Puissance 4

On utilisera pour ce TP `Spyder`.

Le puissance 4 est un jeu d'alignement de pions. La situation initiale est une grille vide de 6 lignes par 7 colonnes. À son tour, un joueur place un jeton de sa couleur (rouge ou jaune) dans une colonne. Ce jeton tombe (par gravité) dans la première case non remplie de la colonne. Un joueur gagne la partie s'il aligne 4 jetons de sa couleur horizontalement, verticalement ou diagonalement.

C'est un jeu d'accessibilité à deux joueurs, les joueurs *rouge* (1) et *jaune* (2). On peut le modéliser par un graphe bipartie tel que :

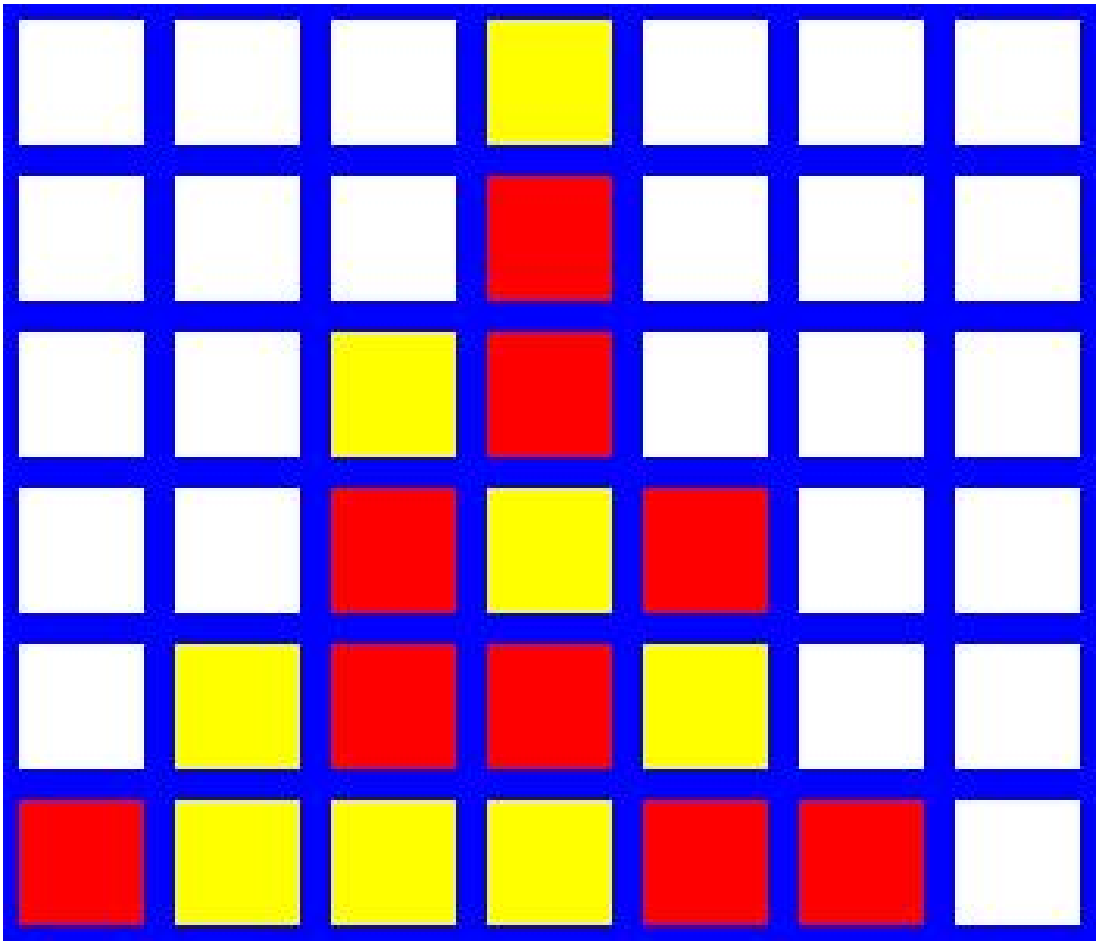
— Les sommets du graphes sont la données du joueur courant (rouge ou jaune) et de la configuration actuelle du plateau. La configuration sera représentée par une matrice 6×7 de 0 (case vide), de 1 (case rouge) et de 2 (case jaune). La ligne d'indice 0 correspondra à la ligne du *haut* du plateau. Par exemple :

```
grille = [[0,0,0,2,0,0,0],
          [0,0,0,1,0,0,0],
          [0,0,2,1,0,0,0],
          [0,0,1,2,1,0,0],
          [0,2,1,1,2,0,0],
          [1,2,2,2,1,1,0]]
```

