8INF878 - Intelligence Artificielle

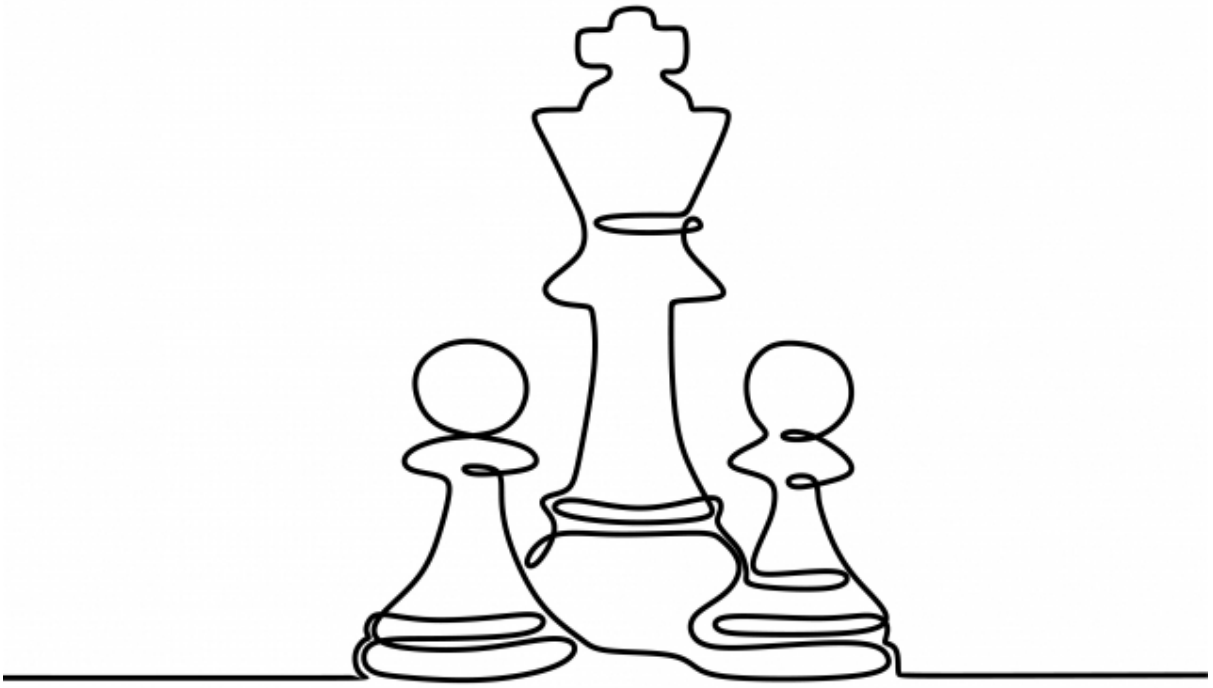
Automne 2022

Cours de Bianca Minetto Napoleão

UQAC

Rapport TP n°2

Création d'une IA d'échec



Aymerick SERGEANT - SERA14129900

Paulin GAILLOT - GAIP01040108

Alix DUFOUR - DUFA05610001

Sommaire

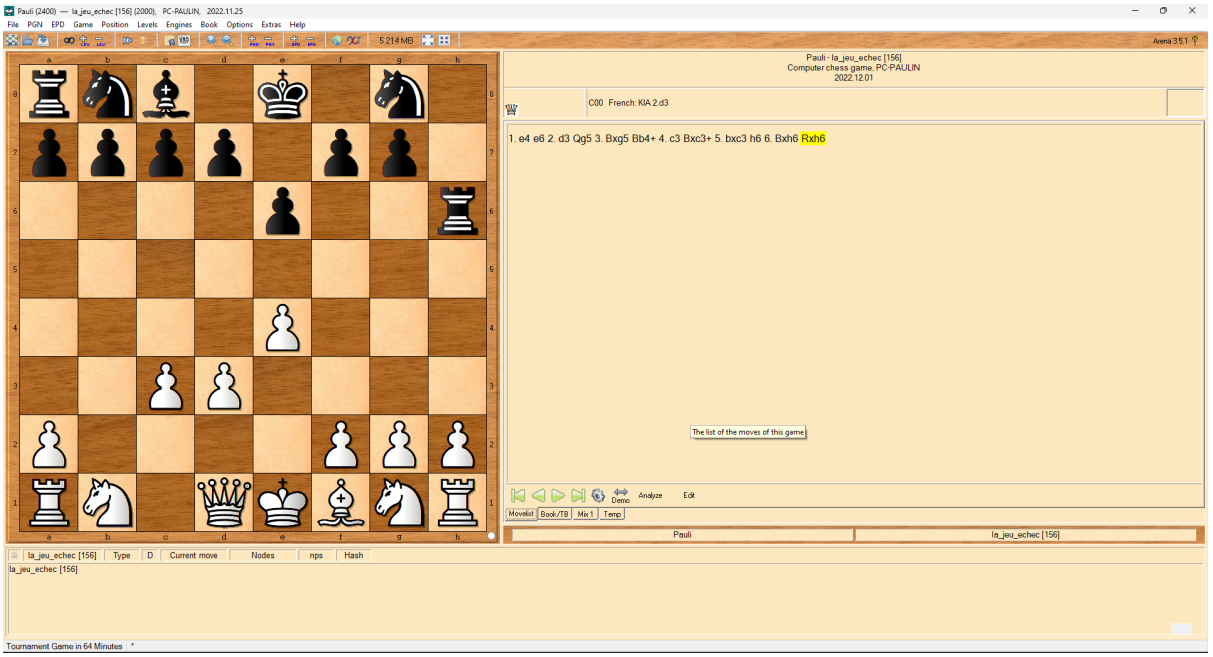
Introduction

Dans le cadre du cours d'Intelligence Artificielle, nous avons réalisé un agent capable de jouer aux échecs dans la plateforme ARENA. L'objectif principal étant de gagner la compétition d'échecs entre IA du cours.

Arena est une plateforme permettant de jouer aux échecs avec une interface graphique (GUI). Elle est gratuite, peut-être utilisée sur Linux, Windows et Mac.
Elle est utilisée car permet d'utiliser des moteurs d'échecs dessus.

Ce projet nous aura permis d'implémenter un minimax dans le cadre d'un projet concret afin de déterminer le meilleur mouvement à jouer lors de chaque tour.

Nous pouvons observer ci-dessous un exemple de comportement de notre IA face à un joueur sur Arena.



Modélisation du jeu

Avant de pouvoir implémenter les algorithmes de l'intelligence ou de le connecter à Arena, il fallait commencer par modéliser le jeu en lui-même, avec comme différentes contraintes les règles du jeu.

Tout d'abord la classe Echiquier permet de modéliser l'échiquier comme un tableau de 8x8 objet de classe Piece (si pas de pièce sur la case alors l'objet est null), elle permet d'appliquer différentes méthodes qui peuvent être utiles à l'IA pour récupérer différentes informations qui l'aideront à prendre une décision par exemple :

