

Aide à la Décision - Othello

Antonin Boyon Quentin Legot Arthur Page

28 février 2021

Table des matières

1	Introduction	2
2	L'algorithme de recherche	2
2.1	Algorithme de base	2
2.2	Algorithme d'élagage	3
3	Mesures	4
3.1	Présentation	4
3.2	AlphaBeta	5
3.2.1	Profondeur 1	5
3.2.2	Profondeur 2	6
3.2.3	Profondeur 3	7
3.2.4	Profondeur 4	8
3.2.5	Profondeur 5	9
3.2.6	Conclusion d'Alphabeta	10
3.3	Negamax	10
3.3.1	Profondeur 1	10
3.3.2	Profondeur 2	11
3.3.3	Profondeur 3	13
3.3.4	Profondeur 4	14
3.3.5	Profondeur 5	15
3.3.6	Conclusion de Negamax	15
3.4	Negamax vs AlphaBeta	15
3.4.1	Profondeur 2	15
4	Difficultés rencontrés	15
5	Expérimentations	16
6	Conclusion	16

1 Introduction

Le but de notre projet était de concevoir un algorithme de recherche performant sur un jeu d' *Othello*. Le jeu est le plus abstrait possible, la partie nous intéressant étant la réalisation d'un algorithme de recherche efficace. Il est ainsi impossible de jouer au jeu, on ne peut que regarder le résultat d'une partie entre deux joueurs artificiels.

Une fois le jeu et l'algorithme de recherche implémentés, nous serons en mesure d'analyser ce dernier pour définir ses paramètres de fonctionnement optimaux. Nous aborderons dans un premier temps l'implémentation du jeu, puis celle de l'algorithme et enfin la présentation et l'analyse des mesures observées.

2 L'algorithme de recherche

2.1 Algorithme de base

Nous avons utilisé un algorithme Negamax pour résoudre le problème.

```
1 public Pair<Point, Point> play(State game) {
2     int bestValue = Integer.MIN_VALUE;
3     Pair<Point, Point> bestMove = null;
4     for(Pair<Point, Point> move : game.getMove(game.getCurrentPlayer()))
5         State nextState = game.play(move);
6         int value = -negamax(nextState, this.depth);
7         if (value > bestValue) {
8             bestValue = value;
9             bestMove = move;
10        }
11    }
12    return bestMove;
13 }
14
15 private int negamax(State state, int depth) {
16     if(depth == 0 || state.isOver()) {
17         return evaluate(state);
18     } else{
19         int m = Integer.MIN_VALUE;
```

```

20         for (Pair<Point, Point> move : state.getMove(state.getCurrentPlayer()))
21             State nextState = state.play(move);
22             complexity++;
23             m = Math.max(m, -negamax(nextState, depth - 1));
24         }
25         return m;
26     }
27 }

```

2.2 Algorithme d'élagage

```

1
2 public Pair<Point, Point> play(State game) {
3     int bestValue = Integer.MIN_VALUE;
4     Pair<Point, Point> bestMove = null;
5     for(Pair<Point, Point> move : game.getMove(game.getCurrentPlayer()))
6         State nextState = game.play(move);
7         complexity++;
8         int value = -alphabeta(nextState, this.depth, Integer.MIN_VALUE, Integer.MAX_VALUE);
9         if (value > bestValue) {
10             bestValue = value;
11             bestMove = move;
12         }
13     }
14     return bestMove;
15 }
16
17 private int alphabeta(State state, int depth, int alpha, int beta) {
18     if(depth == 0 || state.isOver()) {
19         return evaluate(state);
20     } else {
21         for (Pair<Point, Point> move : state.getMove(state.getCurrentPlayer()))
22             State nextState = state.play(move);
23             alpha = Math.max(alpha, -alphabeta(nextState, depth - 1, -beta, -alpha));
24             if(alpha >= beta)
25                 return alpha;
26     }

```

```
27         return alpha ;  
28     }  
29 }
```

3 Mesures

3.1 Présentation

Les graphiques qui vont suivre ont été conçus à l'aide des algorithmes AlphaBeta et Negamax.

Ils sont l'objet de comparaisons entre les algorithmes, en fonction de leur type ou du joueur concerné (premier ou second joueur).