Représentée textuellement par :

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

et graphiquement par :



- Les coups possibles à partir d'une configuration, sont ceux qui ajoutent exactement un jeton de la couleur du joueur courant dans l'une des colonnes, c'est à dire dans la première case vide d'une colonne qui en possède une. Un coup sera représenté par l'indice d'une colonne contenant au moins une case vide. Par exemple, dans la grille précédente : `2` est un coup possible, mais pas `3`.
- Les conditions de victoires sont :
  - Pour *rouge* : toute configuration de la grille avec au moins 4 jetons rouge alignés.
  - Pour *jaune* : toute configuration de la grille avec au moins 4 jetons jaune alignés.
  - Égalité : toute configuration de la grille sans aucune case vide et sans aucune séquence d'au moins 4 jetons alignés de la même couleur.

Un état du jeu (sommets du graphe) sera représenté par un *dictionnaire*, `state` ayant comme association :

- `"grille"` associé à la grille de jeux.
- `"joueur"` associé à 1 ou 2 selon que c'est à *rouge* ou *jaune* de jouer.
- `"vides"` associé au nombre de cases vides de la grille. Cette information est ajouté pour savoir, sans le calculer à chaque fois, si la grille est pleine.
- `"libres"` associé à une liste de longueur 7 telles qui contient à l'indice `i` le nombre de case vide sur la colonne `i` de la grille. Cette information est ajouté pour savoir, sans le calculer à chaque fois, à quel ligne tomberont les prochain jetons ajoutés, et cela pour chaque colonne.

Par exemple :

```
state = {
    "grille" : [[0,0,0,2,0,0,0],
                [0,0,0,1,0,0,0],
                [0,0,2,1,0,0,0],
                [0,0,1,2,1,0,0],
                [0,2,1,1,2,0,0],
                [1,2,2,2,1,1,0]],
    "joueur" : 1,
    "vides" : 25
    "libres" : [5, 4, 2, 0, 3, 5, 6]
}
```

I Fonction utilitaires

On commence par quelques fonctions permettant de jouer un coup.

Exercice 1. *Jouer un coup*

Implémenter une fonction `jouer_coup(state, col)` qui prend en entrée un état du jeu sous la forme décrite plus haut, `state`, un indice de colonne, `col`, représentant un coup à jouer et modifie l'état du jeu pour appliquer ce coup.

La fonction commencera par une *assertion* qui vérifie si `col` est bien un coup valide.

Exercice 2. *Annuler coup*

Implémenter une fonction `annuler_coup(state, col)` qui prend en entrée un état du jeu sous la forme décrite plus haut, `state`, un indice de colonne, `col`, représentant un coup *qui vient d'être* joué et modifie l'état du jeu pour *annuler* ce coup. On supposera que `col` correspond bien au dernier coup joué dans la grille, sans le vérifier.

Exercice 3. *Coup Gagnant*

Implémenter une fonction `coup_gagnant(state, col)` qui prend en entrée un état du jeu sous la forme décrite plus haut, `state`, un indice de colonne valide, `col`, et renvoie un booléen vrai si et seulement si le dernier jeton de la colonne `col` fait partie d'une séquence d'au moins 4 jetons consécutifs.

