IA Solve - DA IODAA

Devoir maison n°2 - Programmation Othello

Décembre 2022

Étudiants:
Hélène PHILIPPE - Louis MANCERON - Luca GORTANA

Dépôt Git:
`https://github.com/helene-philippe/othello.git`

1 Introduction

1.1 Objectif

La première partie de ce devoir consiste en la création d'Othello (**Figure 1**), un jeu de plateau simple à somme nulle et à information complète. L'objectif est de le rendre jouable par des utilisateurs, puis de créer un robot capable d'anticiper les coups et d'être le plus compétitif possible. Pour ceci, nous avons créé plusieurs programmes reposant sur des principes différents.

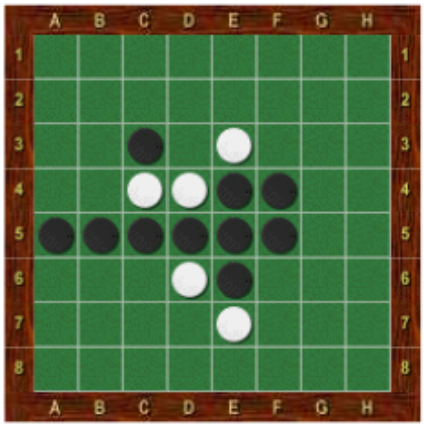

Figure 1: **Plateau du jeu Othello**

1.2 Méthodes

Afin de déterminer le coup optimal à un instant t, il faut créer une **fonction d'évaluation** décrivant les gains associés à une action potentielle. Cette fonction d'évaluation est ensuite utilisée par différents algorithmes afin de sélectionner le **coup optimal**. Les différents algorithmes que nous avons créés afin de déterminer les choix de l'ordinateur sont les suivants:

- **MinMax**, qui suppose que le meilleur choix est celui qui minimise les pertes d'un joueur tout en supposant que l'adversaire cherche au contraire à les maximiser.

- **Alpha-Beta**, une version optimisée de MinMax qui réduit le nombre de noeuds explorés.

- **MCTS (*Monte Carlo Tree Search*)**, basée sur la méthode de Monte Carlo qui utilise un échantillonnage aléatoire pour des problèmes déterministes.

1.3 La programmation du jeu Othello

Nous avons organisé le programme d'othello en 2 fichiers principaux :

- **othellier.py** qui contient la classe Othellier, et les méthodes associées.
- **partie.py** qui mobilise la classe Othellier et orchestre les différentes méthodes entre elles.

Pour chaque question (programmer MinMax, alphaBeta et MCTS), nous avons réalisé un fichier associé éponyme. Les fichiers simulation.py et grid_search_MCTS.py ont été créés pour réaliser les analyses des différents algorithmes. Si vous souhaitez jouer et/ou tester le programme : utilisez **"main.py"**.

2 Fonction d'évaluation

L'évaluation est une fonction qui consiste à **porter une appréciation aussi systématique et objective que possible sur un état de notre partie**. Il s'agit de déterminer la pertinence des différents coups à jouer pour un othellier donné, afin d'en déduire le coup le plus judicieux.

Nous avons choisi deux approches :

- Un coup est favorisé lorsqu'il retourne un maximum de pion. Chaque pion retourné rajoute 1 au poids du coup joué.
- Un coup permettant de positionner un pion sur un bord doit être favorisé. Les coins sont les meilleurs coups à jouer. Pour ceci, des poids bien supérieurs à 1 seront associés aux cases présentes sur les bords et les coins (**Figure 2**).

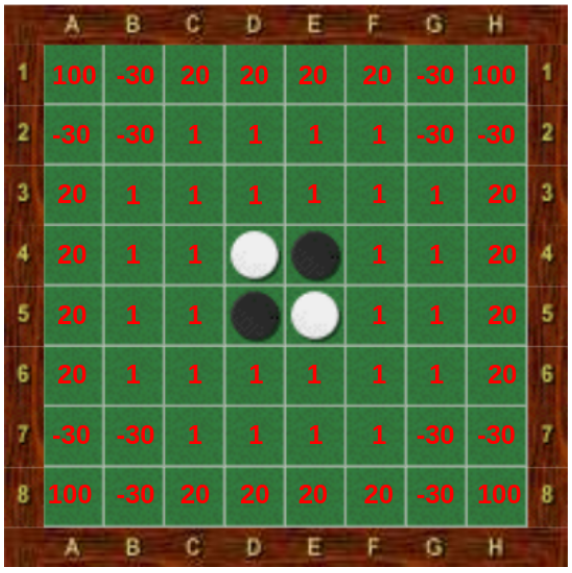

Figure 2: **Exemple de poids possibles pour chaque case de notre othellier**

Les cases juste à côté de ces cases avantageuses ne le sont pas, car elles donnent l'opportunité à l'autre joueur de placer leur pion dans les coins ou sur les bords. Ces cases sont donc défavorisées dans la fonction d'évaluation.

