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

Liste des Acronymes

IA Intelligence Artificielle

MSB Most Significant Bit

LSB Least Significant Bit

Table des figures

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

Dans cette section, nous décrivons la modélisation du jeu et les algorithmes développés. Les principales particularités de notre implémentation sont les suivantes :

- Représentation binaire du plateau de jeu, des pions et des coups valides.
- Algorithmes de recherche Minimax, Alpha-Beta (avec ou sans Negamax)
- Fonction d'évaluation basée sur des heuristiques
- La possibilité de faire jouer n'importe quel algorithme contre un autre, ou contre un joueur humain.
- Classifieur pour prédire le résultat d'une partie en cours. Ce dernier peut également être utiliser pour évaluer la qualité d'une position.

Nous avons réalisé ce projet en Python, pour des raisons de simplicité et de rapidité de développement, notamment à propos des tests, de la visualisation, et du développement du modèle de deep learning via PyTorch [1].

Dans la littérature, un 'coup' est souvent défini comme l'action des deux joueurs pendant un tour, et un 'demi-coup' comme l'action d'un seul joueur. Cependant, dans le cadre de ce rapport, nous utiliserons plutôt le terme 'coup' pour désigner le fait de poser un pion.

1 Modélisation et structures de données

1.1 Structure du projet

Le projet est structuré de la manière suivante :

- `config.yaml` : fichier de configuration, contient les paramètres du jeu, des algorithmes, et des modèles.
- `main.py` : 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.
- `next.py` : contient les fonctions de calcul des coups valides, et celles pour jouer un coup.
- `strategies.py` : contient les algorithmes de recherche, et retourne le plateau après jouer un coup selon la stratégie choisie.

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

D'autres fichiers sont présents dans le doissier `utils/`, contenant des fonctions utilitaires telles que les fonctions d'évaluation, de visualisation, d'opérations logiques, etc.

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 Most Significant Bit (MSB) correspond à la case $h8$, et le Least Significant Bit (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 basse. [Ros05]

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

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

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

— Les coups valides sont en $c3$, $f6$ (noirs), $c6$, $f3$ (blancs).

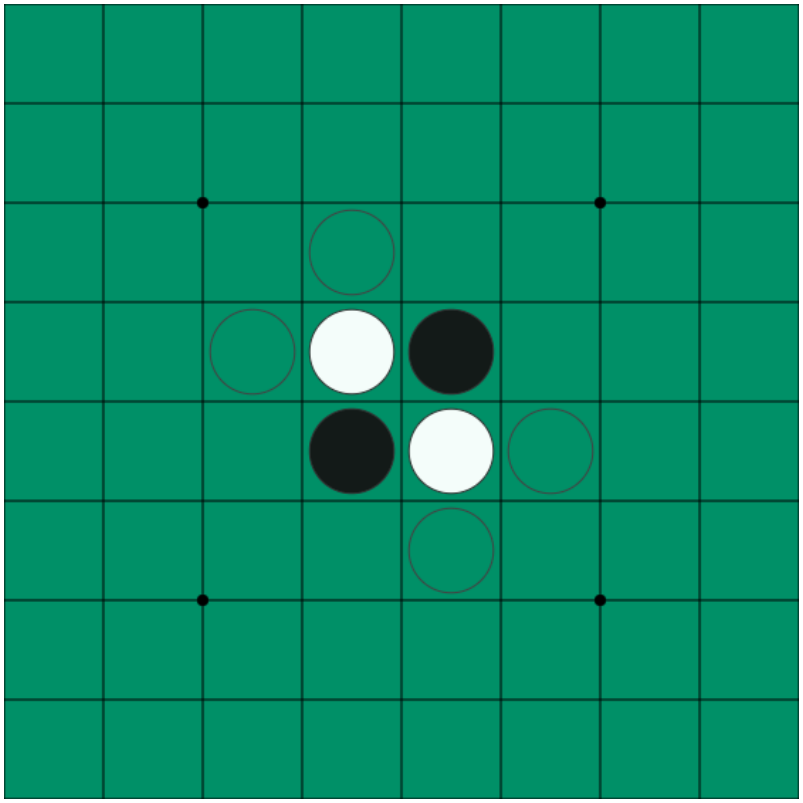

FIGURE 2.1 – Configuration initiale du plateau de jeu. Source : eOthello.

Elle s'encode comme suit. '0b', '0x' indiquent respectivement un nombre binaire et hexadécimal ; les underscores sont là pour faciliter la lecture :

Le plateau noir :

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

Le plateau blanc :

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

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 :

$$\text{pieces} = \text{noir} \quad | \quad \text{blanc}$$
$$\text{vides} = \sim \text{pieces}$$

Avec $\sim$ l'opérateur logique 'non'.

De manière générale, n'importe quel pion ou coup peut être ajouté au plateau avec l'opérateur logique 'ou'.

Comment 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, il va nous falloir être capable de décaler un plateau entier dans les 8 directions cardinales. Nous le faisons en utilisant des opérations logiques, comme le montre le pseudo-code ci-après.

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, ainsi que les codes python correspondant).

2 Algorithmes

2.1 Trouver les coups valides

2.2 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 1.2, 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 2 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.