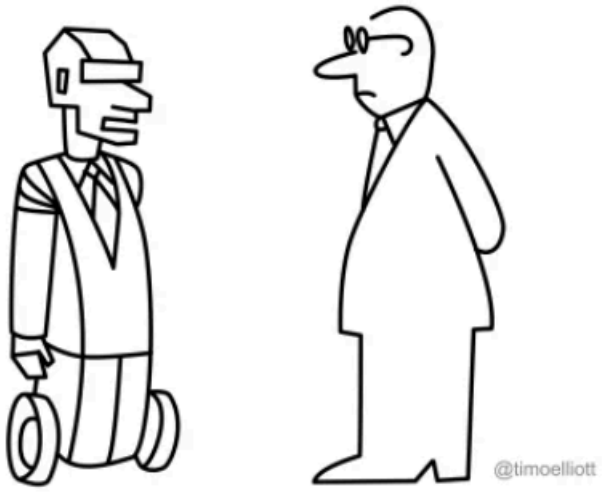
Stratégies d'IA pour Othello



"The good news is I have discovered inefficiencies.
The bad news is that you're one of them."

SAUSSE Sylvain - LAFONTAINE Robin
4A ICy

Fondement de l'IA
R. Mandiau
07-05-2024

Table des Matières

Table des Figures

I. Développement du cadre de travail

Lors du developpement d'une IA il est important d'avoir à sa disposition des outils et un cadre de travail définit afin de pouvoir se concentrer sur le fonctionnement de notre IA et ses heuristiques.

Nous avons décidé d'utiliser le language C++ lors de ce TP pour sa rapiditée d'execution (via la compilation) et son archetype orienté objet qui permet d'encadrer les interactions avec les differentes structures de données.

Cette partie du rapport detaillera les differents algorithme et structures mis en place lors de ce projet pour faciliter l'interaction des IA avec le jeu.

Nous avons réparti les tâches entre nous comme suit :

- **Développement du jeu :** Sylvain SAUSSE
- **Développement des IA et des heuristiques :** Robin LAFONTAINE

Chaque section du rapport a été rédigée par la personne responsable du développement de la partie correspondante.

I.A. Logique du jeu

Othello est un jeu de stratégie abstrait joué par deux joueurs sur un plateau de 8 x 8 cases ou chaque joueur, noir et blanc, pose l'un après l'autre un pion de sa couleur sur le plateau de jeu dit « l'othellier » selon des règles précises. Le jeu s'arrête quand les deux joueurs ne peuvent plus poser de pion. On compte alors le nombre de pions. Le joueur ayant le plus grand nombre de pions de sa couleur sur l'othellier a gagné.

Othello etant un jeu simple, notre implémentation se base sur 3 structures detaillées ci dessous. Toutes ces stuctures et methodes font partie du namespace `othello`.

I.A.1. Enumeration `othello::pawn`

L'enumeration `pawn` est le type de base utilisé dans la représentation du plateau de jeu.

```
typedef enum : unsigned short {
    empty = 0,
    black = 1,
    white = 2,
} pawn;
```

Figure 1: définition de `othello::pawn`

Cette représentation est basée sur le type `unsigned short` qui est codé sur 16 bits. ce choix permet d'avoir une représentation compacte (comparé au type `int` ou `unsigned int`) tout en ayant la capacité d'etre utilisable en tant qu'index pour les tableaux par exemple.

I.A.2. Classe `othello::Board`

La classe `Board` est le coeur même du jeu. Elle contient la représentation du plateau de jeu : un tableau de 64 `othello::pawn` (8x8 cases). le tableau est alloué dynamiquement à la création de l'objet et liberée à sa destruction.

```
class Board {
  private :
    // 64 unsigned short allocated array : contains the state of the board
    pawn *cases;
}
```

Figure 2: `othello::Board.cases` : représentation du plateau

Cette classe gère la logique du jeu au niveau du placement des jetons, mais pas au niveau du déroulement. Par exemple, cette classe est capable de verifier qu'un coup est valide lorsqu'on demande de l'effectuer et donc de l'annuler mais pas de savoir si c'est bien au joueur en question de jouer.

`othello::Board.case` est un attribut privé, il n'est donc modifiable que par des méthodes définies dans la classe `ohtello::Board`. Cela permet de s'assurer que la structure est manipulé seulement de la manière prévue dans les méthodes d'interactions.

I.A.3. Methodes d'interactions

Les methodes d'interactions sont les methodes publiques de la classe `othello::Board`.

I.A.3.a. Systèmes de coordonnées

Deux système de coordonées sont utilisable pour intéragir avec le plateau :

1. L'index dans la stucture de donnée, avec lesquel on peut facilement retrouver les valeur lignes - colonnes :

$$index = colonnes + (lignes \times 8) \iff (lignes \equiv index\,[8] \wedge colonnes = index \div 8)$$

Cette notation correspondant à l'index de la case dans le tableau de pion, elle est rapide mais difficile à comprendre pour une personne.

