Puissance 4 – Exemple d'algorithme MinMax

Algorithme Min Max

Définition – Principe

L'algorithme MinMax est un algorithme utilisé pour prendre des décisions dans des jeux à deux joueurs adversariaux. On a par exemple : les échecs, les dames, le morpion, etc… ou pour l'exmple que nous allons traiter le « Puissance 4 ». L'objectif de cet algorithme est de maximiser les chances de victoire du joueur qui le met en œuvre, tout en minimisant les chances de victoire de son adversaire. L'algorithme MinMax fonctionne de la manière suivante :

1. Le joueur utilisant cet algorithme explore de manière récursive tous les coups possibles à partir de l'état actuel du jeu, jusqu'à atteindre une profondeur maximale définie.

2. Pour chaque coup possible, il attribue une valeur. Cette valeur du coup est donnée par une fonction d'évaluation de l'état résultant de ce coup. La valeur du coup pour un joueur est une fonction croissante de la probabilité qu'a ce joueur de se rapprocher ou d'atteindre un coup gagnant.

3. Le joueur utilisant cet algorithme évalue la valeur pour lui et son adversaire. Puis il choisit le coup qui minimise la valeur de l'état pour son adversaire et maximise la valeur l'état pour lui-même. On peut ainsi définir l'utilité du coup pour un joueur comme la valeur du coup pour lui-même moins la valeur de ce même coup pour son adversaire.

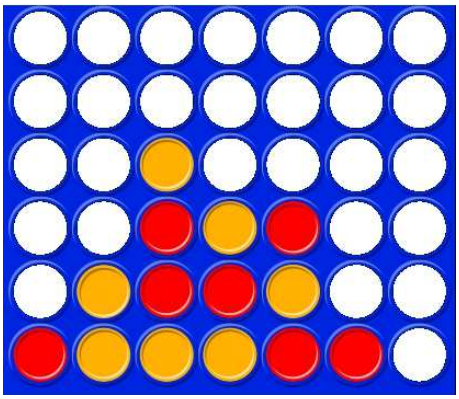
4. Le joueur peut se contenter d'évaluer l'utilité d'un état. La profondeur est alors de 0. Cela n'a pas franchement d'intérêt pour choisir un coup optimal. L'intérêt est dévaluer l'utilité de tous les états atteignables en un seul coup pour retenir l'utilité maximale et le coup optimal correspondant. La profondeur est alors de 1. On peut également le faire sur tous les états possibles après p coups successifs en ne retenant que le coup optimal (ou un des coups optimaux) pour le joueur qui joue un coup. La profondeur est alors de p.

5. Enfin, le joueur prend la meilleure décision possible en choisissant le coup optimal résultant de l'utilité maximisant ses chances de victoire tout en minimisant les chances de victoire de l'adversaire.

Le jeu du Puissance 4

Règles du jeu

Le but du jeu est d'aligner une suite de 4 pions de même couleur sur une grille comptant 6 rangées et 7 colonnes. Chaque joueur dispose de 21 pions d'une couleur (par convention, en général jaune ou rouge). Tour à tour, les deux joueurs placent un pion dans la colonne de leur choix, le pion coulisse alors jusqu'à la position la plus basse possible dans la dite colonne à la suite de quoi c'est à l'adversaire de jouer. Le vainqueur est le joueur qui réalise le premier un alignement (horizontal, vertical ou diagonal) consécutif d'au moins quatre pions de sa couleur. Si, alors que toutes les cases de la grille de jeu sont remplies, aucun des deux joueurs n'a réalisé un tel alignement, la partie est déclarée nulle. (Wikipédia)



Convention pour le codage

Un état du jeu est défini par une grille G qui est une liste de 7 listes contenant 6 entiers (« 0 » lorsque la position n'est occupée par aucun des joueurs, « 1 » lorsqu'elle l'est par le joueur 1 et « 2 » par le joueur 2). Les indices varient de 0 à 6 pour les colonnes et de 0 à 5 pour les lignes.

Ainsi à titre d'exemple pour la grille ci-dessus :

La position en bas à gauche est occupée par le joueur 1 : G[0][0] = 1 , celle juste à droite l'est par le joueur 2 : G[1][0] = 2 et la position en haut à droite par aucun des joueurs : G[6][5] = 0

1- Analyse du code fourni

Ouvrir avec pyzo le fichier « Puissance_4.py » puis exécuter le code correspondant.

Question 1. Que renvoie la fonction **`init()`** ? tester la fonction par les commandes **`G=init()`** puis **`G`**.

Question 2. La fonction **`display(G:list)`** qui prend en argument un liste de listes permet d'afficher la grille correspondante dans le shell. Quel sont les symboles permettant d'afficher une position libre ? Occupée par le joueur 1 ? Par le joueur 2 ? Pourquoi ligne 22 le deuxième indice est : **`5-y`** et pas **`y`** ?