ArrayList<Piece> getPieces() : retourne la liste des pièces sur le plateau après avoir vérifié chaque case de l'échiquier.

ArrayList<Move> getPossibleMoves() : récupérer la liste des mouvements possibles : pour chaque pièce de la couleur de l'IA, elle va calculer tous les mouvements possibles de celle-ci :

```
for (Piece piece : pieces) {
    ArrayList<Coordonnees> coords = piece.listeDeplacementsValides();
```

Ensuite pour chaque mouvement possible il vérifier la nouvelle configuration du plateau dans une copie de l'échiquier :

```
for (Coordonnees coord : coords) {
    Echiquier eq2 = this.clone();
```

Ensuite on vérifie que ceci ne met pas le roi en position d'échecs, car cela rendrait le coup interdit :

```
if (!eq2.isEchec())
```

Si non, on vérifie finalement si il s'agit d'une promotion d'un pion qui le changerait alors en reine afin de mettre à jour l'information, et on ajoute le mouvement à la liste de mouvements possibles.

Une fois la boucle finie, on retourne la liste ainsi construite.

boolean isEchec() : retourne true si on est en position d'échecs : il s'agit tout d'abord de récupérer la pièce correspondant au roi et de lui appliquer sa méthode isEchec(). Celle-ci va alors chercher pour chaque pièce adverse si un de leur déplacement pourrait manger le roi (c'est à dire que le roi est en position d'échec).

