INSA INSTITUT NATIONAL DES SCIENCES APPLIQUÉES HAUTS-DE-FRANCE

Algorithmes de jeu de plateau à deux joueurs

Fondements de l'intelligence artificielle

Boudadi Liam, Caulier Rémi

Université Polytechnique HAUTS-DE-FRANCE

Table des matières

Liste des figures

I. Introduction

Ce papier vient rendre compte du développement et de l'implémentation de différents algorithmes dans le but de jouer au jeu de plateau d'Othello.

On détaillera par la suite l'implémentation du jeu en `C++`, les différentes classes et structures de données mises en place afin de communiquer avec les différentes intelligences artificielles. On présentera ensuite les intelligences qui ont pu être implémentées ainsi que les stratégies mises en place.

I.1. Architecture du projet

Le projet a été développé en C++, l'archive fournie contient donc les sources aux format `.cpp` dans le dossier `src` et les headers au format `.hpp` dans le dossier `includes`. Elle contient également un fichier `makefile` dans le but d'aider le lecteur à la compilation. Sinon cette dernière est possible de manière classique via `g++ src/*.cpp -o main -Iincludes`. Le dossier bin est utilisé pour stocker les fichiers objets lors de la compilation à l'aide de `makefile`.

L'architecture complète est la suivante :

```
Othello/
|- bin/
|  |- *.o
|
|- includes/
|  |- AInterface.hpp
|  |- Board.hpp
|  |- Game.hpp
|  |- MinMax.hpp
|  |- AlphaBeta.hpp
|  |- Random.hpp
|  |- Player.hpp
|- src/
|  |- AInterface.cpp
|  |- Board.cpp
|  |- Game.cpp
|  |- MinMax.cpp
|  |- AlphaBeta.cpp
|  |- Random.cpp
|  |- Player.cpp
|  |- main.cpp
|
|- makefile
|- rendu.pdf
```

Figure 1: Architecture du projet

II. Développement du jeu

La première étape avant l'implémentation d'algorithmes d'intelligences artificielles est le développement du jeu. Pour ce faire, deux classes principales ont été écrites : la classe `Board` et la classe `Game`. La classe `Board` gère le plateau d'une partie d'Othello tandis que la classe `Game` permet le bon déroulement d'une partie.

II.1. Classe `Board`

En premier lieu, la classe `Board` définie deux types d'énumérations :

- Le type `Pawn`
- Le type `Direction`

II.1.a. Type `Pawn`

Le type Pawn servira à représenter un pion dans le reste des sources. Sa définition est :

```
typedef enum : unsigned short
{
    EMPTY = 0,
    WHITE = 1,
    BLACK = 2,
} Pawn;
```

Figure 2: Enumération Pawn

II.1.b. Type `Direction`

Le type `Direction` représentera une direction dans les fonctions de vérification de validité des jeux et de placement des pions. Sa définition est :

```
typedef enum : unsigned short
{
    NONE = 0,
    TOP = 1,
    RIGHT = 2,
    BOTTOM = 4,
    LEFT = 8,
    DTR = 16,
    DBR = 32,
    DBL = 64,
    DTL = 128,
} Direction;
```

Figure 3: Enumération Direction

II.1.c. Coeur de la classe

Le premier objectif de la classe `Board` est de sauvegarder l'état du plateau. Cette sauvegarde est effectuée dans un tableau à une dimension de type `Pawn`. Pour accéder à une case du tableau on effectuera donc l'opération : $ligne * taille + colonne$, c'est le rôle de la fonction `coordToIndex(const std::string& coord) const;` qui prend en paramètre une coordonnée litérale (i.e. "b3") et qui la convertit en un index valide du tableau (i.e. "17").

La classe intègre également le joueur qui doit actuellement jouer (via le champ `Pawn currentPlayer`), cela permettra aux modèles IA d'effectuer les calculs (minimisation et maximisation

par exemple) de manière cohérente avec le joueur courant étant donné qu'un même joueur peut jouer plusieurs fois à la suite.

Enfin la classe gère possède la fonction `bool play(const std::string& coord);` qui place le pion du joueur courant à la coordonnée donnée. Cette fonction, si la coordonnée est valide (dans les bornes du plateau, à un emplacement vide, en respectant les conditions de placement), capture les pionts adverses et met à jour le joueur courant grâce à la fonction `void togglePlayer();`.

La fonction `togglePlayer` passe au joueur suivant, si ce dernier ne peut pas jouer, elle change de nouveau de joueur mais ne vérifie pas si après ce deuxième changement, le joueur peut jouer. Cette vérification est laissée à la classe `Game` qui s'occupe de gérer le bon déroulement d'une partie.

II.2. Classe `Game`

La classe `Game` initialise une instance de `Board` et possède deux fonctions principales :

```
void startGame(const AInterface& interface1, const AInterface& interface2)
```

et

```
Pawn analyseGame(bool verbose, bool displayGrid) const
```

La fonction `startGame` prend en paramètre deux interfaces d'intelligence artificielle (détaillées ci-après) correspondant au joueur noir et au joueur blanc et joue la partie.