2. Les coordonées plus classiques composée d'une lettre et d'un chiffre (ex. "d7") qui sont utilisé pour l'entrée utilisateur.

Verifier si un coup est valide

La verification d'un coup valide se fait via la fonction `canPlaceHere` de la classe `Board`

```
direction canPlaceHere(int index, pawn team) const
```

Figure 3: Fonction canPlaceHere

La fonction prend en paramètre la coordonée où l'on souhaite placer le pion sous forme d'index, et l'équipe qui souhaite le placer. Après avoir verifié pour des raison évidente d'impossibilité (une pièce est deja présente sur la case par exemple) la fonction va ensuite verifier pour chaque direction si il est possible de le placer et ajoute au masque binaire `direction` la valeur correspondant à cette dernière.

```
typedef enum : direction {
    NODIR = 0,
    TOP = 1,
    BOTTOM = 2,
    LEFT = 4,
    RIGHT = 8,
    DTL = 16,
    DTR = 32,
    DBL = 64,
    DBR = 128,
} dirs;
```

Figure 4: Enumération dirs

Par exemple pour le coup suivant (les noirs essaient de placer un pion sur le point rouge), le retour de la fonction canPlaceHere sera de :

$$gauche + haut + diagonale\ haut\text{-}gauche = 4 + 1 + 16 = 21$$

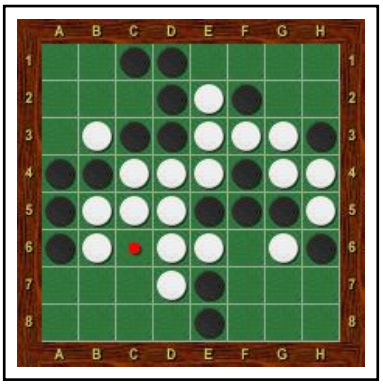


Figure 5: Exemple de coup joué

Lister les coups jouables

Le listage des coups jouables d'un joueur se fait à l'aide de la methode `listAllPlay` de la classe `Board`

```
std::vector<int> listAllPlay(pawn team) const;
```

Figure 6: Fonction canPlaceHere

Le listage des coups jouable est assez simple, on itère sur toute les cases du plateau et on teste la possibilitée de placement sur toute les cases du plateau avec la fonction `canPlaceHere`. La fonction retourne ensuite la liste des coups jouables sous forme de `vecteur` de coordonée sous forme d'index.

Durant no sessions de benchmarking nous avons noté un temps d'execution de $3 \pm 1\mu s$ ce qui va être important par la suite puisque cette fonction va etre appelée frequement par les IAs

I.B. Affichage et benchmarking

Notre jeu d'othello propose 2 interface pour jouer ou faire jouer des IA. Ces interface sont selectionnable et paramétrable via des arguments lors du lancement de l'application.

I.B.1. Affichage avec interface graphique

L'affichage avec iterface graphique utilise SDL2 pour permettre aux utilisateurs humain d'interagir plus naturellement avec le jeu. Pour jouer, il suffit de cliquer sur la case, les coups possibles sont affiché directement sur le board via un carré au centre des cases.

