OPTIMISATION D'ALGORITHME DE COMPLEXITÉ EXPONENTIELLE
Cas du jeu d'échecs

Corto Cristofoli - n° de candidat : 13601

Juin 2023

1 Introduction

Le jeu d'échecs, jeu de société reconnu pour son extrême profondeur stratégique et tactique, ne cesse de captiver les esprits et de défier les capacités humaines en matière de prise de décision. Son évolution est étroitement liée avec l'avancée de l'informatique : l'intelligence artificielle permettant de repousser toujours plus les limites de ce jeu sans pourtant jamais réussir à les atteindre. En 1978, "Chess 4.7" est la première intelligence artificielle (IA) à gagner un match contre un maître (David Levy), mais la première date réellement marquante est la défaite de Garry Kasparov, le champion du monde de l'époque, contre "Deep Blue" en 1997 sur le score de 3.5 à 2.5. Depuis cette date, les intelligences artificielles ont largement dépassé le niveau humain et sont utilisées dans la préparation de tous les joueurs professionnels.

J'ai choisi de m'intéresser à la création d'un jeu d'échecs ainsi qu'à une intelligence artificielle utilisant l'algorithme *alpha-beta* afin d'étudier comment trouver, en un temps minimal, une solution satisfaisante au problème de la recherche du meilleur coup. Mon sujet sera donc orienté vers la compréhension de cette heuristique classique de théorie des jeux et sur son optimisation, autant sur le plan de la fiabilité de la réponse que sur celui de la rapidité d'obtention de celle-ci. Comment implémenter l'algorithme *minmax* et son élagage *alpha-beta* dans le cadre de la recherche du meilleur coup aux échecs ? Par quels moyens améliorer le temps de calcul d'un tel programme dont la complexité est exponentielle ?

2 Théorie des jeux, l'algorithme *minmax* et l'élagage *alpha-beta*

Le jeu d'échecs est, en théorie des jeux, un jeu combinatoire à information complète, c'est-à-dire un jeu opposant deux joueurs qui jouent à tour de rôle et qui ont connaissance de toutes les informations du jeu (information parfaite), où le hasard n'intervient pas. Ce jeu est de plus sans cycle, la partie ne peut être infinie : il y a nécessairement un vainqueur ou un match nul. Nous sommes dans les hypothèses du théorème de Zermelo qui stipule que pour tout jeu combinatoire à information complète, sans match nul et SANS CYCLE alors un des deux joueurs a une stratégie gagnante. Dans le cas des échecs, puisqu'il y a possibilité de nulle, alors, soit le joueur 1 a une stratégie gagnante soit le joueur 2 a une stratégie gagnante ou a une stratégie pour faire nul. Il est donc théoriquement possible de calculer toutes les positions afin de trouver une stratégie gagnante ou nulle pour un des joueurs. Cependant, le nombre de coups possibles par position étant en moyenne d'une trentaine et une partie correcte durant environ 40 coups (soit 80 demis coups) avant mat, nulle ou abandon, on peut alors minorer le nombre de parties possibles par 30^{80} soit environ 10^{120} (c'est le nombre de Shanon [1]). Une machine calculant une partie par microseconde prendrait alors 10^{90} années pour tout calculer, un nombre plus grand que l'âge de l'univers. Il est donc, en pratique, impossible de trouver la stratégie optimale. Il faut alors chercher à l'approximer à l'aide d'une heuristique, j'ai choisi pour cela un algorithme classique : l'algorithme minmax.

2.1 L'algorithme minmax

Le principe est, à partir d'une position donnée, de calculer toutes les positions que le joueur courant (supposons le n°1 sans perte de généralité) peut atteindre avec ses coups (nous sommes alors à une profondeur de 1) puis toutes celles que le deuxième joueur peut atteindre à partir de ces nouvelles positions (profondeur 2). On continue ainsi de suite jusqu'à une certaine profondeur permettant un temps de calcul raisonnable. Une fois arrivé à cette profondeur n on utilise une fonction appelée **fonction d'évaluation** f qui donne une valeur approximative de la position P atteinte, en fonction de si elle est bonne ou non pour le joueur qui a le trait (qui doit jouer) à cette profondeur. Considérons alors qu'une évaluation est bonne si sa valeur

