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

Clément Delteil (DELC12110004), Justin Aubin (AUBJ30080005)

26 novembre 2022

Résumé

Ce document présente les algorithmes et optimisations testés lors de la conception de notre agent intelligent pour le jeu d'échecs. Gagnant de la compétition d'échecs (alias : PVS)

1 Remerciements

Le développement de cet agent intelligent a été grandement facilité grâce au travail préalable de Ben-Huer Carlos Viera Langoni Junior qui a développé la librairie Java *Chesslib* [2]. Celle-ci comprend toutes les fonctions de base pour la gestion d'un jeu d'échecs et nous a ainsi permis de faire abstraction de cette partie lors du développement. De cette façon, nous avons pu nous concentrer nos efforts sur l'intelligence artificielle.

De plus, le site *Chess Programming Wiki* [4] nous a grandement aidé dans le choix des optimisations à implémenter car il répertorie de façon quasi exhaustive toutes les techniques existantes.

2 Universal Chess Interface

Pour communiquer avec l'interface graphique Arena GUI donnée dans le sujet, nous avons dû implémenter le protocole *Universal Chess Interface* développé par Stefan Mayer KHALEN en 2004 [3] et qui est toujours utilisé aujourd'hui.

Malgré son apparence complexe au premier abord, il est plutôt simple à implémenter. Dans notre cas, nous l'avons implémenté sous la forme d'une boucle infinie qui récupère les données textuelles envoyées par le GUI.

Nous n'avons pas implémenté toutes les commandes disponibles, mais seulement celles nécessaires pour le fonctionnement général et le tournoi.

Voici les commandes implémentées :

- uci : Renvoie le nom et les auteurs de l'agent
- isready : Renvoie "readyok" quand l'agent est prêt
- ucinewgame : Réinitialise le jeu
- position [fen | startpos] moves ... : Change l'état du plateau en fonction de la liste de coups donnée ou de la FEN [1].
- go : Lance la recherche du meilleur coup qui doit être affiché en console en moins d'une seconde sous la forme "bestmove xxxx"

Vous trouverez le code associé au protocole UCI dans le fichier du même nom : *UCI.java*.

3 Agent Intelligent

3.1 Recherche

Dans cette partie, nous allons nous intéresser aux algorithmes de recherche du meilleur coup et ses améliorations.

3.1.1 Minimax

Pour commencer, expliquons le principe de l'algorithme Minimax. C'est un algorithme qui consiste à minimiser la perte maximale. A partir d'un noeud racine, l'algorithme va passer en revue toutes les possibilités et évaluer les bénéfices pour le joueur et pour son adversaire. La force de Minimax c'est qu'il prend en compte le mouvement de son adversaire. Il va donc supposer que l'adversaire va jouer à son tour le mouvement qui lui est le plus favorable. Le meilleur choix est celui qui maximise les pertes du joueurs tout en supposant que son adversaire, lui, souhaite les maximiser. En pratique, l'algorithme prend la forme d'un arbre à plusieurs étages. Tout en haut de l'arbre on trouve l'état du jeu actuel, c'est le point de départ de notre algorithme. Au premier étage on trouve le jeu avec les mouvements que l'on peut théoriquement réaliser. L'étage d'en dessous lui décrit le jeu avec le coup de l'adversaire et ainsi de suite.

Le principal problème de cet algorithme c'est qu'il est en pratique impossible de construire l'entièreté de

1. Forsyth-Edwards Notation (FEN) : notation standardisée d'une position d'échecs

l'arbre car bien trop grand. Aux échecs, le mathématicien Claude Shannon [7] a fait les calculs pour nous et il compte pas moins de 10^{120} parties d'échec différentes. A titre de comparaison, on estime à "seulement" 10^{80} le nombre d'atome dans l'univers...

On préfère donc ne réaliser qu'une recherche partielle de l'arbre (dans notre cas, en profondeur 5 car déjà plutôt longue).

Vous trouverez le code associé à cette fonction de recherche dans le fichier *Node.java*.

3.1.2 Élagage Alpha Beta

Une amélioration de Minimax peut-être faite en ne parcourant qu'une partie de l'arbre car il ne sert à rien d'évaluer tous les noeuds. En effet, lors de l'exploration, il n'est pas nécessaire d'examiner les sous-arbres qui conduisent à des configurations dont la valeur ne contribuera pas au calcul du gain à la racine de l'arbre. L'élagage est donc fait sur les noeuds les moins intéressants.

