Jeu du morpion

Le but du projet est d'implémenter un jeu de morpion.

L'intérêt du projet consiste à implémenter différents algorithmes permettant au joueur de jouer contre l'ordinateur
La partie interface est secondaire dans le projet.

1. Préparation

✍ Créer un module **`tic_tac_toe_grid.py`** qui va contenir les opérations de base pour travailler sur une grille de morpion (nb : en anglais, « morpion » (le jeu) se dit « tic tac toe »).

La grille de morpion sera représentée par une matrice 3x3 d'entiers, avec 0 : case vide, 1 : joueur 1, 2 : joueur 2.

✍ Implémenter une fonction **`get_empty_grid()`** sans paramètre qui renvoie une grille de morpion vide.

```
assert(get_empty_grid() == [[0, 0, 0], [0, 0, 0], [0, 0, 0]])
```

✍ Implémenter une fonction **`play(grid, line, col, player)`** qui prend en paramètres une grille, une ligne, une colonne et un joueur, et qui renvoie une **nouvelle grille** qui est une copie de **`grid`** (pour copier la grille utiliser **`deepcopy`**) où l'on a rajouté la marque du joueur **`player`** à la position **`(line, col)`**. Si les informations passées sont incorrectes (valeurs de **`line/col`** hors des bornes, marque déjà présente à l'endroit demandé), la fonction renvoie **`None`**.

```
assert(play([[0, 0, 0], [0, 0, 0], [0, 0, 0]], 1, 1, 1) == [[0, 0, 0], [0, 1, 0], [0, 0, 0]])
assert(play([[0, 0, 0], [0, 1, 0], [0, 0, 0]], 1, 2, 2) == [[0, 0, 0], [0, 1, 2], [0, 0, 0]])
assert(play([[0, 0, 0], [0, 1, 0], [0, 0, 0]], 1, 1, 2) == None)
assert(play([[0, 0, 0], [0, 1, 0], [0, 0, 0]], 1, 3, 1) == None)
assert(play([[0, 0, 0], [0, 1, 0], [0, 0, 0]], -1, 0, 2) == None)
```

✍ Implémentez une fonction **`get_free_cells(grid)`** qui prend en paramètre une grille, et qui renvoie la liste des cases libres dans la grille. Chaque case doit être représentée par un couple (ligne, colonne).

```
assert(get_free_cells ([[0, 0, 0], [0, 0, 0], [0, 0, 0]]) == [(0, 0), (0, 1), (0, 2), (1, 0),
(1, 1), (1, 2), (2, 0), (2, 1), (2, 2)])
assert(get_free_cells ([[1, 2, 0], [1, 0, 2], [0, 2, 1]]) == [(0, 2), (1, 1), (2, 0)])
assert(get_free_cells ([[1, 2, 2], [1, 1, 2], [2, 2, 1]]) == [])
```

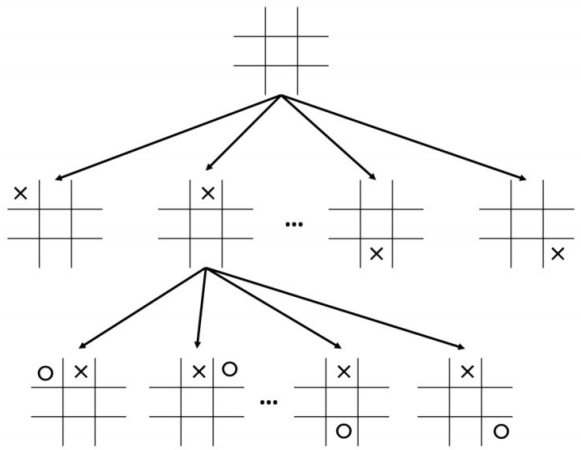
✍ Implémenter une fonction **`check_win(grid)`** qui prend une grille en paramètre et renvoie 0 s'il n'y a aucun alignement dans la grille, 1 s'il y a au moins un alignement pour le joueur 1, et 2 s'il y a au moins un alignement pour le joueur 2.
NB : dans les algorithmes, il ne sera pas possible d'avoir des alignements simultanément pour les joueurs 1 et 2.

```
assert(check_win ([[0, 0, 0], [0, 0, 0], [0, 0, 0]]) == 0)
assert(check_win ([[1, 2, 0], [0, 1, 2], [2, 0, 1]]) == 1)
assert(check_win ([[2, 2, 2], [1, 1, 2], [2, 1, 1]]) == 2)
assert(check_win ([[2, 1, 2], [1, 1, 2], [2, 1, 1]]) == 1)
assert(check_win ([[2, 1, 2], [1, 2, 2], [2, 1, 1]]) == 2)
```