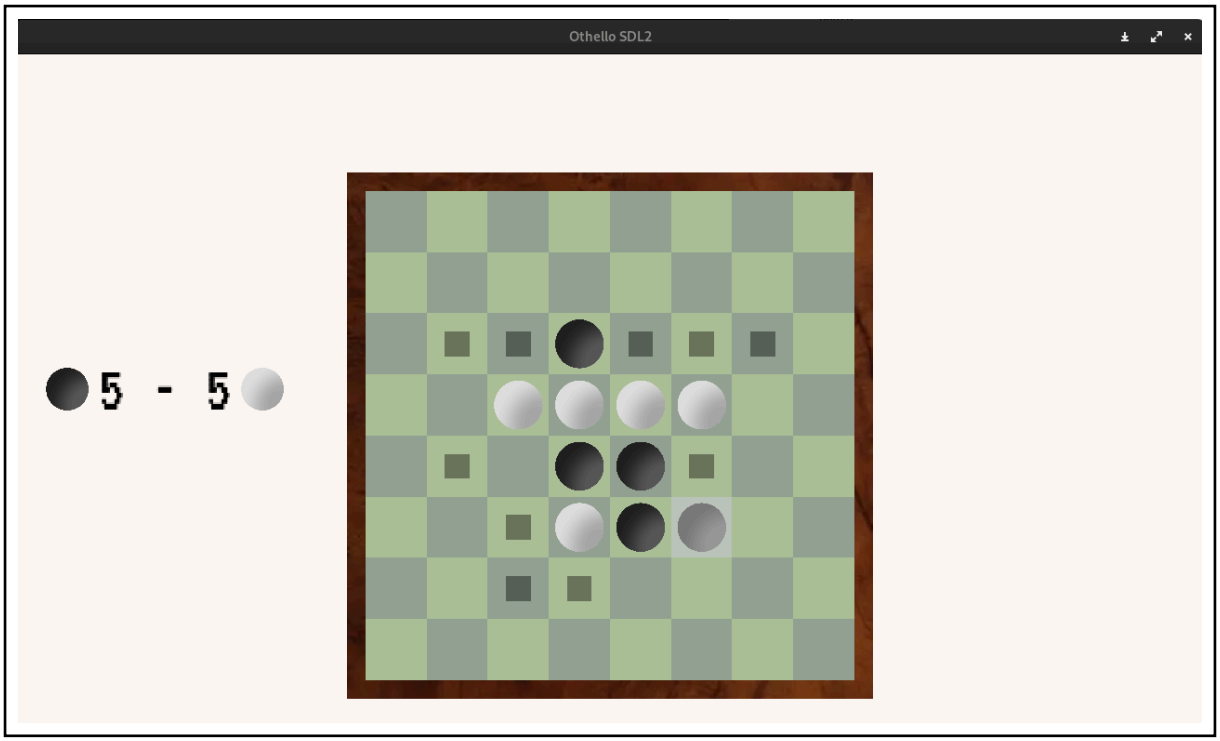


Figure 7: Interface utilisateur du jeu

Cette interface a pour but de permettre a un humain de jouer au jeu dans des condition normale de jeu. Elle n'a pas grande valeur lors du benchmarking.

I.B.2. Affichage CLI

L'affichage CLI est l'affichage pour le benchmarking. Il admet des paramètre supplémentaire qui permettent de tester les IA de manière plus précise, nottament en permettant de jouer plusieurs partie à la suite afin de faire des analyse statistique.

```
                      othello by 0xf7ed0 & Kappaccino
--------------------------------------------------------------------------------

  othello [--gui | --no-gui] [--IA1 <name>=<depth>] [--IA2 <name>=<depth>] [args]

--------------------------------------------------------------------------------

  --gui                     enable the GUI to play
  --no-gui                  play in CLI (default)
  --progress                show real-time game progres (CLI only)
  --no-result               don't show game results (CLI only)
  --gamecount <count>       set the number of games to play
  --IA1 <name>=<depth>      set player1 (black) to AI with depth
  --IA2 <name>=<depth>      set player2 (white) to AI with depth
```

Figure 8: `--help` de l'application

Une fois les parties jouées, un récapitulatif des parties et des temps de jeu est affiché afin de pouvoir facilement evaluer les performances des joueurs.

```
========================= RÉCAPITULATIFS DES SCORES ==========================
10 match(s) gagné par les Noirs.
0 match(s) gagné par les Blanc.
0 match(s) nul(s).
===================== RÉCAPITULATIFS DES AIRES PAR MATCH =====================
80.7812 % du terrain occupé par les Noirs en moyenne
19.0625 % du terrain occupé par les Blanc en moyenne
0.15625 % du terrain non-occupé en moyenne
==================== RÉCAPITULATIFS DES TEMPS D'EXECUTION ====================
IA1 (alphabeta_mixte=6) mean calculation time per move : 66.6095 ms (316 moves
played)
IA2 (random) mean calculation time per move : 0.00325442 ms (283 moves played)
==============================================================================
```

Figure 9: Récapitulatif de fin de session

II. Développement des IA

II.A. Intégration des IA dans le cadre de travail

II.A.1. Classe abstraite `IA::Interface` et son intégration

Pour permettre aux IA de jouer dans l'application, une classe abstraite nommée `IA::Interface` a été créée. Elle intègre toute les fonctions et attribut necessaire pour la communication avec l'interface (CLI ou Graphique) ainsi que des outils pour les IA (tel que la matrice de coup des cases et la fonction `switchTeam`). Il suffit alors de l'heriter pour créer son IA.

```
class IAInterface {
    protected :

        virtual int heuristics(const othello::Board& current_board, const
othello::pawn team ) = 0;

    public :

        static const int payoff_matrix[64];

        virtual int makeAMove(const othello::Board& current_board,othello::pawn
team) = 0;

        virtual void resetAI() {};

        static IAInterface* selectByName(std::string name);

        static othello::pawn switchTeam(othello::pawn p);
};
```