$f(P) = x \in \mathbb{Z}$ est positive et à l'avantage de l'adversaire si négative (nous l'étudierons plus précisément dans la sous partie 4.1). On voit alors que cette position de valeur x pour le joueur 1 est donc de valeur $-x$ pour le joueur 2. Le principe de l'algorithme est alors de remonter la valeur dans l'arbre, par un parcours postfixe, en considérant que chaque joueur joue le meilleur coup possible à chaque fois : le joueur 1 cherchera à maximiser son gain tandis que le joueur 2 cherchera à maximiser le sien donc à minimiser le gain du premier.

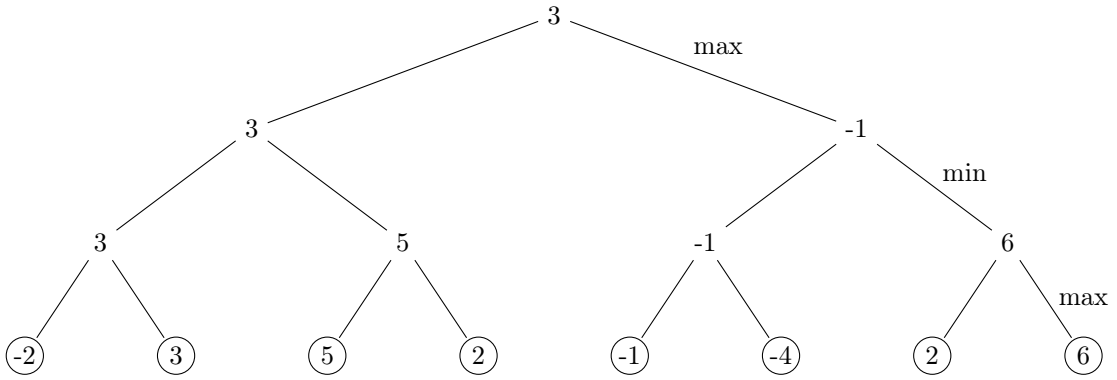


FIGURE 1 – Parcours minmax d'un arbre de jeu à la profondeur 3
Valeurs des positions du point de vue du joueur 1

L'algorithme est alors simple à écrire mais nous l'écrirons dans la convention negamax. Au lieu de considérer la position du point de vue du joueur à la racine, on considère les positions du point de vue du joueur devant jouer la position qui est explorée. Ainsi le score d'une certaine position est égale au maximum des opposés des scores des positions filles. Il n'est alors pas nécessaire de faire une disjonction de cas selon si le noeud visité est un noeud max ou min.

Algorithme 1 : Negamax

Données : Position P dans l'arbre
Résultat : score $\in \mathbb{Z}$
si *P est terminal (est une feuille)* **alors**
  retourner $f(P)$; `/* f est la fonction d'évaluation */`
pour tous *p_i fils de P* **faire**
  score $\leftarrow \max(-\textbf{Negamax}(p_i))$
retourner score

L'algorithme a une complexité en $O(N)$ où N est le nombre de noeuds de l'arbre. Si on note p la profondeur de la recherche, la complexité est en $O(30^p)$ en moyenne, elle est exponentielle. On ne peut alors se restreindre qu'à une profondeur très faible afin de ne pas faire exploser le temps de calcul. Il existe tout de même une amélioration classique de cet algorithme : l'élagage *alpha-beta*.

2.2 L'élagage *alpha-beta*

Cet élagage utilise le fait qu'en pratique de nombreuses positions de l'arbre sont inintéressantes car on sait déjà que l'on a trouvé un meilleur coup. Si l'on prend l'exemple de la Figure 2, on voit qu'une fois les 2 premières positions terminales calculées avec la **fonction d'évaluation**, le joueur choisit le maximum, donc 3. La prochaine feuille ayant 5 pour valeur, quelle que soit la valeur de la suivante, le score du parent sera supérieur à 5 puisqu'il choisit le maximum entre les deux feuilles. Or, son parent à lui doit prendre le minimum entre 3 et ce score (≥ 5), il choisira donc 3. C'est pourquoi il n'est pas intéressant de calculer cette dernière feuille et il en est de même pour la droite de l'arbre (cf Figure 2). Pour l'expliquer d'une autre manière, il existe deux seuils (appelés α et β) tels que, si le score n'est pas dans l'intervalle $[\alpha, \beta]$, le coup n'est pas intéressant puisqu'il existe une manière de forcer les joueurs à se ramener dans cet intervalle. On peut comprendre plus qualitativement α comme le score minimal que le joueur de la racine peut assurer et β comme l'opposé du score minimal que l'autre joueur peut assurer (ou le score maximal que le premier joueur pourrait atteindre) [2].