2. Algorithmes de parcours de graphe

✍ Créer un module **dfs_bfs.py** qui va contenir les fonctions de parcours de graphe en profondeur et en largeur.

Le graphe à parcourir n'existe pas à l'avance. En effet, ce graphe est celui des différents « états » du jeu, un état étant une grille de jeu à un moment donné. Un exemple est donné dans la figure ci-dessous, en partant de la grille vide, et montrant une partie de l'espace de recherche pour deux coups (un coup de **X** et un coup de **O**).



2.1. Parcours en profondeur

✍ Implémentez la fonction **dfs_play(grid, player)**, qui prend en paramètres une grille et un joueur, et qui renvoie la position où le joueur **player** devrait mettre sa prochaine marque. En absence position gagnante la fonction retourne **None**.

Cette fonction est basée sur l'algorithme de parcours en profondeur d'un graphe. C'est une fonction récursive qui génère l'espace de recherche en mode profondeur, et s'arrête dès qu'elle trouve une grille où le joueur **player** a gagné. Elle renvoie alors le premier coup de la chaîne de coups à jouer pour arriver à cette position.

Le pseudo-code

La fonction **dfs_play(grid, player)** appelle pour chaque coup possible (coup de départ) la fonction auxiliaire récursive **dfs_play_aux(grid, player, us)** qui teste tous les coups à partir du coup de départ.

```
fonction dfs_play(grid, player) # retourne la position du coup qui peut amener à la victoire
    lst_pos ← get_free_cells(grid) # on récupère la liste des cases vides (positions possibles)
    pour pos dans lst_pos faire
        grid_tmp ← play(grid, line, col, player) # on crée une grille pour la position donnée
        si (dfs_play_aux(grid_tmp, player%2+1, player) = True) #appel à dfs_play_aux
            retourne (pos)
        fin si
    fin pour
    retourne None


fonction dfs_play_aux(grid, player, us)
    winner ← check_win(grid) # on regarde si le coup est gagnant
    si (winner ≠ 0) # si gagnant
        retourne (winner = us) #on retourne True si us gagne sinon False (fin de la récursivité)
```

```
sinon
    lst_pos ← get_free_cells(grid) # on récupère la liste des cases vides
    pour pos dans lst_pos faire
        grid_tmp ← play(grid, line, col, player) # on crée une grille temporaire
        si (dfs_play_aux(grid_tmp, player%2+1, player) = True) # appel à dfs_play_aux
            retourne True # l'appel récursif a trouvé une victoire
        fin si
    fin pour
fin si
retourne False # us n'a pas trouvé de victoire
```

2.1. Parcours en largeur

✍ Implémentez la fonction **`bfs_play(grid, player)`**, qui prend en paramètres une grille et un joueur, et qui renvoie la position où le joueur **`player`** devrait mettre sa prochaine marque. En absence position gagnante la fonction retourne **`None`**.

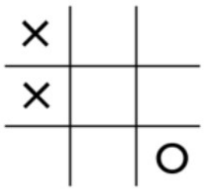
3. Algorithmes d'intelligence artificielle

✍ Créer un module **`ai.py`** qui va contenir les fonctions des algorithmes d'intelligence artificielle.

