Création d'un agent intelligent pour le jeu d'échecs

Yoann Ayoub et Louise Lizé

1 décembre 2022

Résumé

Ce document présente le travail réalisé afin de créer un agent intelligent capable de jouer aux échecs. Pour cela, nous avons implémenté un algorithme MinMax avec un élagage alpha-bêta. Nous avons également choisi une heuristique dont l'évaluation se base sur la valeur des pièces et leur position sur le terrain. Au final, notre IA a fini deuxième lors de la compétition qui a eu lieu en classe.

1 Rappel du sujet

Le sujet donné était le suivant :

1.1 Mise en situation

Le vendredi soir, vous jouez régulièrement aux échecs avec votre conjoint(e) qui est malheureusement beaucoup meilleur(e) que vous. Il/elle vous nargue régulièrement à ce propos et appelle sa mère pour rire dans votre dos de votre médiocre performance. Qu'à cela ne tienne ! Vous décidez d'implémenter un agent intelligent afin de vous donner les coups à jouer pour gagner toutes les parties. Rira bien qui rira le dernier !

1.2 Détails du travail à effectuer

Vous devez implémenter un agent capable de jouer aux échecs dans la plateforme ARENA (http ://www.playwitharena.de/). Le thème est libre pour ce travail. Évidemment, vous pourrez avoir des points bonus pour l'implémentation de techniques difficiles et/ou non vue en classe. Un agent se classant très bien en compétition aura droit à des points bonus. Cependant, pour obtenir une bonne note, il vous suffit d'implémenter minimalement un Minimax.

1.3 Contraintes à respecter

— Vous devez programmer votre agent en Java.
— Vous devez décrire le fonctionnement de votre intelligence artificielle dans un rapport professionnel. Vous devriez donc y inclure une description complète des techniques avec un exemple (simple) de fonctionnement.
— Votre IA doit prendre un maximum de 1 seconde pour choisir son coup.
— Votre IA ne peut pas dépasser 20 Go d'espace mémoire. (Attention, vous devez vous assurez d'avoir un moyen de la transférer sur mon ordinateur)

2 Description de l'implémentation

Nous allons maintenant décrire l'implémentation que nous avons mise en place conformément aux règles du sujet. Notre agent intelligent est disponible à l'adresse suivante : https://github.com/Yoann-Ayoub/ChessAI

2.1 Communication avec Arena et règles du jeu

La communication avec Arena ainsi que les règles du jeu (mouvement des pièces, en-passant, roque, etc) ont été principalement créées à partir des tutoriels du Youtubeur *Logic Crazy Chess* [1]. Elles sont définies dans les classes *UCI* et *BoardGeneration* pour la communication, et *Moves* pour les règles du jeu.

Dans *UCI* on retrouve une boucle infinie qui permet de lire ce que nous renvoie Arena, notamment le coup joué par l'adversaire, et de lui communiquer l'action que l'on souhaite jouer à notre tour (dans *inputGo*, nous reviendrons sur le choix de l'action par la suite). La compréhension de l'action réalisée par l'adversaire repose sur la méthode *inputPosition* (dans *UCI*) qui fait appel à *importFEN* (de *BoardGeneration*).

Nous ne nous attarderons pas plus sur les règles du jeu implémentées dans la classe *Moves*.

2.2 Algorithmes utilisés

2.2.1 Construction de l'arbre

Pour pouvoir symboliser une suite de coups possibles, les algorithmes que nous implémentons utilisent la structure d'arbre. Les premiers fils de notre noeud racine sont donc la liste des prochains coups à jouer. Ensuite, pour chacun de ces fils, on les associe aux coups possibles de l'adversaire, puis leurs fils seront à nouveau associés à nos coups possibles et ainsi de suite, comme le montre l'exemple de la figure 1.

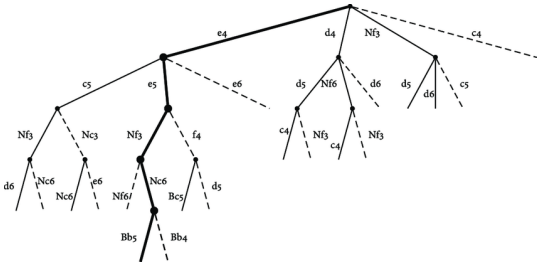

FIGURE 1 – Fragment d'arborescence d'une partie d'échecs (source : Researchgate)

Pour pouvoir représenter cet arbre dans notre code, nous avons ainsi créé la classe *Node*, qui possède les attributs suivants (cf. Table 1).