Les gains de performance sont extrêmement relatifs. En effet, ces deux algorithmes reposent exclusivement sur l'évaluation des premiers noeuds. Dans le cas où l'on trie nos noeuds du plus intéressant au moins intéressant (en théorie), on base notre élagage sur (en théorie) notre meilleur choix, c'est (en théorie) celui qui peut minimiser les pertes. Il va donc élaguer très rapidement notre arbre et donc d'améliorer les performances.

Vous trouverez le code associé à cette fonction de recherche dans le fichier *Node.java*.

3.1.3 Quiescense Search

Lorsque la recherche du meilleur coup à partir d'une position donnée est limitée à profondeur donnée, les IA d'échecs sont confrontées à ce qu'on appelle *l'effet d'horizon*. Par exemple, le meilleur coup trouvé à une profondeur de 4 peut s'avérer être le pire coup si l'on avait pu évaluer la profondeur suivante et se rendre compte que notre meilleure pièce allait se faire capturer, renversant ainsi la partie.

Une première manière simple de limiter cet effet d'horizon est de seulement évaluer des profondeurs impaires. Ainsi, on est sûr d'être la prochaine personne à jouer et on réduit les risques d'effet horizon.

Une autre manière est de seulement évaluer les positions dites *discrètes* ou *calmes*. Une position est considérée comme calme ou discrète si tous les coups disponibles ne constituent pas une capture, un contrôle ou une menace immédiate pour l'opposition.

Ainsi, lorsque Minimax ou PVS (voir 3.1.4) va atteindre une feuille, plutôt que de directement évaluer la position à l'aide de la fonction d'évaluation, on va utiliser la *Quiescense Search* ou *Recherche discrète* afin de jouer tous les coups capturant disponibles. De cette façon, on aura atteint une position dite discrète et l'on pourra être plus confiant concernant l'évaluation de celle-ci.

La fonction ressemble beaucoup à alpha-beta mais elle est fondamentalement différente. On appelle au début la fonction d'évaluation et si le score est suffisamment bon pour forcer un élagage sans que les captures soient tentées, on élague directement. Si l'évaluation n'est pas assez bonne pour provoquer un élagage mais qu'elle est meilleure que alpha, alpha est mis à jour pour refléter l'évaluation statique. Ensuite, les coups capturant sont essayés, et si l'un d'entre eux provoque un élagage, la recherche se termine. Peut-être qu'aucun d'entre eux n'est bon, ce qui n'est pas un problème.

La fonction a donc plusieurs issues possibles. Il est possible que la fonction d'évaluation renvoie un score suffisamment élevé pour que la fonction puisse sortir immédiatement via une coupure beta. Il est également possible qu'une capture puisse entraîner une coupe beta. Il est aussi possible que l'évaluation statique soit mauvaise, et qu'aucune des captures ne soit bonne non plus. Ou il est possible qu'aucune des captures ne soit bonne, mais que l'évaluation statique puisse augmenter un peu alpha.

Vous trouverez le code associé à cette fonction de recherche dans le fichier *Node.java*.

3.1.4 Principal Variation Splitting

Une autre méthode pour améliorer la vitesse du parcours de l'arbre peut être de le parcourir en parallèle. En effet, bien que l'élagage Alpha-Beta semble être un algorithme séquentiel, il est possible de le paralléliser si on fait quelques concessions.

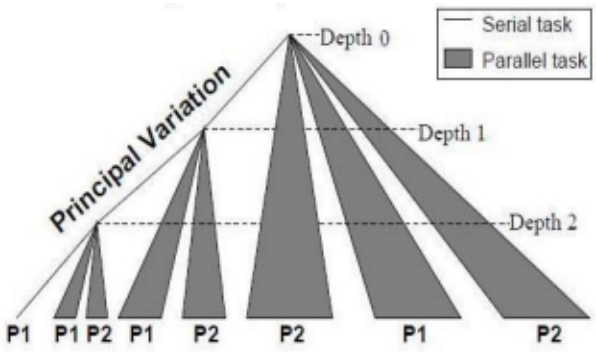

FIGURE 1 – Schéma du Principal Variation Splitting [1]

