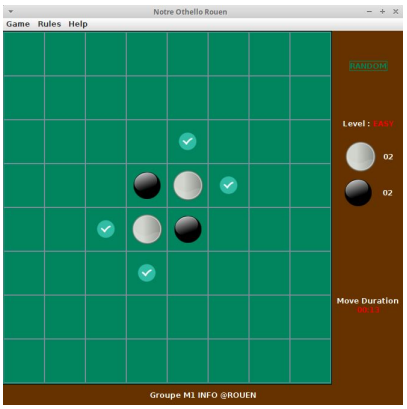
Master 1 Génie de l'Informatique Logicielle

Théorie des jeux – Mémoire
2019 - 2020

Thème:

Développer un jeux OTHELLO



Réalisé par:
Yaser GOUMIDI
Nadjah KARA

Sommaire

1. Sujet

Notre travaille consiste en l'implantation de différentes stratégies algorithmiques en Java sur PC. Le jeu proposé comme modèle est le jeu Othello (ou Reversi). La programmation consistera à implémenter des deux algorithmes classiques de stratégies des jeux : Min-Max et Aplha-Beta.
Pour être efficace, l'utilisation de ces algorithmes devra être basée sur une analyse correcte de jeu.

2. Présentation

OTHELLO est un jeu à deux joueurs, qui s'affrontent sur un plateau de soixante quatre cases. Un joueur possède les pions blancs, l'autre les pions noirs. Le gagnant est celui qui, à la fin de la partie, détient le plus grand nombre de pions. Les jeux de plateau à deux joueurs sont d'une manière générale un terrain propice pour la mise en œuvre des techniques de l'Intelligence Artificielle. Parmi Tous les jeux de ce type, quelques uns se distinguent particulièrement et sont devenus des "classiques", pour le développement, la recherche et la mise en application des techniques de l'Intelligence Artificielle. Ce sont les échecs, les dames et notamment OTHELLO , très étudié pour ses spécificités que nous détaillerons plus loin : nombre de coups fini, pas de cycle dans les positions successives du plateau de jeu, etc.

3. Organisation du code

Nous avons utilisé le langage JAVA et l'IDE Eclips pour réaliser ce projet. afin d'organiser notre code convenablement, nous avons regroupé dans différents packages les différentes parties de notre code.

En effet, nous avons 7 principaux packages, othello.Controller qui contiendra le noyau fonctionnel du jeu et notre classe main, othello.IHM qui contiendra l'IHM, othello.Model contiendra tous les model de notre projet, othello.Strategy pour l'implémentation des algorithme ainsi que othello.Images, othello.Files et othello.Son qui regroupe toutes les images, les fichiers textes et les fichiers audio que nous avons les utilisés comme ressources dans notre projet.

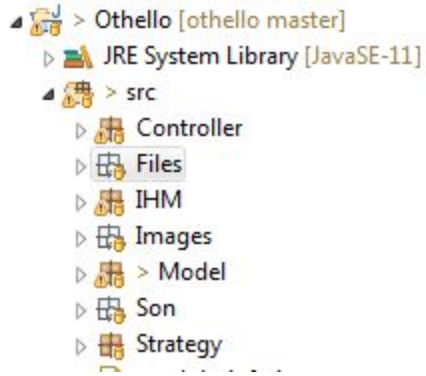


Figure 1: la Liste des packages du projet

4. Le jeu

4.1 Le plateau de jeu

Le plateau de jeu est constitué de soixante quatre cases, sans aucune distinction de couleur (la couleur des cases est généralement verte). Les pions sont des disques dont l'une des faces est blanche, l'autre noire. Un joueur représente le camp blanc, l'autre les pions noirs. Le joueur Noir posera ses pions face noire vers le haut, et vice versa pour les Blancs. La position de départ est montrée sur la figure 1.

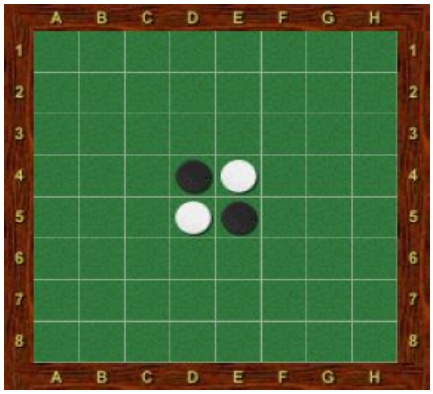


Figure 1

4.2. Les règles du jeu

Ce sont les Noirs qui jouent en premier. Ensuite, chaque joueur joue l'un après l'autre. La partie se termine si toutes les cases du plateau de jeu ont été remplies, ou bien si aucun des deux joueurs ne peut jouer. On compte alors le nombre de pions de chacun et l'on détermine le vainqueur.

- **Poser un pion**

Pour pouvoir poser un pion sur une case, deux conditions doivent être remplies :
1. On ne peut poser un pion que sur une case vide.
2. On ne peut poser un pion sur une case que si cela permet au joueur qui pose ce pion de "retourner" des pions adverses.

Si aucune de ces deux conditions n'est remplie, le joueur doit passer son tour. Si le joueur ne peut poser un pion que sur une seule case et que cela le mette dans une mauvaise posture, il doit jouer quand même ce coup. Il y a obligation de jouer. Il n'est pas permis de passer sans y être obligé.

- **Retourner des pions adverses**

Si le fait de poser un pion sur le plateau permet d'encadrer dans au moins une des huit directions possibles (figure 2), une suite continue de pions adverses, ceux-ci sont retournés, changent donc de couleur, et vous appartiennent maintenant.

Ainsi, dans la position de départ, les Noirs ne peuvent jouer qu'aux seuls endroits marqués d'une croix sur la figure 3. Si par exemple Noir décide de jouer en C4 (figure 4), le pion blanc en D4 est retourné et la position finale devient comme montré sur la figure 5 ; ce sera aux Blancs de jouer ensuite.

Un dernier exemple pour bien visualiser le retournement de pions : dans la position de la figure 6, supposons que c'est aux Noirs de jouer. S'ils décident de jouer en E3, les pions blancs marqués d'une croix sur la figure 7 vont être retournés. On obtiendra la position de la figure 8.

