Ministère de l'enseignement supérieur et de la recherche

Institut National des Sciences Appliquées
des Hauts-de-France

Département d'Informatique et de Cybersécurité

Rapport des Travaux Pratiques

Intelligence Artificielle Appliquée au Jeu d'Othello

Date : 15 Avril 2024

Professeur : René Mandiau - HDR

Elias BOULANGER
INSA Hauts-de-France - ICY
Université Polytechnique Hauts-de-France
elias.boulanger@uphf.fr

Table des matières

1 Introduction **1**

2 Analyse **2**
1 Modélisation et structures de données 2
1.1 Structure du projet 2
1.2 Bitboards 3
2 Algorithmes 5
2.1 Décaler un bitboard 5
2.2 Trouver les coups valides 6
2.3 Jouer un coup 6

3 Validation **7**

4 Discussion **8**

5 Conclusion **9**

A Algorithmes et Code **I**
1 Opérations Logiques I

Liste des Acronymes

IA Intelligence Artificielle

MSB Bit de Poids Fort

LSB Bit de Poids Faible

Table des figures

2.1 Exemple d'une configuration (non complète). 3
2.2 Configuration initiale du plateau de jeu. Source : eOthello. 4

A.1 Opérations logiques pour obtenir l'état du plateau. I
A.2 Opérations logiques pour définir l'état du plateau. I
A.3 Opérations de décalage pour les coups valides. II

Liste des tableaux

Chapitre 1

Introduction

Dans le cadre de ce travail pratique, nous avons abordé la conception et l'implémentation d'une Intelligence Artificielle (IA) pour jouer au jeu d'Othello. Ce jeu de réflexion à deux joueurs sur un damier de 64 cases, avec des pions de deux couleurs, présente un défi intéressant pour l'IA en raison de la complexité de ses règles et de son espace de recherche combinatoire. L'objectif principal était de développer un algorithme capable de jouer contre un joueur humain ou contre une autre IA.

Nous explorerons dans un premier temps la modélisation du jeu et les structures de données utilisées pour représenter le plateau et les pions. Nous décrirons ensuite les algorithmes développés, en particulier l'algorithme Minimax, et plusieurs de ses variantes. Nous présenterons également une fonction d'évaluation pour mesurer la qualité des positions.

De plus, nous détaillerons la conception d'un classifieur pour prédire le résultat d'une partie en cours, ainsi qu'évaluer la qualité d'une position. Nous expliquerons comment ce classifieur a été entraîné et comment il se compare aux autres heuristiques.

Enfin, nous discuterons des résultats obtenus et des améliorations possibles.

Chapitre 2

Analyse

Notre implémentation comprend notamment une représentation binaire, aussi appelé Bitboard, les algorithmes de recherches Minimax et Alpha-Beta, et un classifieur pour prédire le résultat d'une partie en cours. Nous détaillons également les structures de données utilisées, et les algorithmes de décalage pour les coups valides. Nous avons réalisé ce projet en Python, pour des raisons de simplicité et de rapidité de développement, notamment à propos de la visualisation, et du développement du modèle de deep learning via PyTorch [1].

1 Modélisation et structures de données

1.1 Structure du projet

```
Othello-Reversi/
├── config.yaml : fichier de configuration des paramètres globaux du projet
├── main : point d'entrée du programme, permet de réaliser des tests, lancer
│   des parties, faire jouer des algorithmes, etc
├── game.py : contient la logique de la boucle de jeu, permet de lancer une
│   partie selon les paramètres donnés
├── node.py : contient la classe Node, encapsulant un état du jeu, et les informations
│   nécessaires pour l'exploration de l'arbre de recherche, ou pour rejouer
│   une partie
├── strategies.py : contient les algorithmes de recherche, et retourne le plateau
│   après jouer un coup selon la stratégie choisie
├── next.py : contient les fonctions de calcul des coups valides, et celles
│   pour jouer un coup
├── heuristics.py : contient les fonctions d'évaluation basées sur des heuristiques
├── utils/
│   └── Ensemble de fonctions utilitaires telles que des tables d'heuristiques,
│       la visualisation, les opérations logiques, etc
├── model_pipeline.ipynb : notebook Jupyter pour data preprocessing, model training,
│   et evaluation
└── understanding_bitboards.ipynb : notebook Jupyter pour comprendre les bitboards,
    de nombreux exemples illustrent les opérations logiques
```