3 Algorithmes de décision

3.1 MinMax et Alpha-Beta

3.1.1 Conception des algorithmes

Afin d'évaluer la performance de l'algorithme pour différentes profondeurs, nous faisons s'affronter 2 ordinateurs (dont un avec la profondeur d'étude) sur 40 parties. On compare ensuite le nombre de parties gagnées pour évaluer la performance.

Les algorithmes minMax et alphaBeta présentent les mêmes outputs, seule la vitesse d'exécution des 2 algorithmes change. En effet, nous avons calculé le temps moyen d'une partie où deux joueurs s'affrontent en jouant tous les deux avec alphaBeta ou MinMax en profondeur 2. Une partie dure en moyenne 26 secondes avec alphaBeta et 111 secondes avec minMax.

Ainsi, par soucis de gain de temps de calcul, nous avons évalué les performances de alphaBeta et minMax sur **alphaBeta uniquement**.

Nous avons réalisé des simulations pour évaluer la performance selon différentes profondeurs. Les simulations font s'affronter 2 ordinateurs : l'un jouant avec l'algorithme de référence (en vert sur les graphiques), et l'algorithme alphaBeta (en rouge) de profondeur croissante. Dans une partie, le joueur qui commence est légèrement avantagé. Ainsi, afin de s'affranchir de ce biais dans l'évaluation des performances, le joueur qui commence change à chaque partie.

Pour évaluer les performances, nous avons choisi comme **algorithme de référence MinMax de profondeur 1**. Nous avons obtenus les premiers résultats en **Figure 3**.

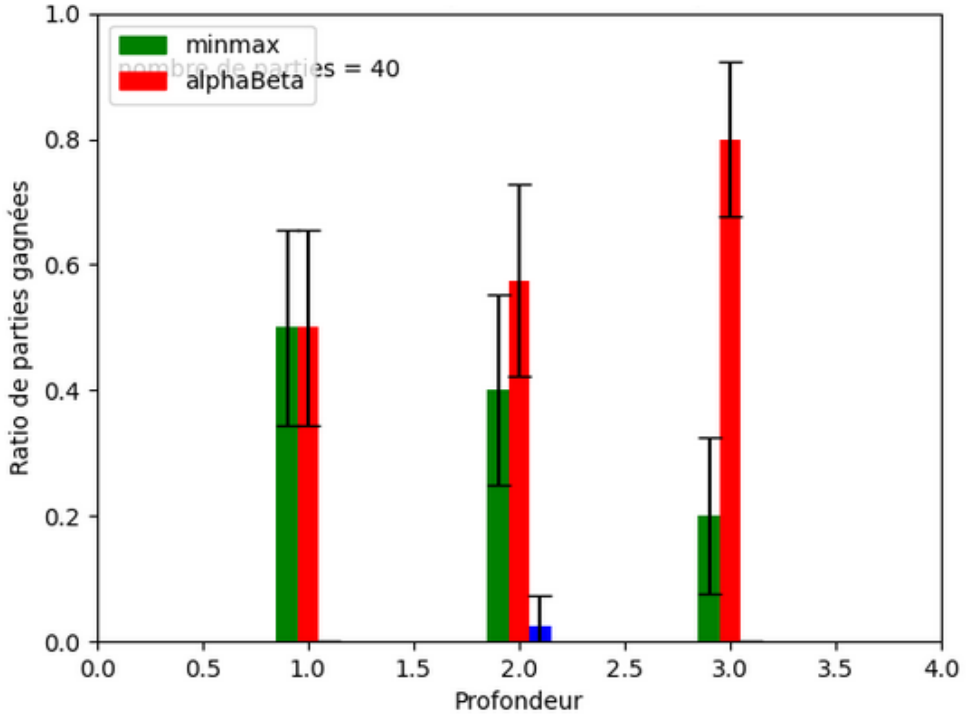

Figure 3: **Performances de alphaBeta face à l'algorithme de référence.** *Résultats de simulations sur des profondeurs allant de 1 à 3 sur 40 parties. En vert: l'algorithme de référence MinMax profondeur constante de 1, en rouge: l'algorithme alphaBeta avec un profondeur variant de 1 à 3. Les barres d'erreur correspondent à un intervalle de confiance à 95% d'une proportion.*

