



UNIVERSITÉ DE CAEN NORMANDIE

L2 INFORMATIQUE

Sécurité et aide à la décision Jeu d'infection

Membre du groupe :
Enzo Durand
Thomas Gignoux

Enseignant :
Grégory BONNET

1^{er} mars 2020

Table des matières

1	Introduction	1
1.1	Présentation du projet	1
2	Algorithme	3
2.1	Minimax	3
2.2	AlphaBeta	5
3	Analyse de données	6
3.1	Nombre de nœuds	6
3.1.1	Par tour	6
3.1.2	Par profondeur	12
3.2	Taux de victoire par rapport a la profondeur et au nombre de coups d'avance	14
3.3	analyse de temps	17
4	Bilan	19
4.1	Problèmes rencontrées	19
4.2	Conclusion	21

Chapitre 1

Introduction

1.1 Présentation du projet

Nous avons programmé un jeu dont les règles étaient données :

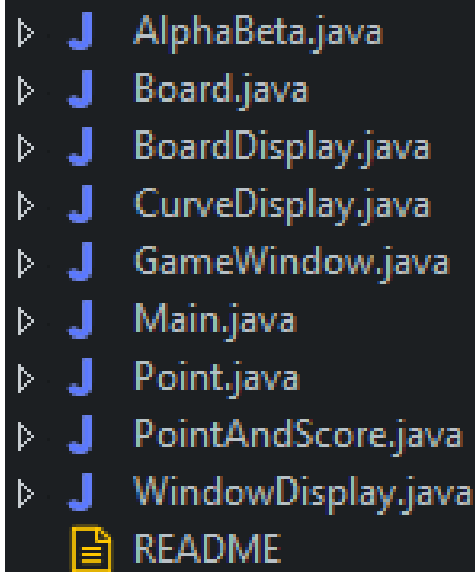
"Soit une grille de $N \times M$ cases. Chaque joueur Blanc et Noir débute la partie avec un pion, respectivement en bas à droite et en haut à gauche. L'objectif du jeu est de posséder en fin de partie le plus de pion possible, la partie se terminant lorsque l'un des joueurs ne dispose plus de pion, ou si aucun joueur ne peut faire de coups légaux. À chaque tour, le joueur actif choisit un pion et effectue une des actions suivantes avant de passer la main à son adversaire :

- Dupliquer le pion en en plaçant un nouveau sur une case libre à une distance de un dans une des quatre directions cardinales (haut, bas, gauche ou droite). Tout pion adverse en contact avec ce nouveau pion est transformé en pion du joueur.

- Déplacer le pion sur une case libre à une distance de deux dans une des quatre directions cardinale; ce mouvement permet de sauter par-dessous un autre pion (quel qu'en soit le propriétaire)."

Le but étant, une fois avoir programmé ce jeu, d'implémenter un algorithme appelé "MinMax" ainsi qu'un algorithme permettant d'optimiser MinMax en explorant moins de possibilités. Après avoir implémenté ces deux algorithmes à notre jeu, nous avons analysé les données par partie ou sur plusieurs parties. Nous avons fait une interface graphique permettant d'afficher différentes courbes extraites d'une partie. Nous avons aussi créer un tableur et récupéré des données sur plusieurs parties.

Voici les différentes classes de notre programme :



```
▶ J AlphaBeta.java
▶ J Board.java
▶ J BoardDisplay.java
▶ J CurveDisplay.java
▶ J GameWindow.java
▶ J Main.java
▶ J Point.java
▶ J PointAndScore.java
▶ J WindowDisplay.java
📄 README
```

Ainsi que les différentes méthodes de notre class Board

- Board(int, int)
- alphabetaCompare(AlphaBeta, int, boolean, boolean, Point) : boolean
- buildBoard() : void
- calculateScore(int) : int
- displayBoard() : void
- getBoard() : int[][]
- getCopy() : Board
- getScoreOnBoard(Point) : PointAndScore
- getValidMoves(int) : List<Point>
- isOver() : int
- max(ArrayList<PointAndScore>) : PointAndScore
- min(ArrayList<PointAndScore>) : PointAndScore
- miniMax(int, boolean, Board, AlphaBeta, boolean, boolean) : PointAndScore
- play(Point, int) : void
- randomMove(int) : Point

Et voici comment se présente le déroulement de la partie dans la console.

```
----- TOUR : 115 -----
```

```
---> Coups valides pour le joueur 1[[5, 4] , Type de move : 2, [6, 4] , Type de move : 2, [5, 4] , Type de move : 1, [5, 4] , Type de move : 1, [7, 5] , Type de move : 1]
```

```
---> Le joueur 1 (minimax) a joue le coup : [6, 4] , Type de move : 2
```

```
---> Nombre de noeuds parcouru par le joueur 1 ( profondeur 1) : 132940
```

```
[0|1|2|3|4|5|6|7|8|9|0|1|2|3|4|
1|#|#|#|#|#|#|#|#|#|#|#|#|#|
2|#|2|2|1|2|1|1|1|1|1|1|1|1|#|
3|#|2|1|1|1|1|1|1|1|1|1|1|1|#|
4|#|2|2|1|2|1|1|1|1|1|1|1|1|#|
5|#|2|1| |1|1|1|1|1|1|1|1|1|#|
6|#|2|2|1| |1|1|1|1|1|1|1|1|#|
7|#|2|2| | |1|1|1|2|1|1|1|1|#|
8|#|2|2|2|1|2|1|2|1|1|1|1|1|#|
9|#|2|2|2|2|2|2|2|2|1|2|1|1|#|
0|#|2|2|2|2|2|2|2|2|2|1|2|1|#|
1|#|2|2|2|2|2|2|2|2|2|2|1|1|#|
2|#|2|2|2|2|2|2|2|2|2|2|2|1|#|
3|#|#|#|#|#|#|#|#|#|#|#|#|#|
4|#|#|#|#|#|#|#|#|#|#|#|#|#|
```

```
---> Score Joueur 1 : 59
---> Score Joueur 2 : 58
```

```
----- TOUR : 116 -----
```

```
---> Coups valides pour le joueur 2[[5, 4] , Type de move : 2, [5, 4] , Type de move : 1, [5, 4] , Type de move : 1, [7, 5] , Type de move : 2, [6, 5] , Type de move : 1]
```

```
---> Le joueur 2 (minimax) a joue le coup : [7, 4] , Type de move : 1
```

```
---> Nombre de noeuds parcouru par le joueur 2 ( profondeur 1) : 132229
```

```
[0|1|2|3|4|5|6|7|8|9|0|1|2|3|4|
1|#|#|#|#|#|#|#|#|#|#|#|#|#|
2|#|2|2|1|2|1|1|1|1|1|1|1|1|#|
3|#|2|1|1|1|1|1|1|1|1|1|1|1|#|
4|#|2|2|1|2|1|1|1|1|1|1|1|1|#|
5|#|2|1| |1|1|1|1|1|1|1|1|1|#|
6|#|2|2|2| |1|1|1|1|1|1|1|1|#|
7|#|2|2|2| |1|1|1|2|1|1|1|1|#|
8|#|2|2|2|1|2|1|2|1|1|1|1|1|#|
9|#|2|2|2|2|2|2|2|2|1|2|1|1|#|
0|#|2|2|2|2|2|2|2|2|2|1|2|1|#|
1|#|2|2|2|2|2|2|2|2|2|2|1|1|#|
2|#|2|2|2|2|2|2|2|2|2|2|2|1|#|
3|#|#|#|#|#|#|#|#|#|#|#|#|#|
4|#|#|#|#|#|#|#|#|#|#|#|#|#|
```

```
---> Score Joueur 1 : 58
---> Score Joueur 2 : 60
```

Chapitre 2

Algorithme

2.1 Minimax

Dans une première partie nous nous sommes intéressés au minmax de profondeur 0, cet algorithme permettait de jouer mais sans récursivité, il ne fonctionnait donc pas en profondeur 1 et plus. Après avoir réussi à faire cet algorithme, nous avons travaillé sur l'implémentation de la récursivité de cet algorithme.

