TP 10 - Théorie des jeux : Puissance 4

Dans ce TP, nous allons illustrer la théorie des jeux sur une application : le puissance 4. L'objectif est d'obtenir un programme capable de jouer "raisonnablement bien" contre nous au jeu Puissance 4.

I Présentation du problème

Ce jeu se joue sur une grille rectangulaire à 6 lignes et 7 colonnes. À tour de rôle, chaque joueur dépose un pion de sa couleur dans l'une des colonnes, à la position non occupée la plus basse (physiquement, on laisse tomber le pion à partir du haut de la colonne). Ceci n'est évidemment possible que lorsque la colonne n'est pas pleine (c'est-à-dire contient strictement moins de 6 pions). Un joueur remporte la partie lorsqu'il parvient à aligner (horizontalement, verticalement, ou en diagonale) 4 pions de sa couleur. La partie est déclarée nulle lorsque toute la grille est remplie mais aucun joueur n'est parvenu à aligner 4 pions de sa couleur. Nous reproduisons sur la figure ci-dessous une position en cours de partie (source : Wikipedia).

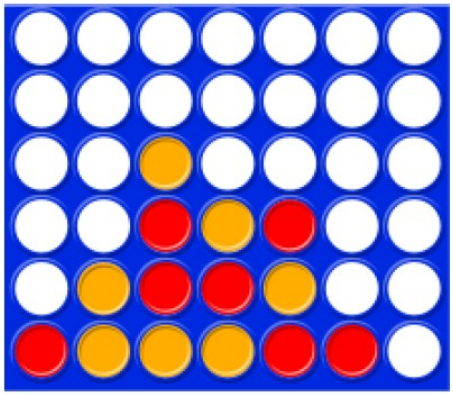

FIGURE 1 – Exemple de position de puissance 4

Q1. Indiquer si les hypothèses d'étude de jeux faites dans le cours sont respectées

On sait depuis 1988 que le joueur 1 dispose d'une stratégie gagnante, et qu'elle commence par un premier coup sur la colonne centrale. Il n'est cependant pas facile de calculer cette stratégie, car le graphe a un nombre de sommets considérable. Dans cet exercice, nous allons utiliser l'algorithme min-max et essayer de construire des stratégies les plus efficaces possibles. Dans toute la suite, nous adopterons la convention suivante. Une configuration de jeu est une liste double G, et $G[x][y]$ vaut 0, 1 ou 2 selon que la case de la colonne x et de la ligne y est vide, occupée par un pion du joueur 1 ou par un pion du joueur 2. Les indices x vont de 0 à 6, et les indices y de 0 à 5, la rangée la plus basse (et donc la première à être remplie) correspondant à $y = 0$.

Q2. Écrire une fonction Python `init ()` qui renvoie (sous forme d'une liste double, avec la convention indiquée précédemment) une grille G correspondant à la position initiale (toutes les cases sont vides)

Q3. Écrire une fonction `display(G)` qui prend en entrée une grille et l'affiche dans la forme ci-dessous. La valeur 0 dans la grille (case vide) est affichée avec un point, la valeur 1 (pions du joueur 1) avec la lettre `o`, et la valeur 2 (pions du joueur 2) avec la lettre `x`.

```
>>> G = init ()
>>> G [3][0] = G [4][0] = 1
>>> G [3][1] = 2
>>> display (G)
. . . . . . .
. . . . . . .
. . . . . . .
. . . . . . .
. . . x . . .
. . . o o . .
```

Pour tester plus facilement les alignements gagnants, et plus généralement pour évaluer plus facilement une position, nous allons construire la liste de tous les alignements possibles de 4 positions sur la grille. On peut montrer qu'il y en a exactement 69 (24 horizontaux, 21 verticaux, et 12 selon chaque diagonale).

Q4. Écrire une fonction Python `tous_alignements()`, qui ne prend aucun paramètre et retourne une liste composée des 69 alignements, chaque alignement étant une liste de 4 tuples à 2 éléments (par exemple, `[(0,0),(1,0),(2,0),(3,0)]` est l'un des alignements horizontaux). Dans toute la suite, nous supposerons que cette fonction est appelée au début du programme et que la liste obtenue est stockée dans la variable globale `tous`.

Q5. Écrire une fonction Python `coups(G)`, qui prend en entrée la grille de jeu G et renvoie la liste des coups possibles, sous la forme d'une liste de tuples de la forme `(x, y)`.

Q6. Écrire une fonction `fini(G)`, qui prend en entrée la grille de jeu, et renvoie la valeur 1 si un alignement a été complètement rempli par le joueur 1, la valeur 2 si un alignement a été complètement rempli par le joueur 2, et la valeur 0 dans les autres cas (on notera que la grille est supposée correspondre à une situation de jeu donc les deux joueurs ne peuvent avoir réalisé tous les deux un alignement). Cette fonction utilisera naturellement la variable globale `tous` évoquée plus haut.

Q7. Avec les fonctions `coups` et `fini` , nous sommes maintenant en mesure de simuler une partie aléatoire. Écrire une fonction `joue_random()`, qui simule une telle partie en choisissant à chaque étape un coup aléatoire parmi tous les coups possibles, et retourne la grille obtenue à la fin de la partie. On pourra utiliser la fonction `randrange` du module `random` pour effectuer le tirage aléatoire.

Q8. Estimer, en appelant un grand nombre de fois la fonction `joue_random`, le score moyen (entre 0 et 1) du joueur 1 pour une partie aléatoire (on conviendra que pour une partie, le score est 1 en cas de victoire, 0 en cas de défaite et 1 2 en cas de match nul). Si les deux joueurs jouent au hasard, quel joueur a l'avantage ?