TABLE 1 – Attributs d'un Node

Variable	Description
List<Node> children	Enfants du noeud
String value	Mouvement correspondant
int score	Score du mouvement
Node sonChoosen	Voir partie MinMax

Grâce à la liste d'enfants, nous pouvons ainsi nous déplacer facilement dans l'arbre.

Pour construire ce dernier, nous faisons appel à la méthode *treeConstruction* disponible dans la classe *SearchAlgorithm*. Cette méthode récursive prend un noeud en paramètre, crée une liste de ses enfants, c'est-à-dire de tous les coups suivants possibles, calcule et leur associe un score. Puis, on appelle à nouveau la fonction en passant en paramètre chacun de ses nouveaux noeuds. On passe également en paramètre l'entier *depth*, qui permet d'arrêter l'appel récursif à une profondeur souhaitée.

De plus, notre arbre prend en compte le fait d'effectuer des mouvements légaux, comme lorsque le roi est en danger et que l'on doit bouger une pièce pour ne plus être dans cette situation.

2.2.2 MinMax

Une fois notre arbre construit, on peut ainsi implémenter MinMax, un algorithme permettant à notre agent de rechercher le meilleur coup. Pour cela, nous nous sommes inspirés du pseudo code présent sur Wikipédia (cf. figure 2) [2]. L'algorithme correspondant est disponible dans notre code dans le fichier *SearchAlgorithm* sous le nom de *MinMax*. Nous l'avons donc adapté à notre structure d'arbre. Cet algorithme retourne la valeur du score et non pas le noeud en question. C'est pourquoi on a ajouté l'attribut *sonChoosen* dans la classe Node. Ainsi, pour chaque noeud, on peut savoir lequel de ses fils a été jugé comme étant le "meilleur". Après avoir fait appel à MinMax, on a ainsi juste à récupérer ce "fils choisi" du noeud racine.

Pseudocode [modifier | modifier le code]

Le pseudocode de l'algorithme minimax de profondeur limitée est présenté ci-dessous :

```
function minimax(node, depth, maximizingPlayer) is
    if depth = 0 or node is a terminal node then
        return the heuristic value of node
    if maximizingPlayer then
        value := −∞
        for each child of node do
            value := max(value, minimax(child, depth − 1, FALSE))
    else (* minimizing player *)
        value := +∞
        for each child of node do
            value := min(value, minimax(child, depth − 1, TRUE))
    return value
```

```
(* Initial call *)
minimax(origin, depth, TRUE)
```

FIGURE 2 – Pseudo code MinMax (source : Wikipédia)

2.2.3 AlphaBeta

Pour améliorer MinMax, on a également choisi d'implémenter un élagage AlphaBeta, permettant de réduire le nombre de nœuds évalué. Nous nous sommes à nouveau inspirés du pseudo code disponible sur Wikipédia (cf. figure 3) [3] en y effectuant quelques changements. Au niveau du code, cet algorithme se situe dans la méthode *AlphaBeta* de la classe *SearchAlgorithm*. L'implémentation de cet algorithme repose sur le fait d'entrer deux entiers en paramètres, Alpha et Beta. Puis, tout au long de l'exécution, on effectue des vérifications sur ces derniers à chaque appel.

Pseudocode [modifier | modifier le code]

Ci-dessous le pseudocode de l'algorithme alpha-bêta : α est initialisé à -∞ et β à +∞

```
fonction alphabeta(nœud, α, β) /* α est toujours inférieur à β */
   si nœud est une feuille alors
       retourner la valeur de nœud
   sinon si nœud est de type Min alors
           v = +∞
           pour tout fils de nœud faire
               v = min(v, alphabeta(fils, α, β))
               si α ≥ v alors  /* coupure alpha */
                   retourner v
               β = min(β, v)
    sinon
           v = -∞
           pour tout fils de nœud faire
               v = max(v, alphabeta(fils, α, β))
               si v ≥ β alors /* coupure beta */
                   retourner v
               α = max(α, v)
    retourner v
```

FIGURE 3 – Pseudo code AlphaBeta (source : Wikipédia)

2.2.4 Heuristique

Concernant l'heuristique, nous avons implementé notre fonction d'évaluation dans la classe *BoardEvaluation*. L'évaluation repose sur deux concepts.