1. PyTorch est une bibliothèque de tenseurs optimisée pour l'apprentissage profond. Voir Documentation PyTorch.

Une partie peut être lancée depuis le programme `main.py`, ce dernier prendra les paramètres définies dans le fichier `config.yaml`. Par exemple, pour lancer une partie avec *Interface* entre un joueur *Humain* et un joueur *Negamax-AlphaBeta*, en utilisant une stratégie *Positionnelle* pour l'évaluation avec une profondeur d'exploration maximale de 4 coups, nous obtenons le fichier de configuration suivant :

```yaml
### Game Parameters ###
# Who plays against who.
# 0: human.
# 1: random (chooses randomly a possible move),
# 2: positional. (uses a heuristic table to define the quality of a position),
# 3: absolute (minimizes/maximizes the number of pieces for the opponent/player),
# 4: mobility (minimizes/maximizes the number of possible moves for the opponent/player),
# 5: mixed (uses a heuristic table to define the quality of a position).
# Mixed signifies using positional, then mobility, then absolute.
mode: [0, 2]

# Which MiniMax version to use.
# 0: Default MiniMax
# 1: Alpha-Beta pruning
# 2: Default Negamax
# 3: Alpha-Beta pruning Negamax
minimax_mode: [3, 3]

# Maximum depth of the search tree (minimax algorithm)
max_depth: [4, 4]

# Which heuristic table to use.
# 0: None
# 1: TABLE1
# 2: TABLE2
h_table: [0, 2]
```

```yaml
# Whether to display the board with a graphical interface
display: False
```

FIGURE 2.1 – Exemple d'une configuration (non complète).

Plus de paramètres sont disponibles, accompagnés de leurs valeurs par défaut, ainsi que des descriptions et commentaires.

1.2 Bitboards

Une représentation classique d'un plateau de jeu d'Othello est une matrice de 8x8, où chaque case peut contenir trois valeurs : par exemple (-1, 0, 1) pour les pions noirs, vides et blancs respectivement. C'est une représentation intuitive, et la première approche que nous avions envisagée. Cependant, nous avons finalement opté pour une meilleure alternative, plus efficace en termes de temps et d'espace : les bitboards.

En effet, au lieu de contenir 64 entiers de $32bits$ chacun (soit $2048bits$), nous pouvons encoder l'ensemble du plateau de jeu dans 2 entiers de $64bits$ chacun (soit $128bits$). Ce qui est 16 fois moins lourd ! De plus, les opérations logiques sur les bitboards sont très rapides,

un avantage supplémentaire pour les algorithmes de recherche.

Comment encoder un pion, un coup, ou un plateau de jeu ?

Nous pouvons en fait tous les encoder de la même manière, en utilisant un bitboard. Chaque bit représente une case du plateau, et est à 1 si la case est occupée par un pion, ou si le coup est valide. Le Bit de Poids Fort (MSB) correspond à la case $h8$, et le Bit de Poids Faible (LSB) à la case $a1$. Les positions sont usuellement notés de $a1$ à $h8$, où a est la colonne la plus à gauche, et 1 la ligne la plus haute. [Ros05]

Par exemple, la configuration initiale du plateau de jeu est la suivante :

— Les pions noirs sont en $d5$ et $e4$.

— Les pions blancs sont en $d4$ et $e5$.

