THÉORIE DES JEUX : OTHELLO

Modélisation Numérique POO – Rapport

Filière de Physique fondamentale d'Orsay
M1/Mag2 – Avril 2022
Élianor Hoffmann et Pierre Boistier

Résumé

Dans le cadre du projet d'informatique du cours de Modélisation Numérique et Programmation Orientée Objet, nous nous intéressons à l'implémentation d'un jeu d'*Othello*, jeu de combinatoire abstrait à deux joueurs, et celle de quelques algorithmes pouvant jouer une partie. Nous comparons ensuite leurs performances en termes de nombre de victoires et de score (nombre de pions en fin de partie). Le choix d'un sujet de théorie des jeux dans une filière de physique fondamentale peut sembler étonnant, mais au-delà du manque évident d'application directe de ce projet à des problèmes physiques, l'apprentissage et l'utilisation de la Programmation Orientée Objet y est à son paroxysme.

Abstract

Within the context of the Numerical Modelisation and Object Oriented Programmation IT project, we focus on the implementation of an *Othello* game, a two players abstract combinatory game, and the implementation of some algorithms able to play. Then, we compare their performance in terms of number of victories and score (which is the number of pieces on the board at the end of the game). Choosing a game theory subject while we are Fundamental Physics students might question, but beyond the evident lack of practical applications of this subject in physical fields, learning and using Object Oriented Programmation reaches there its climax.

Introduction

Reversi (1880, Angleterre, Lewis Waterman et John W. Mollet), aussi connu sous le nom *Othello* (1971, Japon, Goro Hasegawa) est un jeu de société combinatoire abstrait pouvant rappeler les jeux de dames, échecs ou go par son plateau, ses pions et ses mécanismes. Il se joue sur un plateau unicolore de 8x8 cases, appelé othellier (*cf.* Figure 1). Les joueurs disposent de 64 pions bicolores, noirs d'un côté et blancs de l'autre. Les joueurs jouent à tour de rôle, chacun étant tenu de capturer des pions adverses lors de son mouvement. La capture de pions survient lorsqu'un joueur place un de ses pions à l'extrémité d'un alignement de pions adverses contigus et dont l'autre extrémité est déjà occupée par un de ses propres pions (*cf.* Figure 1). Le jeu s'arrête quand les deux joueurs ne peuvent plus poser de pion. Le gagnant en fin de partie est celui qui possède le plus de pions de sa couleur.

Coder un jeu de société est un projet qui se prête particulièrement à une programmation orientée objet. En effet, l'utilisation de classes est naturelle, que ce soit pour coder le plateau, les joueurs humains et les joueurs "ordinateurs". L'hérédité sera notamment pertinente lorsqu'il s'agira de complexifier progressivement les algorithmes de jeu dits "ordinateurs", un joueur (sauf pour le premier) étant une classe fille d'un joueur avec une approche plus simpliste.

L'objectif premier du projet a été de coder, en `C++` et en `Python3` pour le traitement de données, un othellier et des classes de joueurs humains pour jouer en 1v1. Dans un second temps, il a s'agit d'implémenter des classes de joueurs dits "ordinateurs" plus ou moins complexes (joueur aléatoire, joueur qui retourne un maximum de pions chaque tour, et un joueur qui détermine la meilleure stratégie sur plusieurs tours grâce à un algorithme dit MinMax). Enfin, on a comparé l'efficacité de nos algorithmes en les faisant jouer contre un joueur aléatoire, en nombre de victoires et en score final, en ayant bien pris en compte l'influence de l'ordre de jeu.

Objets et Méthodes

Le jeu est intégralement codé en langage `C++`. Nous utilisons le langage `Python3` via un *Notebook* `Jupyter` afin de traiter les données générées par le code en `C++`.