Au début nous nous étions précipités sur le jeu, l'implémentation de MinMax était donc incompatible avec notre jeu. Nous avons donc tout supprimé pour recommencer sur de meilleure base. Après avoir reprogrammé le jeu de base, l'implémentation de MinMax était plus facile.

Les captures d'écrans suivantes sont des tentatives avant d'arriver au minmax final.

```
public int minimax(int depth, int turn) {
    if(getWinner(player2) && this.isOver() == 1) {
        return 1;
    }
    else if(getWinner(player1) && this.isOver() == 1) {
        return -1;
    }
    else if(this.isOver() == 2){
        return 0;
    }
    List<Point> validMoves2 = getValidMoves(2);
    int max = Integer.MAX_VALUE;
    int min = Integer.MIN_VALUE;
    if(turn == player1) {
        for(int i = 0; i < validMoves2.size(); i++) {
            Point point = validMoves2.get(i);
            play(point, player1);
            int currentScore = minimax(depth + 1, player2);
            max = Math.max(currentScore, max);
            if(depth == 0) {
                System.out.println("Point : " + point + ", Score : " + currentScore);
            }
            if(currentScore >= 0) {
                if(depth == 0) {minimaxMove = point;}
            }
            if(currentScore == 1) {
                board[point.x][point.y] = noPlayer;
                break;
            }
            if(i == validMoves2.size() - 1 && max < 0) {
                if(depth == 0) {minimaxMove = point;}
            }
        }
    }
    else if(turn == player2) {
        for(int i = 0; i < validMoves2.size(); i++) {
            Point point = validMoves2.get(i);
            play(point, player2);
            int currentScore = minimax(depth + 1, player1);
            min = Math.min(currentScore, min);
            if(min == -1) {
                board[point.x][point.y] = noPlayer;
                break;
            }
            board[point.x][point.y] = noPlayer;
        }
    }
    return turn == player2 ? max : min;
}
```

```

//ENTREE : depth PROFONDEUR DE RECHERCHER, player JOUEUR UTILISANT L'ALGO
//ACTION :
public int minimax(int depth, int player) {
    if(depth == 0) { //DEPTH = 0
        return 0;
    }
    else { //DEPTH != 0
        if(player == player1) { //PLAYER1
            PointAndScore pointAndScore1 = evaluate(player1, getValidMoves(player1)); //point1 = meilleur point possible pour le joueur 1
            minmaxMove1 = pointAndScore1;
            int currentScore1 = pointAndScore1.getScore();
            System.out.println("currentScore1 : " + currentScore1);
            minimax(depth - 1, player2);
        }
        else if(player == player2) { //PLAYER2
            PointAndScore pointAndScore2 = evaluate(player2, getValidMoves(player2)); //point2 = meilleur point possible pour le joueur 2
            minmaxMove2 = pointAndScore2;
            int currentScore2 = -(pointAndScore2.getScore());
            System.out.println("currentScore2 : " + currentScore2);
            minimax(depth - 1, player1);
        }
    }
    return 0;
}
}

```

Après avoir vraiment compris l'algorithme en général, et après plusieurs tentatives nous avons réussi à implémenter l'algorithme et à le debugger.
La capture d'écran suivante est notre MinMax final.

```

//ENTREE : depth PROFONDEUR DE RECHERCHER, player JOUEUR UTILISANT L'ALGO, virtualBoard COPIE DE LA BORDE AU MOMENT DE L'APPEL DE MINIMAX, ab OBJET CONTENANT ALPHA BETA
//ENTREE : alphabetaUse BOOLEAN UTILISATION DE L'ALGO, playerInitial RENVOYANT LE JOUEUR DE BASE MÊME APRES RECURSION
//RETURN : UN OBJET PointAndScore CONTENANT LE POINT POUR LEQUEL LE SCORE EST MAXIMISE OU MINIMISE SUIVANT LE player
public PointAndScore miniMax(int depth, boolean player, Board virtualBoard, AlphaBeta ab, boolean alphabetaUse, boolean playerInitial) {
    Board virtualBoardN;
    PointAndScore current;
    AlphaBeta AlphaBetaN = new AlphaBeta();
    ArrayList<PointAndScore> tab = new ArrayList<PointAndScore>();
    List<Point> valideMoves = virtualBoard.getValidMoves((player ? 1 : 2));
    int lenght = valideMoves.size();
    //EVITE BUG FIN DE PARTIE
    if (lenght == 0) {
        return null;
    }
    //ON ITERE SUR LES COUPS VALIDES DU JOUEUR1/JOUEUR2
    for(int i = 0; i < lenght; i++) {
        //COMPTAGE NOMBRE DE NOEUDS PARCOURUS JOUEUR1 ET JOUEUR2
        if(player) {
            if(playerInitial) {nbNoeudsJoueur1 = nbNoeudsJoueur1.add(BigInteger.ONE);}
            else {nbNoeudsJoueur2 = nbNoeudsJoueur2.add(BigInteger.ONE);}
        }
        else {
            if(playerInitial) {nbNoeudsJoueur1 = nbNoeudsJoueur1.add(BigInteger.ONE);}
            else {nbNoeudsJoueur2 = nbNoeudsJoueur2.add(BigInteger.ONE);}
        }
        //ON JOUE LES COUPS VALIDES 1 PAR 1 SUR UNE COPIE A CHAQUE ITERATION
        virtualBoardN = virtualBoard.getCopy();
        virtualBoardN.play(valideMoves.get(i), (player ? 1 : 2));
        //PROFONDEUR 0
        if (depth == 0) {
            //ON REMONTE LA VALEUR EVALUER EN DEPTH 0
            current = virtualBoardN.getScoreOnBoard(valideMoves.get(i));
            tab.add(current);
            //ON BREAK SI alphabetaCompare EST true
            if(alphabetaCompare(ab, current.getScore(), player, alphabetaUse, valideMoves.get(i))) {
                breakN++;
                break;
            }
        }
        //PROFONDEUR != 0
    } else {
        //UTILISATION DE LA RECURSION EN depth-1 SUR L'AUTRE JOUEUR
        current = miniMax(depth - 1, !player, virtualBoardN, AlphaBetaN, alphabetaUse, playerInitial);
        //SI L'APPEL DE MINIMAX RENVOIE null CAR PLUS DE COUPS VALIDES ON MET LA VALEUR ACTUEL DANS current
        if (current == null) {
            current = virtualBoardN.getScoreOnBoard(valideMoves.get(i));
        }
    }
}