L'efficacité de cet algorithme dépend de l'ordre dans lequel on regarde les noeuds. Comme le montre l'algorithme 2, si l'on trouve un meilleur score, un "meilleur coup", alors la fenêtre $[\alpha, \beta]$ se précise. Si l'on calcule en premier les meilleurs coups alors la fenêtre sera de suite plus étroite, et plus de coups seront

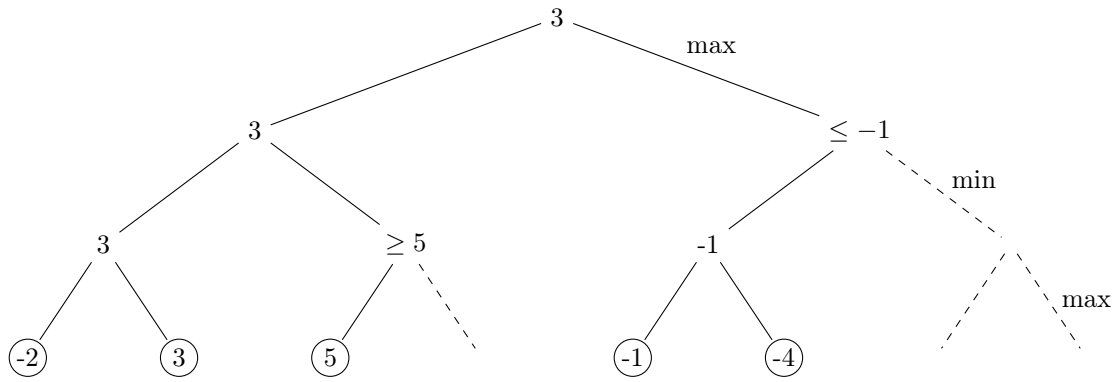


FIGURE 2 – Parcours minmax du même arbre avec l'élagage *alpha-beta*
Valeurs des positions du point de vue du joueur 1

élagués (contrairement au cas où l'on regarde des "mauvais coups" en premier). Nous verrons dans la partie 5 comment optimiser cet élagage en regardant en premier les meilleurs coups probables. Si l'on regarde les noeuds dans le "bon ordre" on peut alors esperer une complexité en $O(\sqrt{N})$ soit $O(\times 30^{p/2})$ noeuds environ (ou N est le nombre de noeuds de l'arbre et p la profondeur)[2].

Algorithme 2 : *Alphabeta*

Données : P, α, β (initialement à $-\infty$ et $+\infty$)
Résultat : score $\in \mathbb{Z}$
si P *est terminal* **alors**
  retourner $f(P)$
pour tous p_i *fils de* P **faire**
  $\alpha \leftarrow \max(\alpha, -\textbf{Alphabeta}(p_i, -\beta, -\alpha))$;
  si $\alpha > \beta$; `/* le coup trouvé est trop bon, il existe un coup forcant au max` β `*/`
  alors
    retourner β
retourner α

La théorie utilisée ayant été définie, nous allons pouvoir procéder à l'implémentation du jeu d'échecs et de l'intelligence dans le language **python**.

3 Implémentation informatique et optimisation des structures de données

J'ai choisi le langage **python** dans ce projet pour deux raisons principales. La première est la facilité d'utilisation et de compréhension du langage permettant de se concentrer sur la partie algorithmique. La seconde est la possibilité d'utiliser des classes, me permettant de structurer le code plus simplement en le gardant lisible (les variables utiles étant stockées dans `self`). Le langage **python** présente néanmoins un souci de performance si on le compare à un langage de plus bas niveau comme le **C**. J'ai simplement choisi de m'intéresser plutôt à la manière d'optimiser mon code de manière algorithmique tout en restant conscient de ce défaut. Mon code est alors divisé en 4 fichiers : `init.py` contient beaucoup de constantes (écrites en majuscule dans mon code) utiles pour le reste, `board.py` s'occupe du jeu en lui-même (ses règles), `main.py` de l'affichage graphique et `engine.py` de la partie IA.