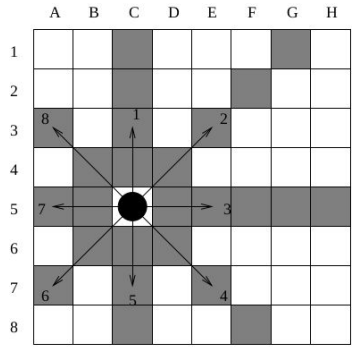


Figure 2

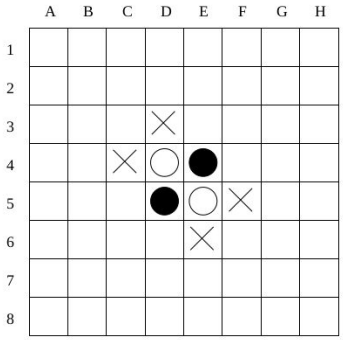


Figure 3

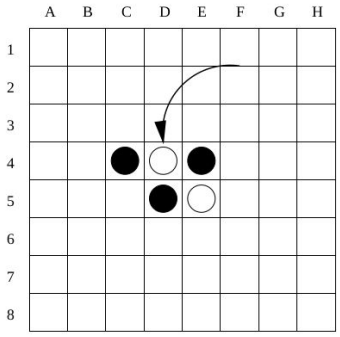


Figure 4

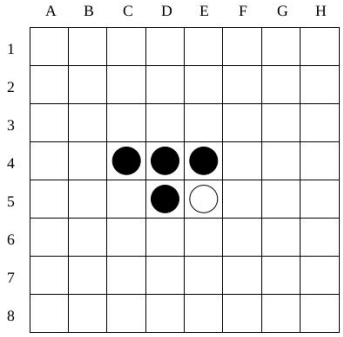


Figure 5

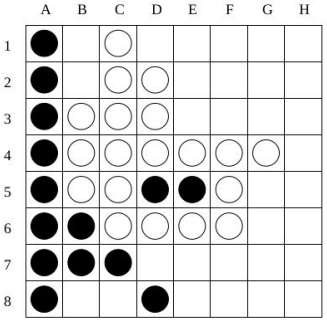


Figure 6

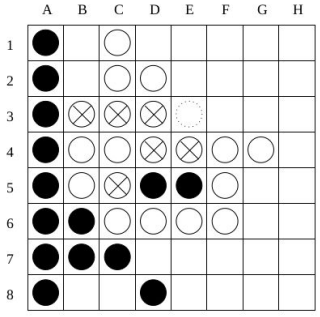


Figure 7

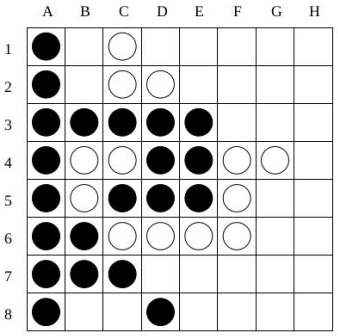


Figure 8

5. Stratégies de base

- **La maximisation de pions**

La première impression qui viendra à l'esprit d'un joueur débutant est de jouer de tel sorte à avoir le maximum de pions.

Malgré que le but d'Othello est d'avoir le maximum de ses propres pions colorés sur le plateau à la fin de la partie, néanmoins cette stratégie limite la mobilité d'un joueur et ne lui assure pas que ces pions ne seront pas retournés par l'adversaire avant la fin de la partie.

- **La strategie positionnelle**

On appelle un pion stable, le pion qui ne pourra pas être retourné par l'adversaire, il est donc définitif.

Le jeu possède différent types de positionnement stratégique sur le plateau de jeu qui assure d'avoir un maximum de pions stables

  - ***les coins***

un pion placé dans un coin (figure 8) est donc l'exemple le plus simple de pion définitif. Les pions adjacents et de la même couleur deviennent aussi des pions définitifs. Tous les pions de la figure 9 sont donc définitifs et Noir est assuré de garder ces pions.

Figure 8

Figure 9

La maniere d'eviter de perdre un coin est donc de ne pas jouer sur les cases adjacentes au coin , que l'on appelle cases X pour celles qui sont sur les diagonales (figure 11) et cases C pour celles qui sont sur les bords (figure 10) . Ces cases sont susceptibles de donner, à terme, un coin à l'adversaire

Figure 10: cases C

Figure 11: cases X

❖ ***Bords***

Les cases dit "de bords" sont représenté sur la figure 12. Les pions positionnés sur ces cases sont moins susceptibles d'être capturés, ils peuvent seulement être capturé sur une seule direction contrairement aux disques positionné au centre du plateau.

Cela peut paraître comme une bonne stratégie , mais les pions positionnés sur les bords ne peuvent être stables que si d'autres disques de la même couleurs ont été positionnés sur les coins pour couvrir la faille.

.

Figure 12

- **La mobilite**

En général, plus vous placez de pions en frontière de la position, c'est-à-dire adjacents à des cases vides, plus votre adversaire a de coups et moins vous en avez. En début et en milieu de partie, il est donc préférable d'avoir peu de pions. Et surtout, il faut les placer au centre de la position et non en frontière. À Othello, l'encerclement est une mauvaise stratégie.

6. Fonction d'évaluation

Après avoir fait une analyse du jeu et les stratégies de base on arrive à conclure qu'on peut avoir trois critères d'évaluation :

– le matériel est mesurée par le nombre de pions d'une couleur donnée
– la mobilité par le nombre de cases jouables par cette couleur
– la force d'une position par la somme des valeurs des cases occupées par cette couleur.

Pour le calcul du critère de force, on va attribuer à chaque case une valeur tactique qui représente l'intérêt qu'on a à l'occuper. Une évaluation possible des cases est donnée dans la figure 13.