Ils traduisent la complexité de l'algorithmes (le nombre de nœuds traversés) au fur et à mesure des tours de la partie.

Le premier joueur est associé à la courbe rouge et le deuxième à la bleue.

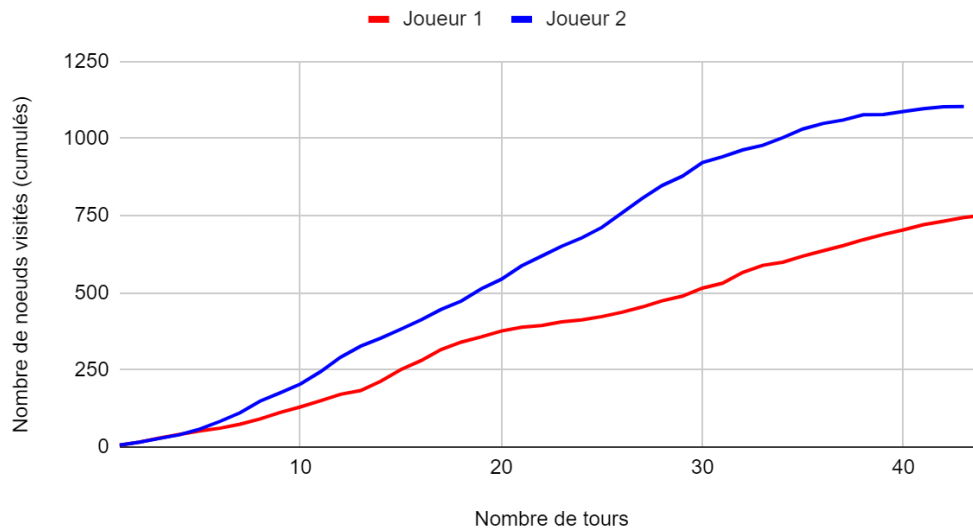
La profondeur de recherche des deux joueurs sera toujours la même.

Tout les tests incluant un temps ont été fait sur la même machine et en même temps : Raspberry pi 3 avec un processeur Quad Core 1.2GHz 64bit sous Raspbian OS 32 bits sans Bureau.

3.2 AlphaBeta

3.2.1 Profondeur 1

J1, J2 : Alphabeta, profondeur 2, J1 gagnant



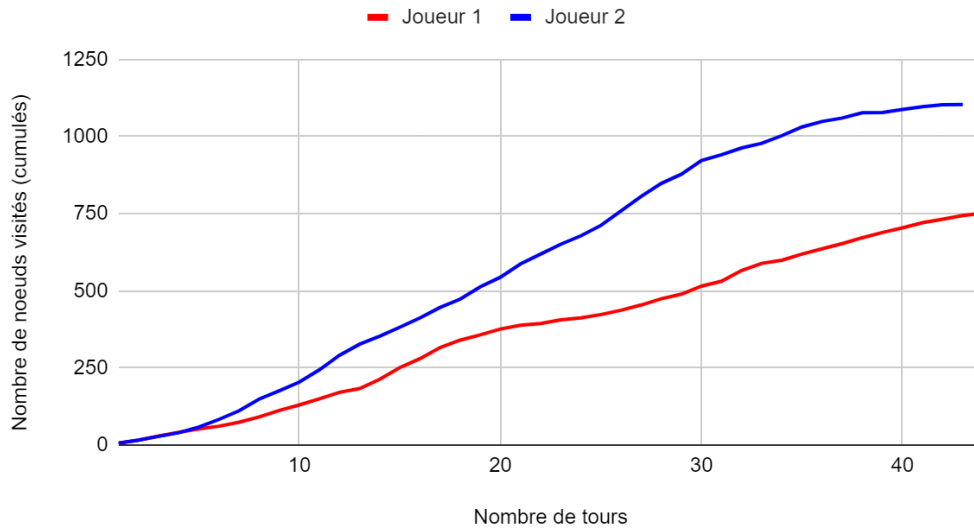
Le joueur 1 obtient assez vite (tour 5) un avantage (il possède plus de possibilités) qui augmente au fur et à mesure des tours. A son maximum (fin de la partie) cet avantage est 69% plus important par rapport au second joueur.

L'augmentation de la complexité est plutôt linéaire.

Il semblerait que jouer en premier est un avantage.