3.1 Considérations à prendre en compte dans l'implémentation

Le jeu d'échecs n'est pas si simple à implémenter, certaines règles complexifient la programmation comme le roque, les échecs, les clouages, les promotions ou les prises en passant. Pour programmer tout cela j'ai utilisé la vidéo de Sebastien Lague [3] dans un premier temps. Afin de vérifier que mon code était bon, c'est-à-dire qu'il générait bien les bons coups, je me suis, pendant tout le projet, référé au "Chess Programming Wiki" [4] (site repertoriant de nombreuses informations sur la programmation du jeu d'échecs) et à leur page *Perf Results* me donnant le nombre de coups possibles à des profondeurs itératives pour certaines positions

"test". Le fait que mon programme renvoie le bon nombre de positions à chaque profondeur permet alors de confirmer le bon fonctionnement du code (voir la fonction `perft` ligne 200 de `board.py`).

Mais, avant même d'implémenter les règles de base, il faut réfléchir aux structures de données utilisées pour représenter le plateau et les pièces. J'ai commencé par une approche utilisant de simples listes avant de me tourner vers une structure nommée *bitboard*.

3.2 Première approche naïve : l'utilisation de liste

Dans ma première implémentation, j'ai considéré l'approche la plus logique : représenter l'échiquier comme un tableau dans lequel sont stockées les pièces présentes sur chaque cases. Avec cette méthode on peut savoir en temps constant la pièce présente sur n'importe quelle case. Pour ensuite générer la liste des coups possibles depuis une position donnée, il suffit de regarder les déplacements possibles de chaque pièces controlées par le joueur qui a le trait. Informatiquement, on distingue alors deux types de pièces : les pièces "sauteuses" et les pièces "glissantes" (*leaping pieces* et *sliding pieces*). Les pièces "sauteuses" sont les pions, le roi et les cavaliers. Elles sont caractérisées par le fait que leurs mouvements ne dépendent pas réellement des autres pièces de l'échiquier (outre les règles spéciales comme l'échec ou les prises en passant, qui sont gérées séparément). La génération de ce type de coups est rapide puisque le nombre de cases à calculer est indépendant de la position actuelle de l'échiquier. Les pièces "glissantes", que sont les fous, les tours et la dame, doivent tenir compte de la position des autres pièces en jeu. Si une pièce se trouve sur leur chemin, ces pièces ne peuvent pas "sauter" par dessus.