	A	B	C	D	E	F	G	H
1	500	-150	30	10	10	30	-150	500
2	-150	-250	0	0	0	0	-250	-150
3	30	0	1	2	2	1	0	30
4	10	0	2	16	16	2	0	10
5	10	0	2	16	16	2	0	10
6	30	0	1	2	2	1	0	30
7	-150	-250	0	0	0	0	-250	-150
8	500	-150	30	10	10	30	-150	500

Figure 13: évaluation des cases

On peut justifier ce tableau a partir des stratégies de base vues précédemment :

– Un pion placé dans un coin est imprenable et constitue donc une solide base de départ pour la conquête des bords.
– Les cases bordant le coin sont à éviter car elles donnent à l'adversaire la possibilité de prendre lecoin.
– Les cases centrales augmente les possibilités de jeu.
– Les cases du bords sont également des points d'appui solides.

7. Stratégies algorithmiques

Dans le cadre de notre projet, le travail de programmation intervient après celui de la conception, Cette dernière concerne la création des diagrammes qui nous faciliteront la programmation de l'application.

La figure suivante nous montre le diagramme de classe pour le package de stratégy.

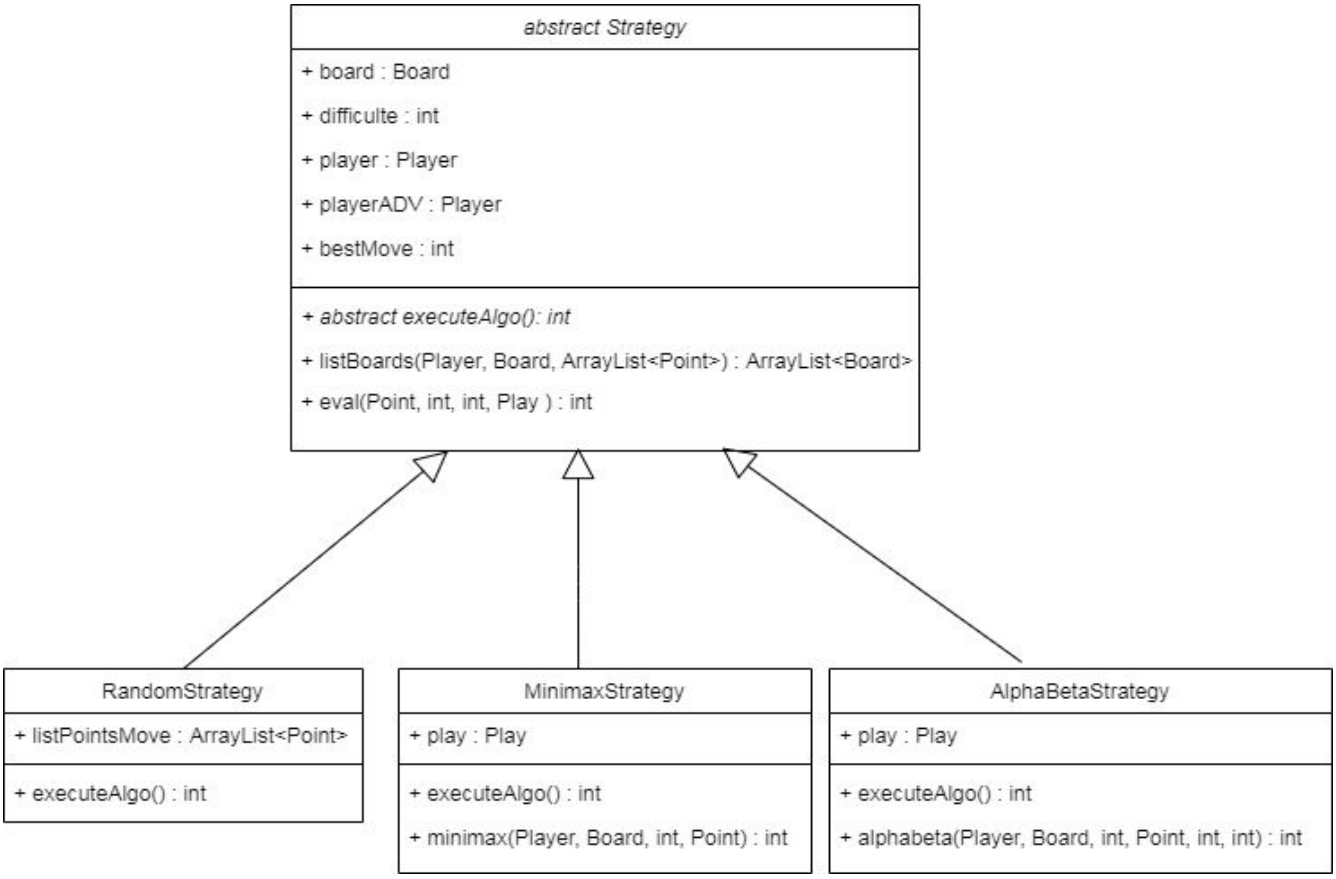


Figure 14: diagramme de classe pour la stratégie

7.1 La Stratégie Minimax

Le principe de cet algorithme c'est que 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.