Prenons comme support la Figure 1. Pour ce faire, on va commencer par diviser les sous-arbres en deux groupes : Le sous-arbre le plus à gauche (LeftSideNode) et l'ensemble de ses frères (Nodes). Puis, on va se déplacer dans la branche la plus au gauche. Pour chaque étage de notre arbre, on va itérer notre processus. Quand on arrive en bas de notre arbre, on donne à un unique processus la tâche d'évaluer le noeud le plus à gauche (LeftSideNode). Cette évaluation va donner une base pour notre élagage. À partir de là, il est possible de paralléliser les noeuds frères (Node) étage par étage. On va alors donner à des threads la tâche de réaliser l'algorithme Alpha Bêta initialisé avec les valeurs du noeud le plus à gauche (LeftSideNode). Chaque thread va alors faire remonter son résultat à son noeud parent qui servira de base à la parallélisation de l'étage du dessus.

Finalement, on se rend compte que l'on perd "l'optimalité" en nombre de noeud de élagage Alpha Bêta car les valeurs alpha et bêta ne sont pas mises à jour globalement pour tous les noeuds mais uniquement à la fin de l'évaluation des noeuds d'une même fratrie. On va donc évaluer plus de noeud que pour l'algorithme séquentiel Alpha Bêta ce qui devrait ralenti le processus. Néanmoins, le fait d'effectuer un parcours d'arbre en parallèle améliore les performances (dans ce cas le temps d'exécution). Il existe néanmoins deux conditions pour optimiser ce gain de performance. La première condition (commune avec Alpha Bêta Séquentiel) est de trier les noeuds afin que le premier noeud soit en théorie le plus intéressant et donc élague notre arbre plus rapidement (voir Optimisations). La seconde condition se porte sur la taille de l'arbre. En effet, si l'on prend un arbre trop petit, les gains de performances sont moindres car la création d'un pool de thread est un processus relativement chronophage. Dans notre programme on considère que les premiers gains se font à partir d'une profondeur 3. Cela signifie également que le noeud le plus à gauche et de sous-arbre de profondeur 3 exécute Alpha Bêta Séquentiel.

En somme, le Principal Variation Splitting (PVS) est relativement efficient car on peut réaliser un parcours de l'arbre en profondeur 7 en moins de 7 secondes comme nous avons pu le voir lors de la compétition d'échec.

Vous trouverez le code associé à cette fonction de recherche dans le fichier *LeftSideNode.java*.

3.2 Évaluation

Il y a deux manières de voir l'évaluation d'une position d'échecs. On peut soit, retourner le score relatif au prochain joueur qui va jouer ou bien considérer que le joueur blanc est le joueur qui sera toujours maximisé.

Dans notre cas, nous avons choisi la seconde option. Ainsi, un score de 3.45 indique un avantage certain pour les blancs alors qu'un score de -4000 indique que les noirs ont largement l'avantage. On comprendra donc qu'un score de 0 indique le début d'une partie, une égalité ou bien une position jugée équilibrée par la fonction d'évaluation.

3.2.1 Évaluation Matérielle

Pour l'évaluation la plus simpliste, on se sert de la valeur relative des pièces d'échecs. Initialement, elles étaient définies comme suit :

- Pion : 1
- Cavalier : 3
- Fou : 3
- Tour : 5
- Reine : 9

Cependant, elles ne sont pas assez représentatives de la réalité. Un fou est par exemple considéré comme étant très légèrement supérieur à un cavalier. Voici des valeurs plus précises et multipliées par 100 pour s'accorder avec les Piece Square Tables (voir 3.2.2) :

- Pion : 100
- Cavalier : 315
- Fou : 320
- Tour : 500
- Reine : 900

Maintenant, si l'on soustrait à la somme des valeurs des pièces du camp blanc, la somme des valeurs des pièces du camp noir, on obtient une première évaluation.

On comprendra que cette vision, simplifiée de la réalité du plateau n'est pas suffisante pour jouer correctement aux échecs. Il faudrait que notre fonction d'évaluation prenne aussi en compte la position des pièces.

3.2.2 Piece Square Tables

Afin de prendre en compte la position des pièces dans l'évaluation, des tableaux ont été créés pour chaque pièce associant à chaque case une valeur. Pour chacun des tableaux qui vont suivre, les noirs se situent en haut et les blancs en bas. Ainsi, les valeurs sont associées au point de vue des blancs.