La fonction `analyseGame` s'apelle lorsque la partie instanciée avec la classe `Game` est terminée. Cette fonction permet l'affichage de différentes statistiques telles que le gagnant, le nombre de pions capturés et la durée de la partie. Si le paramètre `displayGrid` est égal à `true` la fonction affiche également la grille finale.

III. Interfaces d'intelligence artificielle

Afin d'implémenter différents algorithmes d'intelligence artificielle au sein du programme nous avons en premier créé une classe mère disposant de la déclaration commune des différents attributs et fonctions dont chaque algorithme doit diposer.

Cette définition générale correspond à la classe `AInterface` dont la déclaration se trouve dans le fichier `AInterface.hpp`.

III.1. `AInterface.hpp`

Le standard définit par la classe `AInterface` contient les méthodes :

- `virtual std::string play(const Board& board) const = 0;`
- `Pawn getPlayer() const;`
- `void showScores() const;`

La fonction `play()` est décrite comme fonction virtuelle non définie dans le code de classe puisqu'elle est destinée à être implémentée dans les différents algorithmes d'intelligence.

La présence de cette fonction dans l'interface permet à la classe `Game` de l'appeler sans se soucier de l'algorithme utilisé.

Et les attributs :

- `Pawn player;`
- `Pawn ennemy;`
- `Strategy strategy;`
- `int payoff_matrix[64];`

La variable `payoff_matrix` correspond à la matrice des poids statistiques d'une grille `8x8`, ici nous utilisons la matrice présentée par la Figure 4

$$\begin{pmatrix} 500 & -150 & 30 & 10 & 10 & 30 & -150 & 500 \\ -150 & -250 & 0 & 0 & 0 & 0 & -250 & -150 \\ 30 & 0 & 1 & 2 & 2 & 1 & 0 & 30 \\ 10 & 0 & 2 & 16 & 16 & 2 & 0 & 10 \\ 10 & 0 & 2 & 16 & 16 & 2 & 0 & 10 \\ 30 & 0 & 1 & 2 & 2 & 1 & 0 & 30 \\ -150 & -250 & 0 & 0 & 0 & 0 & -250 & -150 \\ 500 & -150 & 30 & 10 & 10 & 30 & -150 & 500 \end{pmatrix}$$

Figure 4: Matrice de poids statistiques utilisée

III.2. `Minmax.hpp`

III.3. `AlphaBeta.hpp`

L'algorithme AlphaBeta est une amélioration de l'algorithme MinMax. Il permet de réduire le nombre de nœuds explorés en élaguant les branches inutiles.

Voici le pseudo-code de l'algorithme AlphaBeta que nous avons implémenté:

Algorithm 1: AlphaBeta

```
   entrées: nœud ; profondeur ; alpha ; beta ; joueurMax
   sortie: valeur heuristique de nœud
1  si profondeur = 0 ou nœud est terminal alors
2  |  retourner heuristique(nœud)
3  si joueurMax alors
4  |  valeur ← −∞
5  |  pour chaque enfant de nœud faire,
6  |  |  valeur ← max(valeur, alphabeta(enfant, profondeur−1, α, β, FAUX))
7  |  |  si valeur ≥ β alors
8  |  |  |  break
9  |  |  α ← max(α, valeur)
10 |  retourner valeur
11 sinon
12 |  valeur ← +∞
13 |  pour chaque enfant de nœud faire
14 |  |  valeur ← min(valeur, alphabeta(enfant, profondeur−1, α, β , VRAI))
15 |  |  si valeur ≤ α alors
16 |  |  |  break
17 |  |  β ← min(β,valeur)
18 |  retourner valeur
   Premier appel : alphabeta(racine, profondeur, -∞, +∞, VRAI)
```

Figure 5: Algorithme AlphaBeta

IV. Stratégies

Les stratégies *positionnelle*, *absolue*, *mobilité* et *mixte* ont été implémentée pour les algorithmes MinMax et AlphaBeta.

L'algorithme choisit l'heuristique correspondant à la stratégie donnée en paramètre lors de son initialisation.