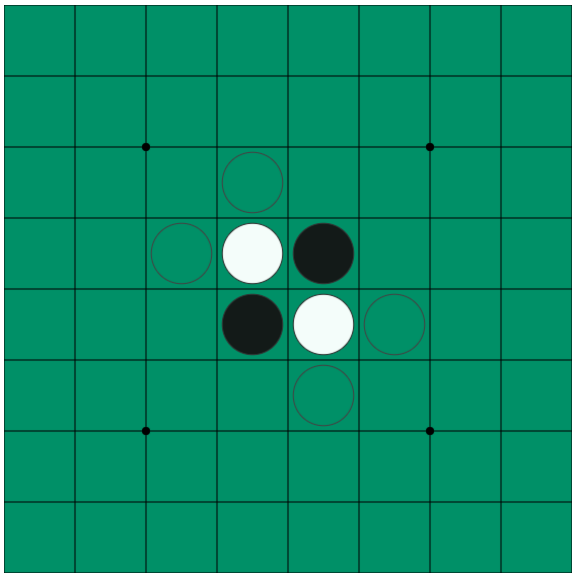

FIGURE 2.2 – Configuration initiale du plateau de jeu. Source : eOthello.

Cette dernière s'encode comme suit, avec dans l'ordre, le plateau noir, puis blanc :

$0b00000000\_00000000\_00000000\_00001000\_00010000\_00000000\_00000000\_00000000$
$0b00000000\_00000000\_00000000\_00010000\_00001000\_00000000\_00000000\_00000000$

Ces deux entiers peuvent être réécris en hexadécimal pour une meilleure lisibilité :

$$0x00\_00\_00\_08\_10\_00\_00\_00 \quad (noir)$$
$$0x00\_00\_00\_10\_08\_00\_00\_00 \quad (blanc)$$

Il est possible de récupérer la valeur d'une case $c_{i,j}$ d'un bitboard b en utilisant la formule suivante :

$$(b \quad \gg \quad (8 \times i + j)) \quad \& \quad 1$$

Avec i la ligne, j la colonne, $\gg$ l'opérateur de décalage à droite, et & l'opérateur logique '`et`'.

Similairement, nous pouvons définir une case $c_{i,j}$ d'un bitboard b en utilisant la formule suivante :

$$b \quad | \quad (1 \quad \ll \quad (8 \times i + j))$$

Avec $\ll$ l'opérateur de décalage à gauche, et | l'opérateur logique '`ou`'.

Nous obtenons toutes les pièces posées et toutes les cases vides avec les opérations logiques suivantes :

$$\begin{aligned} \text{pieces} &= \text{noir} \quad | \quad \text{blanc} \\ \text{vides} &= \quad \sim \text{pieces} \end{aligned}$$

Avec $\sim$ l'opérateur logique 'non'. Les programmes équivalents en python sont donnés dans l'appendice 1. De manière générale, n'importe quel pion peut être ajouté au plateau avec l'opérateur logique '`ou`'. Aussi, `8` peut être remplacé par n'importe quel entier représentant la taille du plateau, ici `8x8`.

2 Algorithmes

2.1 Décaler un bitboard

Le calcul des coups valides est une étape cruciale pour le jeu, c'est par ailleurs l'étape la plus couteuse en temps de calcul. La représentation en bitboards nous permet de calculer les coups valides de manière très efficace, en vectorisant les opérations. Il est en effet possible de calculer simultanément les coups valides qui sont dans la même direction, en utilisant des masques prédéfinis.

Pour cela, nous devons être capable de décaler un plateau entier dans les 8 directions cardinales. Nous y parvenons en utilisant les opérations logiques suivantes :

Algorithm 1 Opérations de décalage pour les coups valides.

```
1:  function NORD(x)
2:      return x ≫ 8
3:  end function
4:  function SUD(x)
5:      return (x & 0x00ffffffffffffff) ≪ 8
6:  end function
7:  function EST(x)
8:      return (x & 0x7f7f7f7f7f7f7f7f) ≪ 1
9:  end function
10: function OUEST(x)
11:     return (x & 0xfefefefefefefefe) ≫ 1
12: end function
```

Les masques permettent d'éviter un débordement ou *overflow*, une sortie de plateau, et de préserver les bords. Ils consistent à mettre à 0 les bits qui sont des positions sensibles dans la direction donnée. Nous obtenons ensuite $Nord - Est$, $Nord - Ouest$, $Sud - Est$, $Sud - Ouest$ en combinant les opérations précédentes. (Voir appendix 1 pour plus de détails).