3.2.2 Profondeur 2

J1, J2 : Alphabeta, profondeur 2, J1 gagnant

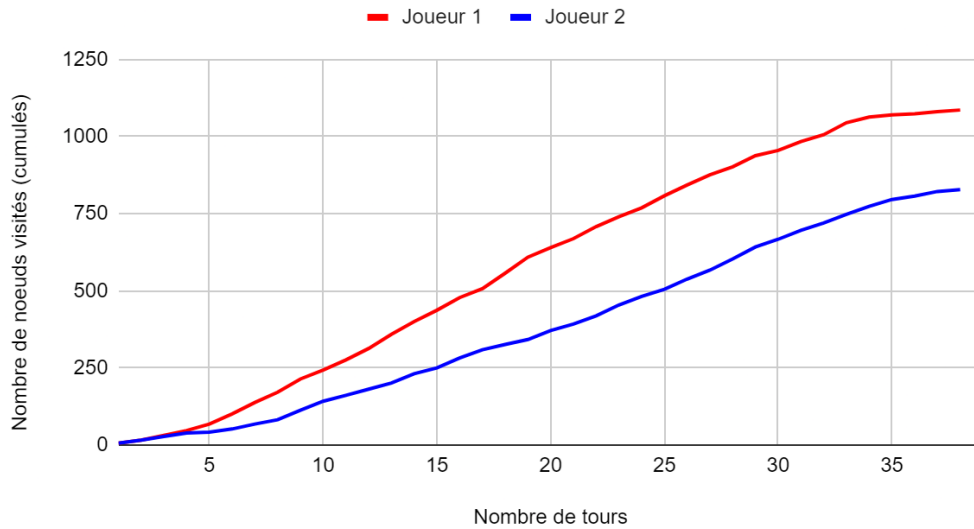


Malgré qu'il soit second à jouer, joueur 2 obtient un avantage au niveau du tour 5 environ. Cet avantage augmente jusqu'au tour 30, avec un pic à 79% par rapport au joueur 1. Il se réduit ensuite jusqu'à la fin de la partie. Le nombre de tour est largement inférieur par rapport au précédent graphique. La complexité du joueur 1 est deux fois moins importante que sur le graphique précédent, malgré la profondeur plus importante. Mais malgré cet avantage, la victoire est pour le joueur 1.

La courbe est linéaire, comme sur la graphique précédent. Être le premier à jouer semble donner un avantage, et le nombre de possibilités du joueur 2 plus important n'est pas suffisant pour le résorber. La profondeur ne semble pas forcément augmenter le nombre de possibilités.

3.2.3 Profondeur 3

J1, J2 : Alphabeta, profondeur 3, J2 gagnant

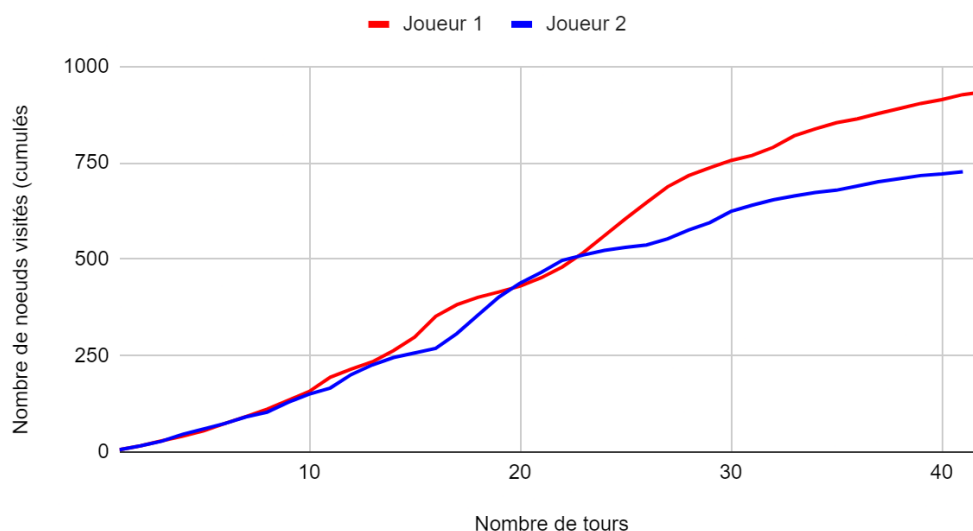


Comme pour la profondeur 1, le joueur 1 a tôt, au tour 5 environ, un avantage qui augmente également au fur et à mesure de la partie. Cet avantage atteint un pic au tour 25 avec 60% par rapport au second joueur. Cependant c'est le second joueur qui l'emporte, malgré son nombre de possibilités de jeu moins important.