3.1.2 Analyse

Premièrement, on observe que pour une profondeur de 1 pour alphaBeta, les résultats sont statistiquement équivalents à celui de référence (minMax profondeur 1). C'est ce que l'on souhaite puisque minMax et alphaBeta doivent retourner les mêmes résultats.

Ensuite, on observe que plus la profondeur augmente, plus la performance de alphaBeta est meilleure. C'est aussi ce que nous attendions, puisque plus la profondeur est importante, plus l'algorithme peut anticiper le gain que lui rapportent ses coups. Nous pourrions nous attendre, à partir d'une certaine profondeur, à voir la performance d'alphaBeta diminuer car les incertitudes dues à la fonction d'évaluation s'accumulent (sur-apprentissage). Cependant, nous ne disposons pas d'une puissance de calcul suffisamment importante pour atteindre cette profondeur.

Nous pouvons maintenant nous interroger sur l'importance que la fonction d'évaluation a sur ces résultats. Si, par exemple, nous attribuions +0 pour chaque case (au lieu de +100, +20 et -30) et que la seule indication était le nombre de pions de la couleur du joueur sur le plateau, qu'obtiendrions-nous? Les résultats de la simulation sont en **Figure 4**.

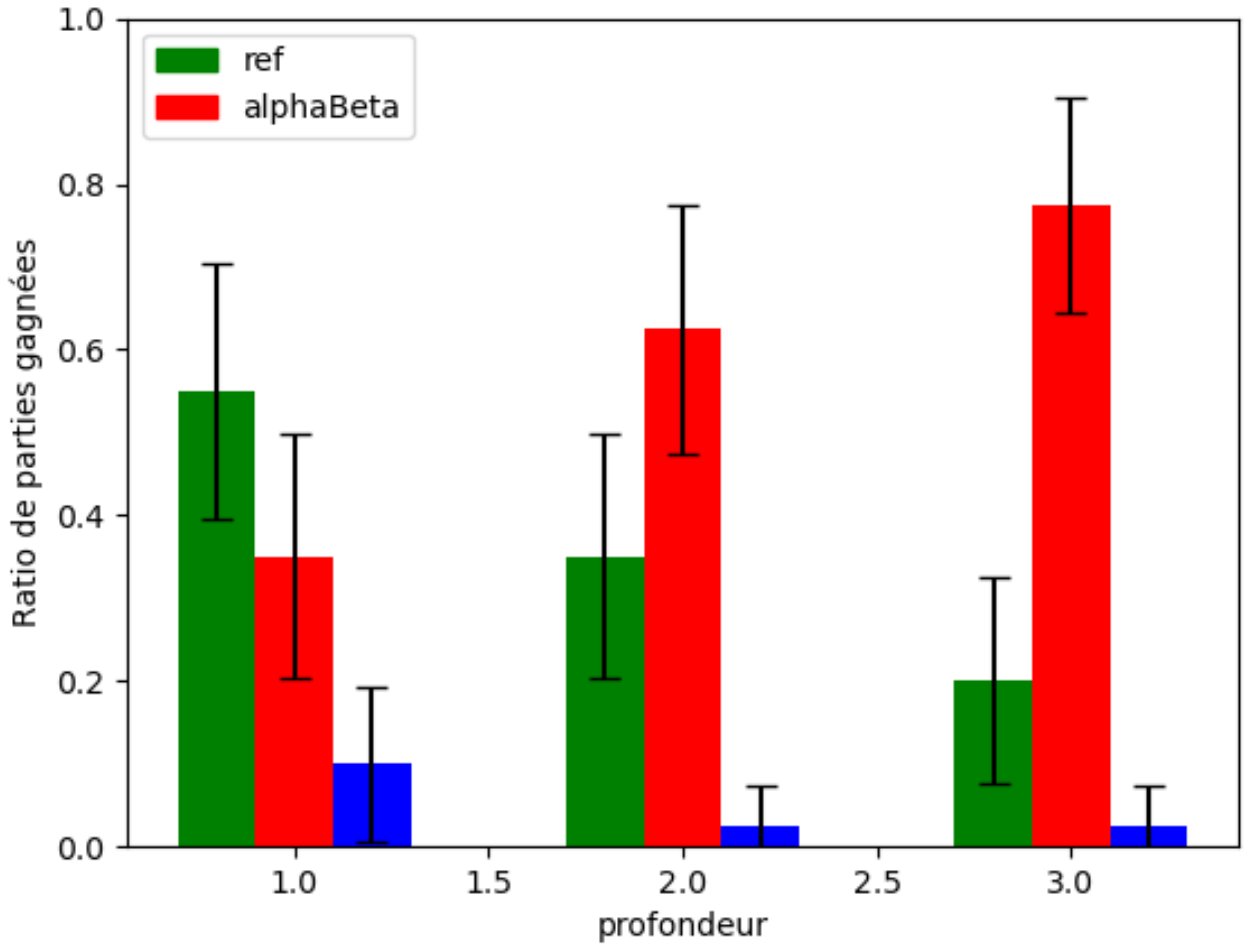

Figure 4: **Performances de alphaBeta avec la deuxième fonction d'évaluation**. *En vert: l'algorithme de référence MinMax profondeur constante de 1, en rouge: l'algorithme alphaBeta avec un profondeur variant de 1 à 3. Les barres d'erreur correspondent à un intervalle de confiance à 95% d'une proportion.*

Nous nous attendions à ce que la performance chute, étant donné que la fonction d'évaluation est simplifiée. Au final, on observe des résultats relativement similaires à ceux obtenus avec la fonction d'évaluation plus élaborée. Ainsi, complexifier un modèle ne permet pas forcément d'obtenir de meilleures performances.

Nous avons aussi testé la performance de alphaBeta/minMax face au hasard (**Figure 5**).

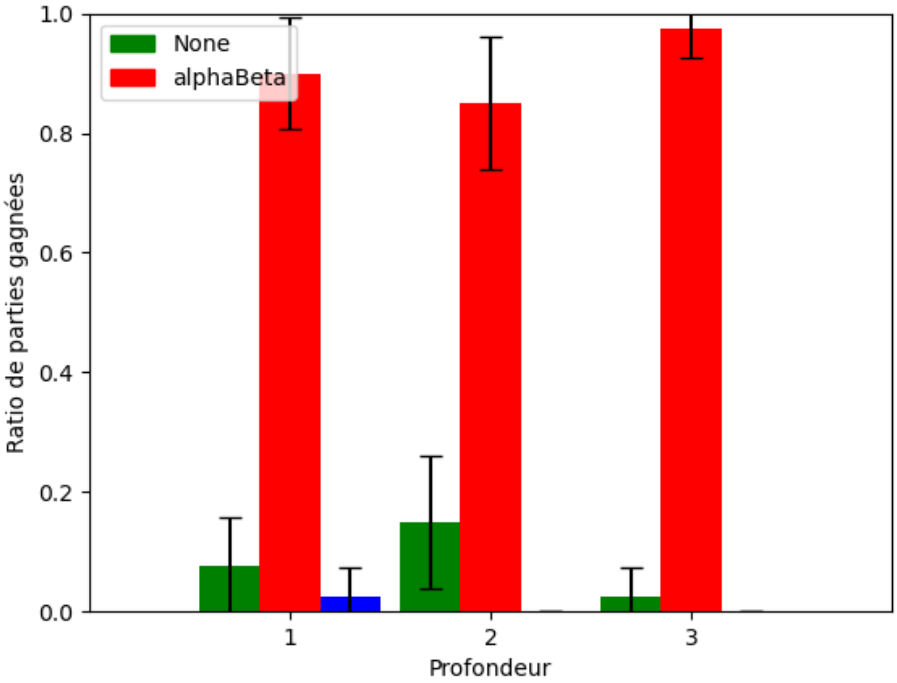

Figure 5: **Performance d'alphaBeta face au hasard**. *En vert: l'algorithme aléatoire, en rouge: l'algorithme alphaBeta avec un profondeur variant de 1 à 3. Les barres d'erreur correspondent à un intervalle de confiance à 95% d'une proportion.*

On remarque que Alphabeta/minMax est significativement bien meilleur que le hasard, et ce dès la profondeur 1.

