README

January 26, 2021

1 Implémentation d'un Othello fonctionnel (sans interface graphique)

Création d'un Othello en C, ainsi que d'un joueur ordinateur capable d'utiliser la meilleure stratégie de jeu selon l'algorithme Min-max.

Les règles du jeu sont décrites ici : https://www.ffothello.org/othello/regles-du-jeu/

1.1 Choix de la représentation :

Nous avons choisi ici de représenter la grille de jeu d'Othello par une matrice 8x8 qui contiendra des 0, des 1 et des 2. Il convient ainsi de préciser que : - 0 représente une case vide de la grille de jeu - 1 représente les pions noirs de la grille (il s'agit des pions du premier joueur) - 2 représente les pions blancs de la grille (il s'agit des pions du second joueur)

En pratique, la grille sera implémentée comme un tableau à deux dimensions `int grille[8][8]`.

1.2 Principales fonctions implémentées :

- `initialisation_grille` : elle initialise la grille de jeu en plaçant les 4 premiers pions.
- `coup_valide` : elle vérifie qu'un coup (i, j) est valide, c'est-à-dire qu'il peut être joué. On doit notamment vérifier que la case (i, j) appartient bien à la grille, et que le coup respecte les règles du jeu.
- `peut_jouer` : elle vérifie, pour un joueur donné, s'il peut encore jouer un coup sur la grille. En d'autres termes cette fonction vérifie qu'il reste ou non des coups valides pour un joueur.
- `partie_finie` : elle vérifie si une partie est terminée (aucun des joueurs ne peut plus jouer). La fonction `gagnant` calcule ensuite le score final de chacun des joueurs (le nombre de pions de chacun sur la grille).
- `jouer` : elle permet de jouer un coup (placer un pion sur la grille et retourner les pions adverses encadrés) en respectant les règles du jeu.
- `strategie_naive` : il s'agit de la première stratégie de jeu de notre IA. La stratégie consiste simplement à tester tous les coups possibles et à jouer le 1er coup valide rencontré.
- `strategie_minimax` : c'est la stratégie la plus aboutie de notre IA pour l'instant. Elle utilise l'algorithme minmax (décrit ici https://cutt.ly/9h1BkoJ) qui se résume au pseudocode suivant (copié depuis Wikipedia) :

```
function minimax(node, depth, maximizingPlayer) is
    if depth = 0 or node is a terminal node then
        return the heuristic value of node
    if maximizingPlayer then
```

```
        value := -ϖ
        for each child of node do
            value := max(value, minimax(child, depth - 1, FALSE))
        return value
    else (* minimizing player *)
        value := +ϖ
        for each child of node do
            value := min(value, minimax(child, depth - 1, TRUE))
        return value
```

L'algorithme minmax appliqué à Othello nécessite une heuristique afin d'évaluer les "noeuds de l'arbre des coups". C'est la fonction `eval_minimax` qui joue ce rôle. Elle associe à chaque case de la grille un score. Ce score est choisi selon les différentes stratégies connues du jeu d'Othello qui maximisent les chances de victoire d'un joueur.

Voici la matrice (que nous utilisons) qui attribue un poids à chaque case :

500	-150	30	10	10	30	-150	500
-150	-250	0	0	0	0	-250	-150
30	0	1	2	2	1	0	30
10	0	2	16	16	2	0	10
10	0	2	16	16	2	0	10
30	0	1	2	2	1	0	30
-150	-250	0	0	0	0	-250	-150
500	-150	30	10	10	30	-150	500

- `alpha_beta`: permet de faire un élagage alpha-bêta selon l'algorithme suivant :

```
fonction alphabeta(nœud,  ,  ) /*   est toujours inférieur à   */
   si nœud est une feuille alors
       retourner la valeur de nœud
   sinon si nœud est de type Min alors
           v = +ϖ
           pour tout fils de nœud faire
               v = min(v, alphabeta(fils,  ,  ))
               si    v alors  /* coupure alpha */
                   retourner v
                 = Min( , v)
    sinon
           v = -ϖ
           pour tout fils de nœud faire
               v = max(v, alphabeta(fils,  ,  ))
               si v    alors /* coupure beta */
                   retourner v
                 = Max( , v)
    retourner v
```