```

```

//SINON ON CREER UN NOUVEAU PointAndScore CONTENANT LE COUP VALIDE ET SON EVALUATION
current = new PointAndScore(valideMoves.get(i), current.getScore());
//ON AJOUTE CETTE VALEUR A LA LISTE
tab.add(current);
//ON BREAK SI alphabetaCompare EST true
if(alphabetaCompare(ab, current.getScore(), player, alphabetaUse, valideMoves.get(i))) {
    breakN++;
    break;
}
}
//ON RETOURNE LE PointAndScore MAX SI JOUEUR MAX OU MIN SI JOUEUR MIN
return player ? max(tab) : min(tab);
}

```

```

//ENTREE : tab LISTE D'OBJET POINTANDSCORE
//RETURN : index DE L'OBJET POINTANDSCORE POUR LEQUEL LE PARAMETRE SCORE EST MINIMUM
public PointAndScore min(ArrayList<PointAndScore> tab) {
    int min = tab.get(0).getScore();
    List<PointAndScore> indexTab = new ArrayList<>();
    for(int i = 0; i < tab.size(); i++) {
        if(tab.get(i).getScore() < min) {
            min = tab.get(i).getScore();
        }
    }
    for(int i = 0; i < tab.size(); i++) {
        if(tab.get(i).getScore() == min) {
            indexTab.add(tab.get(i));
        }
    }
    //CES DEUX LIGNES SERVENT A RENDRE LES PARTIES RANDOM
    //CAR SINON ON PEUT RELANCER MINMAX1 VS MINMAX1 SUR UNE BOARD DE MEME TAILLE
    //LA PARTIE SERA TOUJOURS LA MEME !
    Random r = new Random();
    return indexTab.get(r.nextInt(indexTab.size()));
}

```

```

//ENTREE : tab LISTE D'OBJET POINTANDSCORE
//RETURN : index DE L'OBJET POINTANDSCORE POUR LEQUEL LE PARAMETRE SCORE EST MAXIMUM
public PointAndScore max(ArrayList<PointAndScore> tab) {
    int max = tab.get(0).getScore();
    List<PointAndScore> indexTab = new ArrayList<>();
    for(int i = 0; i < tab.size(); i++) {
        if(tab.get(i).getScore() > max) {
            max = tab.get(i).getScore();
        }
    }
    for(int i = 0; i < tab.size(); i++) {
        if(tab.get(i).getScore() == max) {
            indexTab.add(tab.get(i));
        }
    }
    //CES DEUX LIGNES SERVENT A RENDRE LES PARTIES RANDOM
    //CAR SINON ON PEUT RELANCER MINMAX1 VS MINMAX1 SUR UNE BOARD DE MEME TAILLE
    //LA PARTIE SERA TOUJOURS LA MEME !
    Random r = new Random();
    return indexTab.get(r.nextInt(indexTab.size()));
}

```

2.2 AlphaBeta

L'algorithme AlphaBeta semblait plutôt simple après avoir implémenter MinMax. Nous avons donc essayé de l'implémenter puis de le debugger mais le nombre d'informations même sur une petite grille et une petite profondeur de minmax rendait le problème complexe. Après plusieurs tentatives nous n'avons pas réussi à implémenter AlphaBeta. Le problème étant peu être la structure de MinMax, il y a sûrement besoin de changer MinMax afin de pouvoir implémenter AlphaBeta. Le problème semble être lié la récursivité.

```

//ENTREE : ab OBJET ALPHABETA, toEval LA VALEUR A EVALUER, player LE JOUEUR ACTUEL
//RETURN : true ou false SUIVANT SI BETA EST INFERIEUR OU PAS A ALPHA
public boolean alphabetaCompare(AlphaBeta ab, int toEval, boolean player, boolean alphabetaUse, Point valideMove) {
    if(!alphabetaUse) {
        return false;
    }
    if(player) {
        ab.setAlpha(Integer.max(ab.getAlpha(), toEval));
    }
    else {
        ab.setBeta(Integer.min(ab.getBeta(), toEval));
    }
    return ab.getBeta() <= ab.getAlpha();
}