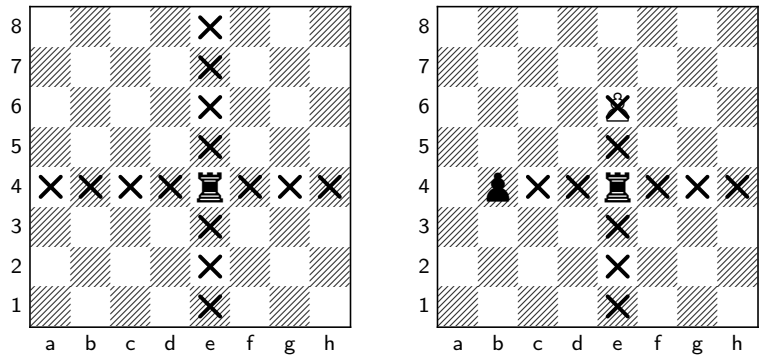


FIGURE 3 – Exemples de cases accessibles pour une tour

Algorithme 3 : Génération des coups d'une tour

Données : case (x, y),plateau
Résultat : liste des cases d'arrivées possibles
L $\leftarrow$ [];
$(i, j) \leftarrow (x, y)$;
tant que (i, j) *dans plateau* **et** *plateau*(i, j) *est vide* **faire**
  L $\leftarrow L + [(i + 1, j)]$;
  $i \leftarrow i + 1$;
$(i, j) \leftarrow (x, y)$;
tant que (i, j) *dans plateau* **et** *plateau*(i, j) *est vide* **faire**
  L $\leftarrow L + [(i - 1, j)]$;
  $i \leftarrow i - 1$;
$(i, j) \leftarrow (x, y)$;
tant que (i, j) *dans plateau* **ou** *plateau*(i, j) *est vide* **faire**
  L $\leftarrow L + [(i, j + 1)]$;
  $j \leftarrow j + 1$;
$(i, j) \leftarrow (x, y)$;
tant que (i, j) *dans plateau* **et** *plateau*(i, j) *est vide* **faire**
  L $\leftarrow L + [(i, j - 1)]$;
  $j \leftarrow j - 1$;
retourner L

Comme on peut le voir dans l'algorithme 3, le calcul des coups possibles pour une pièce "glissante" nécessite des boucles. Le temps de calcul d'un "bon coup" par l'IA dépend beaucoup de la vitesse de calcul d'une position donc des coups possibles. Cette méthode naïve n'est alors pas très efficace puisqu'elle recalcule à chaque fois les cases accessibles pour ce type de pièces. Il faudrait alors envisager de stocker les coups possibles selon la position donnée afin de les récupérer en temps constant, i.e. remplacer la complexité temporelle par une complexité spatiale. C'est pour cela que je me suis intéressé à une autre structure de donnée pour représenter l'échiquier : les *bitboards*.

3.3 L'approche astucieuse des *bitboards*

Le type de représentation de l'échiquier utilisé précédemment est une représentation par rapport aux cases de l'échiquier (*square centric*). On accède en premier lieu à la case et, pour connaitre la position de certaines pièces, il faut alors regarder les 64 cases puisque on ne sauvegarde pas précisément la position de chaque pièce (excepté peut-être le roi). La méthode des *bitboards* est, quant à elle, relative aux pièces (*piece centric*). On connait plus précisément la position de chaque pièce mais il n'est pas direct de savoir ce qui se trouve alors sur une case précise (on peut, en fait, savoir en temps constant si une case est occupée ou non).

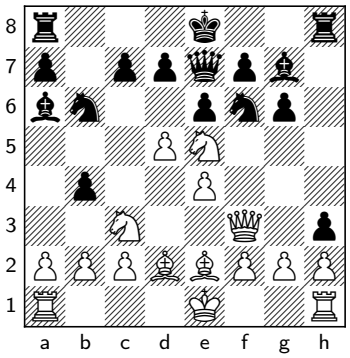
Un *bitboard* est un simple entier codé sur 64 bits. Il permet de stocker dans chacun de ses bits un état préci d'une case. Pour représenter de manière exhaustive le plateau, j'utilise un *bitboard* par type de pièces : `P`,`K`,`N`,`B`,`R`,`Q`,`p`,`k`,`n`,`b`,`r`,`q` dans mon code (les majuscules pour les blancs et les miniscules pour les noirs). Pour bien comprendre, prenons comme exemple `P`. `P` représente la position cases par cases de chaque pions blancs : s'il y a un pion à la case n alors le bit numéro n de `P` sera un 1, de même s'il n'y a pas de pion à la case n' (mais il peut tout à fait y avoir une autre pièce) alors le bit numéro n' de `P` sera un 0 (voir le tout début de `init.py` pour la convention du numérotage des cases). [5] [4]

Une fois ces *bitboards* générés on peut ensuite utiliser des opérations bit à bit : `&`,`|`,$\oplus$ (**et**,**ou**,**ou exclusif**) afin d'obtenir les informations que l'on cherche. On peut par exemple chercher toutes les cases occupées par les blancs :

```
P | K | N | B | R | Q
```

En considérant que l'on connait `A` le *bitboard* de toutes les cases attaquées par les noirs (dans le sens qu'ils ont un coup possible pour déplacer une pièce sur une de ces cases), on peut alors tester si le roi blanc est en échec ou non :

```
K & A > 0
```



bitboard (base 2)

0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	0
0	0	0	0	1	0	0	0
0	0	0	0	0	0	0	0
1	1	1	0	0	1	1	1
0	0	0	0	0	0	0	0

valeur (base 10) :
$2^{27}+2^{36}+2^{48}+2^{49}+2^{50}+2^{53}+2^{54}+2^{55} =$
65020788473856000