Figure 10: Récapitulatif de fin de session

La methode statique `selectByName` permet de generer une IA (classe fille de Interface donc) à partir de texte afin de pouvoir selectionner une IA en particulier facilement.

```
IAInterface* IAInterface::selectByName(std::string name) {
  std::vector <std::string> tokens;
  std::string intermediate; std::stringstream stream(name);
  while(getline(stream, intermediate, '=')) tokens.push_back(intermediate);
  if(tokens[0] == "random") return (new Random());
  else if(tokens[0] == "minmax"){
      return (new MinMax(std::stoi(tokens[1])));
  } else if(tokens[0] == "alphabeta"){
      return (new AlphaBeta(std::stoi(tokens[1])));
  } else if(tokens[0] == "alphabeta_absolute"){
      return (new AlphaBeta_Absolute(std::stoi(tokens[1])));
  } else if(tokens[0] == "alphabeta_mobility"){
      return (new AlphaBeta_Mobility(std::stoi(tokens[1])));
  } else if(tokens[0] == "alphabeta_mixte"){
      return (new AlphaBeta_Mixte(std::stoi(tokens[1])));
  } else {
      std::cout << "No IA named " << name << "." << std::endl;
      throw errors::OutOfBoundError();
  }
}
```

Figure 11: Selection de l'IA via `IAInterface::selectByName`

Cette classe est alors utilisé pour stocker et appeler les IA lors du déroulement de la partie, qu'importe la classe fille d'IA utilisée.

Les IA sont alors appelée par la methode `makeAMove` par l'application pour recuperer leur coup.

```
  // La fonction IA play est une fonction statique qui appelle makeAMove
  // Les methodes de classe ne peuvent pas être utilisé pour un thread tel quel
  if(!gameEnded && ((current_player == othello::pawn::black && isPlayer1AI) ||
(current_player == othello::pawn::white && isPlayer2AI))) {
      if(!IA_launched) {
          IA_thinking = true;
          IA_launched = true;
          if(this->current_player == othello::pawn::black){
              this->IAthread = new std::thread(Window::IAPlay,this->IA1,*(this-
>board),this->current_player,&IA_thinking,&IA_result);
          } else {
              this->IAthread = new std::thread(Window::IAPlay,this->IA2,*(this-
>board),this->current_player,&IA_thinking,&IA_result);
          }
      } else {
          if(!IA_thinking) {
              if(this->IAthread->joinable()) {
                  this->IAthread->join();
                  delete this->IAthread;
              }
              IA_launched = false;
              this->placePawn(IA_result);
          }
      }
  }
```

Figure 12: Execution des coups d'une IA en mode GUI

II.B. Minmax

Avant de procéder à l'implémentation de l'algorithme de Minmax il était judicieux de bien comprendre ce dernier. Nous avons donc commencé par mettre en place le pseudo-code[1] de l'algorithme .

Algorithm 1: Minmax

entrées: *nœud*, *profondeur*, *joueurMax*
sortie: *entier*
debut
  si *profondeur* = 0 **ou** estTerminal(*nœud*) **alors**
    retourner heuristique(*nœud*)
  si *joueurMax* **alors**
    valeur ← −∞
    pour chaque *enfant* de *nœud* **faire**
      valeur ← max(v, minmax(*enfant*, *profondeur* − 1, Faux))
  sinon
    valeur ← +∞
    pour chaque *enfant* de *nœud* **faire**
      valeur ← min(v, minmax(*enfant*, *profondeur* − 1, Vrai))
  retourner *valeur*

Figure 13: Pseudocode Minmax

En suite, nous avons implémenté l'algorithme en C++ en utilisant la classe `othello::Board` pour représenter le plateau de jeu et `othello:pawn` pour représenter le joueur qu'on veut maximiser.

De plus, le code differe légerement du pseudo-code dans l'implémentation du joueur a maximiser. En effet, dans le pseudo-code, le joueur a maximiser est le joueur qui maximise la valeur de la fonction heuristique, alors que dans notre implémentation, le joueur a maximiser est le joueur qui joue le coup. Cela permet de partager la fonction avec les deux joueur via la fonction `MinMax::switchTeam(current_player)`.

```
if (player == team){
        value = INT32_MIN;
        for (int i=0 ; i < possibility ; i++) {
            othello::Board* next_move = new othello::Board(current_board);
            if(next_move->placePawn(choices.at(i),player) == 0) {
                throw -1;
            }
            value = std::max(value,this-
>minmax(*next_move,MinMax::switchTeam(player),depth-1,team));
            delete next_move;
        }
    } else {
        value = INT32_MAX;
        //...
    }
    return value;
```

Figure 14: `MinMax::minmax()` : Implémentation min/max du joueur

[1] https://en.wikipedia.org/wiki/Minmax

II.C. Negamax

L'algorithme de Negamax est une simplification de l'algorithme Minmax. En effet, Negamax est une version simplifiée de Minmax où les valeurs des nœuds sont toujours positives. Cela permet de simplifier l'implémentation de l'algorithme, montré par le pseudocode[2].