Pour répondre aux objectifs de notre projet, il a d'abord fallu commencer par créer un objet plateau. Pour ce faire, nous avons donc créé une classe `grille`, initialement constitué d'un attribut `g` de type tableau 8x8 (la grille de jeu à un instant donné) et d'un attribut `numero_tour` comptant le tour. Le tableau `g` est alors rempli d'entiers associés à une valeur de pion (0 pour vide, 1 pour blanc et 2 pour noir). La méthode `retournerPlacer( · )` prend en argument le coup d'un joueur (les coordonnées (i, j) sur `g`), place le pion et change la couleur des pions capturés. Il est à noter que nous avons un temps pensé à créer une classe `case` équivalente à un type *tuple* en `Python3`, et faire de `grille` un ensemble de `case`s. En effet, le type couple n'existant pas en `C++`, mais les différents joueurs devant jouer sur une case, il a fallu ruser et utiliser des pointeurs tout au long du programme pour permettre aux joueurs de communiquer le coup choisi à la `grille`.

Une fois la classe `grille` créée, nous nous sommes attelés à la création de la classe `humain`, la classe mère de tous les joueurs. Cette classe dispose d'un attribut, sa `couleur` (noir ou blanc), ainsi que d'une méthode `choixCoups( · )` qui prend en argument une `grille` (et deux pointeurs), demande à l'utilisateur sur quelle case jouer, et modifie les pointeurs en conséquence. Pour davantage de lisibilité et d'efficacité, nous avons pensé qu'il serait intéressant de communiquer aux différents joueurs les cases sur lesquelles leur couleur permet de jouer (on dira qu'une case es licite). Pour cela, nous avons adapter l'attribut `g` de `grille` avec un nouveau code équivalent entier (-1 pour vide interdit, 11 pour blanc, 22 pour noir, 1 pour vide licite pour blanc, 2 pour vide licite pour noir, 0 pour vide licite pour blanc et noir). Se faisant, il a également fallu modifier la méthode `retournerPlacer( · )` de `grille` pour modifier la valeur présentes dans les cases vides après un coup.

FIGURE 1 – A droite, othellier en début de partie. A gauche, de gauche à droite et de haut en bas, trois exemples de coups à l'othello. Placer un pion sur la même ligne, colonne ou diagonale qu'un autre des pions de la même couleur capture les pions de la ligne, diagonale ou colonne.

Nous avons alors créé une classe de joueur `ordiAleatoire`, fille de `humain`, dont la méthode `choixCoups( · )`, pour un tour donné et connaissant son attribut `couleur`, parcours l'ensemble des cases de `g` autorisées (de valeur 0, 1 ou 2 en fonction de `couleur`) et joue aléatoirement.

Puis il fallu coder un premier ordinateur un peu malin. Nous nous sommes, pour ce faire, renseigné sur la page de la Fédération Française d'Othello, celle-ci présentant de manière détaillées plusieurs algorithmes à coder en *Othello*. Le premier présenté – MinMax – nous semblant compliqué de prime abord, on décide d'inventer notre propre algorithme, dont le principe est plutôt simple : `retourneMax`. A chaque tour donné, `retourneMax` regarde, pour chaque case licite, le nombre de pions adverses qu'il pourrait retourner en jouant à cet endroit. Afin de tirer avantageusement partie de la classe `grille`, et sachant que pour chaque coup joué, `retournerPlacer( · )` de `grille` modifie déjà la valeur des cases vides de `g`, nous avons décidé de modifier légèrement la structure de `g` de `grille`. Désormais, `g` sera un tableau de type 8x8x3, tel que :

- `g[:][:][0]` représente la grille de jeu comme définie précédemment ;
- `g[i][j][1]` est égale au nombre de pions noirs retournables si un pion blanc est placé en (i, j) (si `g[i][j][0]` n'est pas licite pour blanc, `g[i][j][1]` contient la valeur -1) ;
- `g[i][j][2]` est égale au nombre de pions blancs retournables si un pion noir est placé en (i, j) (si `g[i][j][0]` n'est pas licite pour noir, `g[i][j][2]` contient la valeur -1).