Listing 1 Pawn Square Table

```
private static final short[] PawnTable =
    new short[]
    {
        0,  0,  0,  0,  0,  0,  0,  0,
        50, 50, 50, 50, 50, 50, 50, 50,
        10, 10, 20, 30, 30, 20, 10, 10,
        5,  5,  10, 27, 27, 10,  5,  5,
        0,  0,  0,  25, 25, 0,   0,  0,
        5, -5, -10,  0, 0, -10, -5,  5,
        5, 10,  10,-25,-25, 10, 10,  5,
        0,  0,  0,   0, 0,   0,  0,  0
    };
```

Dans le cas du pion ci-dessus, vous pouvez voir sur la deuxième rangée des blancs (l'avant-dernière du tableau en Java), les valeurs -25 des deux pions centraux vont pousser l'IA à prioriser le développement de ceux-ci. En effet, si les deux pions restent dans cette position indéfiniment, les blancs verront soustrait à leur score 50 points. D'autant plus que si les deux pions atteignent les cases d4 ou e4, ils recevront chacun un bonus de 25.

Aussi, on peut remarquer que plus les pions avancent dans le jeu, plus on essaye de les récompenser afin qu'ils puissent atteindre la dernière rangée et ainsi se promouvoir.

Dans le cas du cavalier ci-dessous, les cases centrales sont bonifiées tandis que celles proches des bords ont des malus, car étant donné que le cavalier se déplace en "L", ses possibilités se voient largement limitées quand il n'est pas vers le centre du jeu.

Listing 2 Knight Square Table

```
private static final short[] KnightTable =
    new short[]
    {
        -50,-40,-30,-30,-30,-30,-40,-50,
        -40,-20, 0,  0,  0,  0, -20,-40,
        -30, 0, 10,  15, 15, 10, 0, -30,
        -30, 5, 15,  20, 20, 15, 5, -30,
        -30, 0, 15,  20, 20, 15, 0, -30,
        -30, 5, 10,  15, 15, 10, 5, -30,
        -40,-20, 0,   5, 5,  0, -20,-40,
        -50,-40,-20,-30,-30,-20,-40,-50,
    };
```

Ainsi, maintenant, en plus de la valeur matérielle de chaque pièce, on ajoute un score associé à la position. De cette manière, on favorise le développement des pièces et le contrôle du centre du jeu. Malgré cela, les valeurs contenues dans les tableaux sont des valeurs générales vouées à être utilisées tout au long de la partie. Or, elles peuvent être amenées à changer. En effet, si on prend l'exemple du roi, il est certes préférable de le garder en retrait lors de l'ouverture et le milieu de la partie, mais dès lors que l'on rentre dans la fin de la partie, il devient beaucoup plus intéressant de le déplacer vers le centre.

Vous trouverez le code associé à cette fonction d'évaluation dans le fichier *BasicEvaluation.java*.

3.2.3 Tapered Evaluation

C'est pour résoudre ce problème qu'a été pensé la *Tapered Evaluation*. Elle permet de moduler les valeurs de certains tableaux et pièces en fonction de la phase de jeu dans laquelle on se trouve.

Au fil de la partie, on calcule une valeur de phase, qui est un nombre entre 1 et 256 qui signifie à quel point la partie est proche de la fin. On tient alors compte de deux scores d'évaluation (1) le score d'ouverture de la partie (2) le score de la fin de la partie.

De cette manière, l'évaluation finale d'une position va varier en fonction de la phase de jeu dans laquelle on se trouve.

$$\text{eval} = \frac{\text{opening} \times (256 - \text{phase}) + \text{endgame} \times \text{phase}}{256}$$

Où *opening* correspond à l'évaluation de la position avec les instructions du début de la partie et *ending* celle de la fin de la partie.