Comme précédemment la courbe est plutôt linéaire. La profondeur plus importante de l'algorithme n'augmente pas le nombre de tour de la partie. Jouer en premier n'est pas obligatoirement avantageux.

3.2.4 Profondeur 4

J1, J2 : Alphabeta, profondeur 4, J1 gagnant



Le premier joueur possède un petit avantage à partir du tour 10 environ. Mais au tour 18 cet avantage se réduit, pour réaugmenter au tour 23 jusqu'à la fin de partie. Il atteint au maximum 28% par rapport au joueur 2. C'est le premier joueur qui remporte la partie.

Comme précédemment la courbe est plutôt linéaire. Le nombre de tour est similaire à l'algorithme de profondeur inférieur. Sans avantage, le joueur 2 ne semble pas pouvoir renverser la tendance et doit s'incliner.

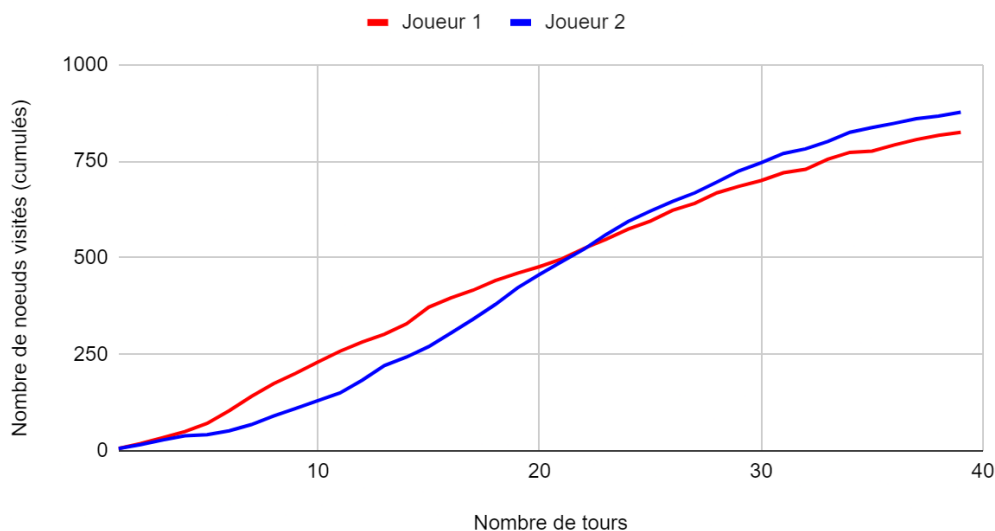
```
29 20
othello.players.AlphaBetaPlayer@88ff2c a gagné la partie
Score joueur 1 -> 0
Score joueur 2 -> 0

real    1m11,886s
user    1m10,713s
sys     0m0,728s
```

Le temps de résolution de cet algorithme est de 1 minute et 12 secondes pour un total de 1 661 nœuds visités (pour les deux joueurs).

3.2.5 Profondeur 5

J1, J2 : Alphabeta, profondeur 5, J1 gagnant



Au tour 5 environ, le premier joueur obtient un léger avantage qui se résorbe au tour 22.

Ensuite c'est le second joueur qui gagne un petit avantage qui se maintient jusqu'à la fin de partie .

Malgré l'avantage du joueur 2 à la fin, c'est le joueur 1 qui l'emporte.

Identique au précédent graphique, la courbe est plutôt linéaire et le nombre de tour n'augmente pas.

L'avantage du second joueur sur la fin de la partie ne semble pas suffisant pour l'emporter.