Ceci étant fait, la méthode `choixCoups( · )` de `retourneMax` a simplement besoin de parcourir les cases de `g[:][:][1]` ou `g[:][:][2]` en fonction de sa `couleur` et de choisir la case possédant la plus grande valeur.

On peut remarquer ici que si l'on avait voulu comparer nos algorithmes en temps, cette méthode de calcul du nombre de cases retournables n'aurait pas été pertinente. En effet, à chaque tour, c'est `grille` qui effectue le calcul des meilleurs coups, et ceci, indépendamment de si le joueur les utilise ou non (comme c'est le cas par exemple pour `ordiAleatoire` et `humain`). La comparaison du temps d'exécution de `retourneMax` aux autres joueurs est donc complètement faussée par cette méthode.

Viens enfin l'implémentation de l'algorithme MinMax, dont le principe exposé ci-après est directement tiré du site internet de la Fédération Française d'Othello :

> *L'idée fondamentale,* [...] *est qu'il faut faire parcourir par le programme un espace de recherche constitué des positions futures potentiellement atteintes par les joueurs pour anticiper la meilleure séquence à jouer. On modélise cet espace de recherche avec des techniques issues de la théorie des graphes* [...].
>
> *La méthode générale est la suivante : à partir d'une position (racine de l'arbre) on génère tous les coups possibles pour le programme. Puis à partir de ces nouvelles positions (niveau 1) on génère toutes les réponses possibles pour l'adversaire (niveau 2). On peut alors recommencer l'opération aussi longtemps que le permet la puissance de calcul de l'ordinateur et générer les niveaux 3, 4, ..., n.*
>
> *Le facteur de branchement dans une partie d'*Othello *équilibrée étant d'environ 10 et le nombre de coups environ 60, on ne peut générer, dans un temps limité, la totalité de l'arbre. On doit donc limiter la profondeur de la procédure. Une fois atteinte cette profondeur, les feuilles (par opposition à la racine) de l'arbre ainsi construit sont évaluées par une fonction d'évaluation.*

On implémente ainsi la classe `ordiMinMax` en lui créant une nouvelle méthode `fonctionMinMax( · )`, réalisant l'algorithme présenté ci-dessus, ainsi qu'un nouvel attribut, `profondeur_max`. L'évaluation d'une des feuilles du graphe se fait simplement en comptant le nombre de pions de la couleur du joueur présents sur le plateau (ceci impliquant que la `profondeur_max` ne peut pas être nulle).

Pour tirer avantageusement partie de la précédente modification de l'attribut `g` de `grille`, on crée une classe `ordiMinMaxRapide` héritant directement de `ordiMinMax`. Le fonctionnement de la méthode `fonctionMinMax( · )` est le même que précédemment, sauf que l'évaluation ne se fait plus aux feuilles du graphe, mais à chacun des nœuds de l'arbre (plutôt que de compter le nombre totaux de pions noirs au niveau des feuilles, on compte le nombre total de pions retournés en descendant en profondeur). L'objectif premier de cette nouvelle classe était de permettre de collecter des statistiques plus rapidement (augmenter la profondeur de 1 correspondant à multiplier le temps de calcul par 7 ou 8), mais lorsque `ordiMinMax` de profondeur 2 réalise 100 000 parties en 837 secondes, `ordiMinMaxRapide` de profondeur 2 les réalise en 761 secondes, ce qui n'est pas ce qu'on peut appeler une amélioration significative. Lors de l'analyse des résultats, on ne se souciera pas de distinguer ces deux classes, et on parlera toujours de `ordiMinMax` pour présenter les résultats de l'un ou de l'autre.

Enfin, on notera que l'on a astucieusement utilisé le polymorphisme des classes joueurs (non sans peine) pour faire sélectionner à l'utilisateur le type des joueurs s'affrontant pendant une partie.