Dans le cas du Roi par exemple, voici sa table positionnelle de fin de partie :

Listing 3 King Ending Square Table

```
private static final short[]
    KingEndingTable =
    new short[]
    {
        -50, -40, -30,-20,-20,-30,-40, -50,
        -30, -20, -10, 0, 0, -10, -20, -30,
        -30, -10, 20, 30, 30, 20, -10, -30,
        -30, -10, 30, 40, 40, 30, -10, -30,
        -30, -10, 30, 40, 40, 30, -10, -30,
        -30, -10, 20, 30, 30, 20, -10, -30,
        -30, -30, 0,  0,  0,  0,  -30, -30,
        -50, -30,-30,-30,-30,-30, -30, -50
    };
```

On peut remarquer que les cases centrales sont très valorisées. En effet, lorsque l'on rentre dans la fin de la partie il est plus optimal pour le roi de se situer vers le centre pour couvrir tous ses pions que de rester cloîtrer derrière sa ligne de pions là où il pourrait se faire faire échec et mat plus facilement.

Enfin, nous avons aussi un système bonus et de pénalités en fonction de la phase de jeu et du nombre de pièces sur le plateau. On attribue une pénalité de -10 lorsque le nombre de cavaliers sur le jeu est supérieur à 1 et une pénalité de -20 lorsqu'il y a plus d'une tour en jeu ou qu'il n'y a plus de pions en jeu. Aussi, on module un bonus ou malus pour les cavaliers, fous et tours en fonction du nombre de pions.

Listing 4 Rook Pawn Adjustment

```
private static final int[]
    ROOK_PAWN_ADJUSTMENT =
        {25, 20, 15, 10, 5, 0, -5, -10,
         -15};
```

Ici pour les tours par exemple, lorsqu'il y a 0 pion en jeu on attribue 25 points bonus pour le nombre de tours en jeu car elles peuvent probablement mieux se déplacer. Au contraire, s'il y a 8 pions en jeu, alors on attribue un malus de 15 points par tour car le plateau est très probablement encore trop plein pour qu'elles puissent être utilisées correctement.

Vous trouverez le code associé à cette fonction d'évaluation dans le fichier *TaperedEvaluation.java*.

3.3 Optimisations

Les optimisations listées ci-dessous permettent à notre IA de donner son meilleur coup en un minimum de temps.

3.3.1 Tri des coups

Le tri des coups est très certainement l'optimisation la plus importante dans notre IA. En effet, si l'on réussit à donner à l'algorithme Alpha Beta le meilleur coup en premier à évaluer alors il pourra élaguer un très grand nombre de branches et ainsi accélérer sa capacité de calcul.

Nous avons implémenté le tri suivant :
- Si le coup est une capture, on retourne la valeur de la pièce attaquée
- Si le coup est une promotion, on retourne la valeur de la pièce après la promotion
- Sinon on retourne la valeur de la nouvelle position de la pièce à l'aide des PST (3.2.2)

Ainsi, en triant les coups après la génération selon le score associé à celui-ci on favorise l'élagage de plus de branches de l'arbre d'exploration.

Vous trouverez le code associé a cette optimisation dans le fichier *Node.java* ligne 123.

3.3.2 Livre d'ouvertures

L'ouverture est une phase très importante dans une partie d'échecs. Elle conditionne le développement des pièces et la sécurité du roi pour la suite. Ainsi, l'utilisation de livres d'ouvertures dans les IA d'échecs plus avancées est quasiment obligatoire.

Le logiciel *Polyglot* est un logiciel très connu dans ce domaine, notamment grâce à son format binaire pour représenter une ouverture. En effet, il présente des avantages importants, en particulier concernant le gain de place, rapidité d'accès et de recherche. Chaque position est stockée comme une valeur de hachage (8 octets) et quelques informations supplémentaires telles que le nombre de fois qu'elle s'est produite, le nombre de parties gagnées par les blancs/noirs/nulles avec cette position, l'ELO moyen/maximal des joueurs jouant cette position d'ouverture, le succès du programme d'échecs avec la position. Pour économiser de l'espace, ces informations supplémentaires sont généralement de 2 à 8 octets.

La lecture d'un tel fichier n'est pas très simpliste. Pour cela nous avons utilisé le code de Alberto Alonso Ruibal[6] que nous remercions.

Dans notre cas, notre livre d'ouvertures contient environ 20 000 parties jouées. Pour des gains de temps, à partir d'une position donnée, nous avons toujours choisi de prendre le premier coup trouvé dans le livre car c'est le coup le plus joué.

Vous trouverez le code associé à cette optimisation dans les fichiers *fenToPolyglot.java* et *openingBook.java*.