Pour commencer nous vérifions le cas particulier de si la pièce adverse est le roi :

```
for (Piece piece : listePieces) {
   // true si la piece a un déplacement possible sur la case du Roi
   if(piece.getNom().equals("roi")) {
       if((piece.getX() == x+1 && piece.getY() == y+1) || (piece.getX() == x+1 && piece.getY() == y) ||
       (piece.getX() == x+1 && piece.getY() == y-1) || (piece.getX() == x && piece.getY() == y+1) ||
       (piece.getX() == x && piece.getY() == y-1) || (piece.getX() == x-1 && piece.getY() == y+1) ||
       (piece.getX() == x-1 && piece.getY() == y) || (piece.getX() == x-1 && piece.getY() == y-1) || (piece.getX() == x && piece.getY() == y)) {
           return true;
       }
```

Finalement nous testons pour les autres pièces :

```
else if ((!piece.getNom().equals("pion") && piece.deplacementValide(x, y)) ||
    (piece.getNom().equals("pion") && piece.deplacementValide2(x, y))) {
    // System.out.println("Mis en echec par : " + piece.getNom());
    return true;
}
```

Sinon aucune de toutes ces conditions a été remplie, elle renvoie finalement false.

boolean isEchecEtMat() : retourne true si on est en position d'échecs et mat

Pour chaque pièce de la couleur de l'IA, elle va tester si après chaque déplacement le roi est toujours en échec.

```
for (Piece piece : listePieces) {
    // recupere toutes les cases ou peut aller la piece
    ArrayList<Coordonnees> listeCasesPossibles = piece.listeDeplacementsValides();
```

Dès qu'elle trouve une possibilité où le roi n'est plus en échecs en retourne False:

```
if (!this.isEchec()) {
    // on revient à l'état de l'échiquier avant le déplacement
    piece.deplacement(x_depart, y_depart);
    this.echiquier.setCase(x_arrivee, y_arrivee, piece_arrivee);
    return false;
}
```

Sinon elle retournera True.

int evaluate() : permet de donner un score de la situation actuelle du plateau en fonction du nombre de pièces restantes dans chaque camp et de la position d'échec ou non, nous la verrons plus en détails dans la partie concernant l'intelligence artificielle.

La classe abstraite Piece permet de définir les attributs que chaque pièce possède sur le plateau (couleur, position…) et des méthodes communes.

Chaque pièce (Roi, Dame, Tour, Cavalier, Fou, Pion) a ensuite sa propre classe afin de définir notamment ses déplacements valides propres à sa classe, et éventuellement des attributs qui leurs sont propres comme l'information de si la pièce a déjà bougé ou non qui est utile uniquement pour le pion, le roi et la tour.

Communication avec Arena

Pour communiquer entre notre logiciel et Arena nous avons dû utiliser le protocole UCI (Universal Chess Interface). Il s'agit d'un protocole de communication libre de droits qui permet à un moteur d'échecs de communiquer avec une interface utilisateur, ici Arena. Elle utilise les entrées et sorties standards.
Dans le fichier App.java est stocké le code qui permet d'interagir avec Arena en utilisant l'UCI, nous allons ensuite voir les fonctions principales du programme :

- inputUCI

Après avoir reçu "uci", la liaison entre notre logiciel et Arena commence. Arena demande avec qui il communique. Notre logiciel va alors renvoyer des informations telles que sa version, son nom et le nom de l'équipe de développement.

- inputIsReady

La liaison entre Arena et le programme est terminé, Arena demande alors au programme si celui-ci est prêt en envoyant "isready". Le logiciel initialise alors le jeu de son côté puis répond "readyok" quand il a fini.

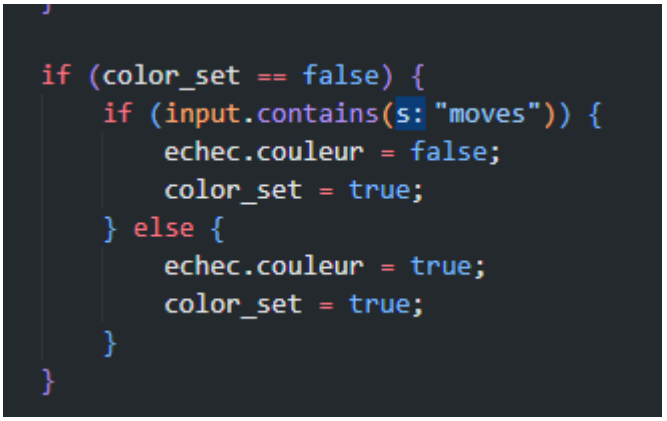
- inputUCINewGame

Arena commence une nouvelle partie. Notre logiciel initialise donc notre échiquier et les différentes pièces à leur état initial.