3.2 MCTS

3.2.1 MCTS face au hasard

Dans un premier temps, nous avons donné à C la valeur 0.5 et réalisé un nombre de play outs par itération de 1. Nous verrons dans un deuxième temps d'autres valeurs pour ces 2 paramètres. Nous comparons les performances de MCTS face à celles du hasard (**Figure 6**).

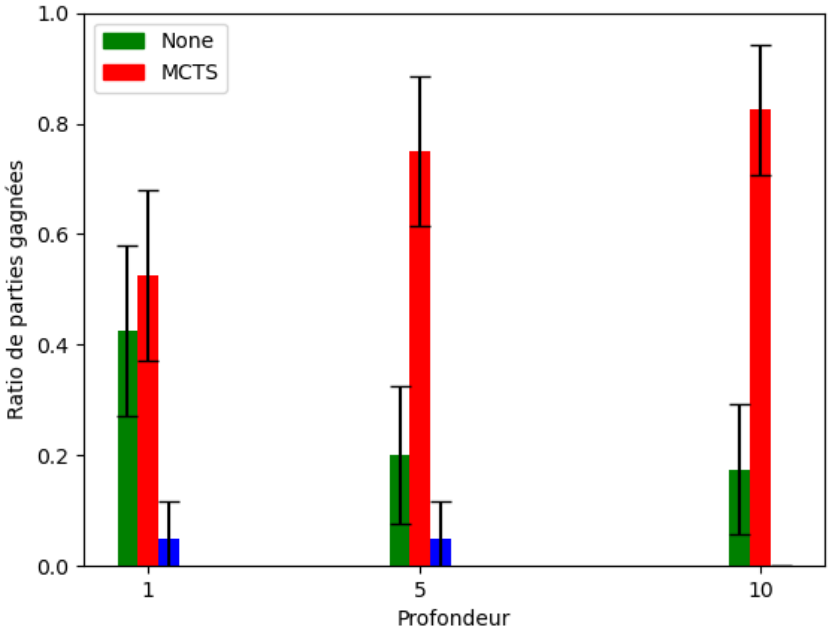

Figure 6: **Performance de MCTS face au hasard.** *En vert: l'algorithme aléatoire, en rouge: l'algorithme MCTS avec les paramètres $C = 0.5$ et play out $= 1$ avec un nombre d'itérations de 1, 5 et 10. Les barres d'erreur correspondent à un intervalle de confiance à 95% d'une proportion sur les 40 parties jouées.*

On observe qu'un nombre d'itération égal à 1 n'est pas significativement meilleur que le hasard. La raison est qu'avec uniquement 1 itération, MCTS n'a pas la possibilité d'envisager l'ensemble de tous les coups possibles. En revanche, un nombre d'itérations égal à 5 permet déjà d'obtenir des résultats significativement plus performants que le hasard. De plus grands nombres d'itérations semblent améliorer la performance.

3.2.2 Influence du paramètre C sur la performance de MCTS

Nous pouvons maintenant nous intéresser à l'influence des valeurs de C et du nombre de play outs dans la performance de MCTS. Nous avons réalisé de nouvelles simulations avec un nombre d'itérations et un play out fixés respectivement à 3 et 1, et en utilisant des valeurs extrêmes pour C à 0 et 10 (**Figure 7**).