3.3.3 Table de transpositions

L'arbre d'échecs peut être vu comme un graphe, où les transpositions peuvent conduire à des sous-arbres qui ont peut-être déjà été examinés. Une table de *hachage* de transpositions peut-être utilisée pour détecter ces cas et ainsi éviter de dupliquer le travail.

Dans certaines positions, telles que les fins de partie roi+pion avec des pions bloqués, le nombre de transpositions est si élevé que la détection des transpositions est une aide si fantastique que d'énormes profondeurs de recherche peuvent être atteintes en quelques secondes.

Pour l'implémentation, la table de transpositions se présente sous la forme d'une *hashmap* où un élément est une paire de <clé, noeud>. La clé qui permet d'indexer la table de hachage est une clé particulière, il s'agit d'une clé de *Zobrist*. C'est une représentation simplifiée d'une position de jeu en 64 bits. Mais 64 bits ne suffisent pas pour représenter complètement une position d'échecs, ainsi on prend le risque d'avoir quelques collisions lors de la recherche. Cette valeur est calculée automatiquement par chesslib au fil du jeu.

Pour le noeud, il s'agit d'une classe simple qui contient 3 attributs. La valeur de la position évaluée, la profondeur à laquelle on l'a évaluée ainsi que son type. En effet, avec la recherche alpha-beta, on trouve rarement la valeur exacte étant donné que l'on supprime des parts importantes de l'arbre. On a donc une valeur approchée de la position, on sait juste qu'elle suffisamment meilleure qu'une autre. Il convient donc de garder en mémoire le type d'évaluation grâce au type du noeud.

On distingue 3 types différents :
- Noeud exact : la valeur du noeud était exactement celle dont on dispose
- Noeud alpha : la valeur du noeud était au maximum celle dont on dispose
- Noeud beta : la valeur du noeud était au moins égale à celle dont on dispose

De cette façon, si l'on ajoute la table de transpositions à la recherche alpha-beta, on va d'abord rechercher si la position n'a déjà été évaluée auparavant. Auquel cas, plutôt que de l'évaluer à nouveau, on va mettre à jour l'évaluation finale ou les valeurs d'alpha et de beta en fonction du type de noeud stocké dans la table. Sinon, on ajoutera une nouvelle entrée à la table lorsque l'on atteindra une feuille dans l'arbre de jeu.

Malgré l'apparence parfaite de cette solution, elle se confronte à quelques problèmes. En effet, il est inimaginable de stocker toutes les évaluations depuis le début de la partie. On perdrait alors tout l'intérêt de cette optimisation. On peut alors limiter la taille de la table et supprimer les données les plus vieilles lorsque l'on atteint la taille maximale.

Nous n'avons pas gardé cette solution pour le tournoi d'échecs par manque d'expérimentation avec celle-ci. De plus, avec l'implantation actuelle de notre table de transpositions, le gain n'était pas assez important pour augmenter la profondeur d'exploration de l'arbre.

Vous trouverez le code associé à cette optimisation dans le fichier *TranspositionTable.java*. Vous trouverez aussi dans ce fichier une fonction AlphaBeta "factice" adaptée à l'utilisation d'une table de transpositions.

4 Exemple simple

Pour présenter rapidement un exemple d'exécution, après le lancement de la boucle UCI (voir 2), on va recevoir de la part d'Arena l'instruction *go* afin de démarrer la recherche d'un coup.

Algorithm 1 Recherche du meilleur coup

Require: Instruction "go" UCI
Ensure: Renvoi d'un coup légal en - de 1 seconde
  $board \leftarrow$ état actuel
  $depth \leftarrow 5$
  if *continueOpening* est VRAI **then**
    $Movemove \leftarrow OpeningBook.getMove(board)$
    if $move = NULL$ **then**
      $continueOpening \leftarrow FAUX$
      $search(board, depth)$ ▷ PVS
    else
      $print(move)$
    end if
  else
    $search(board, depth)$ ▷ PVS
  end if

5 Expérimentation

5.1 PERFT

***Perf**ormance **t**est, move path enumeration* ou PERFT est une fonction de debugging qui permet de vérifier la véracité de la génération des coups d'une IA d'échecs. Elle va compter le nombre de feuilles dans l'arbre à une certaine profondeur. En se basant sur des valeurs de référence on peut ainsi vérifier que tout fonctionne correctement. Aussi, en mesurant le temps écoulé pour chaque itération, il est possible de comparer les performances d'un génération de coups avec un autre.