Pour estimer à quel point une position est favorable (ou défavorable) à un joueur, nous allons construire une fonction d'utilité à partir des alignements calculés par la fonction `tous_alignements` (et stockés dans la variable globale `tous`). Pour tout alignement A (qui, on le rappelle, est un groupe de 4 cases alignées), notons $n_1(A)$ et $n_2(A)$ le nombre de cases de A occupées par des pions des joueurs 1 et 2 respectivement (on a donc $0 \leq n_1(A) + n_2(A) \leq 4$). Si $n_1(A)$ et $n_2(A)$ sont tous deux non nuls (ce qui signifie que l'alignement A est partiellement occupé par les deux joueurs), alors A ne pourra plus participer à la victoire d'un des 2 joueurs, donc il est logique de ne pas le faire contribuer à la

fonction d'utilité. En revanche, si $n_2(A) = 0$ (par exemple), cet alignement peut contribuer à la victoire du joueur 1, et ceci d'autant plus que $n_1(A)$ est élevé (le cas $n_1(A) = 4$ correspond à une victoire du joueur 1, le cas $n_2(A) = 3$ est très favorable, etc.). Nous pouvons donc définir une fonction d'utilité intéressante par :

$$U_k(G) = \sum_{A;n_{3-k}(A)=0} w[n_k(A)] - \sum_{A;n_k(A)=0} w[n_{3-k}(A)]$$

où $k \in 1, 2$ est le numéro du joueur considéré et w une fonction de poids vérifiant $1 = w[1] < w[2] < w[3] < w[4] = +\infty$

La valeur $w[0]$ sera choisie nulle ; elle n'a pas d'importance car les termes de ce type se compensent entre les deux sommes.

Q9. Écrire une fonction Python `utilite(G,k,w)`, qui prend en entrée la grille de jeu `G`, un entier $k \in 1, 2$, et une liste de poids w de taille 5, et retourne la valeur de l'utilité U_k donnée par la définition précédente.

Q10. Écrire une fonction Python `minmax(G,k,w,p)`, qui prend en entrée une grille de jeu `G` un entier $k \in 1, 2$ (numéro du joueur), une liste w de taille 5 (la fonction de poids), et un entier p (la profondeur d'analyse), et retourne l'utilité maximale à p coups $U_k^{(p)}$, ainsi qu'un coup (sous la forme d'un tuple `(x, y)`) choisi aléatoirement parmi ceux qui réalisent cette utilité maximale à p coups. Si aucun coup n'est possible (partie gagnée par l'un des joueurs, grille déjà remplie ou $p = 0$), la valeur `None` sera retournée à la place du coup. Cette fonction pourra être réalisée en suivant le schéma algorithmique suivant (les pointillés sont à compléter) :

$fonctionminmax(G, k, w, p)$
L = liste des coups possibles dans G pour le joueur k
... (traiter le cas $p = 0$ et le cas où la partie est finie)
$U_{opt} \leftarrow -1$
$c_{opt} \leftarrow []$
pour tout coup c de L
    $<jouer>c$ dans G
    $U, c_0 \leftarrow$ minmax(...)
    $U \leftarrow -U$
    ... (mettre à jour U_{opt} et c_{opt})
    $<dejouer>c$ dans G (pour revenir à l'ancien G)
choisir aléatoirement c dans c_{opt}
retourner U_{opt} et c

Q11. Écrire une fonction Python `partie(w,w2,p1,p2,affiche)` qui prend en entrée deux listes `w1` et `w2` (de taille 5), deux entiers `p1` et `p2`, un booléen affiche , et retourne la grille obtenue à la fin de la partie réalisée entre le joueur 1 (utilisant l'algorithme min-max de profondeur p_1 avec la fonction d'utilité définie par `w1`) et le joueur 2 (utilisant l'algorithme min-max de profondeur p_2 avec la fonction d'utilité définie par `w2`). Si le booléen affiche est vrai, la grille de jeu sera affichée à chaque coup. Tester sur quelques exemples avec des profondeurs 1 à 4 et une fonction de poids (égale pour les deux joueurs) donnée par la liste `w = [0, 1, 10, 100,1]`.

Q12. On souhaite comparer, pour une profondeur 2, l'efficacité des listes de poids ci-dessous :

```
w1 = [0, 1, 10, 1000,∞]
w2 = [0, 1, 10, 100,∞]
w3 = [0, 1, 10, 30,∞]
w4 = [0, 1, 2, 4,∞]
```

Pour chacun des 16 duels possibles, jouer 100 parties aléatoires et calculer le score moyen obtenu (le score d'une partie vaut 1 si le joueur 1 l'emporte, 0 si le joueur 2 l'emporte, 1 2 si la partie est nulle). On pourra afficher les résultats dans un tableau à 4 lignes et 4 colonnes. Que peut-on conclure ?

Q13. Écrire une fonction Python `humain_vs_machine` qui prend en entrée une liste de poids `w` et un entier `p`, et gère une partie interactive entre un joueur humain et la stratégie min-max associée à la fonction de poids `w` et la profondeur `p`. Le joueur qui commence sera tiré au hasard. On affichera à chaque étape la grille de jeu, et le programme demandera à son tour au joueur humain un numéro de colonne entre 1 et 7 (numérotation plus intuitive que 0 à 6). Réussirez-vous à gagner contre la machine pour `w = [0, 1, 10, 30,1]` et `p = 5` ?