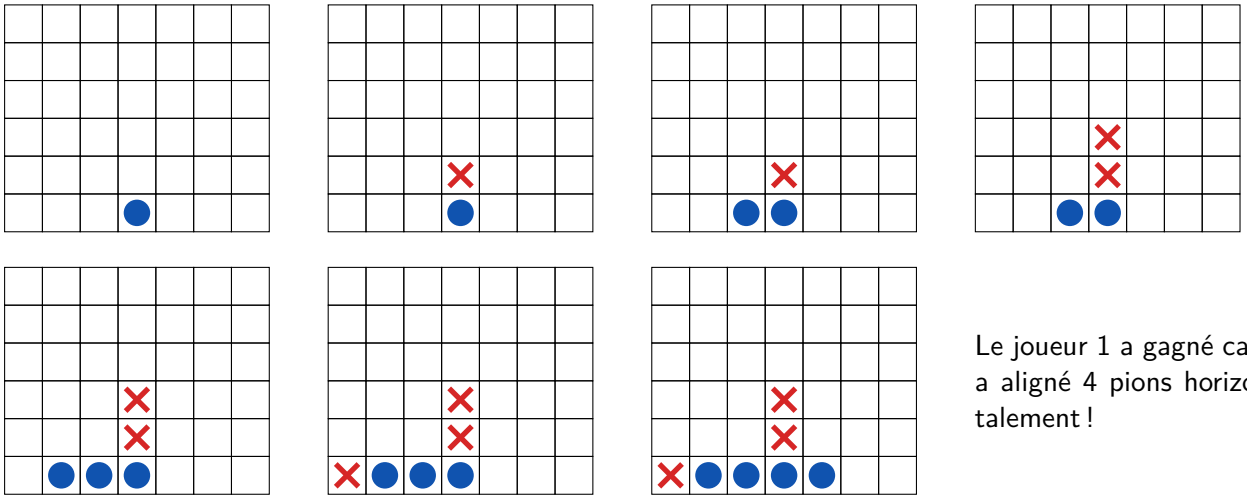
Algorithme minimax au puissance 4

I Présentation du jeu

Le puissance 4 est un jeu à deux joueur, sur un plateau de 6 lignes et 7 colonnes. À tour de rôle, chaque joueur ajoute un pion de sa couleur en choisissant une colonne. Le pion, tombe alors sur la case la plus basse inoccupée. Lorsque qu'un joueur aligne 4 pion (horizontalement, verticalement ou bien en diagonale) à l'issu de son coup, il gagne la partie.

✔ Exemple

Le joueur 1 joue les ronds et le joueur 2 les croix. On peut imaginer la partie suivante :



Le joueur 1 a gagné car il a aligné 4 pions horizontalement !

Le but de cet exercice est de coder une intelligence artificielle jouant le **joueur 2** face de manière efficace, en suivant un algorithme minimax.

II Modélisation

A Fonction heuristique

Chaque situation peut être vue comme un nœud sur un graphe (le graphe de ce jeu possède environ 10^{40} positions possibles !). Il faut trouver un moyen d'évaluer l'avantage que possède un joueur étant donné une position de jeu. L'idée pour cela est d'attribuer un score à chaque case, on calcule ensuite la différence entre la somme des case occupée par le joueur 2 (✖) et celles occupées par le joueur 1 (●). Voici la matrice que l'on va utiliser pour répartir les valeurs des cases :

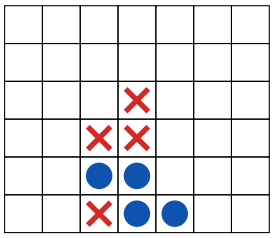
3	4	5	7	5	4	3
4	6	8	10	8	6	4
5	8	11	14	11	8	5
5	8	11	14	11	8	5
4	6	8	10	8	6	4
3	4	5	7	5	4	3

Remarque

La valeur d'une case correspond au nombre d'alignements de 4 pions que l'on peut créer en utilisant cette case. Il s'agit d'une notation totalement arbitraire et on pourrait tout à fait proposer une autre façon d'évaluer les points.