La première est l'évaluation des pièces. Pour cela nous nous sommes basés sur les valeurs proposées par Claude Shanon en 1949 [7] pour laquelle nous avons simplement modifié l'évaluation des fous (*bishop*). Cette modification de la valeur des fous s´intègre dans l'optique de valoriser le fait d'avoir ses deux fous et dévaloriser le fait de n'en avoir plus qu'un, conformément aux indications que l'on retrouve sur la page de *chessprogramming* associée [8].

La seconde est l'évaluation des positions des pièces à partir de *Piece-Square Tables*. Pour cela, l'idée est de créer une table par type de pièce avec une valeur associée à chaque case du plateau. Ce concept permet d'inciter les pièces à prendre des positions dans lesquelles elles seront généralement fortes et ne pas se placer dans des positions où elles seront généralement faibles. Nous avons testé plusieurs valeurs pour nos tables et avons finalement choisi de garder celles proposées sur *chessprogramming* [6] puisque, après plusieurs tests, ce sont celles qui nous donnaient les meilleurs résultats.

2.3 Contrainte à respecter

2.3.1 Communication en moins d'une seconde

Pour respecter la contrainte de communication en moins d'une seconde, nous appelons les méthodes de constructions d'arbres et d'alpha-bêta avec une profondeur de plus en plus élevée, tant que la seconde ne s'est pas écoulée. Cela nous permet de **maximiser la profondeur explorée dans le temps imparti**. Pour le faire, nous avons utilisé la méthode *System.currentTimeMillis()* de Java. A chaque appel d'alpha-bêta ou de la construction de l'arbre, on vérifie qu'on est bien en dessous d'une seconde (990 ms pour laisser le temps de finir). Si ce n'est pas le cas, on arrête la méthode en générant une exception.

3 Améliorations possibles

Dans cette partie nous allons décrire certaines pistes et techniques que nous avons commencé à étudier mais que nous n'avons finalement pas eu le temps d'implémenter. Elles représentent des pistes qui permettraient d'améliorer notre agent.

3.1 Construction de l'arbre

Comme expliqué précédemment, nous avons une boucle while qui permet d'augmenter la profondeur tant que la seconde permise n'a pas été dépassée. Cependant à chaque augmentation de la profondeur, nous faisons appel à la fonction sans réutiliser l'arbre déjà construit précédemment. Une amélioration possible serait de faire en sorte d'appel la méthode *TreeConsruction* sur les feuilles uniquement, afin de ne pas recalculer leurs ancêtres et ainsi gagner du temps de calcul.

3.2 Tri des coups et Principal Variation Search

Une deuxième piste d'amélioration serait de trier nos coups en fonction de celui qui semble être le meilleur à une profondeur donnée. Statistiquement, la meilleure feuille d'un arbre est issue du meilleur des fils à la profondeur 1. Ainsi si nous n'avons pas eu le temps d'explorer suffisamment en profondeur notre arbre et que nous sommes limités par le temps, nous aurons plus de chance de choisir le meilleur coup en ayant effectué un tri en amont. De plus, le tri des coups nous permettrait également de mettre en place l'algorithme *PVS (Principal Variation Search)* qui a de meilleures performances que l'élagage alpha-bêta. Pour cela, nous pourrons nous baser sur les pseudos codes disponibles sur la page *chessprogramming* associée [4].

3.3 Amélioration de l'heuristique

Nous avons plusieurs autres pistes afin d'améliorer l'heuristique permettant de calculer les scores.

La première serait d'avoir des *piece-square tables* différentes pour le milieu de partie et pour la fin de partie [6]. Effectivement, certaines positions peuvent être bonnes quand il ne reste plus beaucoup de pièces en fin de partie et mauvaises en milieu de partie quand il reste encore un grand nombre de pièces (et inversement). C'est pourquoi il serait judicieux de mettre en place un compteur permettant de suivre l'avancée de la partie et d'adapter les *piece-square tables* de sorte à en avoir pour le début et pour la fin de partie.

Une autre amélioration serait d'évaluer la possibilité de mouvement à partir d'une position [5]. Effectivement, plus une pièce a de possibilités de mouvement plus cela est avantageux. Cela se ferait en additionnant un score (plus cinq par exemple) par mouvement possible pour une pièce à partir d'une position.

Enfin, une dernière amélioration de l'heuristique que nous aurions aimé implémenter aurait été d'évaluer les potentiels "d'attaque" à partir d'une position. Un potentiel d'attaque est une situation dans laquelle on met le roi adverse en échec, ce qui force l'adversaire à le déplacer, nous permettant alors de prendre une autre pièce. Dans la figure 4 en plaçant le cheval de la sorte, on force le roi à se déplacer, ce qui nous assure de prendre la tour adverse.