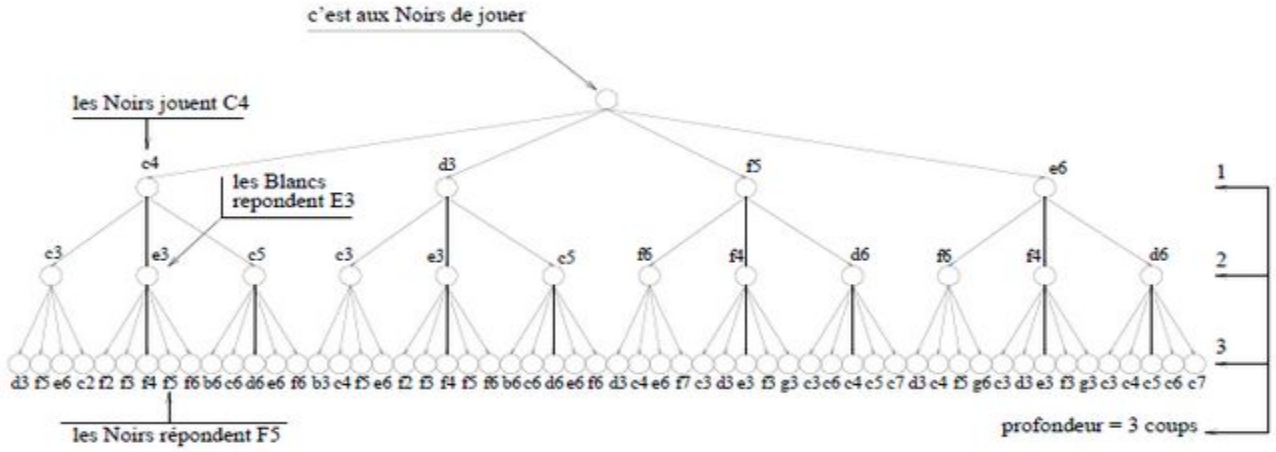


Figure 15: Arbre de recherche pour les trois premier coups

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 la fonction d'évaluation définie ci-dessus. Le programme peut, à partir de la racine, jouer le coup de niveau 1 qui lui garantit le gain maximal contre toute défense de son adversaire, en supposant que celui-ci utilise également une stratégie optimale, c'est-à-dire qu'il joue lui-même à chaque coup le gain qui lui garantit le gain maximal contre toute défense

7.2 Implementation

Notre objectif est de trouver le meilleur coup pour l'ordinateur. Pour ce faire, l'ordinateur prend simplement le nœud avec la meilleur d'évaluation (fonction d'évaluation). donc pour rendre le processus plus intelligent, on peut juste regarder de l'avant et évaluer les mouvements de l'adversaire potentiel. L'algorithme suppose que l'adversaire joue de manière optimale.

Ici, nous définissons formellement les étapes de l'algorithme:

1. Construire l'arbre

En fait, l'algorithme MinMax explore l'arbre de jeu jusqu'à une certaine profondeur donnée, Il fait remonter aux noeuds la meilleur évaluation.
Pour limiter les calculs, on définit une profondeur limite au calcul (ou horizon). Les nœuds ne sont plus développés à partir de cette profondeur, on calcule alors leur valeur heuristique, chaque feuille de l'arbre est donc soit un état final, soit un état non final mais à une profondeur limite.

Pour définir la profondeur du jeu, nous avons implémenté la méthode *getDifficulte* :

```
// methode permet recuperer le profondeur à
// utilisé selon la difficulté du partie
public int getDifficulte(String difficult) {

    if(difficult == "EASY") {
        return 1;
    }else if(difficult == "MUDIEM"){
        return 2;
    }else if(difficult == "HARD"){
        return 4;
    }
    return 0;
}
```

Figure 16: méthode *getDifficulte*

la méthode prend comme argument un stringe et retourner un profondeur pour chaque niveau du jeu.

2. Évaluer des noeud à l'aide de la fonction d'évaluation

La fonction eval() nous permet d'évaluer chaque noeud selon les trois critères indiquée dans la section (2.3).

```
// méthode permet d'évaluer chaque position
public int eval(Point powerPoint, int mobilite, int materiel, Play play) {
    int [][] evalPowerPoint;

    // table d'evaluation selon le critères de la force de position
    evalPowerPoint = new int[][]
            {
                { 500, -150, 30, 10, 10, 30, -150,  500},

                {-150, -250,  0,  0,  0,  0, -250, -150},

                { 30,     0,  1,  2,  2,  1,    0,   30},

                { 10,     0,  2, 16, 16,  2,    0,   10},

                { 10,     0,  2, 16, 16,  2,    0,   10},

                { 30,     0,  1,  2,  2,  1,    0,   30},

                {-150, -250,  0,  0,  0,  1, -250, -150},

                { 500, -150, 30, 10, 10, 30, -150,  500},
            };

            // if nbrcoup <= 30 mobilité et position
            // if nbrcoup > 30 et <= 55  mobilité et position et material
            // if nbrcoup > 55 material
            if(play.getInbrCoup() <= 30)
                return evalPowerPoint[powerPoint.x][powerPoint.y] + mobilite;
            else if ((play.getInbrCoup() > 30) && (play.getInbrCoup() <= 56))
                return evalPowerPoint[powerPoint.x][powerPoint.y] + mobilite + materiel;
            else return materiel;
}
```

Figure 17: méthode *eval*

Maintenant, nous implémentons l'algorithme. Il faut un arbre de jeu pour regarder en avant et trouver le meilleur coup. (bestMove).

- la méthode *listBoards* simule une partie pour un profondeur fixe et retourne la liste des boards possible pour chaque mouvement possible.
- la méthode *listPointsMovePlayer* nous permet de récupérer la liste des mouvements possibles pour un joueur et un board donné.