Résultats et Interprétation

Une fois tous les objets créées, nous avons voulu les comparer. Pour ce faire, nous enregistrons des milliers de parties dans des fichiers `.dat` directement depuis le code en `C++` avant de les analyser en `Python` grâce à un *Notebook* `Jupyter`. Pour pouvoir analyser nos résultats de manière statistique, on attelle systématiquement à enregistrer et analyser nos parties sous forme de paquets, que l'on nomme générations. Toutes les données que l'on présente ici sont ainsi le résultat du traitement systématique de 100 générations de 1000 parties.

La première idée qui nous est apparue est la suivante : est-il plus avantageux de jouer noir (donc de commencer la partie) ou de jouer blanc ? Y a-t-il, dès le début de la partie, un facteur chance lié à la couleur du joueur ? Pour répondre à cette question, on collecte donc des données sur des parties opposant deux joueurs de type `ordiAleatoire`. On observe les résultats présentés en Figure 3.

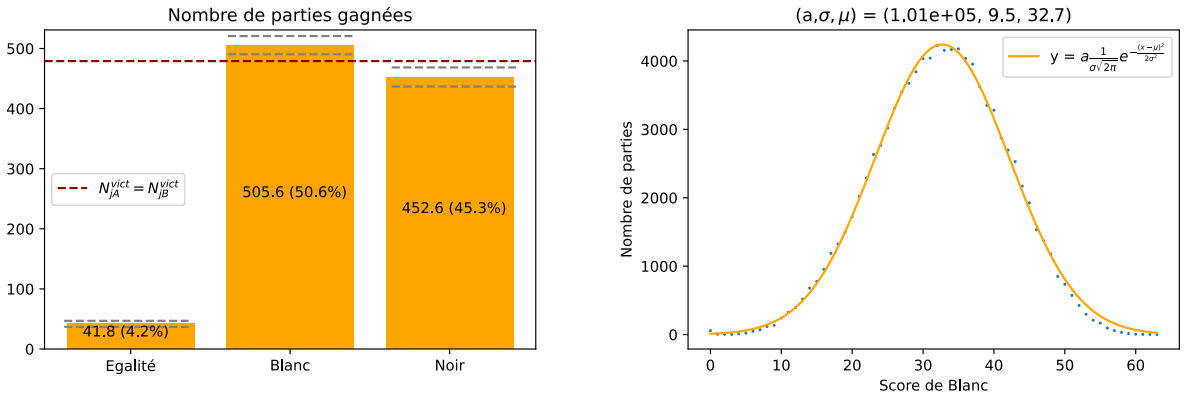

FIGURE 2 – Résultats de parties opposants deux `ordiAleatoire`. A gauche, histogrammes présentant, sur 1000 parties, les résultats moyens obtenus par le joueur blanc et le joueur noir, ainsi que l'incertitude sur ces moyennes (bars pointillées grises). La bar pointillée rouge représente la moitié du nombre de parties, une fois les égalités retirées. A droite, le nombre de parties gagnées en fonction du score du joueur blanc, le score étant le nombre de pions de sa couleur en fin de partie. On fit ces résultats par une gausssienne.