II Stratégies

On implémente une stratégie comme une fonction `Python` qui prend en entrée un état du jeux et renvoie un coup (un indice de colonne valide) à jouer.

Exercice 4. *Stratégie aléatoire*

Implémenter une fonction `strategie_alea(state)` qui prend en argument un état du jeu sous la forme décrite plus haut et renvoie un numéro de colonne choisi aléatoirement parmi les colonnes où un coup est possible.

La fonction commencera par une *assertion* qui vérifie si au moins un coup est possible.

On pourra utiliser la fonction `choice(l)` du module `random` qui renvoie une valeur choisie aléatoirement et uniformément de la liste `l`.

Exercice 5. *Partie*

Implémenter une fonction `partie(stratR, stratJ)` qui prend en argument deux stratégies, `stratR` pour le joueur *rouge* et `stratJ` pour le joueur *jaune*, et renvoie 1 si *rouge* gagne la partie, 0 s'il y a égalité et 2 si *jaune* gagne la partie.

L'état initial du jeux est une grille vide avec *rouge* qui commence.

Sur 10000 parties donnez le pourcentage de victoire de *rouge* sachant que les deux joueurs utilisent des stratégies aléatoire.

Exercice 6. *Heuristique*

Pour écrire une stratégie utilisant un algorithme de MinMax, on définie l'heuristique suivante. Chaque case du plateau rapporte un certain nombre de points, positivement si elle est occupée par le joueur *rouge* et négativement si elle est occupée par le joueur *jaune*. Les points sont données dans cette matrice. Les états gagnant pour *rouge* seront considéré comme ayant pour valeur `float('inf')` et ce *jaune* `-float('inf')`. Un état d'égalité aura comme valeur `0`.

```
points = [[3, 4, 5 , 7 , 5 , 4, 3],
          [4, 6, 8 , 10, 8 , 6, 4],
          [5, 8, 11, 13, 11, 8, 5],
          [5, 8, 11, 13, 11, 8, 5],
          [4, 6, 8 , 10, 8 , 6, 4],
          [3, 4, 5 , 7 , 5 , 4, 3]]
```

Implémenter une fonction `h(state)` qui prend un état du jeu sous la forme décrite en introduction et renvoie la valeur de l'heuristique pour cet état.

Exercice 7. *MinMax*

Implémenter une fonction `strategie_minmax(h,p,state)` qui prend en argument une fonction de calcul d'heuristique, un entier correspondant à une profondeur maximale et une partie de puissance 4 et renvoie le prochain coup à jouer selon l'algorithme du Min-Max.

Sur 50 parties donnez le pourcentage de victoire de *rouge*, utilisant la stratégie minmax à profondeur 5 et *jaune* utilisant une stratégie aléatoire.

On pourra définir la stratégie de rouge comme :

```
def strategie_minmax_5(state):
    return strategie_minmax(h, 5, state)
```

III Intéraction

On produit un affichage et une interaction utilisateur.

Exercice 8. *Interaction Joueur*

Pour faire une interaction joueur on peut implémenter une stratégie qui

- affiche la configuration actuelle de jeux,
- demande à l'utilisateur le numéro de colonne dans lequel il veut jouer, tant que cette colonne est possible.

Implémenter une telle stratégie et testez vos capacités contre un algorithme MinMax à différentes profondeurs…

Pour l'affichage, on pourra afficher (**print**) une grille dans la console, avec les caractère `['.','X','O']` pour les cases vides, rouges et jaunes respectivement.

Pour demander la colonne de l'utilisateur on pourra utiliser l'instruction `col = int(input("Colonne␣de␣jeux␣(0␣à␣`
On vérifiera que la colonne est valide et on redemandera à l'utilisateur si ce n'est pas le cas.