La figure suivante nous explique les fonctionnement de la méthode listBoards

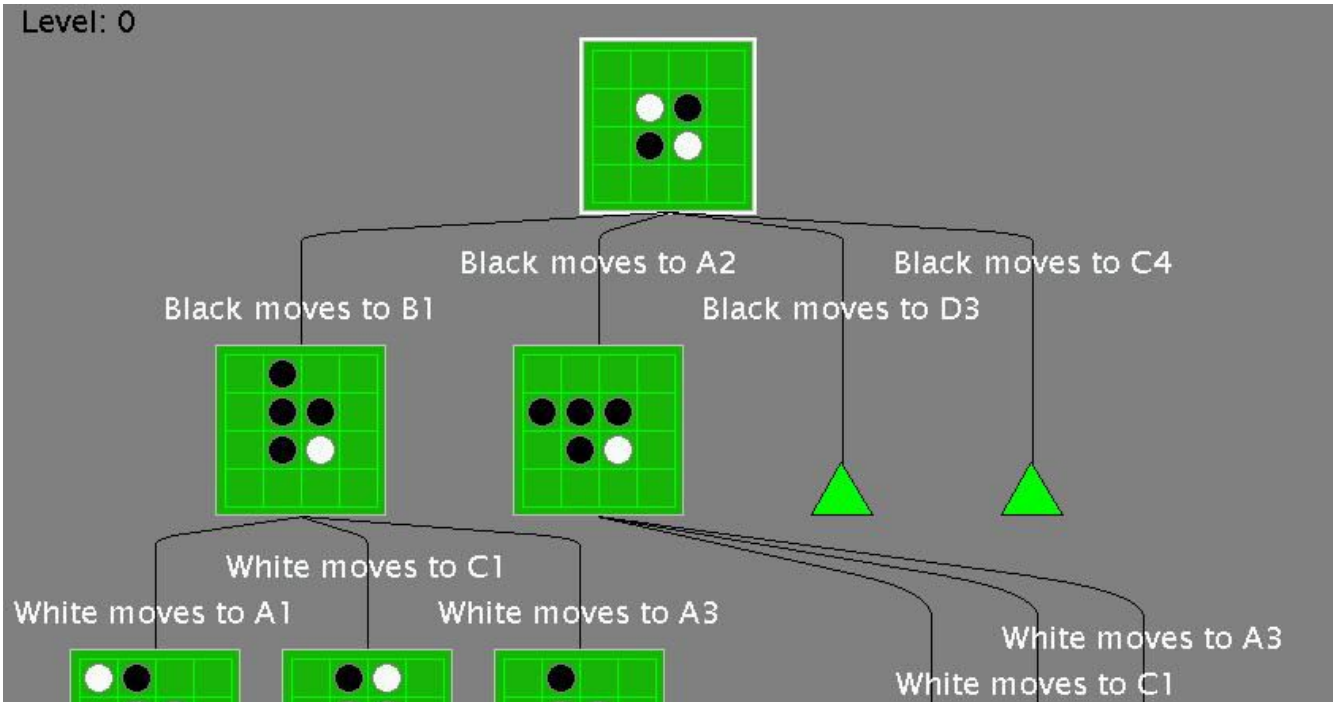


Figure 18: explication sur le fonctionnement de la fonction *listBoards*

Notre implémentation associe un valeur à la racine de l'arbre, selon laquelle le joueur MAX fera le choix de son coup.

l'algorithme visite l'arbre de jeu pour chaque niveau, en montant les feuille jusqu'à la racine

- si le nœud représente une position terminale, nous lui associons la valeur de notre fonction d`évaluation.
- s'il est dans une position intermédiaire et d'un nœud MIN, nous lui associons la plus petite des valeurs associées à ses fils.
- s'il est dans d'une position intermédiaire et d'un nœud MAX, nous lui associons la valeur maximale qui a été associée à ses fils.

une fois l'algorithme peut associer une valeur à la racine *(valeur MiniMax)*, nous faisons le choix de coup qui mène vers le fils qui a cette valeur maximale.

```java
// int fonction minimax (int depth)
public int minimax(Player p, Board board, int depth, Point point) {

    // if (game over or depth = 0)
    if (board.endGame() || (depth == this.difficulte)) {

        //récuperer la mobilité : nombre de cases jouables
        ArrayList<Point> listPointsMove = p.PlayerMove.listPointsMovePlayer(p, board);
        //nombre de pions (matériel).
        int nbPawn = p.getScore();
        //  return winning score or eval();
        return eval(point, listPointsMove.size(), nbPawn, play);
    }

    // int bestScore;
    // move bestMove;
    int bestScore;

    // récuperer la liste des boards
    ArrayList<Point> listPointsMove = p.PlayerMove.listPointsMovePlayer(p, board);
    ArrayList<Board> diffBoardsWithPoints = listBoards(p, board, listPointsMove);

    // if (nœud == MAX) { //=Programme
    if (p.getPawn() == this.player.getPawn()) {
        // bestScore = -INFINITY;
        bestScore = Integer.MIN_VALUE;

        // for (each possible move m) {
        for (int i = 0; i < diffBoardsWithPoints.size(); i++) {

            if(depth == 0) this.bestMovedepth0 = i;
            // make move m;
            Board succBoard = diffBoardsWithPoints.get(i);
            // int score = minimax (depth - 1)
            int score = minimax(this.playerADV, succBoard, depth + 1, listPointsMove.get(i));
            // unmake move m;
            // if (score > bestScore)
            if (score > bestScore) {
                // bestScore = score;
                bestScore = score;
                // bestMove = m ;
                this.bestMove = this.bestMovedepth0;
            }
        }
    }
    // else { //type MIN = adversaire
    else {
        //  bestScore = +INFINITY;
        bestScore = Integer.MAX_VALUE;
        //  for (each possible move m) {
        for (int i = 0; i < diffBoardsWithPoints.size(); i++) {
            // make move m;
            if(depth == 0) this.bestMovedepth0 = i;
            Board succBoard = diffBoardsWithPoints.get(i);
            // int score = minimax (depth - 1)
            int score = minimax(this.player, succBoard, depth + 1,listPointsMove.get(i));

            // unmake move m;
            // if (score < bestScore) {
            if (score < bestScore) {
                // bestScore = score;
```

```
                bestScore = score;
                // bestMove = m ;
                this.bestMove = this.bestMovedepth0;
            }
        }
    }
    // return bestscore ;
    return bestScore;
}
```

Figure 19: l'algorithme minimax

7.3 La Stratégie Alpha-beta

Dans l'algorithme MinMax que nous avons implémenté ci-dessus; les informations ne circulent que dans un seul sens : des feuilles vers la racine. Il est ainsi nécessaire d'avoir développé chaque feuille de l'arbre de recherche pour pouvoir propager les informations sur les scores des feuilles vers la racine.

Le principe de l'algorithme Alpha-Beta, est précisément d'éviter la génération de feuilles et de parties de l'arbre qui sont inutiles.Pour ce faire, cet algorithme repose sur l'idée de la génération de l'arbre selon un processus dit en «profondeur d'abord» où, avant de développer un frère d'un nœud , les fils sont développés.
A cette idée vient se greffer la stratégie qui consiste à utiliser l'information en la remontant des feuilles et également en la redescendant vers d'autres feuilles.Plus précisément, le principe de l'algorithme AlphaBeta est de tenir à jour deux variables α et β qui contiennent respectivement à chaque moment du développement de l'arbre la valeur minimale que le joueur peut espérer obtenir pour le coup à jouer étant donné la position où il se trouve et la valeur maximale. Certains développements de l'arbre sont arrêtés car ils indiquent qu'un des joueurs a l'opportunité de faire des coups qui violent le fait que α est obligatoirement la note la plus basse que le joueur Max sait pouvoir obtenir ou que β est la valeur maximale que le joueur Min autorisera Max à obtenir . L'algorithme Alpha-Beta, donc, permet la suppression de grandes parties de l'arbre complet. Il en résulte un gain de temps substantiel pour le choix d'un coup à une profondeur donnée.

