Algorithme du Min-Max appliqué au Puissance 4

Présentation

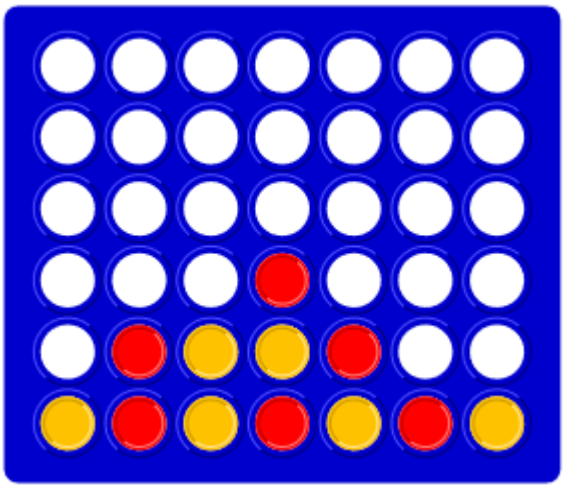
Le Puissance 4 est un jeu à 2 joueurs qui se joue dans une grille de 7 colonnes et 6 lignes.
Chaque joueur à tour de rôle fait tomber dans une colonne un jeton de sa couleur (rouge pour le joueur 1, jaune pour le joueur 2). Chaque jeton tombe alors le plus bas possible dans la colonne.
Le premier joueur qui aligne 4 jetons de sa couleur horizontalement, verticalement ou en diagonale gagne la partie.

Modélisation informatique

Dans la suite, une grille de jeu sera modélisée comme une matrice Python (une liste de listes) de 6 lignes et 7 colonnes.
Les valeurs dans cette matrice seront:

- 0 (pour une case vide, c'est le cas initialement pour toutes les cases),
- 1 (si la case contient un jeton du joueur 1)
- 2 (si la case contient un jeton du joueur 2).

La case en haut à gauche correspond donc la case plateau[0][0], la case en bas à droite à plateau[5][6].
Ainsi, la grille de jeu suivante



est représentée par:

Entrée[6]:

```
plateau1 = [[0, 0, 0, 0, 0, 0, 0],
            [0, 0, 0, 0, 0, 0, 0],
            [0, 0, 0, 0, 0, 0, 0],
            [0, 0, 0, 1, 0, 0, 0],
            [0, 1, 2, 2, 1, 0, 0],
```

Dans le script suivant, les fonctions suivantes sont fournies :

- `plateau_vide()` qui renvoie un plateau sans aucun jeton (toutes les valeurs sont à 0) ;
- `afficher(plateau)` qui réalise un affichage textuel du plateau ;
- `copie(plateau)` qui crée et renvoie une copie du plateau, sans la modifier.

Entrée[23]:

```
plateau1 = [[0, 0, 0, 0, 0, 0, 0],
            [0, 0, 0, 0, 0, 0, 0],
            [0, 0, 0, 0, 0, 0, 0],
            [0, 0, 0, 1, 0, 0, 0],
            [0, 1, 2, 2, 1, 0, 0],
            [2, 1, 2, 1, 2, 1, 2]]

def plateau_vide():
    """ plateau_vide()-> list
        sortie : liste de liste de 6x7 contenant des 0
           Renvoie un plateau vide, c'est-à-dire une matrice de 6 lignes et 7 co
        rempli de 0
    """
    return [[0 for _ in range(7)] for _ in range(6)]

def afficher(plateau):
    """afficher(plateau: list)
        Affiche le plateau, ne renvoie rien.:
        entrée : plateau, liste de liste représentant le plateau de jeu
    """
    conversion = {0:"⚪", 1:"🔴", 2:"🟡"}
    for ligne in plateau:
        for element in ligne:
            print(conversion[element], end="")
        print()
    print()

def copie(plateau):
    """ copie(plateau:list) -> list
        Renvoie une copie du plateau en argument, ne modifie pas plateau
        entrée : plateau, liste de liste représentant le plateau de jeu
        sortie : liste de liste, copie du plateau
    """
```

1- Écrire une fonction `jouer(plateau, joueur, colonne)` qui renvoie une copie de plateau dans laquelle joueur a joué dans la colonne en argument.
Si on généralise le jeu à une grille de n lignes et m colonnes, quelle sera la complexité de cette fonction ?
Comment pourrait-on modifier la structure de données pour avoir la fonction ci-dessus en O (1) ?