- inputPosition

Arena nous renvoie le mouvement qui vient d'être joué par l'adversaire.

Lors du premier tour si les mouvements sont vides alors notre programme comprend qu'il jouera les blancs. Si à l'inverse lors de notre premier tour il y a déjà eu un mouvement, notre programme jouera les noirs.



Notre programme va également mettre à jour l'échiquier lors de cette étape en bougeant la dernière pièce qui a été déplacée dans la liste de mouvements qu'Arena nous a renvoyé.

```
    public String convertPieceDebutUCI() {
        char p1 = (char) (97 + pieceDebut.getX());
        String p2 = Integer.toString(pieceDebut.getY() + 1);

        String result = p1 + p2;
        return result;
    }

    public String convertPieceFinUCI() {
        char p1 = (char) (97 + pieceFin.getX());
        String p2 = Integer.toString(pieceFin.getY() + 1);

        String result = p1 + p2;
        if (this.isPromotion == true) {
            result += "q";
        }
        return result;
    }
```

Le format des coordonnées n'étant pas similaire dans l'UCI (colonnes de A à H et rangées de 1 à 8 en partant du bas à gauche) par rapport à notre IA (colonnes et rangées toutes les deux de 0 à 7 en partant du bas à gauche), nous avons eu besoin de deux fonctions de conversion ainsi que d'une classe mouvement (fichier Move.java) pour récupérer les mouvements sous le bon format pour que l'IA puisse les traiter.

- inputGo

Lorsque Arena nous renvoie "go", notre logiciel va pouvoir commencer à rechercher le meilleur mouvement à exécuter. Arena va alors attendre que notre programme renvoie le mouvement choisi.

Cette étape est l'étape clé de la communication entre Arena et notre programme. Le logiciel va décider du meilleur mouvement grâce à l'intelligence artificielle comme nous verrons plus bas.

Une fois choisi, il faut alors le convertir sous le bon format de la même manière inversée que juste au-dessus pour récupérer les mouvements.

Finalement nous allons renvoyer la commande 'bestmove' suivie du mouvement à exécuter tel que décrit par le protocole UCI.

```
public static void inputGo() {

    Move move = null;

    echec.printEchiquier();
    System.out.println("info : moves possible " + echec.getPossibleMoves().size());

    move = Minimax.maxiFirst(depth: 3, echec, alpha: 5000, -5000);

    String UCI_start_move = move.convertPieceDebutUCI();
    String UCI_end_move = move.convertPieceFinUCI();
    String result = UCI_start_move + UCI_end_move;
    // System.out.println(move.pieceDebut.getX() + "|" + move.pieceDebut.getY());
    // System.out.println(move.pieceFin.getX() + "|" + move.pieceFin.getY());
    // echec.printEchiquier();
    echec.getPieceAt(move.pieceDebut.getX(), move.pieceDebut.getY()).deplacement(move.pieceFin.getX(),
            move.pieceFin.getY());
    System.out.println("bestmove " + result);
}
```

Développement de l'intelligence artificielle

Pour choisir le meilleur mouvement à effectuer nous avons eu besoin d'implémenter une Intelligence Artificielle. Ainsi nous avons donc décidé d'implémenter un Minimax afin de répondre à notre demande.

A- Minimax

Notre Algorithme de Minimax est composé de 3 fonctions. Une première fonction maxiFirst va être appelée : Cette fonction aura comme type de retour une classe Move indiquant les informations sur le meilleur move a effectuer.

```
static HashMap<Integer, Move> bestmove = new HashMap<>();

public static Move maxiFirst(int depth, Echiquier eq, int alpha, int beta) {
    Move bestmove = null;
    int max = -999999;
    for (Move mv : eq.getPossibleMoves()) {
        Echiquier eq2 = mv.eq.clone();
        int score = (int) mini(depth - 1, eq2, alpha, beta).get(index: 1);

        System.out.println(mv.pieceDebut.getX() + "/" + mv.pieceDebut.getY() + " -> " + mv.pieceFin.getX() + "/"
                + mv.pieceFin.getY() + "(score : " + score + ")");
        if (score > max) {
            bestmove = new Move(mv.pieceDebut, mv.pieceFin, eq2);
            max = score;
            if(score > alpha) {
                return mv;
            }
        }
    }
    return bestmove;
}
```

Cette fonction va récupérer la liste des Mouvements possible sur l'échiquier actuel, et va récupérer le Mouvement ayant le meilleur score.