Dans l'exemple de l'image 20, les branches développées par l'algorithme Alpha-Beta sont en trait gras ; les autres parties de l'arbre sont celles développées par MinMax mais pas par AlphaBeta. Si le score d'un fils F d'un noeud Max est supérieur à la valeure courante d'un noeud Min, alors les frères de F n'auront pas besoin d'être explorés

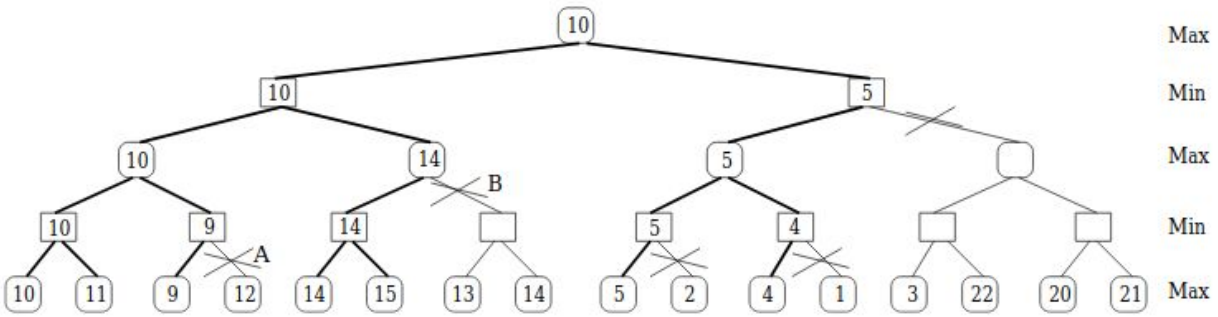


Figure 20:Exemple de déroulement de l'algorithme Alpha-beta

7.4 Implementation

Nous observons que l'algorithme minimax effectue une évaluation pour tous les nœuds de l'arbre de jeu d'un profondeur donné. Sachant qu'il n'est pas nécessaire de calculer les valeurs associées à tous les nœuds de l'arbre pour attribuer un valeur à la racine

L`algorithme d`élagage Alpha-Beta nous conduit à une économie de temps de calcul et de mémoire, en renonçant à l'évaluation des sous arbres dès que leur valeur devient inintéressante puisque nous pouvons souvent savoir à l'avance que certaines branches ne peuvent pas conduire à un noeud avec un valeur désiré.

Pour implémenter l'algorithme Alpha-Beta nous avons associer à chaque nœud de type MAX une valeur appelée *Alpha* égale à la :

- valeur de notre fonction d'évaluation pour la feuille.
- ou bien la meilleur valeur de ses fils trouvée.

Nous avons associer également, une valeur appelée *Beta* pour les nœuds de type MIN, cette valeur égale à la :

- valeur de notre fonction d'évaluation, pour un nœud terminal,
- ou à la valeur minimum de ses fils.

Notre implémentation de l'algorithme Alpha-Beta coupe les sous arbres selon les règles suivantes:

- on arrête la recherche d'un nœud n de type MAX si *Alpha(n) < Beta (Père(n))*
- on arrête la recherche d'un nœud m de type MIN si *Beta(m) > Alpha (Père(m))*

la récursivité de la fonction alphabeta() effectue une recherche en profondeur d'abord, en garantissant une évaluation des valeurs liées aux nœuds en remontant vers la racine.

initialement on a:

```
int alpha = Integer.MIN_VALUE;
int beta = Integer.MAX_VALUE;
```

```
//int alphabêta(int depth, int alpha, int bêta)
public int alphabeta(Player p, Board board, int depth, Point point, int alpha, int beta) {

    //   if (game over or depth <= 0)
    //   return winning score or eval();

    if (board.endGame() || (depth == this.difficulte)) {
        //récuperer la mobilité : nombre de cases jouables
        ArrayList<Point> listPointsMove = p.PlayerMove.listPointsMovePlayer(p, board);
        //nombre de pions (matériel).
        int nbPawn = p.getScore();
        return eval(point, listPointsMove.size(), nbPawn, play);
    }

    //  move bestMove;

    // récuperer la liste des boards
    ArrayList<Point> listPointsMove = p.PlayerMove.listPointsMovePlayer(p, board);
    ArrayList<Board> diffBoardsWithPoints = listBoards(p, board, listPointsMove);

    //   if(nœud == MAX) { //Programme
    if (p.getPawn() == this.player.getPawn()) {

        //  for (each possible move m) {
        for (int i = 0; i < diffBoardsWithPoints.size(); i++) {

            if(depth == 0) this.bestMovedepth0 = i;
            //  make move m;
            Board succBoard = diffBoardsWithPoints.get(i);
```

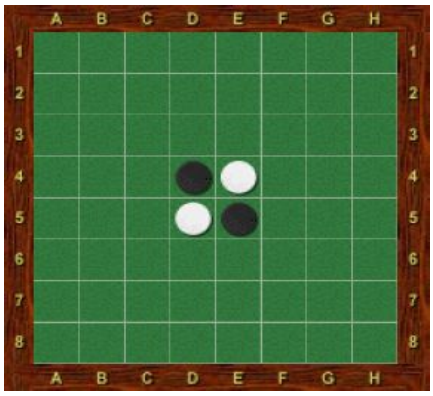
```
            //  int score = alphabêta(depth - 1, alpha, bêta)
            //  unmake move m;
            int score = alphabeta(this.playerADV, succBoard, depth + 1, listPointsMove.get(i),
             alpha, beta);

            //  if (score > alpha) {
            if (score > alpha) {
                // alpha = score;
                alpha = score;
                // bestMove = m ;
                this.bestMove = this.bestMovedepth0;

                if (alpha >= beta)
                    break;
            }
        }
        // return alpha ;
        return alpha ;
    }
    //type MIN = adversaire
    else {
        // for (each possible move m) {
        for (int i = 0; i < diffBoardsWithPoints.size(); i++) {

            if(depth == 0) this.bestMovedepth0 = i;
            // make move m;
            Board succBoard = diffBoardsWithPoints.get(i);

            // int score = alphabêta(depth - 1, alpha, bêta)
            int score = alphabeta(player, succBoard, depth + 1,listPointsMove.get(i), alpha,
             beta);
            //  unmake move m;
```

```
        // if (score < bêta) {
        if (score < beta) {
            //  bêta = score;
            beta = score;

            // bestMove = m ;
            this.bestMove = this.bestMovedepth0;

            if (alpha >= beta)
                break;
        }
    }
    return beta;
}
```