Algorithm 2: Negamax

Entrées: *nœud*, *profondeur*, *couleur*
Sortie: *entier*
1 **debut**
2 **si** *profondeur* $= 0$ **ou** estTerminal(*nœud*) **alors**
3 **retourner** *couleur* $\times$ heuristique(*nœud*)
4 *valeur* $\leftarrow -\infty$
5 **pour chaque** *enfant* de *nœud* **faire**
6 *valeur* $\leftarrow$ max(v, $-$negamax(*enfant*, *profondeur* $- 1$, $-$*couleur*))
7 **retourner** *valeur*

Figure 15: Pseudocode Negamax

Afin de simplifier l'algorithme la notion de couleur est introduite. La couleur est un entier qui vaut 1 si le joueur est le joueur maximisant et -1 si c'est le joueur minimisant.

La couleur est utilisé pour inverser la valeur de la fonction heuristique si le joueur est le joueur minimisant.

```
int color = (player == team) ? 1 : -1;

  if (depth == 0) {
      return color*this->heuristics(current_board,team);
  }
  //...
```

Figure 16: `NegaMax::negamax()` : Implémentation de la couleur

[2] https://en.wikipedia.org/wiki/Negamax

II.D. Alpha-Beta

L'algorithme Alpha-Beta ($\alpha - \beta$) est une amélioration de l'algorithme Minmax qui permet de réduire le nombre de nœuds examinés.

Il évalue les positions potentielles en considérant uniquement les coups les plus prometteurs pour un joueur tout en éliminant les branches moins intéressantes. En comparant les valeurs alpha (la meilleure valeur trouvée pour le joueur maximisant) et bêta (la meilleure valeur trouvée pour le joueur minimisant), l'algorithme peut couper les branches inutiles, améliorant ainsi l'efficacité de la recherche.

Le pseudocode[3] de l'algorithme Alpha-Beta est le suivant :

Algorithm 3: Alpha-Beta

```
   Entrées: nœud, profondeur, α, β, joueurMax
   Sortie: entier
1  debut
2      si profondeur = 0 ou estTerminal(nœud) alors
3          retourner heuristique(nœud)
4      si joueurMax alors
5          valeur ← −∞
6          pour chaque enfant de nœud faire
7              valeur ← max(v, alphabeta(enfant, profondeur − 1, α, β, Faux))
8              si valeur > β alors
9                  retourner valeur
10             α ← max(α, valeur)
11         retourner valeur
12     sinon
13         valeur ← +∞
14         pour chaque enfant de nœud faire
15             valeur ← min(v, alphabeta(enfant, profondeur − 1, α, β, Vrai))
16             si valeur < α alors
17                 retourner valeur
18             β ← min(β, valeur)
19         retourner valeur
```

Figure 17: Pseudocode Alpha-Beta

L'algorithme Alpha-Beta propose plusieurs stratégies de recherche qui permettent d'améliorer les résultats de l'algorithme.

[3] https://en.wikipedia.org/wiki/Alpha-beta_pruning

II.E. Heritage et changement d'heuristiques

Les algorithmes restent les mêmes, peu importe les heuristiques utilisées. Nous avons donc créé des classes filles pour les algorithmes, où nous "overridons" la fonction d'heuristique afin de la modifier pour s'adapter à l'heuristique que nous voulons utiliser.

Nous avons aussi fait le choix de n'utiliser que l'algorithme Alpha-Beta puisque les trois algorithme ci dessus donnent les même résultats pour une heuristique donnée et que Alpha-Beta est l'algorithme qui sera le plus rapide a exectuer.

Voici les stratégies de Alpha-Beta que nous avons utilisées :

- **Alpha-Beta positionnel :** La stratégie positionnelle se concentre sur la position des pièces sur le plateau et évalue leur importance stratégique en fonction de leur position relative et de leur potentiel de contrôle.
- **Alpha-Beta absolue :** La stratégie absolue incorpore une évaluation absolue de la position en attribuant des valeurs numériques précises aux différentes configurations du jeu, sans tenir compte de la stratégie ou de la dynamique du jeu.
- **Alpha-Beta de mobilité :** L'heuristique de la mobilité se concentre sur la capacité des joueurs à effectuer des mouvements dans le jeu. Elle favorise les positions qui offrent plus d'options de mouvement aux joueurs.
- **Alpha-Beta mixte :** Cette approche combine plusieurs heuristiques pour évaluer la position du jeu, telles que la mobilité, la stabilité des pièces et la position sur le plateau. Elle vise à fournir une évaluation plus globale et précise.

III. Analyse des Résultats

Nous avons effectué des tests de benchmarking pour comparer les performances des différentes heuristiques. Les tests ont été effectués sur un ensemble de parties jouées entre l'IA qui choisit ses coups de manière aléatoire, et les différentes IA qui utilisent des heuristiques, en utilisant des paramètres de profondeur de recherche variés.