Pour 1 702 nœuds visités, l'algorithme dure 6 minutes et 54 secondes.

```
27 22
othello.players.AlphaBetaPlayer@88ff2c a gagné la partie
Score joueur 1 -> 0
Score joueur 2 -> 0

real    6m54,460s
user    6m49,528s
sys     0m2,353s
```

3.2.6 Conclusion d'Alphabeta

Jouer en premier donne un avantage, le premier joueur gagne plus souvent. Mais le second joueur surpasse des fois le premier joueur, alors qu'il n'a pas de réelle avantage. En effet, alors qu'il avait 60% de possibilité en moins par rapport au premier joueur, le second l'a emporté avec l'algorithme Alphabeta de profondeur 3. De plus, le second joueur possède un avantage de 79% au maximum par rapport au premier joueur avec l'algorithme Alphabeta de profondeur 2. Et malgré cet avantage, c'est le joueur 1 qui remporte la partie. Il semblerait que le nombre de possibilité n'influe pas sur la victoire avec cet algorithme, malgré ce qu'on pourrait logiquement penser.

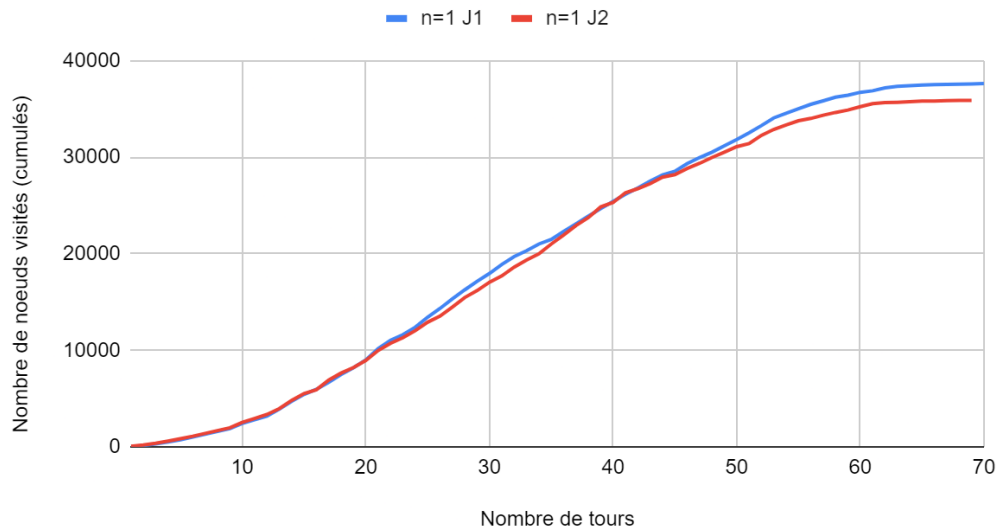
L'augmentation de la profondeur de l'algorithme AlphaBeta n'augmente pas forcément la complexité de ce dernier. Cependant l'augmentation de la complexité en fonction du nombre de tour est relativement linéaire. Le temps de résolution des algorithmes pour des petites profondeurs (1, 2, 3) est de quelques secondes mais augmente drastiquement avec la profondeur, AlphaBeta de profondeur 5 s'exécute durant plusieurs minutes.

3.3 Negamax

3.3.1 Profondeur 1

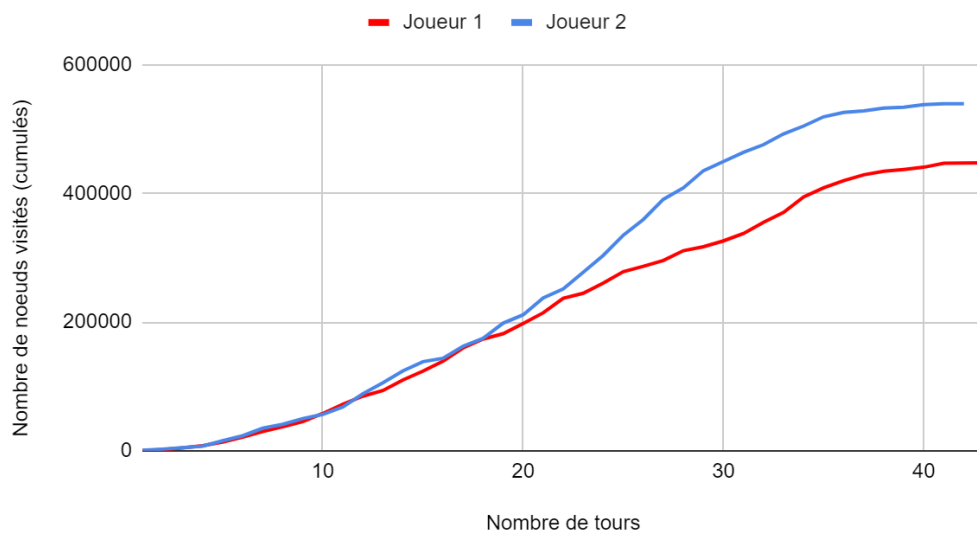
Aucun des joueurs n'a d'avantage particulier. La complexité est environ 20 fois plus importante qu'AlphaBeta. Le joueur 1 est le gagnant. Avec cet algorithme aussi il semblerait que le premier joueur possède un avantage. L'augmentation de la complexité est moins importante au début et à la fin de partie mais est assez linéaire.