```

Chapitre 3

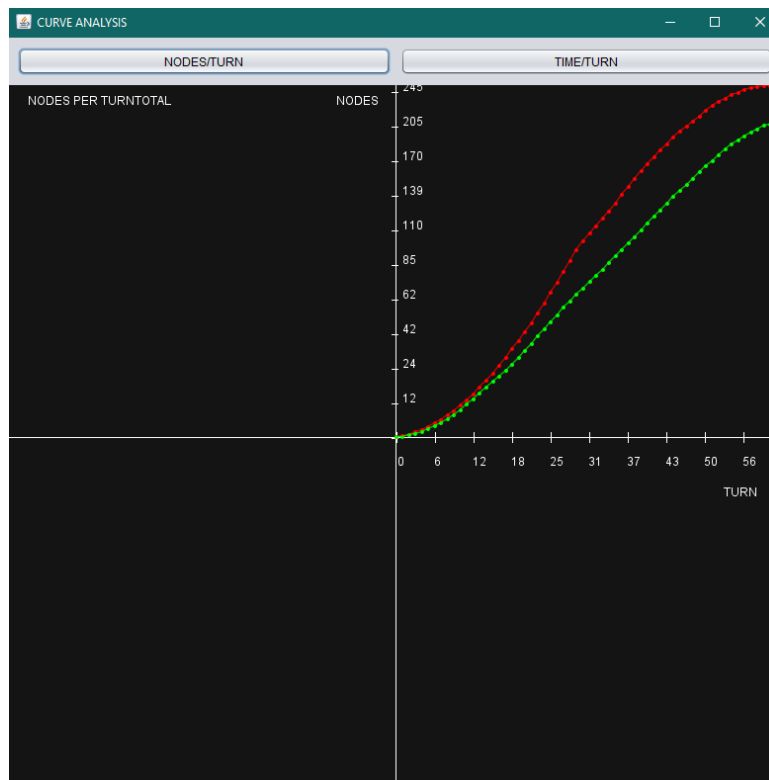
Analyse de données

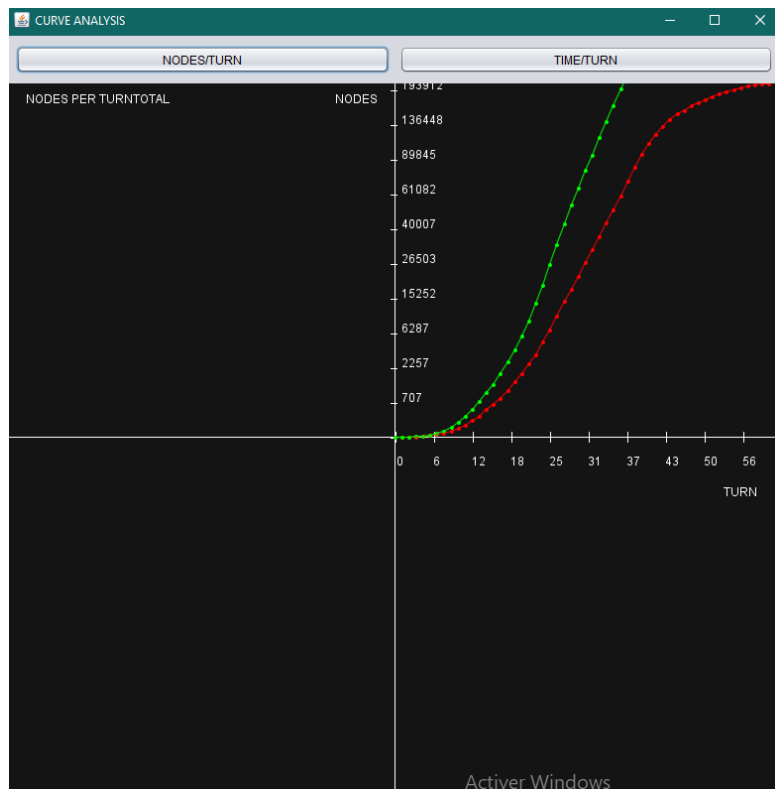
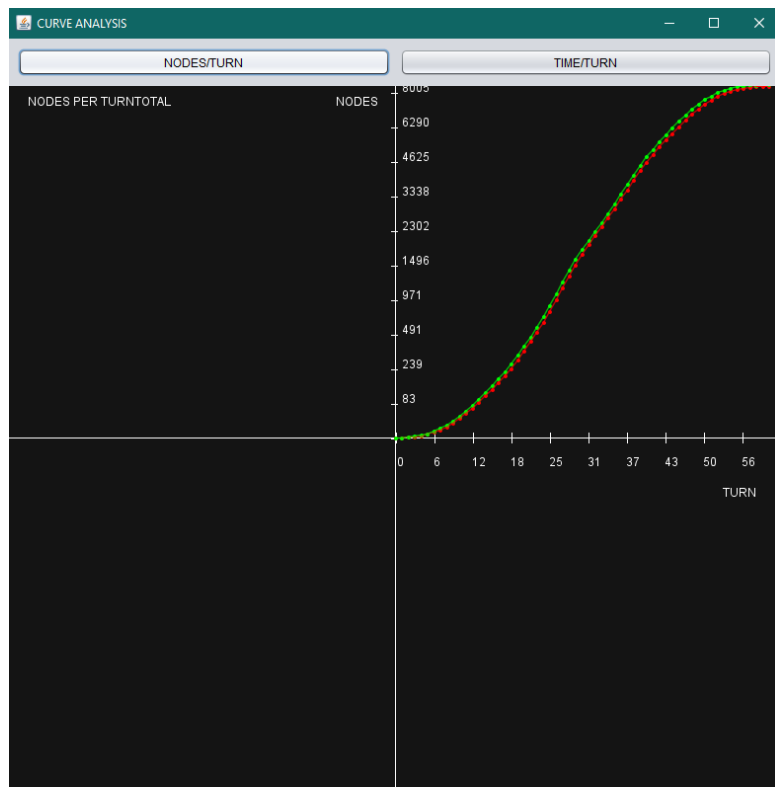
3.1 Nombre de nœuds

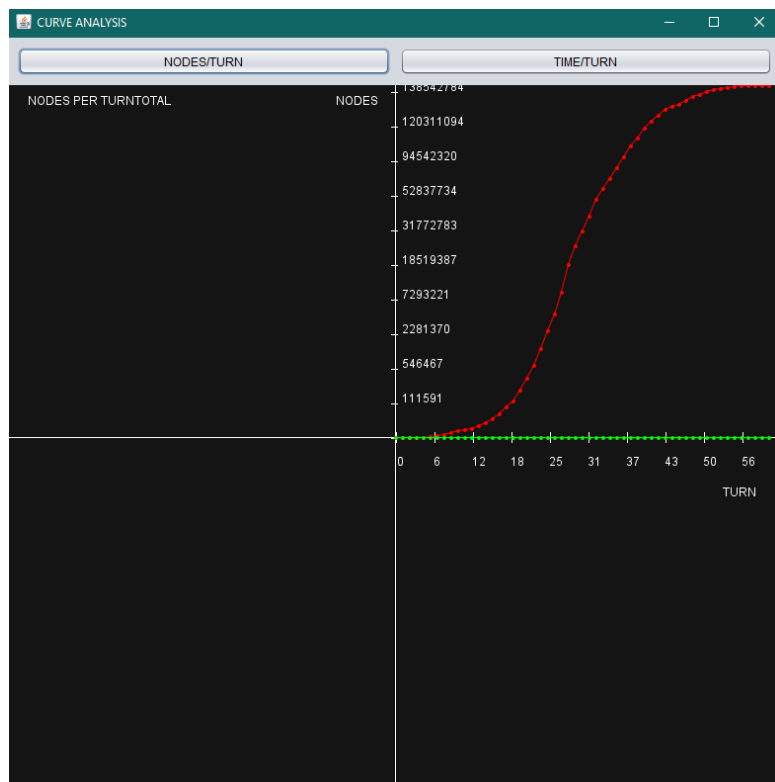
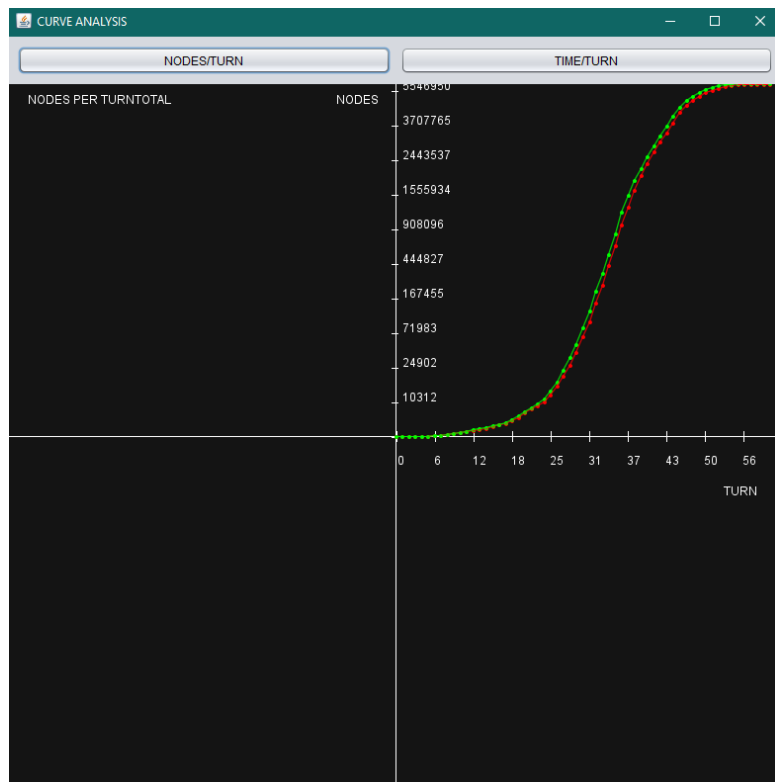
3.1.1 Par tour

Pour pouvoir analyser le nombre de nœuds explorés pendant une partie par un joueur utilisant MinMax, nous avons implémenter une interface graphique qui trace une courbe à la fin de la partie. Nous avons donc lancé plusieurs parties puis nous avons tracé les courbes suivant le nombre de tour. Les courbes suivantes sont des courbes représentant le nombre total de nœuds parcourus par tour ainsi que les courbes représentant le nombre de nœuds à chaque tours.