Les résultats des tests ont montré que les différentes heuristiques ont des performances variables en termes de temps de calcul par coup, de taux de victoire et de qualité des coups joués. Les résultats détaillés des tests sont présentés dans la Partie V.B.

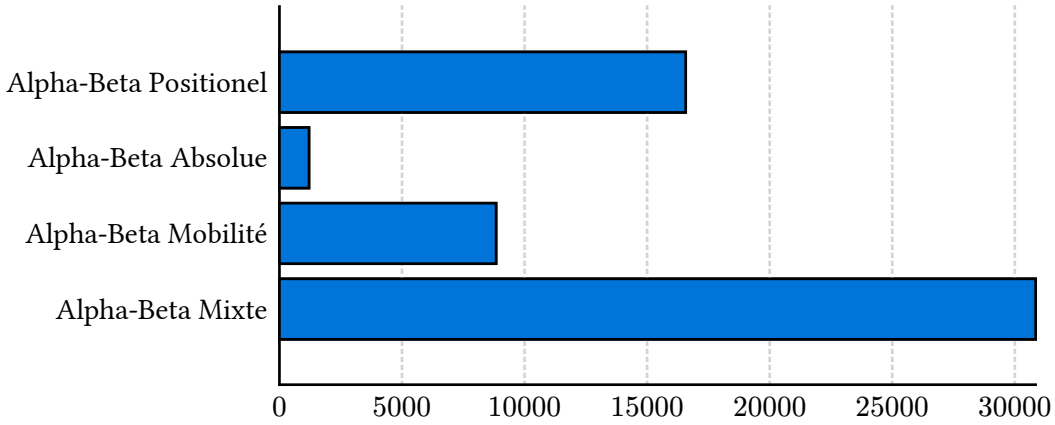


Figure 18: Temps d'exécution moyen par coup (ms) `depth=10`

Dans le premier graphique nous pouvions voir les temps d'execution moyen par coup pour les différentes IA à une profondeur de 10. On peut voir que l'heuristique Mixte est la plus lente ce qui est logique car la verification du nombre de coup joué est couteuse, et que les IA Alpha-Beta Absolue et Alpha-Beta Positionelle sont plus rapides. L'heuristique de mobilité etant la plus simple est aussi la plus rapide.

En termes de performances, les IA ont remporté la totalité de leurs matchs ou n'en ont perdu qu'un seul. Utiliser cette métrique pour évaluer leur performance n'était donc pas le plus optimal pour les départager. Nous avons alors utilisé le pourcentage de cases occupées par les deux équipes à la fin des parties pour essayer de comprendre quelles heuristiques fonctionnaient le mieux.

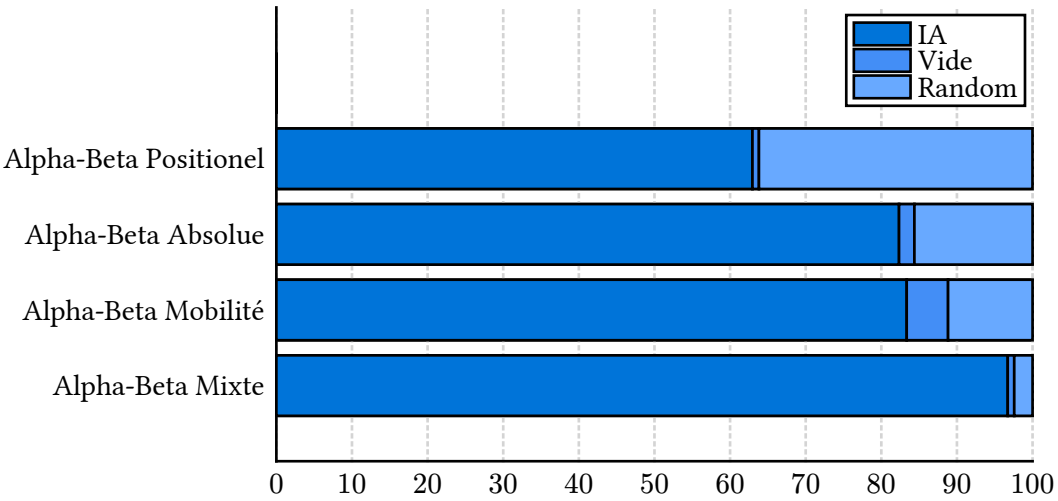


Figure 19: Occupation moyenne du terrain par IA (%)

On remarque plusieurs tendances grâce à ce graphe. Premièrement, l'heuristique mixte, qui combine astucieusement les 3 autres heuristiques, finit en général avec la quasi totalité du plateau couverte de ses pions. La deuxième est que l'heuristique positionnelle gagne ses parties avec un nombre de pions

bien plus faible que les autres heuristiques, et que l'heuristique de mobilité finit ses parties avec seulement 5% des cases vides, ce qui signifie qu'elle parvient à bloquer l'adversaire avant que le plateau soit rempli.