FIGURE 4 – *Bitboard* des pions blancs d'une position donnée

3.4 Le principe du "magic-bitboard"

Afin d'améliorer l'efficacité de la génération des coups, on va réaliser tous les calculs au démarage et stocker pour chaque case et chaque pièce le bitboard représentant les cases d'attaque de la pièce à cette case dans une liste python (voir "initialisation des attaques" dans `board.py`, ligne 350). Comme expliqué précédemment, les attaques des pions, roi et cavaliers ne dépendant pas réellement de la position actuelle de l'échequier, ces listes contiennent 64 éléments (1 *bitboard* pour chaque case). Pour les pièces "glissantes" cela se complique grandement et c'est ici qu'entre en jeu le principe des "magic-bitboards". L'idée première est d'enregistrer pour chaque case le bon bitboard pour chaque situations (i.e. positions) possibles, soit $64 \times 63 \times 11^{61}$ au maximum (choix des positions des 2 rois puis de toutes les autres pièces ou cases vides, exceptée la case de la pièce dont on calcule ses coups). En se rendant compte que le type de pièces présentes

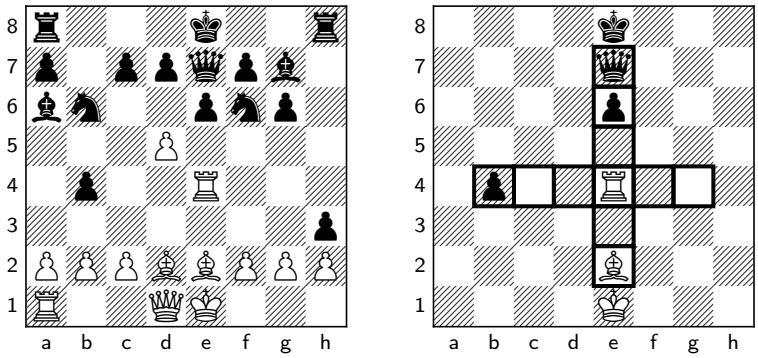


FIGURE 5 – Cases "intéressantes" pour une tour située en e4
Notons que les cases près du bord ne sont pas réellement intéressantes car, qu'il y ai une pièce ou non, la tour ne pourra continuer plus loin.

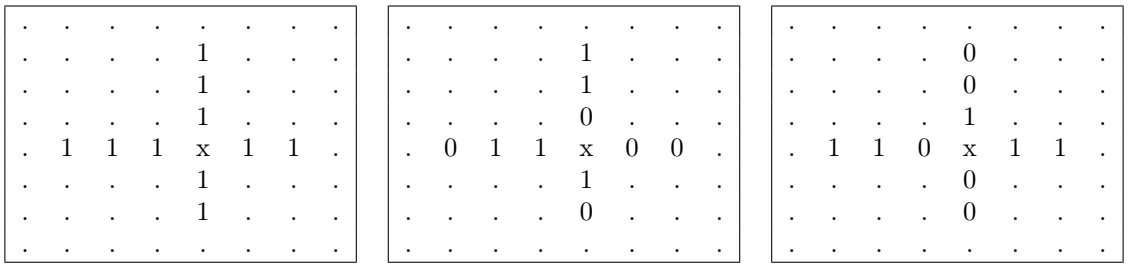


FIGURE 6 – Différents états d'"occupancy" pour une tour situé en e4

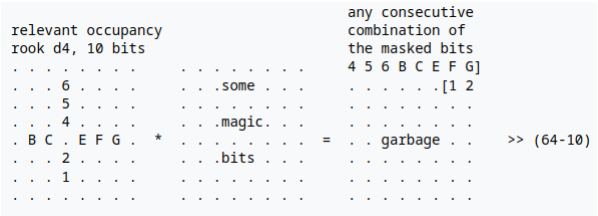


FIGURE 7 – Fonctionnement synthétique du "magic bitboard"
https ://www.chessprogramming.org/Magic_ Bitboards

