Rapport du projet en 2I013:
Développement d'agents intelligents pour les jeux Othello et Awélé

Encadrants:

LAMPRIER Sylvain

FRANCESCHI Jean-Yves

Toure Momar Faly

BOUAZIZ Dillan

Sommaire:

I- Les fonctions d'évaluation:

A - Othello:

Nous avons remarqué que les stratégies qu'on peut retrouver parmi les joueurs d'othello peuvent être implémentées en privilégiant certaines cases plutôt que d'autres. En effet, par exemple pour s'assurer d'avoir une mobilité importante, ou d'avoir des pions stables, c'est à dire des pions qui ne pourront pas par la suite être enlevés, il suffit par exemples les cases qui sont au coins. C'est ainsi donc que nous avons conçu la fonction composées des fonctions suivantes.

nombres_de_cases_coins(jeu):

Cette fonction retourne le nombre de pions qui se situent sur les coins après avoir jouer. Etant donné l'importance de s'accaparer des coins, cette fonction sera affectée d'un coefficient de **66**

nombre_de_mauvaises_cases_X(jeu):

Cette fonction retourne le nombre de pions qui se situent sur les cases qui sont adjacentes diagonalement aux coins, si ce coin est vide. Occuper ces cases, c'est prendre le risque de voir l'adversaire jouer aux coins et donc retourner l'ensemble des cases qui se trouvent dans toutes les directions de à partir du coin. Il est donc logique de lui attribuer un coefficient négatif. On a choisi de lui attribuer **-63**

nombre_de_mauvaises_cases_C(jeu):

Cette fonction retourne le nombre de pions qui se situent sur les cases qui sont adjacentes horizontalement et verticalement aux coins. Comme la fonction précédemment définie, on voit tout de suite l'importance de ne pas occuper ces cases. D'où le coefficient de **-60** qu'on lui a attribué

Au delà de ces fonctions, la fonction d'évaluation vérifie d'abord si le coup joué ne nous amène pas dans une situation de fin jeu, si c'est le cas, en fonction de l'éventualité, renvoie un résultat qui prime sur les autres. Et si c'est pas le cas, elle fait appelle aux fonctions définies plus haut.

$$\text{Evaluation(jeu)} = \begin{cases} \text{jeufini(jeu)} & \text{if game.finJeu(jeu)} \\ \sum_i w_i e_i & \text{else} \end{cases}$$

Les coefficients que nous avons eu ont été choisis arbitrairement suite à plusieurs simulations.

Résultats:

Nous avons donc testé la fonction d'évaluation avec une profondeur 1 et nous avons obtenu sur un ensemble de 1000 parties contre un joueur aléatoire un pourcentage de victoirs de **73%**

Interprétation:

Le pourcentage obtenu est assez raisonnable mais nous avons pensé qu'un des faiblesses de cette fonction d'évaluation, c'est le fait qu'elle traite les différents états de la même façon quelle que soit la phase de jeu dans laquelle on est. Par exemple une case du plateau n'a pas la même importance à travers toute la partie.

Pour pallier à ce problème, nous avons décidé de considérer l'environnement du jeu comme étant un système dynamique.

Partitionnement des etats de jeu:

Nous pouvons remarquer que Othello est un jeu de maximum 60 tours. Cette caractéristique permet de répartir toutes les configurations qu'on peut avoir en 60 grands groupes dans lesquelles les configurations ont en commun le fait qu'elles sont à la même phases de jeu.

Pour obtenir le tour dans lequel on est, nous avions eu une première idée qui était d'initialiser une variable **tour** à 0 et de l'incrémenter de 1 à chaque fois qu'on fait appel à la fonction saisieCoup(jeu). Puis nous avons remarqué que pour un plateau donné, le tour dans lequel on se trouve correspond à **nombre de pions qui se trouvent sur le plateau - 4 .**

On se retrouve donc avec une nouvelle fonction d'évaluation que l'on espère plus performant

Fonction d'évaluation dynamique:

On définit cette fois-ci une variable global :
tour = game.getScores(jeu)[0] + game.getScores(jeu)[1]
dans la fonction saisieCoup(jeu).