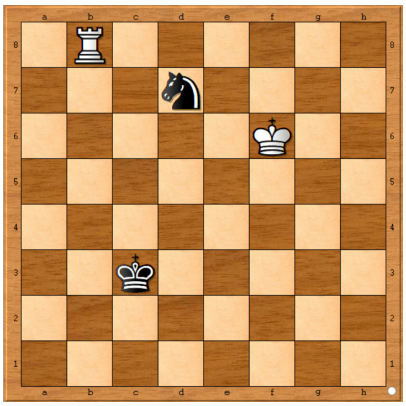

FIGURE 4 – Exemple de situation d'attaque

3.4 Ouverture et fermeture

La dernière piste d'amélioration que nous aimerions implémenter pour la suite serait d'utiliser des bibliothèques d'ouverture et de fermeture. Cela nous permettrait de jouer plus rapidement en début de partie en nous assurant de ne pas faire de faute. Mais également de nous assurer de gagner une partie si on atteint une position à partir de laquelle on est sûr de pouvoir finir. Cela pourrait également nous permettre d'éviter les matchs nuls dans certaines circonstances. Pour les ouvertures, il suffit d'en jouer une jusqu'à ce qu'elle se finisse, mais il sera plus compliqué d'implémenter les fermetures puisqu'il faudra reconnaître les positions dans lesquelles nous avons un pattern de fin.

4 Exemple

Prenons le jeu figure 5 :

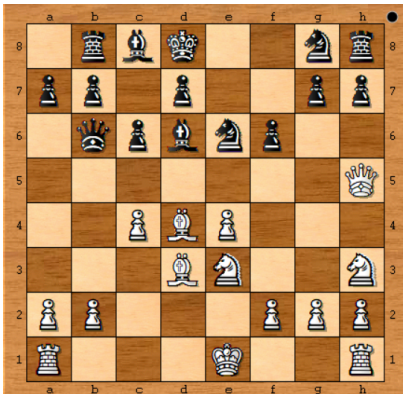

FIGURE 5 – Exemple de situation d'attaque

Dans cet exemple, notre agent joue les pions noirs. Il calcule alors tous les coups suivants possibles et leurs fils. Ici le meilleur coup qu'il puisse effectuer semble être de bouger le cavalier e6 en d4. Pour donner un ordre d'idée, on passe alors d'un plateau ayant un score de 325 à 520. Cette différence s'explique par le fait que l'on retire un fou à notre adversaire et que notre cavalier prend une case centrale sur laquelle il aura un bon score avec les piece-square tables. Il peut être difficile d'imaginer tous les coups futurs, mais il semblerait que bouger ce cavalier ne le mette pas en danger, en tout cas pour les un ou deux coups à venir, puisque notre agent est descendu à une profondeur de 3 dans la seconde qu'il a eu pour choisir son coup.

5 Conclusion

En conclusion, nous avons réussi à implémenter un agent qui a fini deuxième à la compétition d'échec réalisée en classe et qui à atteint un ELO moyen de 700 sur *chess.com*. Nous pensons être capable de continuer à l'améliorer avec plus de temps, et c'est un projet que nous continuerons à développer à la suite de ce cours.

Références

[1] Jonathan from the *Logic Crazy Chess* Youtube Chanel, playlist : "Programming an Advanced Java Chess Engine - Logic Crazy"

[2] Wikipedia, Minimax, [en ligne], adresse url : https ://en.wikipedia.org/wiki/Minimax

[3] Wikipedia, Alpha-Beta pruning, [en ligne], adresse url : https ://en.wikipedia.org/wiki/Alpha-beta_pruning

[4] Chessprogramming, Principal Variation Search, [en ligne], adresse url : https ://www.chessprogramming.org/Principal_Variation_Search

[5] Chessprogramming, Mobility, [en ligne], adresse url : https ://www.chessprogramming.org/Mobility

[6] Chessprogramming, Piece-Square Tables, [en ligne], adresse url : https ://www.chessprogramming.org/Piece-Square_Tables

[7] Chessprogramming, Material, [en ligne], adresse url : https ://www.chessprogramming.org/Material

[8] Chessprogramming, Bishop Pair, [en ligne], adresse url : https ://www.chessprogramming.org/Bishop_Pair