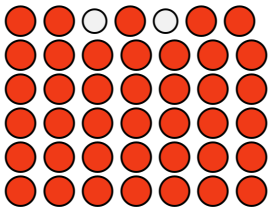
Entrée[24]:

```
def jouer(plateau, joueur, colonne):
    """ jouer(plateau :list, joueur: int, colonne : int) -> list
    Renvoie une copie du plateau dans lequel le joueur (1 ou 2) a joué dans la c
    colonne (en supposant que c'est possible)
        entrées : plateau, liste de liste représentant le plateau de jeu
                : joueur, entier, numero du joueur (1 ou 2)
                : colonne, entier, numero de la colonne dans laquelle on veut me
        sortie : liste de liste, représentant le plateau dans lequel a été ajout



    """
    pass

# afficher(plateau1)
```

2- Écrire une fonction `coups_possibles(plateau)` qui renvoie la liste des colonnes dans lesquelles on peut jouer (autrement dit elles ne sont pas complètement remplies).
Quelle est la complexité de cette fonction ?

Entrée[26]:

```
def coups_possibles(plateau):
    """ coups_possibles(plateau:list) -> list
        Renvoie la liste des indices de colonnes dans lesquels on peut jouer.
        entrées : plateau, liste de liste représentant le plateau de jeu
        sortie : liste comportant les indices des colonnes possibles
    """


plateauTest = [[1 for j in range(7)] for i in range(6)]
plateauTest[0][2] = 0
plateauTest[0][4] = 0
afficher(plateauTest)
print(coups_possibles(plateauTest)) #[2, 4]
```



None

Création d'une heuristique

On appelle segment une suite de 4 cases alignées (dans n'importe quelle direction).
Un segment est décrit par M (i , j) les coordonnées d'une case à l'extrémité, et

$$\overrightarrow{v} = (d_i, d_j)$$

le vecteur entre deux points consécutifs du segment ; le segment est donc constitué des points

$$(M, M+\overrightarrow{v}, M+2\overrightarrow{v}, M+3\overrightarrow{v})$$

Le score d'un segment est lié au nombre de jetons de chaque joueur dans le segment.

- Si on trouve des jetons de chaque joueur dans le segment ou s'il n'y a aucun jeton, alors ce score est 0.
- Sinon, on trouve la valeur dans un dictionnaire donné en argument. Dans l'exemple du `dico_score` ci-dessous, on peut lire que si un segment contient 3 jetons du joueur 1 et 0 jeton du joueur 2, alors le segment vaut 3 points. Un segment contenant 4 jetons de même couleur signifie que la partie est terminée, on donne donc un score suffisant dans ce cas pour déterminer le vainqueur.

Entrée[28]:

```
inf = float('inf')
dico_score = {(0, 1): -1, (0, 2): -2, (0, 3): -3, (0, 4): -inf,
```

3- Combien y a-t-il de segments dans un plateau au total ?

Entrée[ ]:

```

```

4- Écrire une fonction `score_segment(plateau, i, j, di, dj)` qui calcule le score d'un segment décrit par ses paramètres.

Entrée[30]:

```
def score_segment(plateau, i, j, di, dj, dicoS=dico_score ):
    """Calcule le score lié au semgent (i, j), (i + di, j + dj),
    (i + 2*di, j + 2*dj), et (i + 3*di, j + 3*dj)
        entrées : plateau, liste de liste représentant le plateau de jeu
                : i,j: entiers, coordonnees du jeton
                : di,dj : entiers, direction du segment
                : dicoS, dictionnaire des scores au format (Nb de jetons joueur1
        sortie : liste comportant les indices des colonnes possibles
    """

print(score_segment(plateau1, 5, 0, 0, 1, dico_score))
# 0
print(score_segment(plateau1, 2, 1, 1, 0, dico_score))
# 2
print(score_segment(plateau1, 2, 1, 1, 1,dico_score))
# -2
```

```
None
None
None
```

5- Écrire une fonction `score_plateau(plateau)` qui calcule le score du plateau, ie la somme des scores de tous les segments.
Remarque : il faut comprendre ici qu'un plateau avec un score élevé correspond à une situation plutôt avantageuse pour le joueur 1, un score faible correspond à une situation plutôt désavantageuse pour le joueur 1

Entrée[32]:

```
def score_plateau(plateau, dicoS=dico_score):
    """ Calcule le score lié au plateau entier
    (teste toutes les lignes, colonnes et diagonales).
        entree : plateau, liste de liste représentant le plateau de jeu
        sortie : score, entier, score du plateau entier
    """
print(score_plateau(plateau1))
#0
```

```
None
```

Une stratégie gloutonne

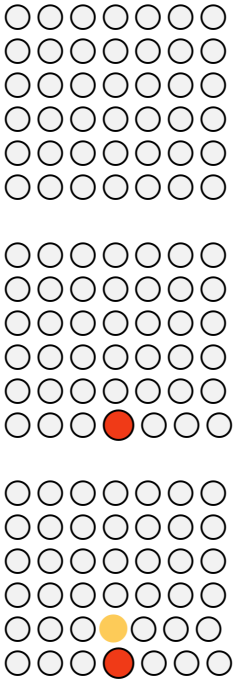
On s'intéresse dans un premier temps à la stratégie gloutonne : le joueur 1 (respectivement 2) joue, parmi les coups possibles, celui qui lui donne le score maximal (respectivement minimal). En cas d'égalité entre plusieurs coups possibles, on jouera dans la plus petite colonne possible.
6- Écrire les fonctions `strategie_j1_glouton(plateau, score_plateau)` et `strategie_j2_glouton(plateau, score_plateau)` qui renvoie une copie du plateau sur laquelle le coup optimal du joueur 1 et 2 respectivement a été joué, suivant la stratégie gloutonne.

