

# **Algorithmes de jeu de plateau à deux joueurs**

Fondements de l'intelligence artificielle

Boudadi Liam, Caulier Rémi

## Table des matières

I. Introduction .....	3
I.1. Architecture du projet .....	3
II. Développement du jeu .....	4
II.1. Classe Board .....	4
II.1.a. Type Pawn .....	4
II.1.b. Type Direction .....	4
II.1.c. Coeur de la classe .....	4
II.2. Classe Game .....	5
III. Interfaces d'intelligence artificielle .....	5
III.1. AInterface.hpp .....	5
III.2. Minmax.hpp .....	6
III.3. AlphaBeta.hpp .....	6
IV. Stratégies .....	6
IV.1. Positionnelle .....	6
IV.2. Absolue .....	6
IV.3. Mobilité .....	6
V. Utilisation du CLI .....	6
VI. Statistiques .....	6
VII. Problèmes rencontrés .....	7
VIII. Perspectives d'amélioration .....	7
IX. Conclusion .....	7

## Liste des figures

Figure 1: Architecture du projet .....	3
Figure 2: Matrice de poids statistiques utilisée .....	6

# 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 au 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 définit 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;
```

#### 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;
```

#### II.1.c. Cœur 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 littérale (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 pions 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'appelle 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 disposer.

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

### III.1. `AInterface.hpp`

Le standard défini 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 enemy;`
- `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 2

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

Figure 2: Matrice de poids statistiques utilisée

### III.2. Minmax.hpp

### III.3. AlphaBeta.hpp

## 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

### IV.2. Absolue

### IV.3. 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

Comparaison des différentes stratégies et algorithmes mis en place.

## **VII. Problèmes rencontrés**

## **VIII. Perspectives d'amélioration**

Threading

Affinement des heuristiques

Implémentation negamax et montecarlo

## **IX. Conclusion**