Ces quatre heuristiques ont des comportements différents que l'on remarque grâce à cette métrique. Cependant, la quantité de pions à la fin de la partie n'est pas le meilleur indicateur de performance pour ce jeu, car il suffit d'avoir plus de pions que son adversaire pour gagner et qu'une partie peut rapidement s'inverser en raison des règles du jeu. Pour une analyse de performance plus approfondie, il aurait fallu utiliser d'autres métriques.

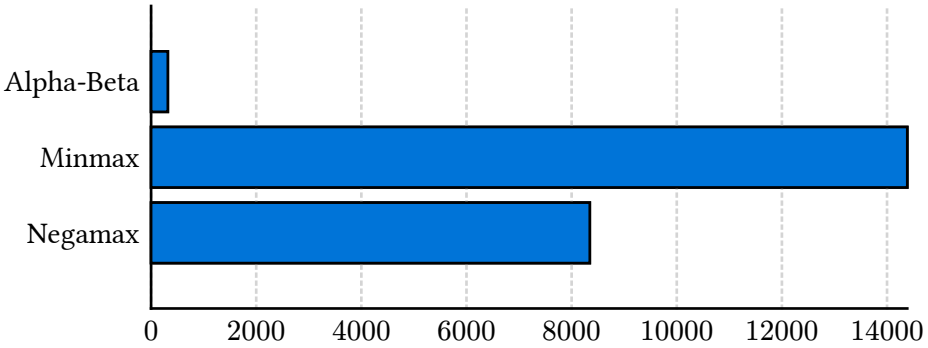


Figure 20: Temps d'exécution moyen par coup (ms) `depth=8`

Dans le deuxième graphique, nous pouvons voir les temps d'execution moyen par coup pour les différentes IA à une profondeur de 8. On peut voir que l'IA Alpha-Beta est la plus lente, mais que les IA Minmax et Negamax sont plus rapides.

IV. Conclusion et Perspectives

Dans le domaine de l'IA, il est largement admis que le meilleur algorithme d'Othello peut surpasser les joueurs humains les plus talentueux. Ce succès majeur dans la recherche en intelligence artificielle est illustré par l'utilisation de divers algorithmes dans cette étude sur Othello.

Dans ce projet, nous avons mis en œuvre plusieurs stratégies d'IA pour le jeu d'Othello en utilisant des algorithmes de recherche classiques tels que Minmax, Negamax et Alpha-Beta. Nous avons également exploré diverses heuristiques pour évaluer la position du jeu et orienter les décisions de l'IA.

Les résultats des tests de benchmarking ont révélé des performances variables entre les différentes stratégies d'IA, en termes de temps de calcul par coup, de taux de victoire et de qualité des coups joués. En général, les stratégies basées sur l'algorithme Alpha-Beta ont produit de meilleurs résultats en matière de temps de calcul et de qualité des coups joués.

En ce qui concerne les perspectives, il serait intéressant d'explorer davantage d'heuristiques et d'autres algorithmes de recherche, tels que la recherche arborescente Monte-Carlo, afin d'améliorer les performances des IA. L'exploration d'approches d'apprentissage automatique pour entraîner les IA à jouer à Othello de manière plus efficace et stratégique serait également pertinente.

De plus, l'amélioration de la gestion des threads pourrait permettre à l'IA de jouer de manière plus fluide et efficace. En effet, l'utilisation de threads peut entraîner des problèmes de synchronisation et de performance susceptibles d'impacter les performances de l'IA.

Enfin, l'exploration de nouvelles métriques pour évaluer les performances des IA, telles que la moyenne des cases occupées tout au long de la partie pourrait aider a mesurer la maîtrise de l'IA et donc pourrait fournir une mesure plus fine de la qualité des coups joués par l'IA et de son efficacité stratégique.

V. Annexes

V.A. Code source

Le code source complet du projet est disponible sur notre dépôt Github[4]

V.B. Résultats des tests

```
========================== RÉCAPITULATIFS DES SCORES ==========================
20 match(s) gagné par les Noirs.
0 match(s) gagné par les Blanc.
0 match(s) nul(s).
====================== RÉCAPITULATIFS DES AIRES PAR MATCH ======================
62.9588 % du terrain occupé par les Noirs en moyenne
36.1719 % du terrain occupé par les Blanc en moyenne
0.859375 % du terrain non-occupé en moyenne
==================== RÉCAPITULATIFS DES TEMPS D'EXECUTION ======================
IA1 (alphabeta=10) mean calculation time per move : 16573.5 ms (606 moves played)
IA2 (random) mean calculation time per move : 0.00394168 ms (583 moves played)
================================================================================
```