Bien sûr, si la position est gagnante pour le joueur 2, la fonction heuristique renverra $+\infty$; et si elle perdante (donc que le joueur 1 y gagne), elle renverra $-\infty$.

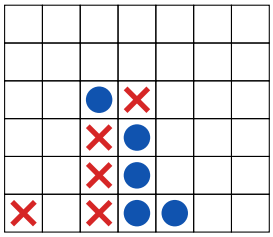
Exemple



$\Longrightarrow$

$$\textbf{joueur 1} = 8 + 10 + 7 + 5 = 30$$
$$\textbf{joueur 2} = 4 + 11 + 14 + 14 = 43$$
$$\textbf{score} = 43 - 30 = 13$$

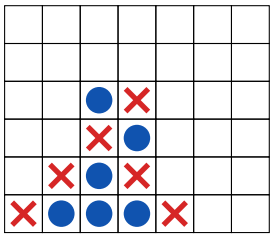
Cette position est donc avantageuse pour le joueur 2.



$\Longrightarrow$

$$\textbf{joueur 1} = 7 + 10 + 14 + 5 + 11 = 47$$
$$\textbf{joueur 2} = 5 + 8 + 11 + 14 + 3 = 41$$
$$\textbf{score} = 41 - 47 = -6$$

Cette position n'est donc pas avantageuse pour le joueur 2.



$\Longrightarrow$ Dans cette situation, le joueur 2 a gagné donc la fonction heuristique renvoie

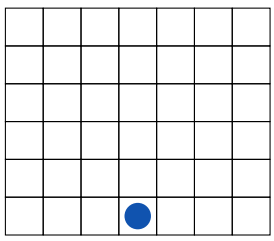
$$\textbf{score} = +\infty$$

B Algorithme minimax

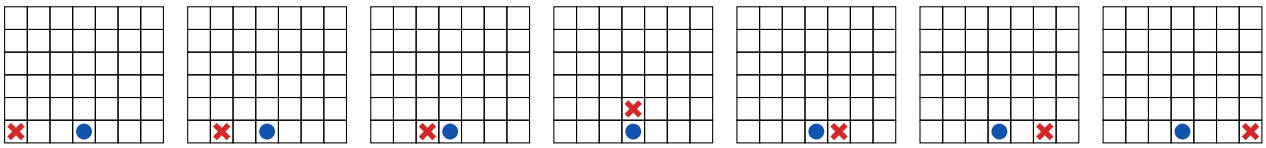
Le but est de coder une intelligence artificielle suivant un algorithme minimax dont on pourra régler le nombre de coups d'avance. L'idée est d'évaluer chaque coup possible et de choisir celui offrant le plus grand avantage, en supposant que le joueur en face joue de façon idéale.

Exemple

On est au début de la partie, le joueur 1 (●) vient de jouer dans la colonne centrale, et c'est au joueur 2 (✕) de répondre :

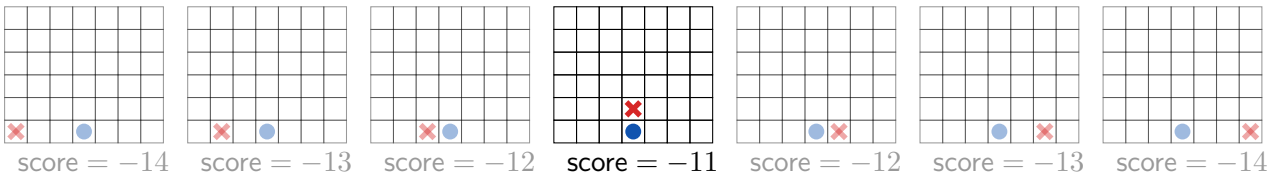


Le joueur 2 (✕) a alors 7 possibilités directes de jeu :

