DM Othello

Luis RACCA, Jules MARÇAIS, Clément EDET

1 Introduction

La programmation du jeu d'Othello a été réalisée en Python.

Le jeu en lui-même a été fait en Programmation Orientée Objet en créant une classe Game qui a différents attributs décrivant l'état de la partie (liste de pions noirs et blancs, tour du joueur blanc ou noir, liste des coups légaux, etc) et différentes méthodes permettant de jouer (calcul des coups légaux, des pions à retourner lors d'une prise, vérification de fin de partie, etc).
Une interface graphique a également été développée avec le package Tkinter pour permettre au joueur humain de suivre la partie et de jouer.

Une fonction IA_play a été codée pour permettre au joueur humain de jouer contre l'ordinateur, ou de simuler une partie IA contre IA. A chaque exécution de cette fonction, l'ordinateur a quatre façons de choisir son prochain coup :

1. Choix au hasard d'un coup parmi la liste des coups légaux
2. Choix du meilleur coup par l'algorithme MinMAx
3. Choix du meilleur coup par l'algorithme AlphaBeta
4. Choix du meilleur coup par l'algorithme MCTS

Pour les algorithmes MinMAx et AlphaBeta, nous avons codé une fonction d'évaluation simple qui prend un array 8x8 de valeurs de position pour chaque case et le multiplie terme à terme avec la position actuelle représentée sous forme d'array 8x8, avec un 1 pour une case occupée par un pion blanc, un -1 pour un pion noir et un 0 pour une case vide. On prend alors la somme des valeurs de l'array résultant comme évaluation.
Un score positif correspond à un avantage pour les blancs, un score négatif à un avantage pour les noirs et un score nul à une position égale pour les deux joueurs. Plus le score est élevé en valeur absolue plus la position est avantageuse pour l'un des deux joueurs.
Exemple :

Valeurs de position

100	-20	10	5	5	10	-20	100
-20	-50	-2	-2	-2	-2	-50	-20
10	-2	-1	-1	-1	-1	-2	10
5	-2	-1	-1	-1	-1	-2	5
5	-2	-1	-1	-1	-1	-2	5
10	-2	-1	-1	-1	-1	-2	10
-20	-50	-2	-2	-2	-2	-50	-20
100	-20	10	5	5	10	-20	100

$*$

Position à évaluer

1	-1	0	0	0	0	0	0
0	1	0	0	0	0	0	0
0	-1	1	-1	0	0	0	0
0	0	0	1	-1	0	0	0
0	0	0	-1	1	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

$=$

Matrice résultat

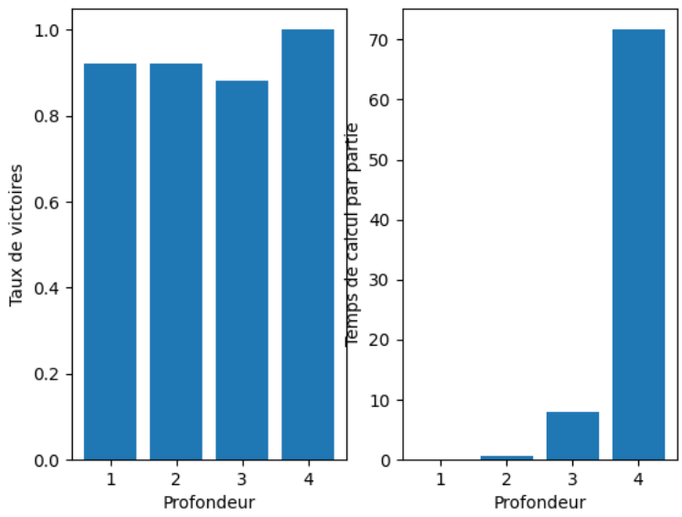
100	20	0	0	0	0	0	0
0	-50	0	0	0	0	0	0
0	2	-1	1	0	0	0	0
0	0	0	-1	1	0	0	0
0	0	0	1	-1	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

Ici cette position est évaluée à 72, les blancs ont donc l'avantage.

2 Évaluation des performances

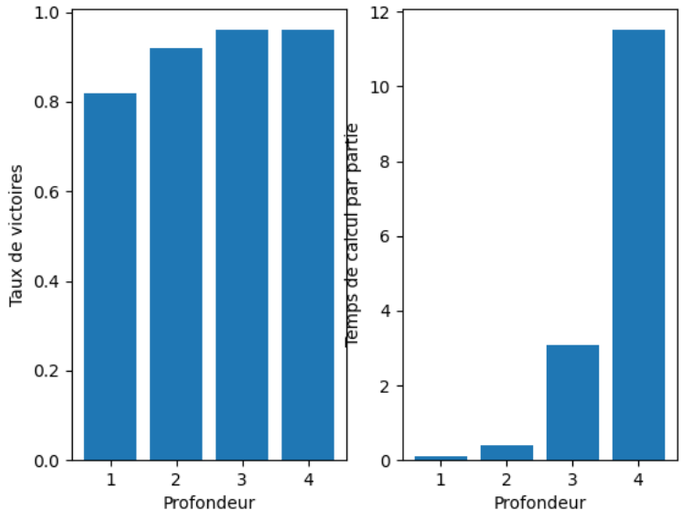
Pour implémenter les algorithmes de choxi du meilleur coup, nous avons crée en Programmation Orientée Objet une classe TreeNode qui permet de créer des arbres et ainsi de partir d'une position initiale et de parcourir les positions suivantes.
Dans chacun des cas suivants, on simule 50 parties IA contre IA où les blancs jouent avec un algorithme de décision et les noirs jouent de façon aléatoire. Les taux de victoires affichés correspondent au taux de victoires pour le joueur blanc.