L'élagage alpha-beta permet d'optimiser grandement l'algorithme min-max qu'on a déjà implémenté (sans en modifier le résultat). Pour cela, il ne réalise qu'une exploration partielle de l'arbre des

possibilités.En effet, l'élagage alpha-beta n'évalue pas des nœuds dont on peut penser, si la fonction d'évaluation est à peu près correcte, que leur qualité sera inférieure à un nœud déjà évalué.

- `partie_2_joueurs` : simule une partie entre deux joueurs non machine.
- `partie_vs_computer` : simule une partie entre un joueur non machine et une IA.
- `computer_vs_computer` : simule une partie entre deux IA.

1.3 Usage :

Nous avons implémenté deux versions d'othello (chacune ayant ses propres fichiers sources) : la première sur une grille 4x4 (pour se débarrasser de la contrainte de l'heuristique et explorer entièrement l'arbre des possibilités) et une autre sur une grille 8x8. Pour tester les fonctions sus-décrites il suffit de compiler les fichiers sources (par exemple en utilisant gcc).

1.4 Difficultés rencontrées :

- Pour appliquer l'algorithme minmax à notre jeu, notre programme teste les coups jouables pour retenir celui avec la meilleure évaluation. Le problème était qu'à chaque appel récursif, la grille allait être modifiée en profondeur (chaque branche de l'arbre des possibilités va être pousuivie jusqu'au cas terminal) ; or on souhaite que pour deux noeuds de même niveau, la grille évaluée soit la même. De plus, annuler simplement un coup est une opération très difficile et coûteuse (il ne suffit pas d'enlever le pion placé mais il faut également regarder les pions adversaires retournés suite au coup). Pour contourner ce problème, on a décidé de travailler sur une copie de la grille pour une même profondeur dans l'arbre des choix.
- Notre fonction `strategie_minimax` retourne des scores de façon récursive. Or notre objectif final n'est pas d'avoir le score optimal, mais le coup qui réalise ce score. Vu qu'il y a plusieurs appels récursifs, il faut retourner ce coup à la bonne étape. On a utilisé une deuxième variable `depth`, constante au cours des appels de la fonction, et qui a la même valeur que le paramètre `profondeur` initialement, lui variable.
- Après nos premiers tests, nous avons observé des résultats assez contre-intuitifs : une IA qui utilise la stratégie minmax avec une profondeur 3 gagne contre une IA qui utilise la stratégie minmax avec une profondeur 4 ; une IA qui joue de façon naïve gagne parfois contre une autre qui utilise la stratégie minmax. Ces résultats nous ont conduits à douter de la validité de notre implémentaion et de l'heuristique que nous avions choisie. Pour vérifier que le corps de notre algorithme était correct, M. Zanni nous a suggérés d'effectuer des tests qui ne dépendent pas de notre heuristique : notamment explorer l'arbre des choix jusqu'aux feuilles (fin du jeu). Sur une grille de taille 8x8, parcourir entièrement l'arbre des choix est extrêmement coûteux. Nous avons donc effectué notre test sur une grille de plus petite dimension (4x4). On observe que dans cette configurtion, que la stratégie minmax (de profondeur "infinie") l'emporte systématiquement contre une IA naïve.
- Après avoir vérifié que notre algorithme mini-max était bien implémenté, on a constaté que son exécution était très lente pour de grandes profondeurs (à partir de 5 sur une grille 8x8). De la même façon, sur une grille 4x4, lorsque notre IA (avec une stratégie min-max) commence la partie, elle met énormément de temps pour jouer son premier coup (l'ensemble des coups possibles à explorer est trop grand en début de partie). C'est pour ces raisons, entre autres, que nous avons décidé d'implémenter un élagage alpha-bêta. Grâce à cette amélioration, on a gagné significativement en temps de calcul.