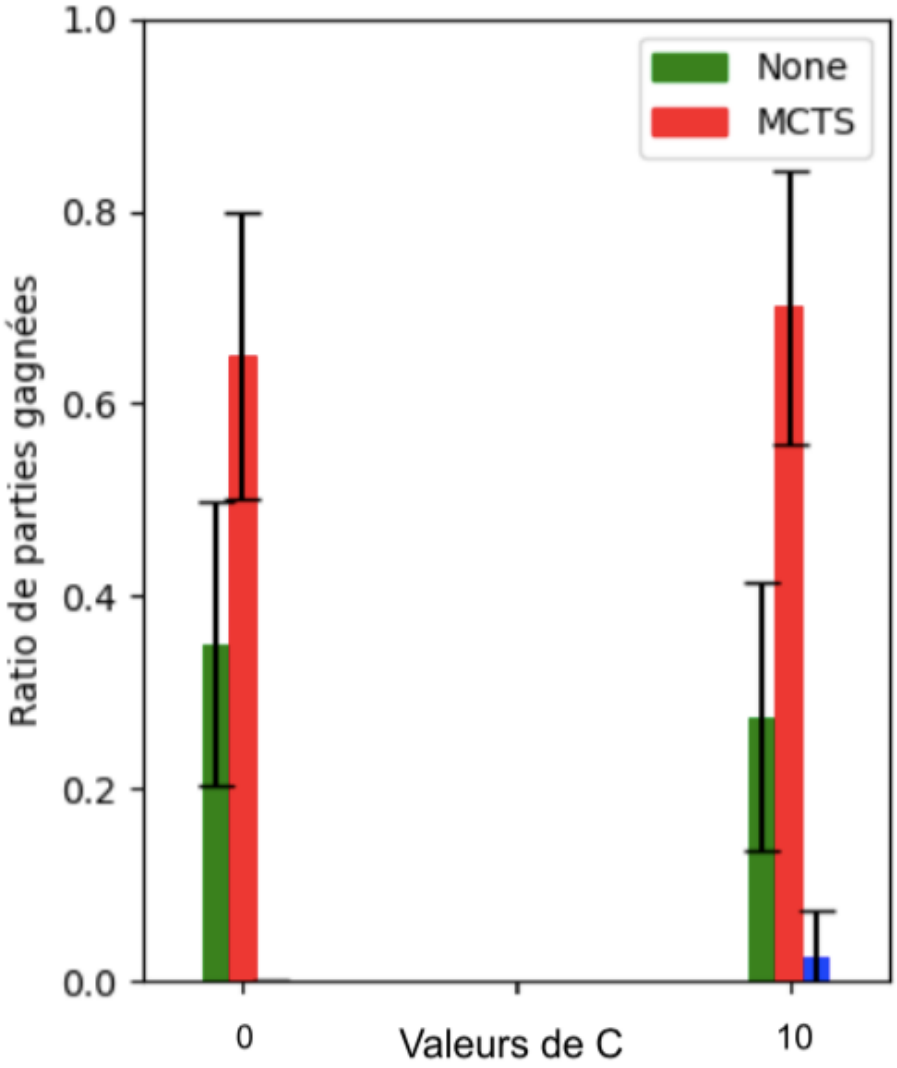

Figure 7: **Influence du paramètre C sur la performance de MCTS.***En vert: l'algorithme aléatoire, en rouge: l'algorithme MCTS avec les paramètres suivants : nombre d'itérations = 3, play out = 1 et C = 1 ou C = 10. Les barres d'erreur correspondent à un intervalle de confiance à 95% d'une proportion sur les 40 parties jouées.*

Plus C est grand, plus la part accordée à l'exploration dans MCTS est importante. Ainsi, à un nombre d'itération de 3 et un play out de 1, réaliser plus d'explorations semble améliorer la performance. Nous ne nous attendions pas forcément à ce résultat : augmenter le hasard dans un algorithme est rarement avantageux! Nous pensons que comme les simulations n'utilisent qu'un seul play out, l'algorithme ne dispose que de peu d'informations. Ainsi, face à ce peu d'information, l'algorithme en "tentant sa chance ailleurs" va récolter de meilleurs résultats. Avec d'autres valeurs de play out et nombre d'itérations, l'influence de C aurait peut être été différente. Nous verrons ceci dans la partie "Variations conjointes de C et play out".

3.2.3 Influence du paramètre "nombre de play outs" sur la performance de MCTS

Nous allons désormais évaluer l'influence du paramètre "nombre de play outs" sur la performance de MCTS. Nous avons réalisé de nouvelles simulations avec un nombre d'itérations et une valeur C fixés respectivement à 3 et 0.5, et en utilisant des valeurs de play-out variant entre 1 et 10 (**Figure 8**).