IV.1. Positionnelle

L'heuristique positionnelle calcule le score du noeud en se basant sur la matrice de récompense vu en Figure 4. Le code est le suivant :

```
int MinMax::heuristic_pos(const Board &B) const
{
    int score = 0;
    for (int i = 0; i < B.getSize(); i++)
    {
        for (int j = 0; j < B.getSize(); j++)
        {
            int c = i * B.getSize() + j;
            if (B.getCoord(c) == this->player)
            {
                score += this->payoff_matrix[c];
            }
        }
    }
    return score;
}
```

Figure 6: Calcul heuristique positionnel

IV.2. Absolue

La stratégie absolue attribue une valeur au noeud en fonction du score du joueur. Le score étant représenté par le nombre de pions du joueurs correspondant, le code de l'heuristique est assez simple. Néanmoins on fera attention a la subtilité en fonction du joueur joué par l'algorithme pour ne pas avoir une valeur de noeud négative :

```
int MinMax::heuristic_abs(const Board &B) const
{
    // Care to the sign for the operation
    if (this->player == Pawn::BLACK)
    {
        return B.getBlackScore() - B.getWhiteScore();
    }
    else
    {
        return B.getWhiteScore() - B.getBlackScore();
    }
}
```

Figure 7: Calcul heuristique absolu

IV.3. Mobilité

L'heuristique mobilité se sert du dernier pion placé pour prioriser les déplacements dans les coins du plateau en se basant sur la matrice de récompense pour retourner le score scorrespondant.

Si le déplacement n'est pas joué dans un coin alors la valeur est le nombre de déplacement possible.

```
int MinMax::heuristic_mob(const Board &B, std::string move) const
{
    if (this->payoff_matrix[B.coordToIndex(move)] > 400)
    {
        return this->payoff_matrix[B.coordToIndex(move)];
    }

    return B.getValidMoves(B.getCurrentPlayer()).size();
}
```

Figure 8: Calcul heuristique mobilité

V. Utilisation du CLI

Une fois compilé (cf. Architecture du projet), le programme peut être lancé en ligne de commande de la façon suivante:

```
othello BLACK WHITE [--benchmark MONTANT] [--display-grid] [--only-final]
```

BLACK	Obligatoire. Interface jouant les pions noirs. Les valeurs possibles sont: `minmax`, `alphabeta`, `random` et `player`
WHITE	Obligatoire. Interface jouant les pions blancs. Les valeurs possibles sont: `minmax`, `alphabeta`, `random` et `player`
`--benchmark MONTANT`	Le programme joue autant de parties que `MONTANT` lui indique.
`--display-grid`	Affiche les grilles de fin de partie pour la ou les parties jouées.
`--only-final`	Affiche les résultats une fois que toutes les parties demandées ont été jouées. Remarque: Les accumulateurs statistiques sont uniquement affiché lorsque toutes les parties ont été jouées et non après chaque partie.

VI. Statistiques et Critiques

Comparons les différents algorithmes alimentés des différentes stratégies.

VI.1. Random - Random

Dans un premier temps vérifions que l'affrontement entre 2 algorithmes complètement aléatoire tend vers 1 partie gagnée sur 2 pour chaque joueur.

```
$ ./main random random --benchmark 1000 --only-final
Game 1000/1000

====== Résultats ======
[NOIRS ] Victoires:    442 (44.2%)
[BLANCS] Victoires:    518 (51.8%)
[******] Egalités:      40 (4%)
[EXEC  ] Temps moyen d'une partie: 0.887ms
```

Figure 9: Affrontement Random - Random sur 1000 parties

Malgré la proximité des 50% de parties gagnées pour les blancs, on peut néamoins remarquer un avantage pour ces derniers en jouant de manière totalement aléatoire. On peut potentiellement expliquer cet avantage comme étant dû à l'ordre de jeu. En effet, jouant en deuxième, les blancs peuvent capturer le pion joué par les noirs aux premier tour, offrant potentiellement un avantage.

VI.2. Minmax - Random

VI.2.a. Stratégie positionnelle

VI.2.b. Stratégie absolue

VI.2.c. Stratégie mobilité

VI.2.d. Stratégie mixte

VII. Problèmes rencontrés

VIII. Perspectives d'amélioration

VIII.1. Threading

Une première amélioration majeure à apporter est le threading de la recherche heuristique des algorithmes. En effet, de nos jours les ordinateurs possèdent de multiples coeurs et ne pas les utiliser nous prive d'une grande partie de la puissance de calcul disponible.

Un premier threading efficace pourrai être la création d'un thread par branche initiale de l'arbre de recherche. De cette façon on divise au premier tour par 4 la durée d'exploration de l'algorithme, et plus encore en milieu de partie.

VIII.2. Affinement des heuristiques

VIII.3. Implémentation de nouveaux algorithmes

Les algorithmes `minmax` et `alphabeta` sont des algorithmes déterministes, il est donc inutile d'y jouer plus d'une partie étant donné que ces dernières seront toutes identiques. Actuellement, le seul moyen de le départager reste de les faire jouer un nombre important de parties contre l'algorithme aléatoire et d'ensuite comparer les résultats.

Pour améliorer ce processus l'optimal serait d'implémenter un algorithme non déterministe tel qu'un algorithme de Monte-Carlo. Celui-ci s'exécuterai en un temps déterministe (non infini et non aléatoire) mais dont le résultat contient une part d'aléatoire. De cette façon les parties contre un algorithme déterministe ne seront jamais identiques et il sera possible de départager ces dernier de façon plus intéressante qu'avec un algorithme complètement aléatoire.

IX. Conclusion