On a finalement donc la fonction suivante:

$$\text{Evaluation(jeu)} = \begin{cases} \text{jeufini(jeu)} & \text{if game.finJeu(jeu)} \\ \sum_i w_i e_i & \text{else} \end{cases}$$

$$\begin{cases} e_1 = \text{Nombre de pions aux coins} \\ e_2 = \text{Nombre de pions aux cases X} \\ e_3 = \text{Nombre de pions aux cases C} \\ e_4 = \text{Score obtenu} \end{cases}$$

Variables	Valeur statique	Valeur dynamique
W_1	66	66 - tour
W_2	-63	-63 + tour

W_3	-60	-60 + tour
W_4	0	if t>48 : 1

Après avoir effectuer un ensemble de 1000 parties, le nouveau joueur avait un pourcentage de victoires de **87 %** comparé au **73%** obtenu avec le joueur statique. Notre intuition était donc bonne, comme le montre le tableau ci-dessous:

Score	**Horizon 1**	**Alpha Beta 2**	**Alpha Beta 3**
Statique	**87,2%**	**91,2%**	**95,8%**
Dynamique	**73,9%**	**94,5%**	**97,3%**

Résultats obtenus en 1000 parties contre un joueur aléatoire

B- Awele :

Sur le jeu Awele il existe différentes manières de distinguer ce qui est un mauvais coup de ce qui est un bon coup c'est pourquoi nous allons nous intéresser à des éléments différents du jeu afin d'établir le score résultant de la fonction d'évaluation. Pour cela nous avons établis les fonctions suivantes :

getCasesDes(jeu,joueur):

Cette fonction retourne le nombre maximale des cases successives ayant seulement 2 ou 3 graines. En effet comme posséder des cases avec seulement 2 ou 3 graines est désavantageux car ces graines peuvent être capturés par l'adversaire, le score que retourne cette fonction sera négatif.

getCasesAv(jeu,joueur):

Cette fonction retourne le nombre maximale des cases successives chez notre adversaire ayant seulement 2 ou 3 graines. En effet comme posséder des cases avec seulement 2 ou 3 graines est désavantageux pour notre adversaire car ces graines peuvent être capturés par nous, le score que retourne cette fonction sera positif.

getNbGraines(jeu,joueur):

Cette fonction retourne la différence de graine entre le joueur et son adversaire.
En effet plus un joueur possède de graine moins il est susceptible de posséder des cases faibles et donc des cases ou les graines risquent d'être prises et inversement moins il a de graine plus il est susceptible de posséder des cases faibles (cases avec 2 ou 3 graines)

getFinDePartie(jeu,joueur):

Cette fonction retourne la valeur de fin de partie, ainsi elle indique le joueur qui a gagné à l'issue de la partie.

getPrises(jeu,joueur):

Résultats:

Nous avons donc testé la fonction d'évaluation avec une profondeur 1 et nous avons obtenu sur un ensemble de 1000 parties contre un joueur aléatoire un pourcentage de victoirs de **85%.**

Etant donné qu'il y avait encore de la marge de progression, nous avons décidé de concevoir un joueur utilisant l'algorithme minimax afin d'avoir une vision plus poussée lors de la prise des décisions.

II- L'algorithme minimax

La fonction minimax peut être principalement divisées en 3:

- une partie décision qui renvoie le coup estimé le meilleur
- une partie d'évaluation constituée par l'heuristique définie plus haut

- une partie estimation qui à partir d'un état donné fait une simulation jusqu'à la profondeur désirée

Résultats:

Avec le temps imparti sur le serveur, nous n'avons pu aller jusqu'à une profondeur de 2, tant pour awélé que pour othello. Ce qui s'explique par le fait par l'exploration de noeuds inutiles résultant ainsi à une perte de temps ; D'où l'idée de faire un élagage alpha-beta pour palier à ce problème.

III- L'algorithme Alpha beta:

Son objectif reste le même que celui de minimax, sauf que contrairement à minimax, alpha beta permet de faire un élagage permettant ainsi de n'exploiter que les noeuds qui nous seraient utiles

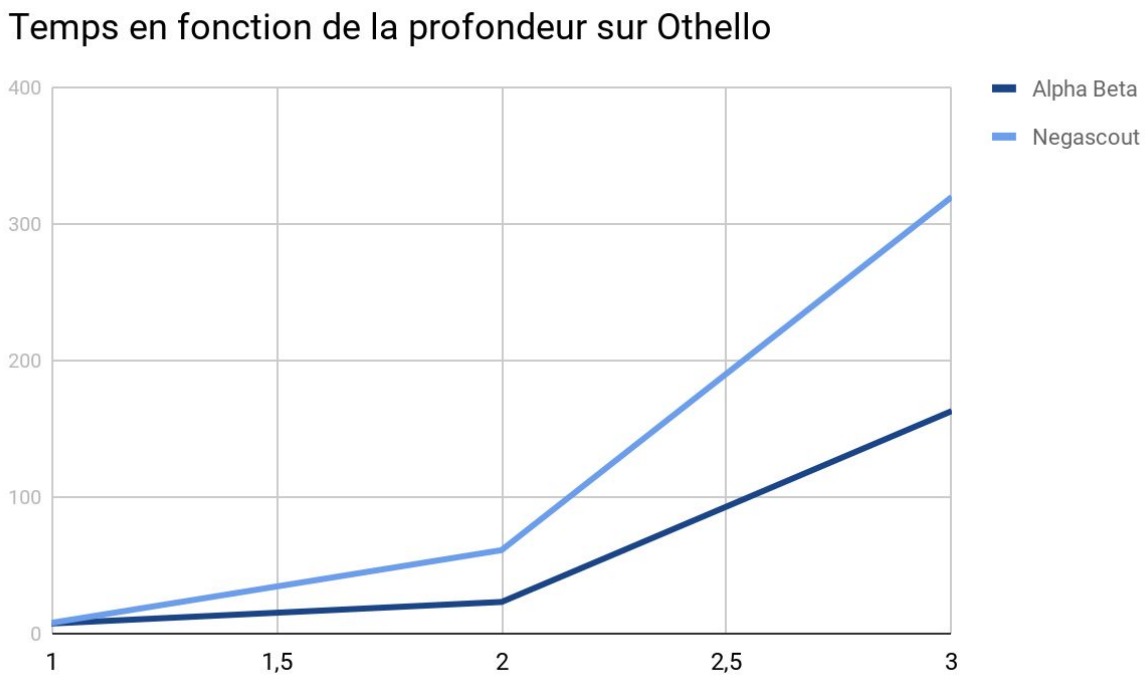
Résultats:

