Compte-rendu Datascience IA – Puissance 4

Voici le rapport concernant les choix que nous avons fait concernant notre programme. Nous parcourons, plus ou moins brièvement, toutes les fonctions dans l'ordre par soucis de lisibilité.

Choix dans nos méthodes/test

Affiche(), Actions(), Result()

Tout d'abord, les fonctions classiques comme Affiche(), Actions() et Result() respectent plus ou moins ce que nous avons pu voir en TD, sans trop de modifications. L'unique devoir a été de modifier les tailles pour respecter les contraintes de la grille du puissance 4 (6 lignes, 12 colonnes).

TerminalTest()

Ensuite, la fonction TerminalTest() permet de savoir si le jeu est arrivé à un stade terminal, soit les 42 pions sont posés, soit un des deux joueurs gagnent par ligne, colonne ou diagonale.
La vérification par ligne et colonne est relativement la même qu'au morpion, sauf que ce sont des lignes et colonnes de 4.

Pour ce qui est de la vérification des diagonales, nous avons découpé cela en deux parties : les diagonales croissantes et décroissantes. Ces deux parties reposent sur le même principe : à partir d'un pion placé dans la grille, l'idée est de pouvoir redescendre et trouver les coordonnées du point de départ de la diagonale, qui contient le pion en question.
Cela passe alors par un jeu d'incrémentation/décrémentation des lignes et colonnes pour pouvoir retrouver le point de départ d'une diagonale croissante ou décroissante. Et par le même jeu d'incrémentation/décrémentation, on est capable de trouver le point d'arrivée de cette diagonale. Il ne reste alors qu'à parcourir cette diagonale et vérifier si le caractère en question revient 4 fois de suite sur la diagonale.

Utility()

Concernant la fonction Utility(), nous avons dû faire plusieurs choix pour permettre à notre IA de faire les bons choix dans le jeu, sans pour autant la surcharger et tomber dans des situations sans intérêt.

Premièrement, une grille de points a été implémentée, pour inciter notre IA à jouer, lors des premiers tours au milieu de la grille, qui est l'endroit où statistiquement il est plus favorable de compléter des lignes, colonnes ou diagonales. Pour exemple, notre IA préfère jouer colonne 6 ou 7 lors des premiers tours, car placer le pion dans ces colonnes lui fait augmenter son utility de 14, contrairement à la colonne 1 ou 12, qui ne lui augmente son utility que de 3.

Ensuite, nous avons fait le choix de récompenser généreusement les alignements de 4 pions, que ce soit en colonne, ligne ou diagonale, mais aussi les alignements de 3 pions qui peuvent être convertis en alignement 4. On accordera naturellement plus de points à un alignement de 4 qu'à un alignement de 3, mais le fait de récompense un tel alignement de 3 incitera notre IA à jouer de façon plus offensive.

Les récompenses qu'on accorde à une utility sont passées en paramètre de la fonction, ce qui nous a facilité la tâche lors de nos tests et notamment pour la fonction BestParam() (cf. ultérieurement).

De plus, notre fonction Utility présente également les récompenses qu'on décide d'attribuer à notre adversaire (dans le cas où on fait s'affronter notre IA contre elle-même, avec des paramètres différents, cf. BestParam()).

Pour tout ce qui est vérification de lignes/colonnes/diagonales, on a fait de simples boucles for pour balayer toutes les possibilités. On a également fait appel à une variable *previous* qui va nous permettre de vérifier si la case se trouvant avant une ligne de 3 par exemple est vide, nous permettant par la suite de récompenser cette action, car future ligne de 4 en perspective.
Le système de récompenses (des lignes/colonnes/diagonales de 3 pouvant donner une future victoire) est toujours présent si la ligne/colonne/diagonale en question se présente comme X X _ X et qu'il suffit de compléter le trou du milieu pour obtenir une victoire.

Jouer1(), Jouer2(), Jouer3()

Simplement, nos fonctions qui permettent, respectivement, de jouer face à l'IA en jouant en premier, de jouer face à l'IA en jouant en deuxième et de faire jouer l'IA face à elle-même avec des paramètres différents pour les deux IA. Pour cette dernière, il a simplement suffi d'appeler une fois notre fonction Alpha_Beta() (cf. ultérieurement) lors du tour de 'X', et une nouvelle fois lors du tour de 'O'.
On peut souligner l'utilisation de la bibliothèque time qui nous permet d'afficher le temps pris par l'IA pour jouer.

Elagage Alpha_Beta()

Cette fonction est celle qui va nous renvoyer la valeur et les coordonnées du pion à placer dans la grille. Cette fonction contient peu de choses, car fait appel à la fonction Max_Value_Alpha_Beta() (cf. ultérieurement). Elle permet uniquement de désigner le joueur qui jouera au prochain tour et retourne la valeur de Max_Value_Alpha_Beta().
De plus, elle prend en paramètre les récompenses qu'on a cité précédemment, ce qui nous a facilité la tâche lorsqu'on voulait faire affronter deux IA entre elles, sans avoir à tout rechanger dans les fonctions en elles-mêmes.

Max_Value_Alpha_Beta()

Cette fonction-ci est la fonctionqu'on utilise lorsqu'on arrive à un nœud n de l'arbre, où on souhaite maximiser l'utility du joueur.
C'est dans cette fonction-là (et Min_Value_Alpha_Beta(), cf. ultérieurement) qu'on retrouve, dans les paramètres, la profondeur, entre autres. Notre fonction respecte dans les grandes lignes l'algorithme de Max_Value vu en cours.
C'est notamment dans cette fonction qu'on fait appel à un système récursif, pour obtenir l'utility la plus élevée entre la valeur du nœud où on se trouve actuellement et la valeur du Min_Value_Alpha_Beta() précédent

Min_Value_Alpha_Beta()

Cette fonction-ci est la fonction qu'on utilise lorsqu'on arrive à un nœud n – 1 de l'arbre, où on souhaite minimiser l'utility de l'adversaire. C'est dans cette fonction-là (et Max_Value_Alpha_Beta()) qu'on retrouve, dans les paramètres, la profondeur, entre autres. Notre fonction respecte dans les grandes lignes l'algorithme de Max_Value vu en cours.
C'est notamment dans cette fonction qu'on fait appel à un système récursif, pour obtenir l'utility la plus élevée entre la valeur du nœud où on se trouve actuellement et la valeur du Max_Value_Alpha_Beta() précédent.

Best_Param()

La fonction Best_Param() est une mini-IA dans l'IA. Basée sur un système d'apprentissage supervisé, cette fonction récupère le résultat des paramètres du vainqueur lorsqu'on fait s'affronter notre IA avec elle-même. Et à chaque qu'un lot de paramètres bat le lot précédent, il prend sa place, ce qui nous permet d'obtenir le meilleur set de paramètres, après un nombre n d'itérations.
On fait notamment appel à la bibliothèque random et la fonction random.randrange(), pour faire jouer une part de hasard dans la création de nouveaux lots de paramètres et sélectionner le meilleur set au bout de 100 itérations, par exemple. Ce meilleur set pourra ensuite être récupérer manuellement et réimplanter dans cette fonction pour relancer 100 itérations à partir de ce meilleur set etc.