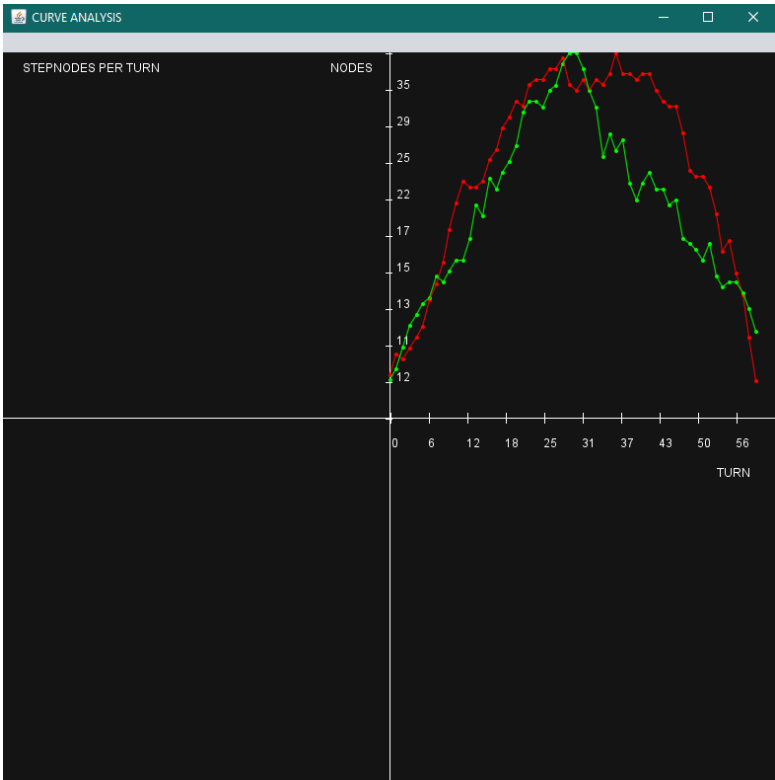
On voit sur les premières types de courbes que au début de la partie il n'y a pas beaucoup de nœuds parcourus car il y a peu de pions sur la grille donc peu de possibilités. Ensuite le nombre de nœuds accélère rapidement car il y a de plus en plus de possibilités. A la fin de la partie le nombre de nœuds redescend car il n'y a plus beaucoup de place libres dans la grille donc peu de coups valides.

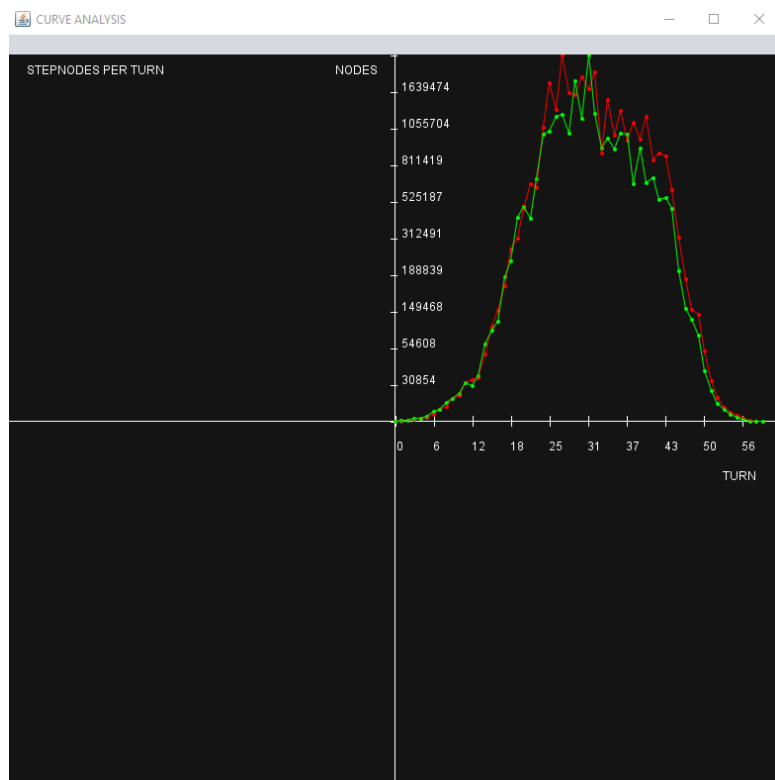
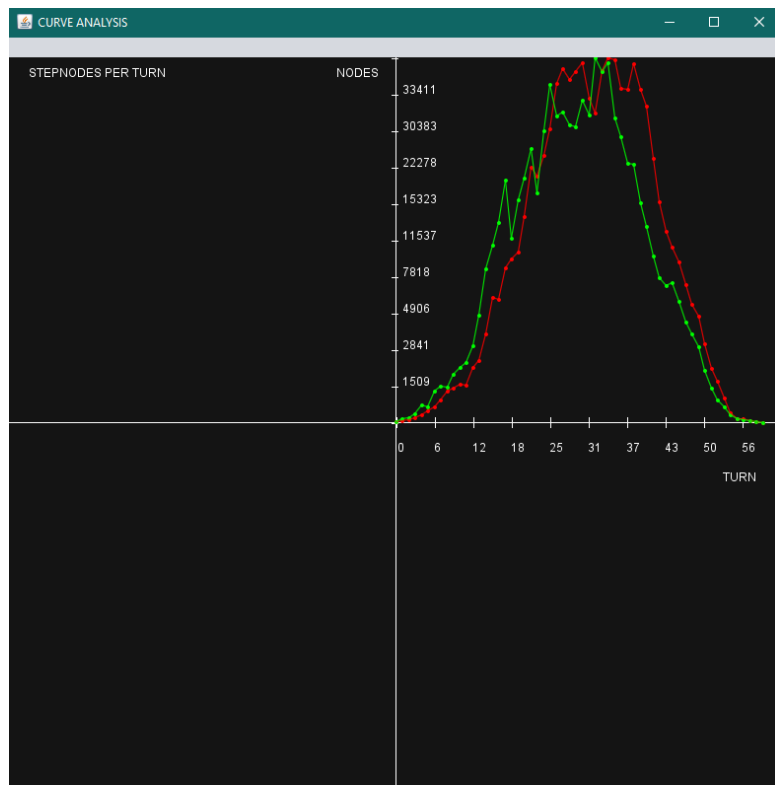