Figure 20: l'algorithme minimax

8. Exemples d'exécutions et performance

Nous allons illustrer les exemples d'exécutions avec la figure suivante :



On appelle l'ordinateur le joueur maximisant (joueur blanc) et le joueur réel le joueur minimisant (joueur noir). Ici, dans notre exemple le nombre du coup < 30 avec le niveau moyen donc:

- notre fonction d'évaluation prend la mobilité et la force de position.
- avec un profondeur égale à 2.

avec le minimax :

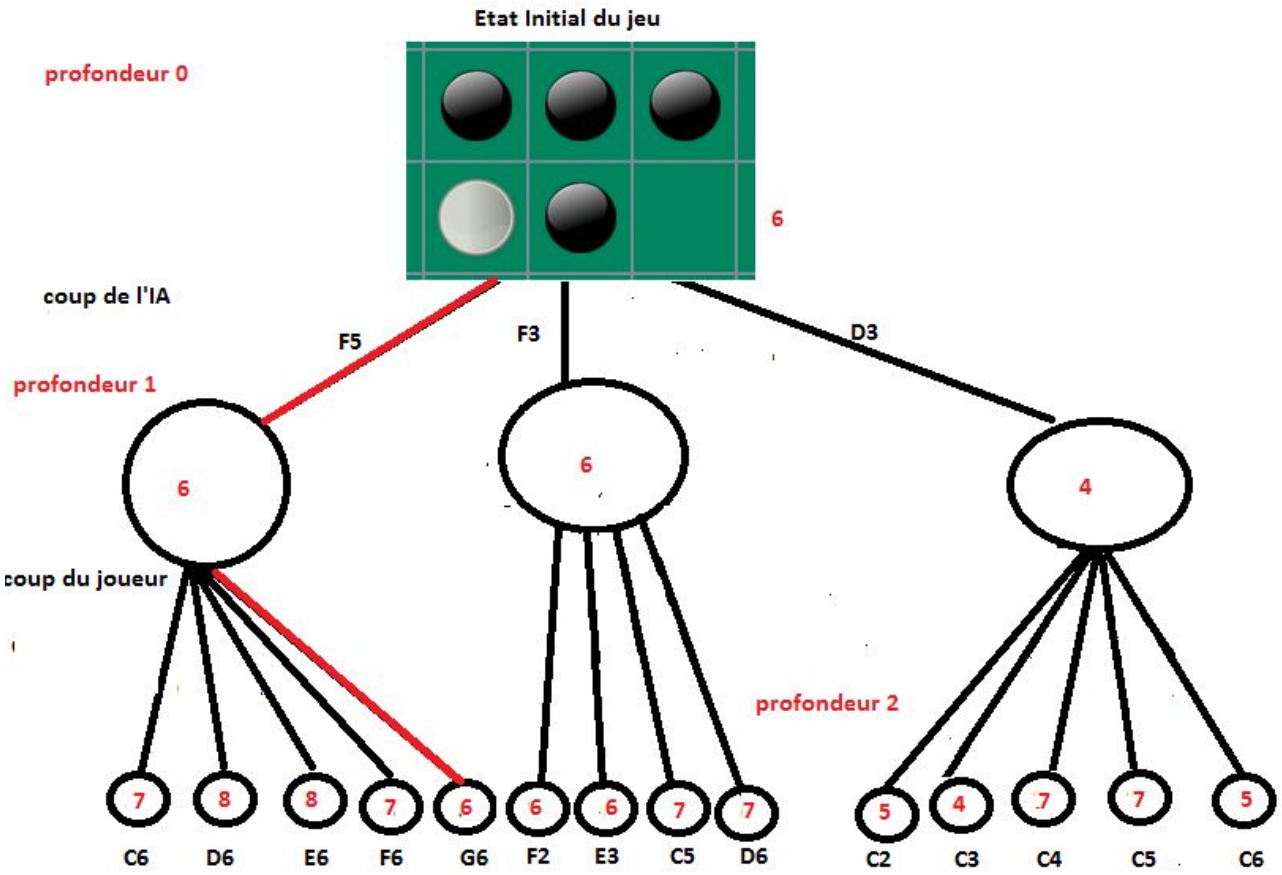


Figure 21: exemple d'exécution avec l'algorithme minimax

La figure nous montre que l'intérêt la plus haute que l'ordinateur (joueur max) peut avoir est 6, donc l'ordinateur choisira la case F5. le chemin par l'arbre de jeu est mis en évidence dans la Figure à l'aide d'une ligne rouge.

avec alphabeta :