Cette fois-ci sur le serveur, nous avons pu aller jusqu'à une profondeur de 5 pour awélé et avoir un pourcentage de victoires de **59 %.**
Par contre, pour othello malgré le score de **59%**, on est resté à une profondeur de 2

IV- Negascout

Pour améliorer la profondeur de notre joueur pour othello, nous avons voulu essayé l'algorithme negascout, mais il se trouve ce dernier prend encore plus de temps. Ce qui peut s'expliquer probablement par l'ordre des actions qui ne se trouve pas dans la configuration la plus optimale. Nous pouvons le constater avec les résultats suivants:

Après avoir constaté l'algorithme minimax et ses variantes ne nous permettait pas d'avoir un bon résultat dans un temps convenable, nous avons donc essayé d'implémenter une nouvelle solution avec l'algorithme monte-carlo Tree search



Temps d'exécution (en secondes) en fontion de la profondeur maximale

V- MonteCarlo Tree Search

Contrairement à l'algorithme minimax, MCTS n'a pas besoin d'une heuristique, il est donc indépendant de nos biais ; ce qui constitue un de ces atouts principaux.
L'approche qu'on a choisi était de faire jouer l'algorithme contre lui-même pour un certain temps afin qu'il puisse établir une base donnée complète que l'on stockera grâce à la bibliothèque pickle.

Le fonctionnement de l'algorithme est résumé par les schéma suivants:

avec UCB1=

$$v_i + C \times \sqrt{\frac{\ln N}{n_i}}$$

https://towardsdatascience.com/monte-carlo-tree-search-in-reinforcement-learning-b97d3e743d0f

Et pour plus de détails, vous pouvez consulter les fichiers **entrainementmcts.py** et **joueurmcts.py** qui ont été commentés pour plus de clarté

Résultats:

Nous n'avons malheureusement pas pu avoir des résultats. Ce qui s'explique par la limitation des moyens disponibles. En effet, pour qu'on puisse utiliser le modèle il faut qu'il puisse être entrainé assez longtemps de façon à ce qu'il puisse gérer toutes les situations. Malheureusement, après deux heures d'entrainement, nous avons rencontré un **MemoryError**

Amélioration de monteCarlo Tree Search:

Hybride entre minimax et Montecarlo tree search

Une des faiblesses de MCTS c'est que lors de la phase de simulation (rollout), les coups sont choisis aléatoirement. Ce qui fait qu'il consomme beaucoup de temps. Pour palier à ce problème, on peut utiliser à ce stade un algorithme minimax qui pourrait

nous donner le coup le plus prometteur, nous épargnant ainsi donc de faire trop de simulations non nécessaire et gagner finalement du temps.

Utilisation d'un réseau neuronal

On utilise un réseau neuronal qui prend en argument état de jeu et qui renvoie un valeur comprise entre -1 et 1 en fonction de l'utilité du coup et une politique de choix des coups suivants.
L'utilisation de MCTS permettrait d'améliorer cette estimation de la politique de choix des coups.

VI- Résultats finaux:

Les résultats finaux sont établis dans le tableau suivant, pour 100 parties :

	Random	Horizon 1	Alpha Beta 2	Alpha Beta 3
Random	41%	13%	4%	0%
Horizon 1	87%	/	35%	0%
Alpha Beta 2	96%	65%	/	/
Alpha Beta 3	100%	100%	/	/

Tableau de confontration entre les joueurs pour le jeu Othello

	Random	Horizon 1	Alpha Beta 2	Alpha Beta 5
Random	44%	21%	10%	0%
Horizon 1	79%	/	0%	0%
Alpha Beta 2	90%	100%	/	/
Alpha Beta 5	100%	100%	/	/

Confrontation entre les joueurs pour le jeu Awélé