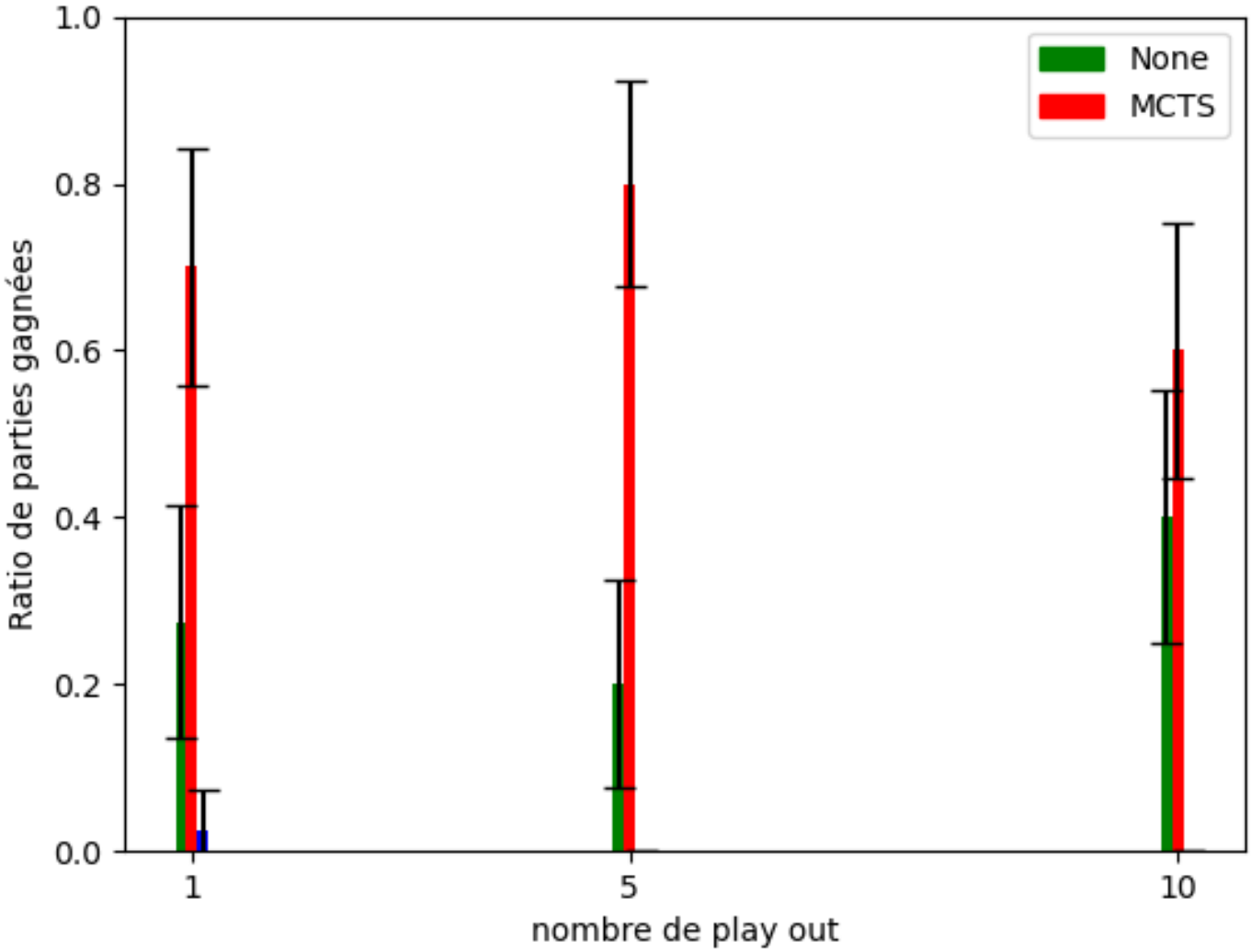

Figure 8: **Influence du play outs sur la performance de MCTS.** *En vert: l'algorithme aléatoire, en rouge: l'algorithme MCTS avec les paramètres suivants : nombre d'itérations = 3, C = 0.5 et play out variant entre 1 et 10. Les barres d'erreur correspondent à un intervalle de confiance à 95% d'une proportion sur les 40 parties jouées.*

La performance de MCTS est meilleure que celle de l'algorithme aléatoire pour un nombre de play outs = 5. En revanche pour play outs = 10, la performance n'est pas statistiquement meilleure que celle de l'algorithme aléatoire.

Ainsi, augmenter le nombre de play outs augmente la performance de MCTS par rapport à celle de l'algorithme aléatoire mais ce jusqu'à un certain point. Nous pensions voir la performance augmenter continuellement avec des play-out importants, car cela semble offrir plus d'informations. Cependant, l'information que nous apportent les play outs est approximative, étant donné que les parties y sont jouées aléatoirement. Peut-être que ces approximations, si elles sont trop nombreuses, s'accumulent jusqu'à ce que l'information remontée par les play-outs en devienne fausse. Ceci pourrait expliquer ce que l'on observe pour play outs = 10 sur la **Figure 8**.