2.2 Trouver les coups valides

Algorithm 2 Génération des Coups Valides avec Bitboards

```
1: function GénérerCoups(joueur, ennemi, taille)
2:     casesVides ←~ (joueur ∨ ennemi)
3:     coupsUniques ← []
4:     sautsDir ← {}
5:     for all direction in [nord, sud, est, ouest, nord_ouest, ..., sud_est] do
6:         compteur ← 0
7:         victimes ← direction(joueur) ∧ ennemi
8:         if victimes = faux then
9:             continue
10:        end if
11:        for i ← 1 to taille do
12:            compteur ← compteur + 1
13:            prochainePiece ← direction(victimes) ∧ ennemi
14:            if prochainePiece = faux then
15:                break
16:            end if
17:            victimes ← victimes ∨ prochainePiece
18:        end for
19:        captures ← direction(victimes) ∧ casesVides
20:        while captures ≠ 0 do
21:            capture ← captures ∧ −captures
22:            captures ← captures ⊕ capture
23:            if capture ∉ sautsDir then
24:                Ajouter capture à coupsUniques
25:                sautsDir[capture] ← []
26:            end if
27:            Ajouter (direction, compteur) à sautsDir[capture]
28:        end while
29:    end for
30:    return coupsUniques, sautsDir
31: end function
```

2.3 Jouer un coup

Chapitre 3

Validation

Chapitre 4

Discussion

Chapitre 5

Conclusion

Annexe A

Algorithmes et Code

1 Opérations Logiques

En 1.2, nous avons vu comment encoder en pseudo-code un pion et un plateau.

En Python :

```python
def get_state(bitboard: int, x: int, y: int, size: int):
      """Return the state of the cell by shifting the board
      to the right by x * size + y and taking the LSB"""
      return (bitboard >> (x * size + y)) & 1
```

FIGURE A.1 – Opérations logiques pour obtenir l'état du plateau.

```python
def set_state(bitboard: int, x: int, y: int, size: int):
    """Add a bit to the board by shifting a 1 to the left
    by x * size + y and performing a bitwise OR with the board"""
    return bitboard | (1 << (x * size + y))
```

FIGURE A.2 – Opérations logiques pour définir l'état du plateau.

En 2.1, nous avons vu comment décaler un bitboard dans 4 directions cardinales. Voyons maintenant comment décaler un bitboard dans les 4 directions diagonales, et leur équivalent en python.

Algorithm 3 Opérations de décalage en diagonales.

function NordEst(x)
    return $Nord(Est(x))$
end function
function NordOuest(x)
    return $Nord(Ouest(x))$
end function
function SudEst(x)
    return $Sud(Est(x))$
end function
function SudOuest(x)
    return $Sud(Ouest(x))$
end function

En Python :

```python
def N(x):
    return x >> 8
def S(x):
    return (x & 0x00ffffffffffffff) << 8
def E(x):
    return (x & 0x7f7f7f7f7f7f7f7f) << 1
def W(x):
    return (x & 0xfefefefefefefefe) >> 1
```

```python
def NW(x):
    return N(W(x))
def NE(x):
    return N(E(x))
def SW(x):
    return S(W(x))
def SE(x):
    return S(E(x))
```

Figure A.3 – Opérations de décalage pour les coups valides.

Bibliographie

[Ros05] Brian Rose. *Othello : A Minute to Learn, A Lifetime to Master*. 2005. URL : https://www.ffothello.org/livres/othello-book-Brian-Rose.pdf.