J1, J2 : Negamax, profondeur 1, J1 gagnant



3.3.2 Profondeur 2

J1, J2 : Negamax, profondeur 2, J1 gagnant



La complexité est 40 fois plus importante qu'avec la profondeur précédente.

La forme de la courbe est similaire au graphique précédent.

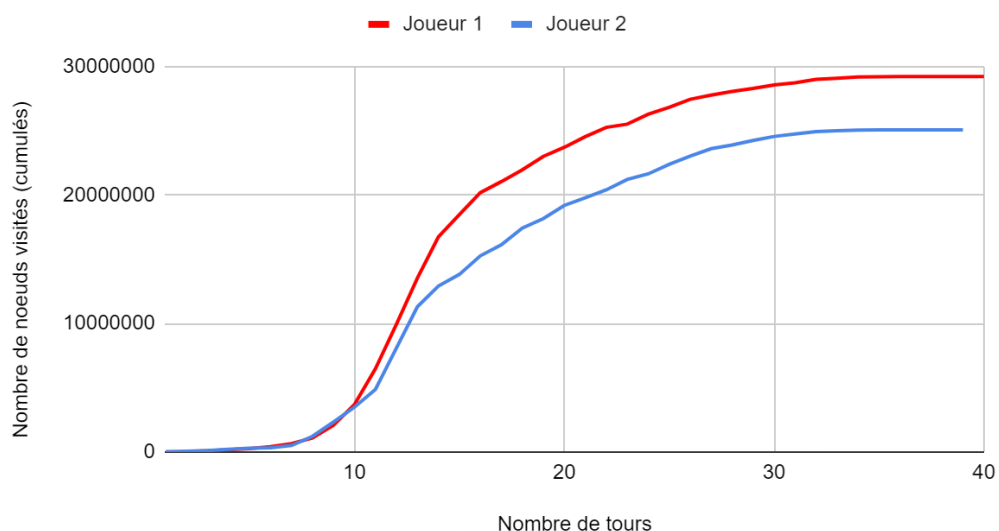
Aucun joueur n'a d'avantage majeur au cours de la partie. Le second joueur a un avantage qui commence au tour 19 jusqu'à la fin de la partie. Cet avantage atteint au maximum 37% par rapport au joueur 1.

Cependant c'est le premier joueur qui l'emporte, alors qu'il n'avait pas un grand avantage.

La complexité est bien plus importante qu'avec le précédent algorithme mais le nombre de tour est plus petit.