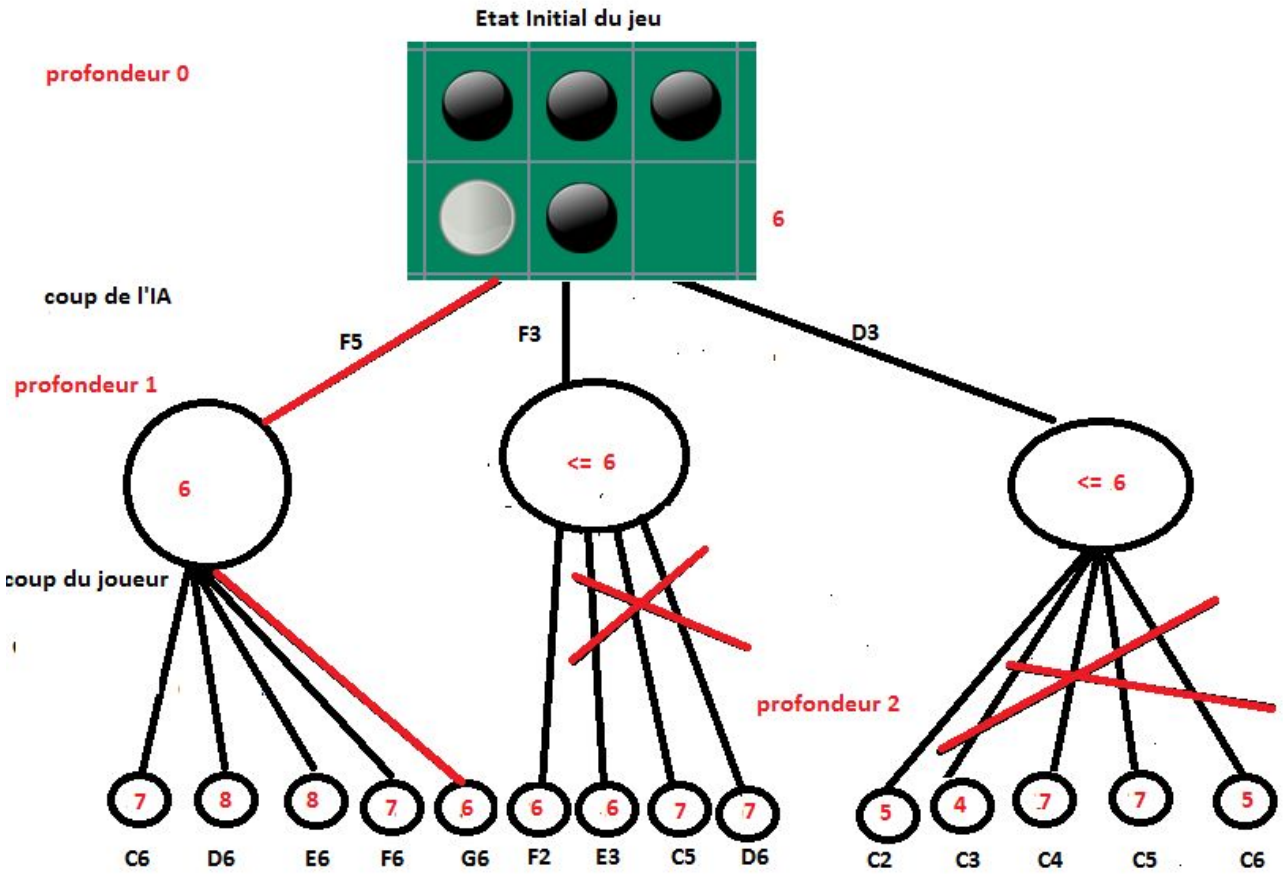


Figure 22: exemple d'exécution avec l'algorithme alphabeta

Avec l'algorithme alphabêta nous arrivons au même résultat que lorsque nous avons utilisé la recherche Minimax. Ce n'est pas une coïncidence, car l'élagage alpha-bêta fournit toujours une solution optimale puisque l'élagage de l'arbre de jeu n'affecte pas le résultat final.

Si l'algorithme MiniMax fonctionne bien sur des profondeur et coups simples, il est en revanche bien trop lent pour des profondeur et des coups plus complexes.

En effet, il effectue une exploration complète de l'arbre de recherche jusqu'à un niveau donné, alors qu'une exploration partielle de l'arbre est généralement suffisante : lors de l'exploration, il n'est pas nécessaire d'examiner les sous-arbres qui conduisent à des configurations dont la valeur ne contribuera sûrement pas au calcul du gain à la racine de l'arbre.

L'élagage Alpha/Bêta nous permet de réaliser ceci. Ainsi, il permet d'optimiser efficacement et de manière sûre l'algorithme MinMax.

9. Organisation :

Nous n'avons été que deux pour réaliser ce projet.
Au départ, nous n'étions pas au niveau, et nous nous sommes très vite rendu compte que nous allions devoir fournir une quantité importante de travail pour réaliser à bien ce projet, que nous ne voulions pas bâcler.

Pour commencer, nous nous sommes mis d'accords sur la conception du jeu, et la fonction d'évaluation, nous avons travaillé chaque jour sur le code du jeu.

Nous nous sommes envoyés l'avancée de notre travail chaque soir sur le dépôt GIT, et nous nous retrouvions deux fois par semaine en visioconférence (Zoom) pour avancer dessus ensemble.

Lorsque nous avons terminé le code du jeu,nous nous somme occupés de rédiger le guide d'utilisation, le rapport, la présentation et de faire tous les tests du jeu.
Enfin pour terminer, nous nous sommes retrouvés ensemble afin de créer un exécutable et finaliser le projet.

10. Problèmes rencontrés et les bugs référencés

Le principal problème que nous avons rencontré était le confinement, qui dit confinement ne dit pas vacances. Même enfermé à la maison à cause de l'épidémie de coronavirus, nous devrons travailler pour suivre notre programme et faire nos projets et nos TPs. Mais c'est vraiment, pas facile de s'y mettre. Et le manque de temps, car ayant eu à faire tous les projets en même temps, il était compliqué de se consacrer à 100% sur celui-ci.

La solution a été simple : nous avons énormément travaillé pour prendre le minimum de retard sur notre projet .

Après avoir testé l'application plusieurs fois, pas de bugs référencés.

11. Conclusion

Le projet nous permet d'implanter les algorithmes (minimax , alpha-bêta) vu en cours afin de concevoir une intelligence artificielle efficace et performante, nous avons également essayé de les tester avec différentes profondeurs d'arbre, il s'avère qu'avec l'élagage utilisé par l'alpha-bêta, nous obtenons des gains de performances significatif et visibles dans le jeu.

Ce projet nous a donné vraiment une occasion pour apprendre des nouveaux connaissances concerne le model MVC et l'utilisation des plusieurs notions comme Thread, Son, … etc intègrent dans le langage JAVA.

12. Réferences

http://turing.cs.pub.ro/auf2/html/chapters/chapter3/chapter_3_4_3.html
https://fr.qwe.wiki/wiki/Alpha%E2%80%93beta_pruning#Heuristic_improvements
https://fr.wikipedia.org/wiki/%C3%89lagage_alpha-b%C3%AAta
http://www.ffothello.org/othello/
http://www.ffothello.org/informatique/algorithmes/