Notre fonction va également appeler la fonction mini() en décrémentant la profondeur.

Notre fonction mini fonctionnera de manière similaire mais va récupérer la liste des mouvements que l'adversaire peut effectuer et va récupérer le mouvement ayant la pire incidence sur notre score.

```
public static ArrayList mini(int depth, Echiquier eq, int alpha, int beta) {
    if (depth == 0) {
        ArrayList array = new ArrayList<>();
        array.add(e: null);
        array.add(eq.evaluate());
        return array;
    }
    ArrayList min = new ArrayList<>();
    min.add(e: null);
    min.add(e: 999999);
    for (Move mv : eq.getPossibleMovesEnnemi()) {
        Echiquier eq2 = mv.eq.clone();
        int score = (int) maxi(depth - 1, eq2, alpha, beta).get(index: 1);
        if (score < (int) min.get(index: 1))
            min.set(index: 0, mv);
            min.set(index: 1, score);
            if(score < beta) {
                return min;
            }
    }
    return min;
}
```

Par la suite nous allons enchaîner entre les fonctions mini() et maxi() jusqu'à ce que notre variable *depth* atteigne 0.

Une fois cette variable à 0 nous allons appeler la fonction evaluate de notre échiquier afin d'évaluer notre jeu sur le plateau obtenu après notre enchainement d'actions.

Ainsi notre fonction minimax va alterner entre récupérer la meilleur ou la pire valeur pour récupérer le mouvement ayant la meilleur incidence sur son jeu.

Cependant cette fonction peut encore être améliorée, notamment en utilisant l'élagage Alpha-Beta.

B- Élagage Alpha Beta

Pour améliorer notre algorithme nous pouvons ignorer la découverte de certains nœuds. En effet si nous trouvons un nœud nous permettant de faire échec et mat a coup sûr il serait alors inutile de développer les autres nœuds.

Nous allons donc utiliser deux paramètres supplémentaires alpha et beta qui vont nous permettre de déterminer à partir de quel moment un nœud peut être considéré comme bon ou mauvais.

Dans notre fonction mini(), si notre score est inférieur à beta que l'on peut prendre égal à -500 alors il sera inutile de découvrir la suite de notre arbre puisqu'il s'agira du pire (ou meilleur pour l'adversaire) mouvement possible à

```
if(score < beta) {
    return min;
}
```

exécuter. Nous avons choisi d'associer 500 comme valeur pour un échec et mat. Notre algorithme renverra alors ce nœud comme étant le pire nœud sans continuer la recherche de nœuds.

```
max.set(index: i, score
if(score > alpha) {
    return max;
}
```

A contrario dans notre fonction maxi(), si notre score est supérieur à alpha, que nous prendrons égal à 500, alors nous renverrons la valeur de ce mouvement sans découvrir la suite de notre arbre.

Le choix de notre valeur alpha et bêta se fera dans notre fonction inputGo() et ne changera pas au cours de la partie. Il est nécessaire de ne pas trop facilement considérer qu'un mouvement est bon (c'est à dire d'avoir un alpha et beta trop faibles) pour éviter de prendre de mauvais choix même si cela permettrait de gagner un certains temps.

```
move = Minimax.maxiFirst(depth: 3, echec, alpha: 500, -500);
```

C- Fonction d'évaluation

Notre fonction d'évaluation est essentielle au fonctionnement de notre MiniMax. Elle consiste à donner une note à une situation donnée. Le but de cette fonction étant de retourner une valeur la plus haute possible si pour une situation très avantageuse pour l'IA (par exemple, être en supériorité numérique au niveau des pièces, ou bloquer le jeu de son adversaire selon le placement des pièces), et à l'inverse de retourner une valeur négative la haute si elle à l'avantage de l'adversaire

.

Sans cette fonction d'évaluation il nous serait impossible d'évaluer si un mouvement est bon ou mauvais et notre Minimax ne pourrait tout simplement pas fonctionner.