3.3.3 Profondeur 3

J1, J2 : Negamax, profondeur 3, J2 gagnant

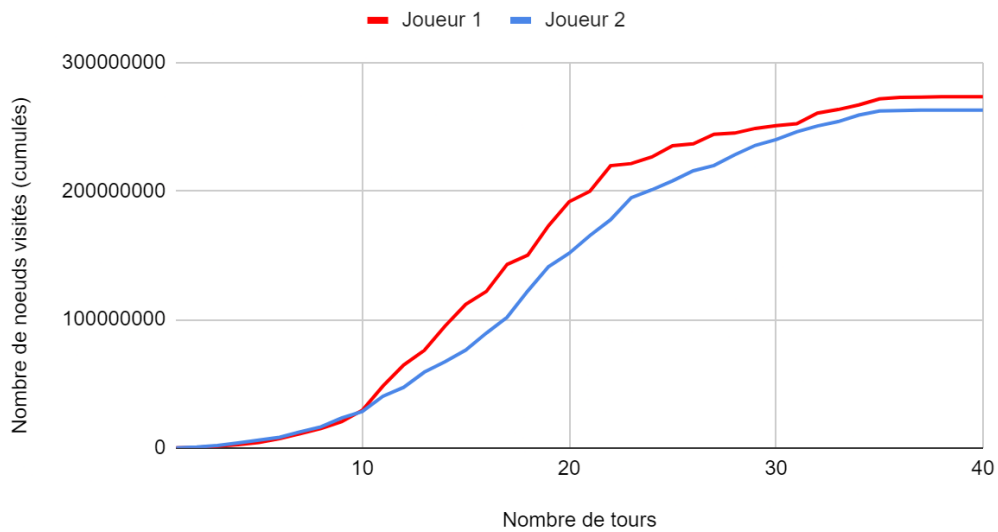


La complexité de cet algorithme est 55 fois supérieur au précédent. Le joueur 1 commence à avoir un avantage au tour 10, avantage qui augmente jusqu'à la fin de la partie. Cependant cet avantage n'est pas suffisant et c'est le second joueur qui gagne.

Les courbes évoluent de façon similaire. La complexité est bien plus grande qu'avec le précédent graphique mais le nombre de tour est le même.

3.3.4 Profondeur 4

J1, J2 : Negamax, profondeur 4, J2 gagnant



La complexité est environ 10 fois supérieure à Negamax de profondeur 3. Le premier joueur possède un avantage au tour 10 qui augmente jusqu'au tour 22 et se réduit ensuite jusqu'à la fin de la partie.

Malgré cet avantage c'est le second joueur qui remporte la partie.

```
32 17
othello.players.NegamaxPlayer@1851384 a gagné la partie
Score joueur 1 -> 0
Score joueur 2 -> 0

real    69m42,907s
user    69m22,313s
sys     0m13,479s
pi@raspberrypi:~/othello-4 $
```

Pour le parcours des 536 329 836 nœuds, l'algorithme met 69 minutes et 43 secondes.

3.3.5 Profondeur 5

Nous avons estimé la complexité de Negamax de profondeur 5 à au moins 8.2 milliards de nœuds.

Nous avons essayé de le lancer mais après 28 heures d'exécution sur le Raspberry Pi celui-ci n'était pas terminé, de plus la complexité stockée comme attribut de la classe Player est stockée sur un entier signé 32 bits dont la limite positive est $2^{31} - 1$ soit inférieur à 8 milliards.

3.3.6 Conclusion de Negamax

L'augmentation de la complexité en fonction de la profondeur de l'algorithme est exponentielle.

La victoire d'un joueur n'a pas l'air influé par son avantage ou par son ordre de jeu.

Du à sa complexité exponentielle, le temps d'exécution de cet algorithme l'est également. Cependant le nombre de tour de jeu n'a pas l'air influé par la complexité de Negamax.

3.4 Negamax vs AlphaBeta

3.4.1 Profondeur 2, AlphaBeta premier joueur, Negamax second

Dès le premier tour, le joueur 2 avec l'algorithme Negamax, possède un très grand avantage qui augmente fortement par rapport au premier joueur. Avec cette avantage il gagne la partie, malgré qu'il soit second à jouer.

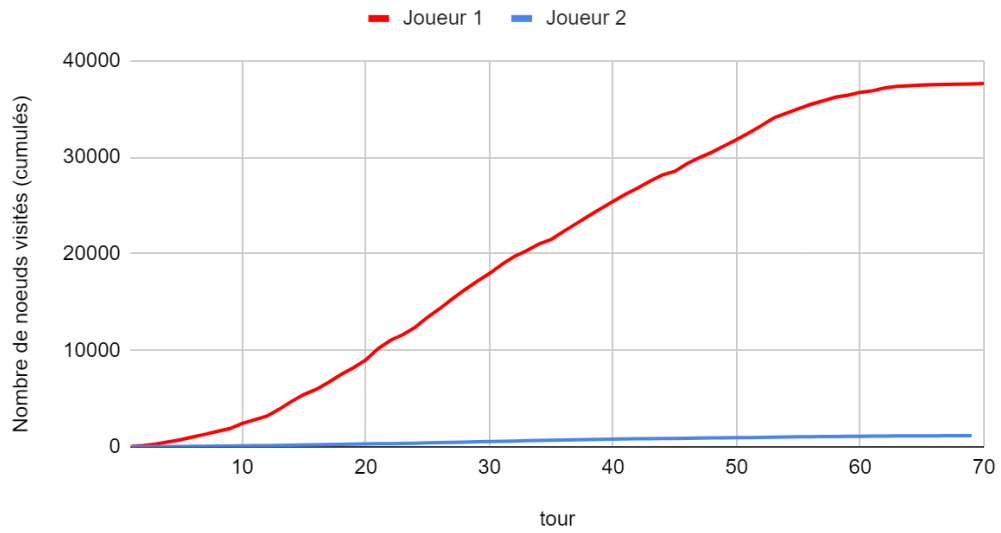
3.4.2 Profondeur 2, Negamax premier joueur, AlphaBeta second