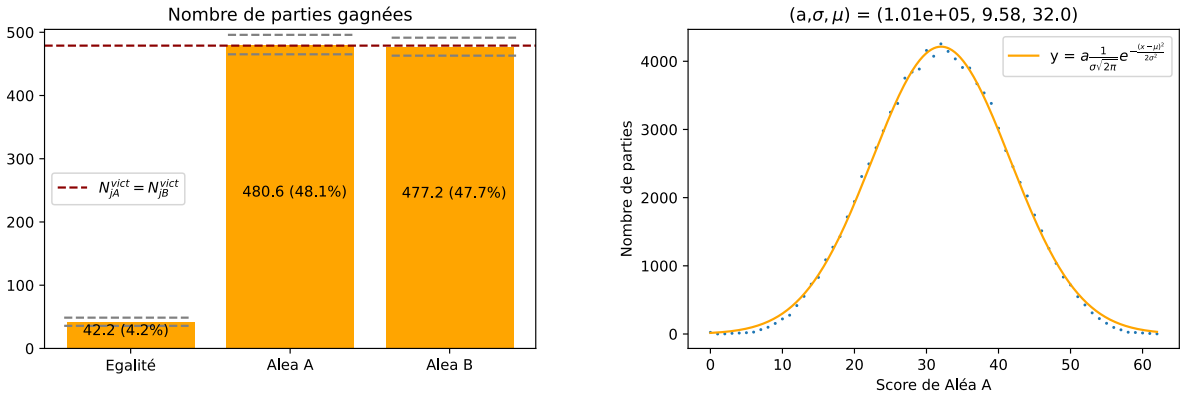

FIGURE 3 – Résultats de parties opposants deux `ordiAleatoire`, en prenant en compte le biais des couleurs. Même description qu'en FIGURE 3.

Comme on peut le voir, il est en effet plus avantageux de jouer blanc à l'*Othello*, et l'effet se ressent de manière significative lorsque l'on joue de manière aléatoire. On remarque en effet que les courbes d'incertitudes sont bien situées respectivement au-dessus pour blanc et en-dessous pour noir de la courbe du nombre moyen de victoires. Le résultat se confirme lorsque l'on regarde le nombre de parties en fonction du score du joueur blanc : moyenne de la gaussienne est située au-delà de 32 points (même si très léger), lorsqu'on s'attend à une moyenne de exactement 32 points pour un plateau de 64 cases. Cet effet n'est pas étonnant : tout le monde s'est déjà rendu compte en jouant au morpion qu'il est plus aisé de gagner en commençant. Des effets de ce type existent dans tous les jeux se faisant s'affronter deux adversaires, et dans certains comme les échecs, c'est toute la suite de la partie qui peut découler du premier coup du premier joueur (on parle d'ouverture). On prend en compte ce biais dans la suite de nos résultats, et on compare systématiquement les joueurs entre eux en les faisant jouer 50% de parties noir et 50% de parties blanc. On observe bien le résultat attendu, une fois ce biais pri en compte, en FIGURE 4.

On compare ensuite l'efficacité de nos algorithmes comparés à un joueur `ordiAleatoire`, et on présente nos résultats en FIGURE 4, selon la même méthodologie qu'en FIGURE 3. On note dans un premier temps que nous n'avons pu faire des statistiques de `ordiMinMax` que jusqu'en profondeur 3, la profondeur 4 impliquant 12h consécutives de mesures, la profondeur 5, 100 heures et ainsi de suite...