Question 3. La fonction **`tous_align()`** renvoie une liste de tuples de 4 couples correspondant aux positions voisines alignées (verticalement, horizontalement ou en diagonale). Ce sont tous les alignements possibles sur une grille de 6 lignes × 7 colonnes. Combien a-ton de tels alignements. Vérifier cela par la commande **`len(tous_align())`**.

La ligne de code 49 : **`A_gagnants = tous align()`** permet de stocker cette liste de tous les alignements de 4 positions (potentiellement gagnants) dans une variable globale.

Question 4. Que renvoie la fonction **`fini(G)`** qui prend en argument une grille **`G`** ? Remarque un booléen est égal à 0 si c'est **`False`** et 1 si c'est **`True`**.

Question 5. Que renvoie la fonction **`coups(G)`** qui prend en argument une grille **`G`** ? Quel est la longueur maximale de cette liste ? Quelle est sa longueur si toutes les positions de la grille sont occupées (la grille est remplie) ?

La fonction **`joue_random()->(int,list)`** permet de jouer une partie de puissance 4 au hasard (L'ordinateur joue pour les deux joueurs). Elle renvoie le gagnant de cette partie et l'état de la grille à la fin de la partie. Le gagnant est 0 si elle se termine par un match nul (Toute la grille est remplie sans qu'il y ait de gagnant).

Question 6. Combien de fois au maximum est effectué la boucle **`while`** ? Quelle commande permet de changer de joueur ?

Les fonctions **`test_random()`** et **`probabilité_hasard(k)`** permettent respectivement de jouer une partie au hasard ou k parties au hasard. La première renvoie le gagnant et affiche la grille, la seconde compte le nombre de fois où chacun des joueurs a gagné et donne pour chaque joueur sa probabilité de gagner. Tester ces fonctions par les commandes **`test_random()`** et **`probabilité_hasard(2000)`**

Question 7. On constate qu'en général le joueur 1 gagne dans environ 55% des cas, qu'il y a moins de 1% de matchs nuls et donc que le joueur 2 ne gagne que dans environ 45% des cas. Qu'est-ce qui explique une telle différence ? (Vous n'êtes pas obligé de calculer ces probabilités, à moins que vous ayez un peu de temps à perdre et de belles compétences en probabilités)

2- Codage de l'algorithme MinMax

La fonction **`nbr_pions(A:tuple,G:list,joueur:int):->(int,int)`** prend en argument **`A`** : un tuple de 4 couples correspondant à un alignement, **`G`** : une liste de listes correspondant à une grille et **`joueur`** : un entier 1 ou 2 correspondant à un joueur. Elle doit retourner un couple d'entier **`n_joueur`** et **`n_autre_joueur`** correspondant respectivement pour la grille **`G`** aux nombre de pions du **`joueur`** dans l'alignement **`A`** et aux nombre de pions de l'autre joueur dans ce même alignement

Question 8. Implémenter la fonction **`nbr_pions(A:tuple,G:list,joueur:int):->(int,int)`**

Principe du calcul de l'utilité d'une grille pour un joueur « j »

Pour calculer la valeur d'une grille pour un joueur « j » qui est une fonction croissante de la probabilité pour ce joueur « j » de gagner nous allons utiliser une liste poids : W qui est une liste de 5 flottants. $W = [w_0, w_1, w_2, w_3, w_4]$

Pour chaque alignement parmi tous les alignements gagnants possibles nous allons déterminer les points qu'apporte cet alignement à la grille pour le joueur « j ». Pour cela on compte le nombre de pions $n_j(A)$ du joueur « j » et $n_{3-j}(A)$ du joueur « 3−j » présents dans l'alignement A. On utilise pour cela la fonction précédente.

Le nombre de points apporté par cet alignement A est de : $w_{n_j} = W[n_j(A)]$ si $n_{3-j}(A) = 0$. Si $n_{3-j} \neq 0$ l'alignement ne peut plus être gagnant pour le joueur « j » donc le nombre de point apporté par cet alignement est nul.

Ainsi la valeur d'une grille pour un joueur est la somme de ces points pour tous les alignements.

Enfin l'utilité d'une grille pour un joueur j est cette valeur de la grille du joueur « j » moins la valeur de la grille pour l'autre joueur : le joueur « 3−j ». Cette utilité se calcule donc avec la relation :

$$\mathbf{U_t(G) = \sum_{A:n_{3-j}(A)=0} W[n_j(A)] - \sum_{A:n_j(A)=0} W[n_{3-j}(A)]}$$

Nous allons tester l'algorithme MinMax pour différentes listes de poids W. Celles-ci sont données dans les lignes 111 et 112 du code. On a toujours $w_4 = +\infty$ car aligner 4 pions correspond à une victoire.

La fonction **`utilite(G:list,j:int,W:list)->float`** prend en argument une liste de poids **`W`** un joueur **`j`** et une grille **`G`**. Elle retourne l'utilité de cette grille **`G`** pour le joueur **`j`**. On utilisera pour cela la formule ci-dessus.