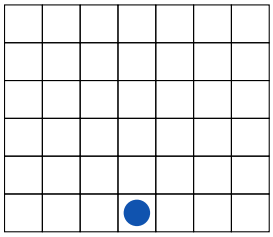


On pourrait évaluer directement le score de ces positions avec notre fonction heuristique, mais notre intelligence artificielle n'aurait alors pas du tout anticipé les coups suivants. L'idée est donc d'envisager pour chacune des positions, les 7 nouvelles possibilités de réponse du joueur 1 (●). Ce qui fait alors $7 \times 7 = 49$ issues envisageables :

En supposant que le joueur 1 (●) joue au mieux, cela revient à considérer qu'il cherche à **minimiser** le score. Donc le score attribué aux 7 situations initiales est le minimum des scores au sein d'une colonne (grilles mises en valeur).



Le meilleur coup pour le joueur 2 (✖) est celui qui **maximisera** son score. Ce score est donc attribué à la grille initiale :



score = -11

Ici, le coup à jouer pour atteindre ce score est la colonne centrale.

Ainsi on peut résumer le fonctionnement de l'algorithme de la manière suivante :

- pour attribuer un score à une situation où le joueur 1 (●) doit jouer, il faut choisir le **minimimum** des scores suivants ;
- pour attribuer un score à une situation où le joueur 2 (✖) doit jouer, il faut choisir le **maximum** des scores suivants.

C Notations utilisées

Dans cet exercice, on sera amenés à utiliser les variables globales suivantes :

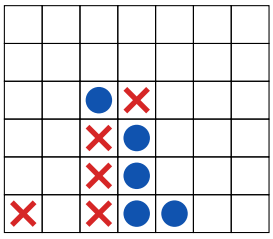
- `ROWS = 6` indique le nombre de lignes du plateau.
- `COLS = 7` indique le nombre de colonnes du plateau.

Et les fonctions que l'on utilisera pourront faire appel aux arguments suivants :

- `player` un indice représentant le joueur dont c'est le tour (vaut 1 ou 2).
- `board` représente une situation du jeu. Il s'agit d'un tableau numpy de taille `(ROWS, COLS)`, contenant dans chaque case l'un des entiers suivants :
  - 0 si la case est vide ;
  - 1 si la case est occupée par le joueur 1 (●) ;
  - 2 si la case est occupée par le joueur 2 (✖).

  On utilisera l'indice `ROWS - 1` pour la ligne du bas, et 0 pour la ligne du haut.

✔ Exemple



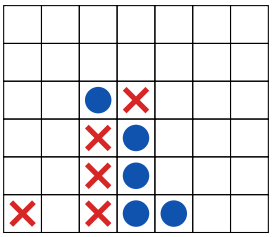
→

```
board = np.array([
    [0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0],
    [0, 0, 1, 2, 0, 0, 0],
    [0, 0, 2, 1, 0, 0, 0],
    [0, 0, 2, 1, 0, 0, 0],
    [2, 0, 2, 1, 1, 0, 0],
])
```

III Implémentation

A Fonctions utilisées

Pour coder correctement notre algorithme, il est important de le décomposer en sous-problèmes plus simples ("*diviser pour mieux régner*"). On aura besoin des fonctions détaillées ci-dessous. Pour les exemples, on utilisera le plateau suivant :



`get_first_row(board, col)`

Action : Trouve la première ligne vide dans la colonne donnée.

Arguments :

- `board` représente la situation de jeu
- `col` est l'indide de la colonne dans laquelle on souhaite jouer

Sortie :

- la fonction doit renvoyer l'indice de la première ligne vide dans la colonne ciblée
- si cette colonne est pleine, renvoie `None`

Dépendances : aucune

Exemples :

```
>>> get_first_row(board, 0) # La première colonne contient 1 pion
1
>>> get_first_row(board, 3) # La quatrième colonne contient 4 pions
4
>>> get_first_row(board, 6) # La septième colonne ne contient aucun pion
0
```

`get_next_board(board, col, player)`

Action : Construit la situation suivante, si `player` joue en colonne `col`.