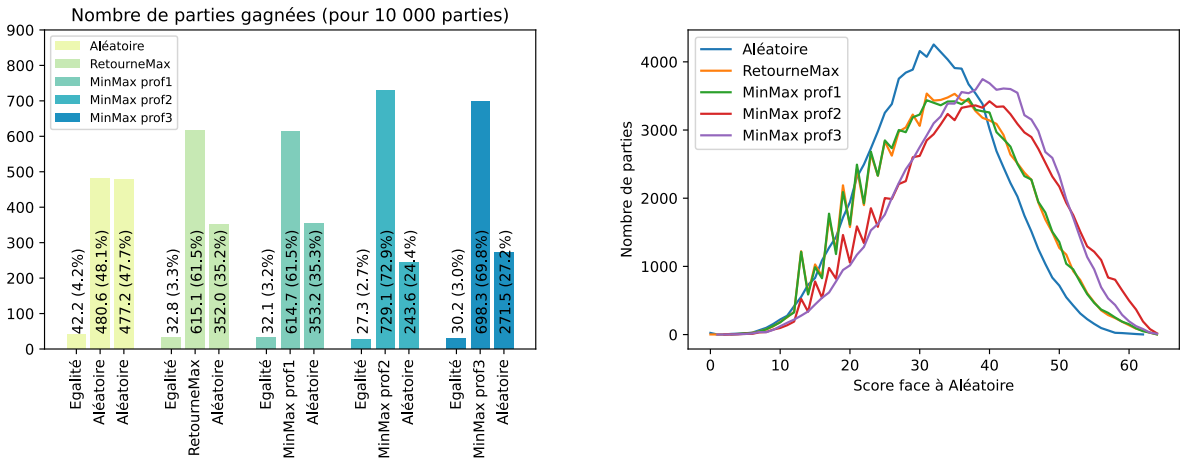

FIGURE 4 – Résultats superposés de parties opposants toutes les classes de joueur à un `ordiAleatoire`. On ne fait pas ici de fit gaussien ni d'incertitudes, les données sont présentées brutes et moyennées.

Le premier constat que l'on peut faire est que notre algorithme `retourneMax` est déjà nettement meilleur que `ordiAleatoire`. Comme prévu, `ordiMinMax` de profondeur 1 donne les mêmes résultats que `retourneMax`, les deux algorithmes effectuant la même chose à cette profondeur. En ce qui concerne `ordiMinMax` de profondeur 2, on observe une nette amélioration par rapport à la profondeur 1. Mais cette amélioration est en baisse en passant à la profondeur 3. Lorsqu'on l'on regarde les scores effectués par les différents algorithmes face à `ordiAleatoire`, on remarque que la différence entre `ordiMinMax` de profondeur 2 et 3 ne réside pas dans la moyenne du score mais dans l'élargissement de la courbe. Nous pensons que cet effet est la conséquence de deux phénomènes. Tout d'abord, nous supposons qu'il existe un effet lié à la parité de la profondeur. En effet, une profondeur paire ou impaire change légèrement la manière dont réfléchit le joueur : en parité paire, il calcule le meilleur score qu'il pourrait faire en s'arrêtant sur un coup de son adversaire, là où en parité impaire il s'arrête sur un de ses propres coups. Pour confirmer cette hypothèse, il aurait fallu faire des mesures à profondeurs supérieures, ce qui n'est pas possible en l'état actuel de nos algorithmes, qui sont beaucoup trop lents. Le deuxième effet pourrait venir du fait que l'algorithme MinMax essaye de maximiser son score en supposant que son adversaire essaye de le minimiser. Pour cette raison, faire jouer un `ordiMinMax` contre des joueurs humains auraient probablement des résultats plus probants que contre un `ordiAleatoire`. Les résultats observés sont les résultats minimaux que peuvent faire les algorithmes `ordiMinMax`.

Conclusion

Cette étude nous a permis de montrer quelques techniques de jeu à l'*Othello* permettant d'augmenter ses chances de gagner. Après avoir éliminer le biais de couleur, on se rend compte que prévoir ses coups et ceux de l'adversaire permet de considérablement augmenter ses chances de réussite. Une première manière de prolonger l'étude serait d'améliorer les temps de calcul des `ordiMinMax` via, par exemple, des coupures AlphaBeta (il n'est pas nécessaire de parcourir tout l'arbre pour déterminer le chemin optimal), qui est une technique purement informatique. D'autres méthodes consisteraient à s'intéresser à des subtitilités spécifiques au jeu *Othello*, comme décrit sur le site de la FFOthello. Enfin, nous pourrions redéfinir les classes `grille` et `retourneMax` afin de comparer en temps les différents algorithmes, et ce dans l'objectif de choisir celui qui allie à la fois performance et rapidité d'exécution.

Références

[1] *Page Wikipedia de Othello (jeu)*. adresse : `fr.wikipedia.org/wiki/Othello_(jeu)`.

[2] *Site internet de la Fédération Française d'Othello*. adresse : `www.ffothello.org`.