Figure 21: Résultats de Alpha-Beta Positionel

```
============================ RÉCAPITULATIFS DES SCORES ============================
19 match(s) gagné par les Noirs.
1 match(s) gagné par les Blanc.
0 match(s) nul(s).
====================== RÉCAPITULATIFS DES AIRES PAR MATCH ======================
82.3438 % du terrain occupé par les Noirs en moyenne
15.625 % du terrain occupé par les Blanc en moyenne
2.03125 % du terrain non-occupé en moyenne
==================== RÉCAPITULATIFS DES TEMPS D'EXECUTION ======================
IA1 (alphabeta_absolute=10) mean calculation time per move : 1205.58 ms (629 moves
played)
IA2 (random) mean calculation time per move : 0.00413578 ms (545 moves played)
================================================================================
```

Figure 22: Résultats de Alpha-Beta Absolue

```
============================ RÉCAPITULATIFS DES SCORES ============================
20 match(s) gagné par les Noirs.
0 match(s) gagné par les Blanc.
0 match(s) nul(s).
====================== RÉCAPITULATIFS DES AIRES PAR MATCH ======================
83.3594 % du terrain occupé par les Noirs en moyenne
11.1719 % du terrain occupé par les Blanc en moyenne
5.46875 % du terrain non-occupé en moyenne
==================== RÉCAPITULATIFS DES TEMPS D'EXECUTION ======================
IA1 (alphabeta_mobility=10) mean calculation time per move : 8838.38 ms (638 moves
played)
IA2 (random) mean calculation time per move : 0.0035 ms (492 moves played)
================================================================================
```

Figure 23: Résultats de Alpha-Beta Mobilité

[4] https://github.com/f7ed0/othello-cpp

```
========================= RÉCAPITULATIFS DES SCORES =========================
20 match(s) gagné par les Noirs.
0 match(s) gagné par les Blanc.
0 match(s) nul(s).
====================== RÉCAPITULATIFS DES AIRES PAR MATCH ======================
96.7188 % du terrain occupé par les Noirs en moyenne
2.42188 % du terrain occupé par les Blanc en moyenne
0.859375 % du terrain non-occupé en moyenne
==================== RÉCAPITULATIFS DES TEMPS D'EXECUTION ====================
IA1 (alphabeta_mixte=10) mean calculation time per move : 30848.2 ms (680 moves
played)
IA2 (random) mean calculation time per move : 0.00397053 ms (509 moves played)
================================================================================
```

Figure 24: Résultats de Alpha-Beta Mixte

```
========================= RÉCAPITULATIFS DES SCORES =========================
20 match(s) gagné par les Noirs.
0 match(s) gagné par les Blanc.
0 match(s) nul(s).
====================== RÉCAPITULATIFS DES AIRES PAR MATCH ======================
62.0312 % du terrain occupé par les Noirs en moyenne
37.9688 % du terrain occupé par les Blanc en moyenne
0 % du terrain non-occupé en moyenne
==================== RÉCAPITULATIFS DES TEMPS D'EXECUTION ====================
IA1 (minmax=8) mean calculation time per move : 14385 ms (606 moves played)
IA2 (random) mean calculation time per move : 0.00394781 ms (594 moves played)
================================================================================
```

Figure 25: Résultats de Minmax

```
========================= RÉCAPITULATIFS DES SCORES =========================
15 match(s) gagné par les Noirs.
4 match(s) gagné par les Blanc.
1 match(s) nul(s).
====================== RÉCAPITULATIFS DES AIRES PAR MATCH ======================
64.7656 % du terrain occupé par les Noirs en moyenne
35.2344 % du terrain occupé par les Blanc en moyenne
0 % du terrain non-occupé en moyenne
==================== RÉCAPITULATIFS DES TEMPS D'EXECUTION ====================
IA1 (negamax=8) mean calculation time per move : 18345.5 ms (612 moves played)
IA2 (random) mean calculation time per move : 0.00398299 ms (588 moves played)
================================================================================
```

Figure 26: Résultats de Negamax

```
========================= RÉCAPITULATIFS DES SCORES ==========================
19 match(s) gagné par les Noirs.
1 match(s) gagné par les Blanc.
0 match(s) nul(s).
====================== RÉCAPITULATIFS DES AIRES PAR MATCH ======================
62.1094 % du terrain occupé par les Noirs en moyenne
37.7344 % du terrain occupé par les Blanc en moyenne
0.15625 % du terrain non-occupé en moyenne
==================== RÉCAPITULATIFS DES TEMPS D'EXECUTION ======================
IA1 (alphabeta=8) mean calculation time per move : 318.496 ms (606 moves played)
IA2 (random) mean calculation time per move : 0.0038902 ms (592 moves played)
================================================================================
```

Figure 27: Résultats de Alpha-Beta