Arguments :

- `board` représente la situation de jeu
- `col` est l'indide de la colonne dans laquelle on souhaite jouer
- `player` est l'indice du joueur plaçant son pion

Sortie :

- renvoie la nouvelle configuration du plateau, après que le coup ait été posé
- si le coup est interdit (colonne pleine), renvoie `None`

Dépendances :

- `get_first_row`

Exemples :

```
>>> get_next_board(board, 2, 1) # Le joueur 1 joue dans la troisième colonne
np.array([
    [0, 0, 0, 0, 0, 0, 0],
    [0, 0, 1, 0, 0, 0, 0],
    [0, 0, 1, 2, 0, 0, 0],
    [0, 0, 2, 1, 0, 0, 0],
    [0, 0, 2, 1, 0, 0, 0],
    [2, 0, 2, 1, 1, 0, 0],
])
```

`check_winner(board, row, col)`

Action : Vérifie si le coup est un coup gagnant pour l'un des joueurs.

Arguments :

- `board` représente la situation de jeu
- `row` est l'indice de la ligne de la case dans laquelle on vient de jouer
- `col` est l'indice de la colonne dans laquelle on vient de jouer

Sortie : entier indiquant quel joueur gagne

- 1 si la case ciblée fait partie d'un alignement de 4 pions du joueur 1
- 2 si la case ciblée fait partie d'un alignement de 4 pions du joueur 2
- 0 sinon

Dépendances : aucune

Exemples :

```
>>> check_winner(board, 3, 2)   # Le pion posé en 4e ligne, 3e colonne ne
    fait gagner aucun des joueurs
0
```

`get_board_score(board, player)`

Action : Donne le score d'une situation, du point de vue du joueur donné (fonction heuristique)

Arguments :

- `board` représente la situation de jeu
- `player` est l'indice du joueur considéré

Sortie :

- si le plateau est impossible, le coup précédent était interdit, donc la fonction renvoie $+\infty$
- si la situation est gagnante pour `player`, renvoie $+\infty$
- si la situation est gagnante pour le joueur opposé, renvoie $-\infty$
- sinon renvoie la différence entre la somme de toutes les cases occupées par `player` et la somme des cases occupées par le joueur opposé

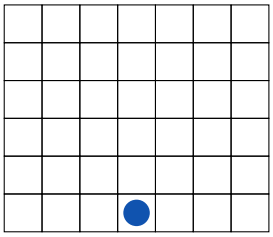
Dépendances :

- `check_winner`

Exemples :

```
>>> get_board_score(board, 2)    # 41 - 47
-6
```

Pour la fonction suivante, on reprend l'exemple détaillé en partie II- B :



`eval_board(board, player, k)`

Action : Mise en place de l'agorithme minimax, afin d'évaluer la colonne la plus optimale, du point de vue du joueur donné.

Arguments :

- `board` représente la situation de jeu
- `player` est l'indice du joueur
- `k` est la profondeur qu'il reste à explorer (décrémente jusqu'à 0)

Sortie : tuple `(score, col)` où

- `score` est le score attribué à `board` du point de vue de `player`
- `col` est l'indice de la meilleure colonne à jouer pour `player`

Dépendances :

- `get_board_score`
- `get_next_board`

Exemples :

```
>>> eval_board(board, 2, 2) # Avec k=2 coups d'avance, pour le player=2, il
    vaut mieux jouer la colonne du milieu (col = 3), qui lui assure un score
    maximal de -11
(-11, 3)
>>> eval_board(board, 2, 1) # Pour 1 coup d'avance, le score maximal est
    10-7 = 3, toujours en jouant dans la 4e colonne
(3, 3)
```

B Travail à effectuer

Chacun d'entre vous devra au moins coder la fonction `eval_board` ainsi qu'une autre de votre choix. Le but est de se répartir les fonctions restantes pour qu'elles soient toutes implémentée dans votre groupe.