3.1. Evaluation de la grille de jeu

Il faut d'abord créer une fonction qui va évaluer la valeur d'une grille de jeu.
✍ Implémentez la fonction **`heuristic(grid, player)`**, qui prend en paramètres une grille et un joueur, et qui renvoie un score évaluant la qualité de la grille pour **`player`**.
L'heuristique proposé est le suivant :
Score = nombre de lignes ouvertes pour **`player`**
+ nombre de colonnes ouvertes pour **`player`**
+ nombre de diagonales ouvertes pour **`player`**
- nombre de lignes ouvertes pour adversaire
- nombre de colonnes ouvertes pour adversaire
- nombre de diagonales ouvertes pour adversaire.

Une ligne/colonne/diagonale est ouverte (si l'adversaire n'y a pas de pions), il y a des places libres (si l'adversaire n'y jouait pas on pourrait la compléter et gagner).
Par exemple dans cette situation :



Le joueur X a 2 lignes ouvertes, 2 colonnes ouvertes et 1 diagonale ouverte, et le joueur O a 1 ligne ouverte, 2 colonnes ouvertes et 1 diagonale ouverte. Le score du point de vue de X est donc : 2+2+1-1-2-1 = 1.

```
assert(heuristic([[0, 0, 0], [0, 1, 0], [0, 0, 0]], 1) == 4)
assert(heuristic([[2, 0, 0], [0, 1, 0], [0, 0, 0]],2) == -1)
```

3.2. Algorithme min_max

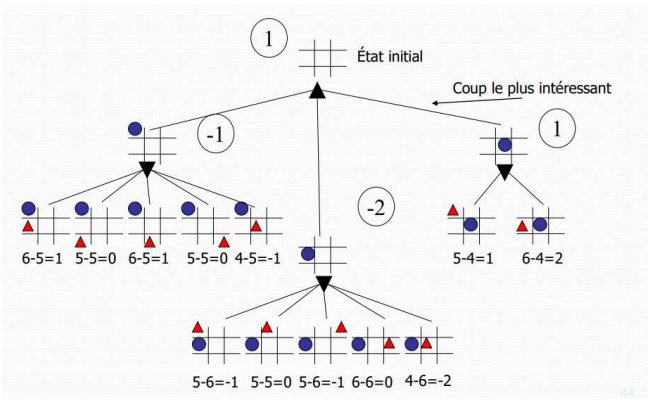
Fonctionnement de l'algorithme min_max

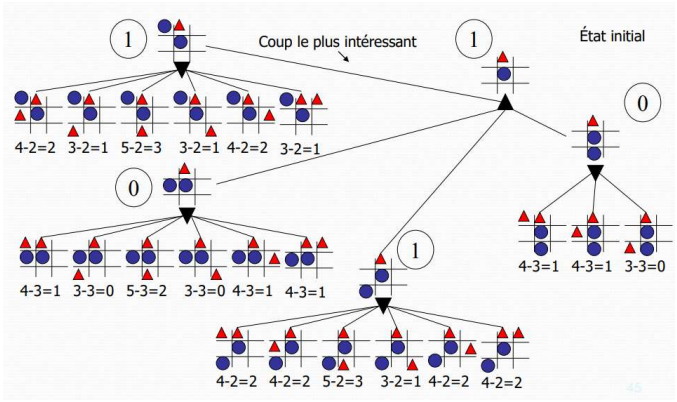
Hypothèse de départ

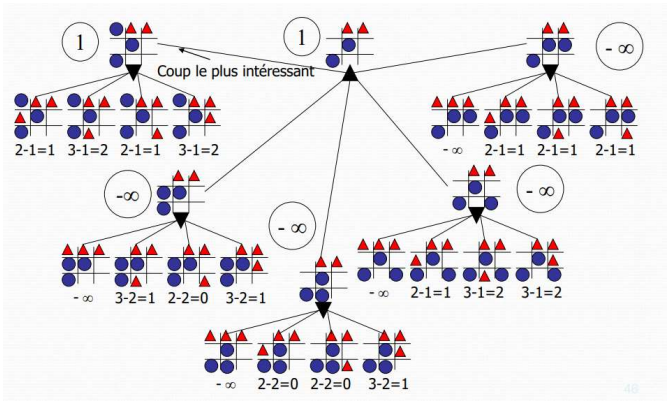
- La fonction d'évaluation (heuristique) est toujours du point de vue de **player**.
- Le coup le plus intéressant est celui qui a la valeur maximale.
- L'adversaire fait toujours les meilleurs choix pour lui (les plus contraignants pour **player**).

