[IA02] TP feuille 4 – Jouons au morpion

Information	Valeur
Auteur	Sylvain Lagrue (sylvain.lagrue@utc.fr (mailto:sylvain.lagrue@utc.fr))
Licence	Creative Common CC BY-SA 3.0 (https://creativecommons.org/licenses/by-sa/3.0)
Version document	0.1.3

I. Présentation du sujet

L'objectif est de ce sujet est d'implémenter les règles du Tic-Tac-Toe, une interface textuelle pour pouvoir jouer et enfin différents agnts artificiels, le tout en Python.

Les règles sont accessibles ici : https://fr.wikipedia.org/wiki/Tic-tac-toe (https://fr.wikipedia.org/wiki/Tic-tac-toe)

Un site sympathique si vous êtes intéressés par les mathématiques de ce jeu : https://zestedesavoir.com/billets/3672/combien-y-a-t-il-de-positions-au-morpion/ (https://zestedesavoir.com/billets/3672/combien-y-a-t-il-de-positions-au-morpion/)

II. Implémentation des règles

Structures de données

On représentera la grille de jeu à l'aide de tuples de tuples d'entier. L'entier `0` représente une case vide, `1` représente une case marquée par `X` et `2` une case marquée par `O`. Cette grille représente un état du jeu. L'objectif est de travailler avec des représentaion non-mutables.

Une action est un tuple des coordonnées de la case. Un joueur est représenté par un entier.

```
# Quelques structures de données

Grid = tuple[tuple[int, ...], ...]
State = Grid
Action = tuple[int, int]
Player = int
Score = float
Strategy = Callable[[State, Player], Action]

# Quelques constantes
DRAW = 0
EMPTY = 0
X = 1
O = 2
```

Préliminaires

Afin de faciliter les changements entre représentations mutables et non mutables de la grille (ex: quand vous aurez besoin de changer l'état du jeu après une action), écrire les fonctions suivantes.

```
def grid_tuple_to_grid_list(grid: Grid) -> list[list[int]]
```

```
def grid_list_to_grid_tuple(grid: list[list[int]]) -> Grid
```

Règles de base

Écrire les fonctions suivantes implémentant les règles du jeu.

```
def legals(grid: State) -> list[Action]
```

`legals` est une fonction qui étant donné un état renvoie l'ensemble des actions légales.

```
def line(grid: State, player: Player) -> bool
```

`line` renvoie vrai si la grille `grid` contient une ligne pleine du joueur `player`, faux sinon. Ne pas hésiter à décomposer cette fonction en plusieurs sous-fonctions...

```
def final(grid: State) -> bool:
```

`final` renvoie vraie si la grille `grid` est un état final, c'est à dire si un joueur a gagné ou s'il n'est plus possible de marquer une case.

```
def score(grid: State) -> Score
```

`score` est une fonction qui étant donné un état final, retourne le score du jeu.

```
def pprint(grid: State)
```

`pprint` est une fonction qui étant donnée un état fait une joli affichage de la grille. Par exemple :

```
O . X
X X X
. O O
```

```
def play(grid: State, player: Player, action: Action) -> State
```

`play` est une fonction qui étant donné un état, un joueur et l'action jouée par ce joueur, retourne le nouvel état de jeu.

III. Ajout de premiers joueurs et boucle de jeu

```
Strategy = Callable[[State, Player], Action]

# exemple
def strategy(grid: State, player: Player) -> Action
```

Une strategie est une une fonction qui étant donné un état et un joueur renvoie l'action choisie par ce joueur.

Joueur humain

Voici un exemple de joueur humain, c'est-à-dire une interface texte permettant de jouer.

```
def strategy_brain(grid: State, player: Player) -> Action:
    print("à vous de jouer: ", end="")
    s = input()
    print()
    t = ast.literal_eval(s)

    return t
```

Boucle de jeu

```
def tictactoe(strategy_X: Strategy, strategy_O: Strategy, debug: bool = False) -> Score
```

Écrire une fonction `tictactoe` qui étant données deux stratégies s'occupe de la boucle de jeu, du gagnant et renvoie le score à la fin du jeu.

Joueur first_action

```
def strategy_first_legal(grid: State, player: Player) -> Action
```

Ce joueur joue toujours la première action légale disponible.

Joueur aléatoire

```
def strategy_random(grid: State, player: Player) -> Action
```

Ce joueur joue au hasard l'une des actions légales disponibles.

IV. Joueurs intelligents

Minmax de base

```
def minmax(grid: State, player: Player) -> Score
```

Écrire la fonction minmax qui étant donné un état de jeu renvoie le score optimal de la partie.

Minmax avec une action

Écrire la fonction minmax qui étant donné un état de jeu renvoie sous forme de tuple l'action amenant au score optimal de la partie et ce score.

```
def minmax_action(grid: State, player: Player, depth: int = 0) -> tuple[Score, Action]
```

En déduire la fonction suivante.

```
def strategy_minmax(grid: State, player: Player) -> Action
```

Minmax indéterministe avec choix d'actions

Écrire la fonction minmax qui étant donné un état de jeu renvoie dans un tuple les actions amenant au score optimal de la partie et ce score.

```
def minmax_actions(
    grid: State, player: Player, depth: int = 0
) -> tuple[Score, list[Action]]
```

Utiliser cette fonction pour créer un minmax indéterministe.

```
def strategy_minmax_random(grid: State, player: Player) -> Action
```

Ajout d'un cache

Ajouter un cache aux fonctions précédentes en utilisant le *pattern* de mémoïsation.

Stratégie α-β

Améliorer les algorithmes précédents en utilisant les coupures α-β.

Pour aller plus loin…

Reprendre l'ensemble des questions précédentes en considérant une profondeur limitée et une fonction d'évaluation.

Améliorer le cache pour qu'il prenne en compte les symétries du jeu.

Annexes

Quelques grilles prédéfinies

```
EMPTY_GRID: Grid = ((0, 0, 0), (0, 0, 0), (0, 0, 0))
GRID_0: Grid = EMPTY_GRID
GRID_1: Grid = ((0, 0, 0), (0, X, O), (0, 0, 0))
# (0, 0, 0),
# (0, X, O),
# (0, 0, 0))

GRID_2: Grid = ((O, 0, X), (X, X, O), (O, X, 0))
#((O, 0, X),
# (X, X, O),
# (O, X, 0)

GRID_3: Grid = ((O, 0, X), (0, X, O), (O, X, 0))
#((O, 0, X),
# (0, X, O),
# (O, X, 0))

GRID_4: Grid = ((0, 0, 0), (X, X, O), (0, 0, 0))
#((0, 0, 0),
# (X, X, O),
# (0, 0, 0))
```