Profondeur	# Positions	Temps (ms)
1	20	10
2	400	14
3	8902	71
4	197 281	347
5	4 865 609	1 939
6	119 060 324	33 469

TABLE 1 – Résultats PERFT

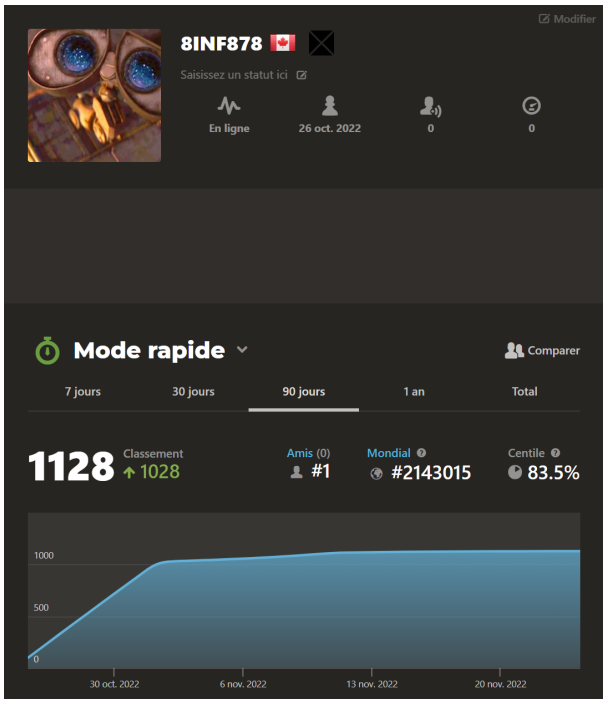

FIGURE 2 – Résumé des statistiques

Comme on peut le voir ci-dessus, dès les profondeurs 5 et 6, le nombre de noeuds et le temps écoulé croissent énormément. Par ailleurs, nous avons vérifié et nos valeurs correspondent bien aux valeurs de référence [5].

5.2 Chess.com

Pour évaluer l'ELO d'une intelligence artificielle aux échecs, il existe des listes de classement d'échecs par ordinateur maintenues par des groupes de passionnés qui font jouer les IA les unes contre les autres. Pour évaluer la nôtre, il faudrait la faire jouer contre un grand nombre d'entre elles afin de la classer. Cependant, c'est un processus compliqué à mettre en place. À la place, nous avons fait jouer notre IA sur le site chess.com afin d'avoir une idée de son ELO.

Vous trouverez en Figure 2 un résumé des statistiques de celle-ci.

Comme vous pouvez le voir, nous avons atteint un ELO final de 1128. Ce qui, pour un joueur humain est un ELO moyen. Dans notre cas, nous étions plutôt satisfaits du résultat, car c'est une bonne démonstration de la cohérence de notre IA sur un échantillon assez important de parties (108).

Plus en détails en Figure 3, vous pouvez voir que notre IA a une précision moyenne (sur 7 parties analysées) de 71,9%. C'est une mesure donnée par chess.com en comparant nos coups à ceux de Stockfish. Plus le choix de nos coups est similaire, plus notre précision augmente. Aussi, vous pouvez voir parmi les 16% de défaites, 50% ont été causées par un abandon de notre part. En effet, étant donné que l'on devait reproduire manuellement l'état de la partie sur Arena, la moindre erreur nous obligeait à abandonner directement.

Cette expérimentation nous a permis d'apercevoir les limites de notre IA. En effet, dans certaines situations, celle-ci ne voulait pas faire évoluer le jeu, car la fonction d'évaluation lui indiquait que cela pénalisait sa position. Ainsi, elle jouait une pièce, puis la faisait revenir à sa case initiale, etc. Le joueur humain pouvait ainsi exploiter cette faille et tenter de faire nul via la répétition de la même position 3 fois de suite. Aussi, ça nous a fait rendre compte du réel problème de l'effet d'horizon. Notamment en fin de partie où si l'on avait augmenté la profondeur de l'arbre à 7, notre IA aurait directement trouvé le mat, mais étant limité à 5 jouait des coups sans grand intérêt.

Vous pouvez trouver le profil de notre IA ici. Afin d'évaluer son niveau réel pour le jour du tournoi, nous l'avons toujours fait jouer en 1 seconde à une profondeur de 5.