Entrée[34]:

```
def strategie_j1_glouton(plateau, score_plateau):
    """Renvoie une copie de plateau sur laquelle le coup optimal du premier joue


def strategie_j2_glouton(plateau, score_plateau):
    """Renvoie une copie de plateau sur laquelle le coup optimal du premier joue

```

On suppose que les joueurs suivent cette stratégie.
7- Afficher les 3 premiers plateaux d'une partie.
Quel joueur va gagner cette partie ?

Entrée[35]:
```
1
```



Algorithme du Min-Max

Désormais chaque joueur veut prendre en compte les coups ultérieurs dans la partie. On définit de manière récursive le coup optimal à profondeur p , comme cela a été vu en cours.

Principe de l'algorithme

- Si le plateau est plein, on ne fait rien.
- Si p = 0, alors on joue un coup qui maximise (respectivement minimise) le score.
- Sinon, on cherche un coup optimal de l'adversaire à profondeur p −1. On joue un coup qui maximise (respectivement minimise) ce score optimal.
- Si on arrive sur une partie gagnée (score égal à +∞ pour le joueur 1 ou −∞ pour le joueur 2), c'est ce coup qui est joué

Par exemple, pour obtenir le coup optimal à profondeur 1, on regarde tous les coups possibles. Si un coup permet de gagner, ce coup est automatiquement considéré comme ayant un score de victoire. Pour les autres, on regarde le pire score possible après un coup de l'adversaire (on suppose qu'il joue le mieux possible), et on choisit le coup qui donne, parmi ces pires scores, le meilleur score (pour jouer le mieux possible).
8- Écrire des fonctions `maximin(plateau, p, score_plateau)` trouvant le coup optimal pour le joueur 1, c'est-à-dire avec le score maximal, et `minimax(plateau, p, score_plateau)` trouvant le coup optimal pour le joueur 2, c'est-à-dire avec le score minimal, mutuellement récursives. On pourra renvoyer la copie du plateau sur lequel ce coup optimal a été joué.

Entrée[36]:

```
def maximin(plateau, p, score_plateau):
    """Renvoie le coup optimal à profondeur p pour le joueur 1
        entrées : plateau, liste de liste représentant le plateau de jeu
                : p, entier, profondeur
                : score_plateau : nom de la fonction heuristique
        sortie : liste de liste, grille obtenue avec le meilleur coup
    """


def minimax(plateau, p, score_plateau):
    """Renvoie le coup optimal à profondeur p pour le joueur 2
            entrées : plateau, liste de liste représentant le plateau de jeu
                : p, entier, profondeur
                : score_plateau : nom de la fonction heuristique
        sortie : liste de liste, grille obtenue avec le meilleur coup
    """
```

Faire s'affronter la stratégie min-max avec p = 3 (pour le joueur 1) contre la stratégie gloutonne (pour le joueur 2).
Quelle stratégie gagne ? Faire de même avec différents niveaux de profondeur pour les stratégies min-max

Entrée[38]:

```
def jeu(p,score_plateau):
     """ Joueur 1 joue min-max avec profondeur p et Joueur 2 joue glouton
         entrées : p, entier, profondeur
         score_plateau : nom de la fonction heuristique
    """
    plateau = plateau_vide()
    joueur = 1
    while coups_possibles(plateau) != []:
        # à compléter
        afficher(plateau)
        s = score_plateau(plateau)
        if s == inf:
            pass # à compléter
        elif s == -inf:
            pass # à compléter
    # à compléter


jeu(3,score_plateau)
```

Pour aller plus loin :

Negamax

Reprendre l'algorithme du min-max pour utiliser négamax comme présenté dans le cours.

Entrée[ ]:

```
def negamax(plateau, p, score_plateau, signe):
    """     entrées : plateau, liste de liste représentant le plateau de jeu
                    : p, entier, profondeur
                    : score_plateau : nom de la fonction heuristique
                    : signe, entier valant 1 ou -1
        sortie : liste de liste, grille obtenue avec le meilleur coup
    """
```

changer d'heuristique

Essayer une autre heuristique. Vous pouvez modifier celle présentée dans ce sujet, ou bien utiliser celle présentée en cours, ou encore en concevoir une de votre propre imagination. Essayer de confronter les différentes heuristiques entre elles pour tester leur qualité.

Entrée[4]:

```
## Heuristique du cours

tableau_score = [[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]]

```