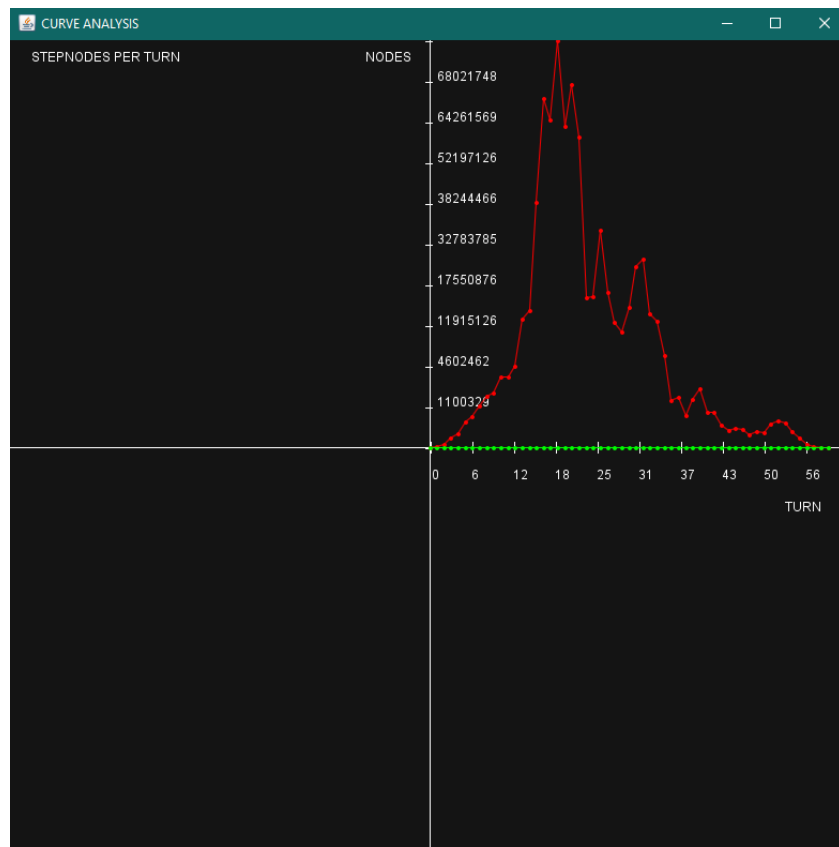




Sur les deuxièmes types de courbes nous voyons le même processus mais sous un autre angle.





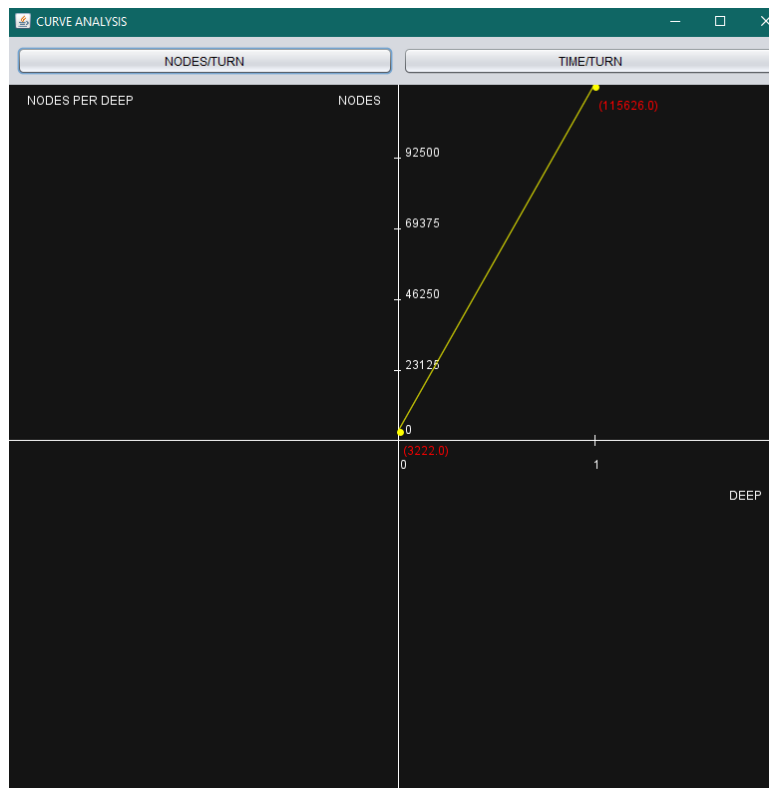


Si nous avons réussi à implémenter alphabeta nous aurions tracé les courbes avec deux joueurs de même profondeur minmax, mais l'un avec alphabeta activé et l'autre sans. Nous aurions donc observé que la courbe du joueur avec alphabeta activé aurait été en dessous de l'autre joueur.

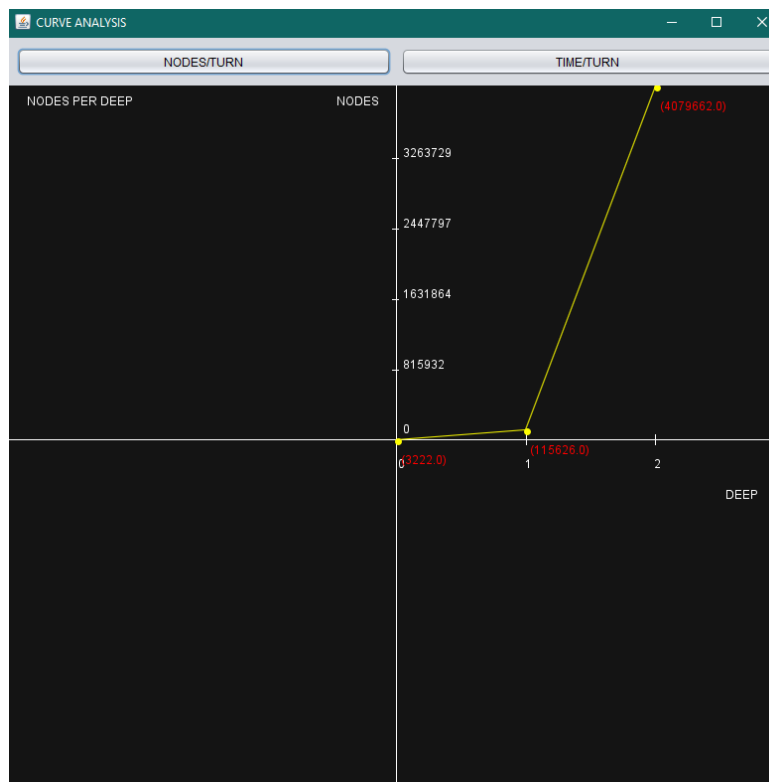
3.1.2 Par profondeur

Pour comprendre à quel point le nombre de nœuds croît suivant la profondeur donnée à l'algorithme, nous avons tracé les courbes des nombres de nœuds totaux par rapport à la profondeur donnée à MinMax.

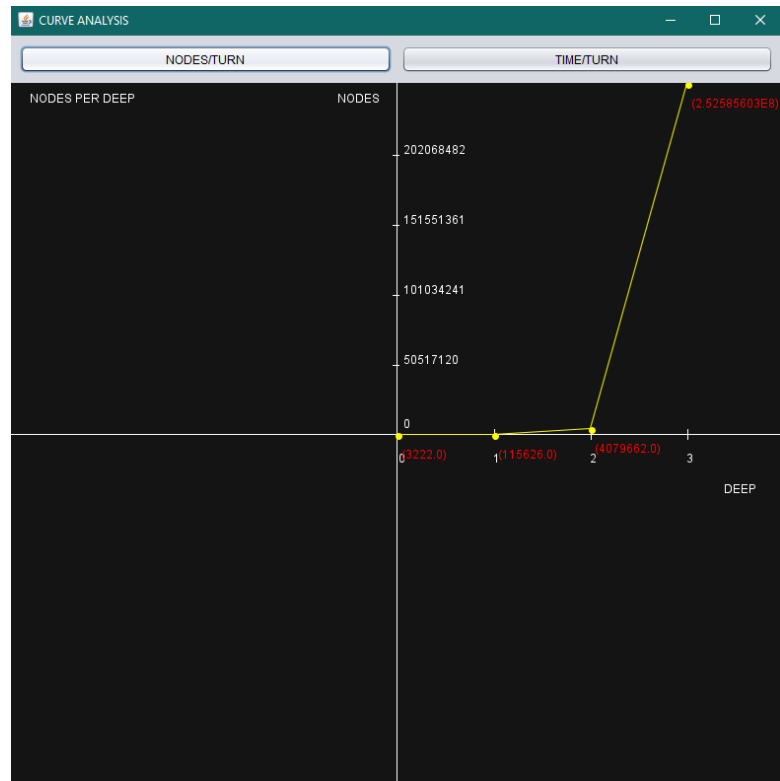
Profondeur 0



Profondeur 1



Profondeur 3



Nous observons ici des courbes exponentiels, ceci étant logique vu que le nombre de nœuds explorés est égale à : $\text{CoupsValidesAvecTour}^{\text{profondeur}}$