Identiquement au graphique précédent, le premier joueur utilisant l'algorithme Negamax prend un avantage dès le premier tour. Son avantage augmente rapidement et se poursuit jusqu'à la fin de la partie pour enfin la remporter.

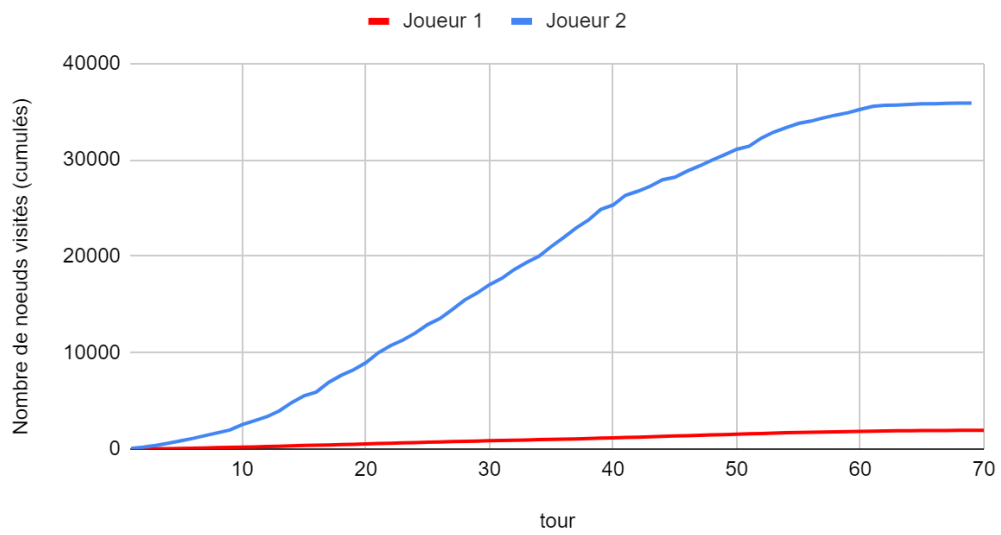
3.4.3 Conclusion de AlphaBeta vs Negamax

L'algorithme Negamax parcourt beaucoup plus de nœuds, il prend rapidement un très gros avantage. Cet algorithme est plus efficace de AlphaBeta, mais il est plus long à exécuter.

J1 : Negamax, J2 : AlphaBeta, profondeur 2, J1 gagnant



J1 : AlphaBeta, J2 : Negamax, profondeur 2, J1 gagnant



4 Difficultés rencontrés

Nous avons rencontrés quelques difficultés durant la réalisation du jeu, notamment du à des incompréhension des règles du jeu :

- Lors du clonage toutes les cases même inoccupées étaient modifiées.
- Le saut était possible même quand il n’y avait pas de pion en dessous et ne transformait pas les pions adverses.

5 Expérimentations

Nous avons mis en place un logger afin de pouvoir exporter plus facilement les données notamment pour faire les graphiques.

Le flux s’ouvre au lancement du programme et se ferme à sa fermeture ce qui fait que les fichiers de log resteront vides pendant le fonctionnement du programme, ce qui rend impossible la lecture de la complexité durant le fonctionnement du programme notamment pour le negamax de profondeur 5 ou plus qui demande beaucoup de performances et de temps de calcul.

6 Conclusion