3.2.4 Variations conjointes de C et play outs

Il est maintenant intéressant de voir comment évolue la performance de MCTS sous l'influence conjointe des variations de C et du nombre de play outs. Voici en **Figure 9** les résultats que nous avons obtenus.

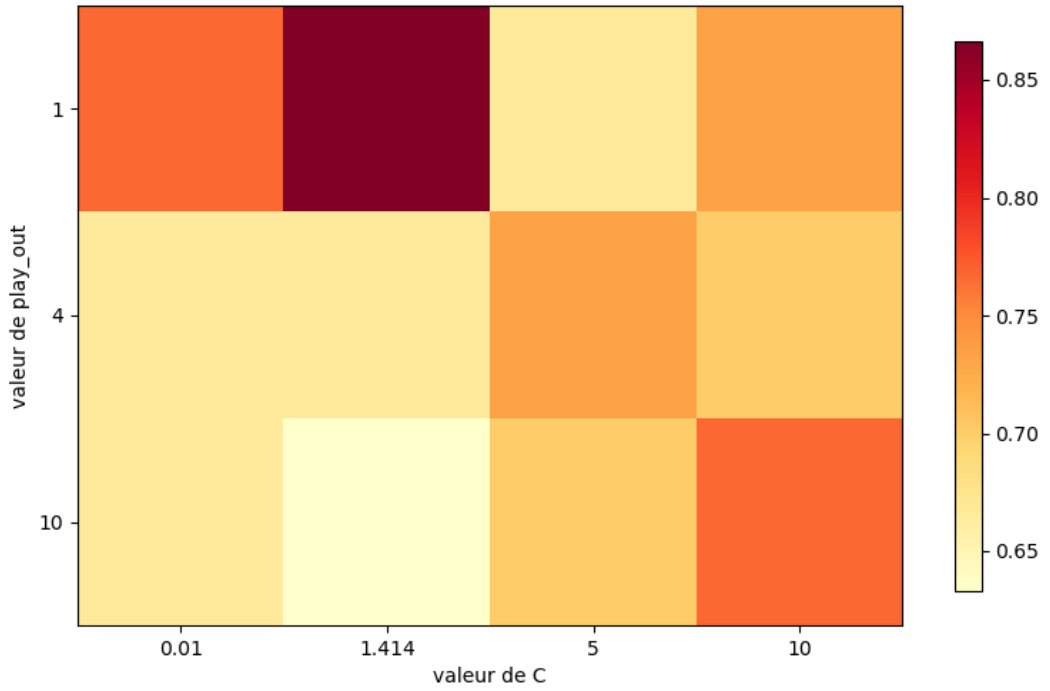

Figure 9: **Influence conjointe des paramètres C et play outs sur la performance de MCTS.** *Le nombre d'itérations utilisé ici est fixé à 3.*

La littérature indique que C varie habituellement entre 0 et 10. C'est ce que nous avons donc testé ici, et en particulier la valeur $\sqrt{2}$, souvent citée comme étant un bon compromis exploration/exploitation. En effet, nous pouvons remarquer que c'est avec cette valeur que la meilleure performance est obtenue, en combinaison avec la valeur de play out égale à 1. En revanche, il est aussi important de noter que la plus faible performance est aussi obtenue avec C = $\sqrt{2}$, cette fois-ci avec un play out de 10.

Il n'y a pas de tendance particulièrement marquée. Peut-être pouvons nous observer que pour des faibles valeurs de C, un nombre de play out petit est avantageux. En revanche, pour de grandes valeurs de C, alors il est plus avantageux d'avoir un play out important.

Cette étude n'est pas exhaustive : nous pourrions refaire la même étude mais avec un autre nombre d'itérations (ici, le nombre d'itération est fixé à 3). Les résultats seraient sans doute différents.

3.2.5 Est-il possible de comparer MCTS et MinMax/alphaBeta ?

Nous souhaitons maintenant comparer les performances de MCTS par rapport à celles de MinMax. Pour ce faire, nous devons répondre à plusieurs questions auparavant : Quel nombre d'itérations choisir pour MCTS ? Il faut comparer ce qui est comparable : une profondeur de 1 en minMax ne représente pas du tout la même valeur qu'un nombre d'itérations de 1 en MCTS.

Nous avons donc essayé de trouver une équivalence. Nous n'en avons pas trouvé : **alphaBeta ou minMax profondeur 1 sont toujours équivalents ou meilleurs que MCTS** (tant dans le ratio de parties gagnés que par le temps de calcul).