3.2 Taux de victoire par rapport a la profondeur et au nombre de coups d'avance

Les captures suivantes sont des analyses de plusieurs parties suivant la profondeur et le nombre de coups d'avance. Ces parties ont toutes été jouées sur des grille de taille (11,11), CD veut dire "Coups d'avances" dans ces tableaux.

MinMax 1 vs MinMax 0

CD	PTSJOUEUR1	PTSJOUEUR2	GAGNANT	JOUEUR J1	
0	27	21	W	% W (0 A 5 CD)	% W (5 A 10 CD)
0	25	19	W	80	5
0	29	19	W		
0	27	19	W	% W (0 a 2 CD)	% W (3 a 10 CD)
0	25	23	W	100	25
1	31	18	W		
1	27	22	W		
1	25	23	W		
2	25	24	W		
2	25	22	W		
2	22	14	W		
3	25	23	W		
3	21	28	L		
3	26	23	W		
4	24	23	W		
4	25	23	W		
4	25	23	W		
5	19	25	L		
5	24	24	D		
5	22	26	L		
6	23	23	D		
6	25	23	W		
6	22	27	L		
7	15	28	L		
7	23	25	L		
7	23	26	L		
8	22	23	L		
8	23	23	D		
8	23	23	D		
9	20	29	L		
9	24	27	L		
9	27	28	L		
10	28	31	L		
10	22	25	L		
10	24	27	L		

MinMax 2 vs MinMax 0

CD	PTSJOUEUR1	PTSJOUEUR2	GAGNANT	JOUEUR J1	
0	29	20	W	% W (0 A 5 CD)	% W (5 A 10 CD)
0	27	24	W	65	5
0	28	27	W		
0	31	28	W	% W (0 a 2 CD)	% W (3 a 10 CD)
0	25	22	W	90,90909091	16,66666667
1	27	24	W		
1	19	19	D		
1	25	23	W		
2	22	19	W		
2	25	22	W		
2	23	21	W		
3	23	22	W		
3	22	22	L		
3	24	19	W		
4	25	18	W		
4	21	27	L		
4	23	26	L		
5	22	24	L		
5	22	22	D		
5	17	28	L		
6	21	27	L		
6	20	28	L		
6	27	22	W		
7	23	23	D		
7	24	25	L		
7	24	25	L		
8	23	26	L		
8	20	29	L		
8	24	27	L		
9	27	28	L		
9	28	31	L		
9	22	26	L		
10	24	25	L		
10	22	22	D		
10	21	21	D		

MinMax 3 vs MinMax 0

CD	PTSJOUEUR1	PTSJOUEUR2	GAGNANT	JOUEUR J1	
0	26	23	W	% W (0 A 5 CD)	% W (5 A 10 CD)
0	27	22	W	60	20
0	29	16	W		
0	22	22	D	% W (0 a 2 CD)	% W (3 a 10 CD)
0	27	21	W	90,90909091	25
1	25	23	W		
1	26	23	W		
1	24	16	W		
2	25	24	W		
2	26	21	W		
2	28	9	W		
3	24	25	L		
3	24	25	L		
3	23	26	L		
4	26	19	W		
4	22	26	L		
4	24	25	L		
5	23	26	L		
5	24	24	D		
5	26	22	W		
6	26	23	W		
6	21	27	L		
6	23	26	L		
7	22	24	L		
7	22	22	D		
7	17	28	L		
8	21	27	L		
8	20	28	L		
8	27	22	W		
9	25	10	W		
9	21	28	L		
9	24	23	W		
10	24	28	L		
10	24	25	L		
10	22	27	L		

D'après ces tableaux on peut voir que lorsque l'on donne moins de 3 coups d'avances au joueur MinMax0 MinMax1 gagne toujours. Alors que pour plus de 3 coups d'avances, MinMax0 commencent à battre MinMax1. Cela fonctionne pareil pour MinMax2 et MinMax3. Avec des analyses plus poussées, nous aurions pu faire beaucoup d'autre tableur comme ceux-ci, mais en changeant la taille de la grille. Grâce a cela nous aurions peu être pu mettre en évidence une corrélation entre la taille de la grille et le nombre de coups d'avance.

3.3 analyse de temps

Nous avons récupéré des données sur les temps maximums de décisions de MinMax suivant la profondeur donnée. Premièrement nous avons affiché a chaque fin de partie le temps maximum de MinMax pour choisir un coup, le temps total de la partie ainsi que des informations sur le nombre de nœuds parcourus.