Question 9. Implémenter la fonction **`utilite(G:list,j:int,W:list)->float`**

Vous pouvez tester votre fonction avec les commandes : **`G=init()`** puis **`G[3][0]=1`** et enfin **`utilite(G,1,W1)`** qui doit renvoyer **`7`**. En effet si le joueur 1 joue le premier pions dans la colonne du milieu (position [3][0]) il y a 7 alignements possibles passant par cette position (un vertical deux en diagonale et 5 horizontaux. Comme on a qu'un seul pion des cas alignements l'utilité est de **`7×W1[1]`**

Nous arrivons maintenant au cœur de l'algorithme MinMax. Pour cela nous allons écrire la fonction **`minmax(G:list,j:int,W:list,p:int)->(float,tuple)`** . Cette fonction prend en argument une grille **`G`** , un joueur **`j`** , une liste de poids **`W`** et un entier p profondeur de l'algorithme MinMax. Elle doit retourner l'utilité optimale (maximale) et le coup optimal , celui qui maximise l'utilité suite à ce coup.

Cette fonction sera récursive. Ainsi si on souhaite maximiser ses chances de gagner avec une profondeur p nous calculerons par récursivité les utilité et coup optimaux de l'adversaire au coup suivant avec une profondeur p−1 pour en prendre son opposé.

Cette récursivité s'arrêtera lorsque la profondeur sera nulle que la partie sera gagnante (pour un joueur ou l'autre) ou terminée car la grille est remplie. Dans ce cas on renvoie l'utilité et **`None`** car il n'y a plus de coup possible.

Cette fonction sera réalisée en utilisant le schéma algorithmique suivant :

Fonction minmax(**G,j,W,p**)

On mémorise la liste des coups possibles pour la grille donnée (variable **L**)

On test les conditions de fin de récursivité

dans le cas où l'une des conditions est vrai on retourne l'utilité (**G,j,W**) et **None**

On initialise l'utilité maximale à $-\infty$ (variable **Ut_opt**)

On initialise la liste des coups optimaux conduisant à l'utilité maximale (variable **L_c_opt**)

Pour tous les coups possibles (dans **L**)

On joue le coup dans la grille **G**

On mémorise l'utilité optimal (profondeur : **p-1**) de l'adversaire après ce coup (variables **Ut**)

On prend l'opposé de cette utilité (**Ut = -Ut**)

Si l'utilité **Ut** est supérieure à l'utilité optimale

On met à jour l'utilité optimale : **Ut_opt = Ut**

On réinitialise la liste des coups optimaux

Si l'utilité **Ut** est supérieure ou égale à l'utilité optimale

On ajoute le coup à la liste des couts optimaux

On retire le coup pour revenir à la grille d'avant ce coup et tester le coup suivant.

On tire au sort un coup parmi la liste des coups optimaux qu'on mémorise dans la variable **c_opt**

On retourne l'utilité optimale : **Ut_opt** et le coup optimal tiré au sort : **c_opt**

Question 10. Implémenter la fonction **minmax(G:list,j:int,W:list,p:int)->(float,tuple)**

Vous pouvez tester votre fonction avec les commandes : **G=init()** et enfin **minimax(G,1,W1,1)** qui doit renvoyer **(7, (3, 0))**. Pour une grille vide le coup optimale est de joueur au centre de la grille. Coup pour lequel on a 7 alignements possibles avec un pion dans ces alignements. Pour les autres coups on a que 3, 4 ou 5 alignements possibles.

Avec les profondeurs 2 et 3 (**minimax(G,1,W1,2)** et **minimax(G,1,W1,3)**) c'est un peu plus compliqué à expliquer mais la fonction ne renvoie pas toujours la même chose car on effectue un tirage au sort entre des coups qui sont symétriques.

3- Test de l'algorithme MinMax

La fonction : **joue_partie(W:list,p1:int,p2:int,affiche:bool)->(int,list)** permet de jouer une partie avec la liste des poids W et des profondeurs différentes pour le joueur 1 (p1) et le joueur 2 (p2). Le dernier argument (un booléen) permet d'afficher la grille en fin de partie (si affiche = True) ou pas (si affiche = False).

La fonction : **mesure_force(W:list,p1:int,p2:int,k:int)** permet de jouer k parties avec les même arguments mais sans afficher les grilles pour juste compter la probabilité qu'a chaque joueur de gagner des parties.

Tester ces fonctions avec les listes de poids **W1** et **W2** pour différentes valeurs de **p1** et **p2** entre 1 et 4. Attention à partir d'une profondeur de 4 car les parties commencent à être longues à jouer. Conclure sur l'importance de la liste de poids et la différence de profondeur entre les deux joueurs.

La dernière fonction **joue_partie_h_m(W:list,p:int)** vous permet de jouer une partie contre l'ordinateur. Parviendrez vous à gagner avec la liste de poids **W2** et une profondeur de l'ordinateur de 4 ?