Évaluation

- La racine est la situation du jeu au moment où **player** doit jouer.
- On développe l'arbre jusqu'à une profondeur **deph**.
- Les feuilles sont évaluées et leur valeur est remontée jusqu'à la racine.
- Au nœud **OU** (choix de **player**), on associe le maximum des valeurs.
- Au nœud **ET** (choix de l'adversaire), on associe le minimum des valeurs (choix le plus contraignant pour **player**).








```
fonction min_max_play(grid, player, depth)
    """renvoie le meilleur coup à jouer dans la grille pour le joueur
    depth détermine la profondeur d'exploration de minMax"""
    val_max = - ∞ (valeur inférieure à la valeur min donnée par l'heuristique)
    coup_a_jouer = rien (None)
    on récupère les mouvements possibles (cases libres)
    pour chaque mouvement possibles faire
        si le mouvement est gagnant pour player
            alors on retourne le coup_a_jouer (mouvement)
        score ← mini_min_max(grille temporaire, player, deph - 2)
        si score est supérieure val_max
            alors on met à jour val_max et le coup_a_jouer
    on retourne le coup_a_jouer
```

```
fonction mini_min_max(grid, player, depth)
    """retourne la valeur mini d'une feuille ou des feuilles
    pour une profondeur donnée calculé à l'aide de l'heuristique"""
    val_mini = + ∞ (valeur supérieure à la valeur max donnée par l'heuristique)
    on récupère les mouvements possibles (cases libres)
    si il n'y a plus de mouvements possibles (si feuille de l'arbre)
        alors on calcule la valeur (val_mini) de l'heuristique pour la grille
        on retourne cette valeur (val_mini)
    sinon
        pour chaque mouvement possible faire
            si le mouvement est gagnant pour l'adversaire
                alors on retourne - ∞
        val_mini ← minimum de val_mini et de maxi_min_max(grid temporaire, player, depth)
    on retourne val_mini

fonction maxi_min_max(grid, player, depth)
    """retourne la valeur maxi, et la position de départ qui donne cette valeur,
    d'un nœud pour une profondeur donnée calculé à l'aide de l'heuristique
    pour une position de jeu d'un joueur"""
    on récupère les mouvements possibles (cases libres)
    #si feuille ou profondeur atteinte
    si il n'y a plus de mouvements possibles ou si la profondeur est de 0
        alors on calcule la valeur (val_max) de l'heuristique pour la grille
        on retourne cette valeur (val_maxi)
    sinon
    val_max = - ∞
        pour chaque mouvement possible faire
            on crée la grille temporaire
        val_maxi ← maximum de val_maxi et de mini_min_max(grid temporaire, player, depth - 2)
        on retourne val_maxi
```

Implémentation de min_max

✍ Implémentez les fonctions **`min_max_play(grid, player, depth)`**, **`mini_min_max(grid, player, depth)`** et **`maxi_min_max(grid, player, depth)`** qui renvoie le meilleur coup à jouer pour **`player`**. Cette fonction se base sur l'algorithme **`min_max`** (https://fr.wikipedia.org/wiki/Algorithme_minimax) pour décider du meilleur coup à jouer, et utilise l'heuristique de la question précédente.
Le paramètre **`depth`** détermine la profondeur d'exploration de **`min_max`**. Par exemple **`depth = 2`** veut dire explorer 2 coups plus loin (un coup pour **`player`** et un coup pour l'adversaire).

4. Module de jeu

✍ Créer un module **`play_tic_tac_toe.py`** qui va permette de jouer contre l'un des algorithmes définis au préalable.