2.1 MinMax



On observe un taux de victoires de 0.9 ± 0.05. On s'attendrait à ce que le taux de victoires augmente avec la profondeur mais peut-être que trop peu de parties ont été simulées.
On observe également une augmentation exponentielle du temps de calcul (affiché en secondes) avec la profondeur.

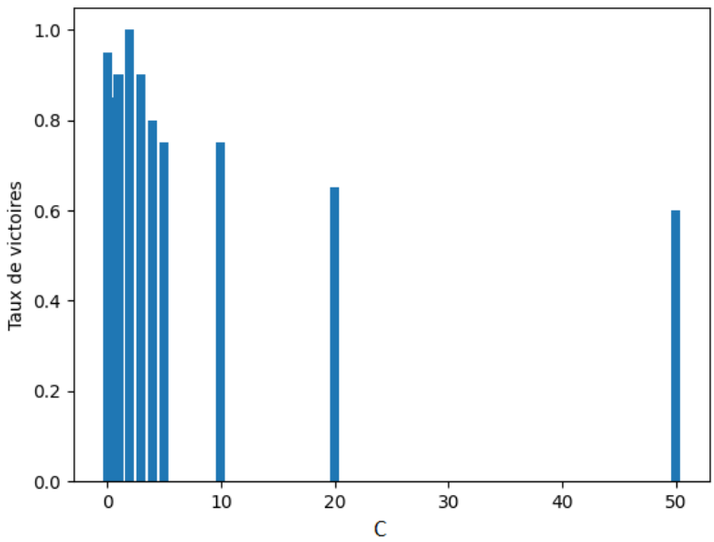
2.2 AlphaBeta



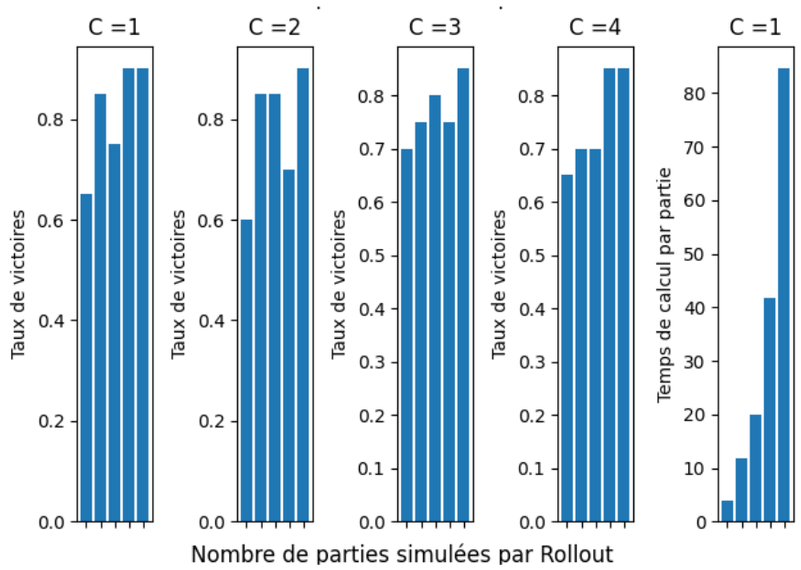
Cet algorithme renvoie le même coup que l'algorithme MinMax mais n'explore pas l'arbre entier pour le trouver.
Ainsi le taux de victoires est du même ordre de grandeur, de 0.9 ± 0.05 même si cette fois-ci on semble voir une augmentation de ce taux avec la profondeur.
On observe de même une augmentation exponentielle du temps de calcul avec la profondeur mais les calculs sont bien plus rapides avec une diminution du temps de calcul d'un facteur 6 environ.

AlphaBeta et MinMax étant tous deux déterministes, les parties simulées sont toutes identiques et conduisent à une victoire des noirs, c'est pourquoi nous n'avons pas jugé pertinent de calculer des statistiques dessus.

2.3 MCTS



On note "mcts_simul" le nombre de parties par Rollout.
Nous avons d'abord testé différentes valeurs de C à mcts_simul constant.
On observe un taux de victoires maximal pour C = 2 et une décroissance pour des valeurs grandes (>10)



Nous avons ensuite testé différentes valeurs de mcts_simul pour des valeurs de C proches de 2.
Chaque valeur de C a été testée sur 20 parties et pour 5 valeurs de mcts_simul : 1, 3, 5, 10, 20
On observe un taux de victoires croissant avec mcts_simul ce qui s'explique par une meilleure évaluation des nœuds et donc un meilleur choix du coup à jouer.
Cependant, le temps de calcul pour chaque partie est proportionnel à mcts_simul, il y a donc un choix à faire entre performance et vitesse.

3 Pour aller plus loin

La fonction d'évaluation très simple que nous avons utilisée pour MinMax et AlphaBeta fonctionne assez bien en première approche car l'IA va chercher à récupérer les coins qui sont des cases qui donnent un avantage. Mais son défaut est qu'elle est statique, en effet les cases adjacentes aux coins sont considérées mauvaises mais ce n'est plus le cas lorsque les coins ont été pris.
Une solution aurait été par exemple de créer un réseau de neurones dense avec en entrée un vecteur colonne 64x1 correspondant à la position, quelques couches cachées et une couche de sortie 1x1 pour renvoyer un score sous forme de scalaire avec une fonction d'activation sigmoïde par exemple.
On pourrait alors générer un grand nombre de positions différentes et simuler des parties (avec des coups aléatoires) pour déterminer une fréquence de victoire empirique et utiliser cette fréquence comme étiquette Y.
On utiliserait ensuite ces positions et leurs étiquettes pour entraîner le réseau à évaluer une position.