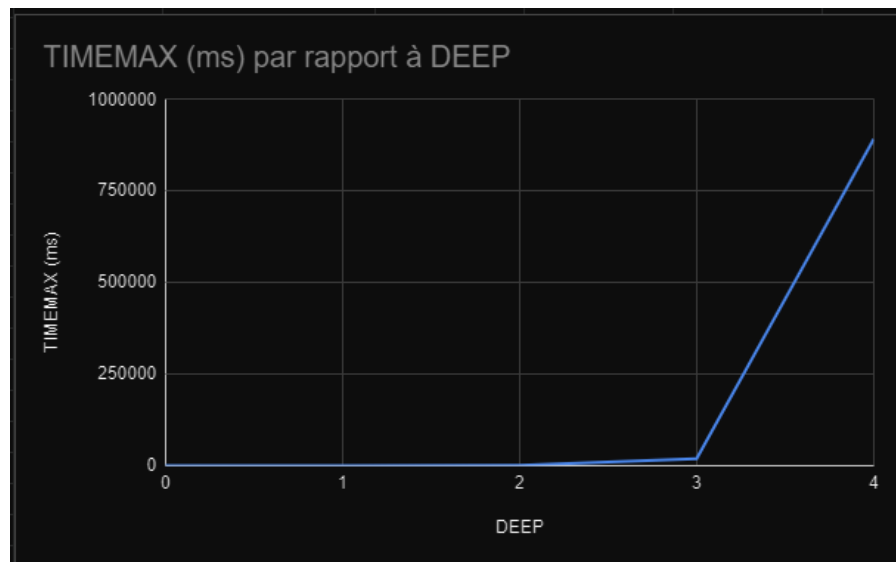
```
----> SCORES :
----> Score final Joueur 1 (minmax, 3) : 62
----> Score final Joueur 2 (minmax, 3) : 59
----> Grand gagnant : JOUEUR 1

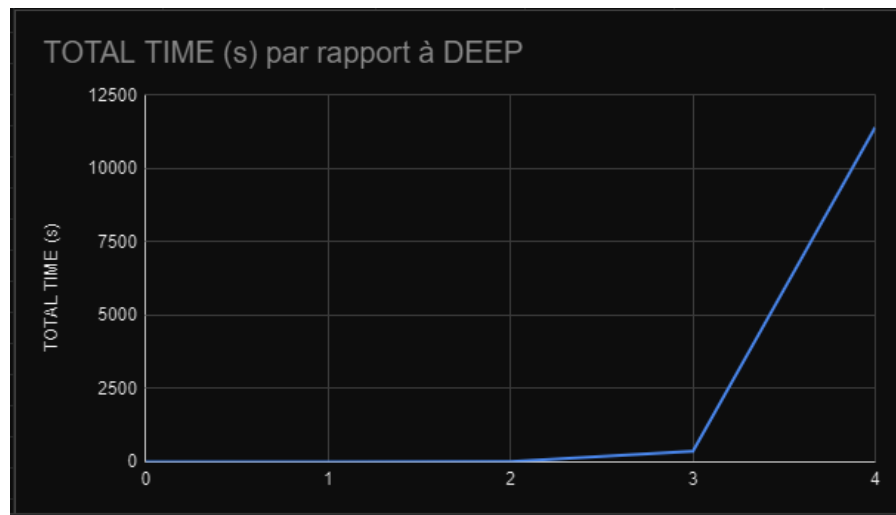
----> MINMAX :
----> Nombre de noeuds parcouru par le joueur 1 (profondeur 3) : 619673282
----> Nombre de noeuds parcouru par le joueur 2 (profondeur 3) : 613636900

----> DUREES :
----> Duree de la partie : 2168.7451171875 secondes (36.145751953125 minutes)
----> Duree maximum de fonction minimax (profondeur 3) : 45.619998931884766 secondes (45619.998931884766 ms)
----> Duree maximum de fonction minimax (profondeur 3) : 48.900001525878906 secondes (48900.001525878906 ms)
```

Puis nous avons tracé des courbes après avoir récupéré ces données avec plusieurs profondeurs données :

GRILLE (15,15)	ALPHABETA : OFF	
DEEP	TIMEMAX (ms)	TOTAL TIME (s)
0	1	0,25
1	17	0,5
2	686	14
3	19242	360
4	891273	11400





Ces courbes ressemblent fortement aux courbes du 1.1.2, la rapidité du MinMax en temps étant liée au nombre de nœuds parcourus.

Chapitre 4

Bilan

4.1 Problèmes rencontrés

La plupart des problèmes étaient liés au MinMax et surtout à la façon de gérer plusieurs objet Board. En effet, pendant le fonction MinMax, énormément de coups sont joués sur des Boards différentes. Nous avons donc finit par réussir en comprenant mieux comment fonctionne les objets.

```
//ON JOUE LES COUPS VALIDES 1 PAR 1 SUR UNE COPIE A CHAQUE ITERATION
virtualBoardN = virtualBoard.getCopy();
virtualBoardN.play(valideMoves.get(i), (player ? 1 : 2));
```

Alphabeta ne fut pas implémenté probablement à cause de problème de référencement et de la classe AlphaBeta. Le problème est peut être aussi le même que lors des premières tentatives d'implémentation de MinMax. MinMax était incompatible avec notre jeu, il est possible qu'ici AlphaBeta soit incompatible avec notre version actuelle de MinMax.

```
//ENTREE : ab, OBJET ALPHABETA, toEval LA VALEUR A EVALUER, player LE JOUEUR ACTUEL
//RETURN : true ou false SUIVANT SI BETA EST INFERIEUR OU PAS A ALPHA
public boolean alphabetaCompare(AlphaBeta ab, int toEval, boolean player, boolean alphabetaUse, Point valideMove) {
    if(!alphabetaUse) {
        return false;
    }
    if(player) {
        ab.setAlpha(Integer.max(ab.getAlpha(), toEval));
    }
    else {
        ab.setBeta(Integer.min(ab.getBeta(), toEval));
    }
    return ab.getBeta() <= ab.getAlpha();
}

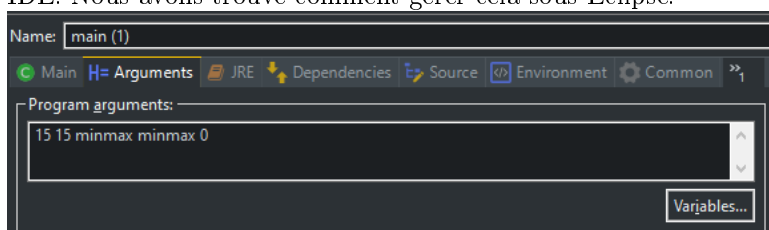
//ON BREAK SI alphabetaCompare EST true
if(alphabetaCompare(ab, current.getScore(), player, alphabetaUse, valideMoves.get(i))) {
    breakN++;
    break;
}
```

Nous n'avons pas réussi à tracer la courbe nodesperdeep4 car les méthodes pour dessiner la courbe n'acceptaient pas les BigInteger en paramètre, et le nombre de nœuds total lors d'une partie jouée par un MinMax4 est supérieur à 2^{15} . Nous étions donc obligé de récupérer ce nombre dans une variable BigInteger.

```
//ON ITERE SUR LES COUPS VALIDES DU JOUEUR1/JOUEUR2
for(int i = 0; i < lenght; i++) {
    //COMPTAGE NOMBRE DE NOEUDS PARCOURUS JOUEUR1 ET JOUEUR2
    if(player) {
        if(playerInitial) {nbNoeudsJoueur1 = nbNoeudsJoueur1.add(BigInteger.ONE);}
        else {nbNoeudsJoueur2 = nbNoeudsJoueur2.add(BigInteger.ONE);}
    }
    else {
        if(playerInitial) {nbNoeudsJoueur1 = nbNoeudsJoueur1.add(BigInteger.ONE);}
        else {nbNoeudsJoueur2 = nbNoeudsJoueur2.add(BigInteger.ONE);}
    }
}

graphics.drawString("(" + tmpStr + ")", x+5, y.add(BigInteger.valueOf(25)));
graphics.setColor(Color.yellow);
graphics.fillOval(x-1, y.subtract(BigInteger.ONE), 7, 7);
if(lx != 0) {
    graphics.drawLine(prevx, prevy, x, y);
}
```

Le passage de paramètre à Main est différent quand on exécute le programme dans un terminal ou dans un IDE. Nous avons trouvé comment gérer cela sous Eclipse.



Enfin, il y avait une redondance du code dans le Main, l'objectif était donc de factoriser le code, pour résoudre ce problème nous avons créé une fonction `CurrentPlayer` afin de ne pas répéter les opérations du joueur 1 et du joueur 2

4.2 Conclusion

Ce projet nous a permis de découvrir plusieurs fonctionnalités du langage Java. Il nous a aussi permis de travailler sur un algorithme très intéressant, et de mieux comprendre la récursivité ainsi que les structures arborescentes.