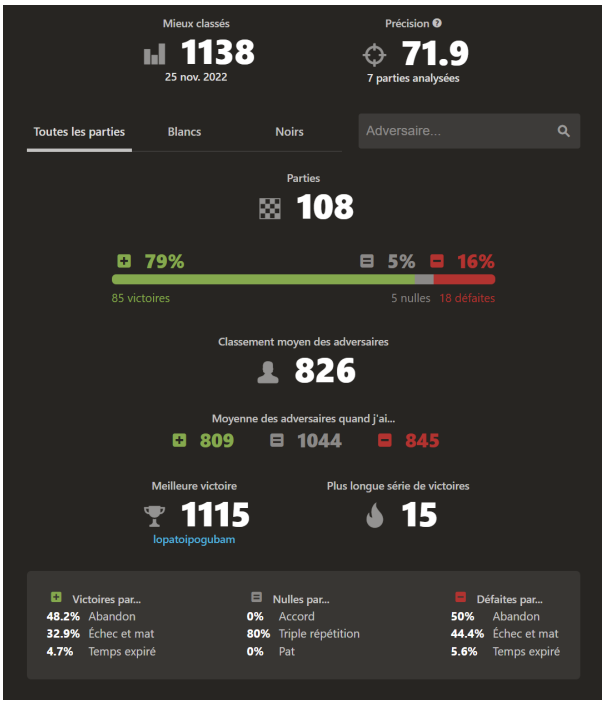

FIGURE 3 – Statistiques détaillées

6 Conclusion

En conclusion, nous avons pu expérimenter la conception d'un agent intelligent pour le jeu d'échecs. Pour ce faire, nous avons d'abord implémenter des algorithmes simples puis nous avons cherché des optimisations pour parfaire les mouvements de notre agent et optimiser ses performances. De surcroît, nous avons pu démontrer l'intérêt de la parallélisation avec notre algorithme PVS distribué sur différents threads.

La condition de renvoi du meilleur coup en 1 seconde nous a poussée à optimiser au maximum notre code afin d'éviter toutes les opérations inutiles. Au fil du développement de l'agent, nous nous sommes rendu compte d'à quel point nous étions dans une boucle sans fin. En effet, les ressources concernant les IA d'échecs sont en abondance sur Internet. Ainsi, si l'on avait eu 2 mois de plus, on aurait pu passer tout ce temps à améliorer chaque partie de notre agent avec des optimisations plus avancées.

À ce jour, il reste bien évidemment encore quelques imperfections à notre agent, mais à ce stade, il est nécessaire d'avoir une bonne connaissance des échecs pour se rendre compte des problèmes présents. C'est d'ailleurs un point positif dans notre équipe, car l'un d'entre nous avait des bonnes connaissances théoriques sur le jeu d'échecs qui ont permis de juger plus facilement et rapidement si les modifications apportées à l'IA amélioraient ou non le niveau de celle-ci. À ce niveau de détails, il devient de plus en plus difficile de voir l'impact des modifications et une solution peut fixer un cas précis et en casser 200 autres.

Références

[1] Khondker HASAN et al. « Implementation of a Distributed Chess Playing Software System Using Principal Variation Splitting. » In : jan. 2010, p. 92-96.

[2] Ben-Hur Carlos Vieira Langoni JUNIOR. *Chesslib*. https://github.com/bhlangonijr/chesslib. 2022.

[3] Stefan Meyer KAHLEN. *UCI protocol*. Avr. 2004. URL : http://wbec-ridderkerk.nl/html/UCIProtocol.html.

[4] *Main page*. Juin 2021. URL : https://www.chessprogramming.org/Main_Page.

[5] *Perft results*. URL : https://www.chessprogramming.org/Perft_Results.

[6] Alberto Alonso RUIBAL. *Carballo*. https://github.com/albertoruibal/carballo. 2022.

[7] Claude E. SHANNON. « Programming a Computer for Playing Chess ». In : *Computer Chess Compendium*. Sous la dir. de David LEVY. New York, NY : Springer New York, 1988, p. 2-13. ISBN : 978-1-4757-1968-0. DOI : 10.1007/978-1-4757-1968-0_1. URL : https://doi.org/10.1007/978-1-4757-1968-0_1.