Cette fonction d'évaluation est exécutée sur toutes les feuilles lors de l'exécution de notre minimax. Notre Minimax remonte ensuite petit à petit l'arbre afin de prendre les valeurs d'évaluations la plus haute ou la plus basse selon le cas considéré (selon s'il s'agit du tour de l'IA ou de l'adversaire).

Concentrons nous maintenant sur notre fonction d'évaluation. Celle-ci est composé de 5 parties:

- Nous ajoutons tout d'abord des points pour chaque pièce de la couleur de notre IA présentes sur le plateau selon leur importance (50 points pour la Reine, 5 points par pion et 30 points pour chaque autre pièce)

```
ArrayList<Piece> pieces;
if (couleur == true)
    pieces = this.getPiecesBlanches();
else {
    pieces = this.getPiecesNoires();
}

for (Piece p : pieces) {
    if (p.getNom().equals(anObject: "pion")) {
        result += 10;
    } else if (p.getNom().equals(anObject: "reine")) {
        result += 200;
    } else {
        result += 60;
    }
}
```

- De même, nous retirons ensuite des points pour toutes les pièces de la couleur de l'adversaire sur le plateau (50 points pour la Reine, 5 points par pion et 30 points pour chaque autre pièce)

```
if (couleur == true)
    pieces = this.getPiecesNoires();
else {
    pieces = this.getPiecesBlanches();
}

for (Piece p : pieces) {
    if (p.getNom().equals(anObject: "pion")) {
        result -= 5;
    } else if (p.getNom().equals(anObject: "reine")) {
        result -= 50;
    } else {
        result -= 30;
    }
}

result += 2 * (this.getPossibleMoves().size());
```

- Nous ajoutons ensuite des points pour tous les mouvements que notre IA peut effectuer. Ainsi chaque zone où notre IA peut se déplacer lui apporte 2 points. Cela permet de donner un certains nombre de points selon la taille de la zone que notre IA contrôle.

```
result += 2 * (this.getPossibleMoves().size());
```

- Nous donnons des points pour le fait que notre IA mette notre adversaire en échec. Nous pouvons remarquer que nous ne donnons que 5 points ce qui est la valeur d'un pion. En effet, nous ne voulons pas que notre IA sacrifie ses pièces afin de mettre le roi adverse en échec, car l'échec peut être intéressant mais si il ne permet pas l'échec et mat alors il ne faut pas sacrifier inutilement ses pièces.

```
if (this.isEchec()) {
    result -= 5;
}

if (this.isOpponentEchec()) {
    result += 5;
}
```

- Finalement, nous donnons un très grand nombre de points à l'échec et mat. Nous donnons 500 points dans le cas d'un échec et mat et nous retirons 500 points dans le cas où notre IA peut être échec et mat. Ces valeurs correspondent également au valeur de bon ou mauvais mouvement que nous avons assigné (Alpha et Beta) lors de notre élagage Alpha Beta.

```
if (this.isEchecEtMat()) {
    result -= 500;
}

if (this.isOpponentEchecEtMat()) {
    result += 500;
}
```

Conclusion

Pour conclure, nous avons pu observer lors de ce projet la manière dont peut être implémenter un minimax dans un projet concret.

De plus, nous avons pu observer l'importance de la fonction d'évaluation evaluate() dans le fonctionnement de notre MiniMax. En effet, une bonne fonction evaluate() permet d'obtenir de valeur de bestMove plus proche de la réalité.

Nous avons également pu remarquer l'importance de l'optimisation de nos fonctions et du code en général dans ce genre de projet. En effet, puisqu'à chaque récursion tous les mouvements possibles sont appelés, de nombreux calculs sont effectués, la moindre optimisation dans le code aura une influence sur le temps total d'exécution de notre programme.

Nous avons pu remarquer que notre algorithme ne s'exécute pas du tout à la même vitesse selon nos ordinateurs. Ainsi, nous pouvons en déduire qu'afin d'obtenir le meilleur mouvement possible, les algorithmes tels que Deep Blue, qui battent aujourd'hui les champions du monde d'échecs, nécessitent d'avoir des machines extrêmement puissantes.

Enfin, notre projet pourrait être amélioré en utilisant un délai maximal d'exécution de notre fonction minimax afin d'obtenir un résultat le plus précis possible pour la durée demandée, ici 1 seconde.