n'importe pas pour savoir si la pièce est ou non bloquée, on réduit à 2^{63} le nombre de positions différentes possibles à prendre en compte (on appelle alors bloqueur toute pièce, blanche ou noire, qui bloque le trajet de la pièce), mais c'est toujours beaucoup trop pour envisager de stocker tout cela dans un dictionnaire ou un tableau. En affinant encore le raisonnement, il apparait qu'en fonction de chaque case de départ, les cases de bloqueurs potentiels ne sont pas au nombre de 63 mais de beaucoup moins, ce nombre de cases intéressantes dépend de la case de départ et du type de pièce "glissante" mais on peut le majorer pour les tours par 12 et par 9 pour les fous (la dame et la fusion de ces 2 pièces donc on ne s'y intéresse pas) (voir `BISHOP_RELEVANT_BITS` et `ROOK_RELEVANT_BITS` dans `init.py`). Nous avons alors seulement à stocker ces cases "intéressantes" dans nos tableaux comme le montre la figure 5, ce qui ne fait plus que $2^9 = 512$ et $2^{12} = 4096$ entrées (entièrement stockables).

Il ne reste plus qu'à trouver un moyen de convertir une position donnée en un indice en fonction des cases "intéressantes" (appelé "occupancy" en anglais). On cherche en effet une fonction de *hashing parfait*, c'est-à-dire une fonction injective transformant l'"occupancy" en un indice unique afin de pouvoir lui associer le bon *bitboard* d'attaque. C'est ici qu'intervient un "magic number" tel que ce nombre multiplié par la configuration de l'"occupancy" renvoie le bon indice. Ce nombre est différent pour chaque pièce et chaque case, il est généré par force brute en considérant des nombres aléatoirement (voir `find_magic_number` à la ligne 1170 de `board.py` et la figure 7). [4]

En se concentrant sur des positions "test" présentes sur la page *Perf Results* de [4], j'ai pu comparer les performances moyennes de génération de coup, en utilisant des listes et des *bitboards* avec et sans "magic bitboard" (voir Figure 8).

Une fois le jeu programmé efficacement, j'ai pu me concentrer sur l'implémentation informatique de mon

Positions	Listes	Bitboards	Magic bitboards
initiale	13s	3,9s	2,8s
4	33s	9,4s	7s
5	160s	46s	34s
6	313s	77s	58s

FIGURE 8 – Performance du calcul des coups sur différentes positions test à une profondeur de 4

IA (sauf mention contraire, dans ces 2 prochaines parties, toute référence au code sera celui présent dans `engine.py`). J'ai tenté d'améliorer la qualité des coups de diverses manières.

4 Alpha-beta : approcher une meilleure solution

4.1 La fonction d'évaluation

La première manière d'améliorer mon heuristique est d'améliorer la fonction d'heuristique elle-même : la **fonction d'évaluation**. Cette fonction doit renvoyer une estimation précise de l'état d'une position. La question est donc de savoir comment juger une position aux échecs. J'ai choisi de m'intéresser aux facteurs souvent étudiés et analysés par des joueurs humains pour juger d'une position. Le premier paramètre, qui est le plus classique et le plus simple, est la prise en compte du matériel. On attribue classiquement aux échecs une valeur à chaque pièce : 1 pour le pion, 3 pour le cavalier et le fou, 5 pour la tour et 9 pour la dame. En sommant pour chaque joueur les valeurs de chaque pièces puis en calculant la différence entre les deux on a alors une évaluation grossière de la position : si un joueur a un fou de plus que son adversaire, il a certainement une meilleure position, s'il a une dame de plus, cette position est encore meilleure. Mais ce paramètre, bien qu'important ne rend pas compte de la complexité de certaines positions ; de nombreux autres paramètres peuvent être pris en compte : la sécurité du roi, le développement des pièces sur des bonnes cases (fous sur diagonales ouvertes, cavaliers au centre, tours sur colonne ouvertes...), le contrôle du centre, la paire de fous, les pions doublés ou isolés... J'ai implémenté une partie de ces paramètres (voir ligne 1000 de `board.py`).

Le problème de cette approche est qu'elle se fonde sur des paramètres que les humains considèrent comme bons et non bons dans l'absolu (puisque le jeu d'échecs n'est pas résolu, on ne sait théoriquement pas si une position est bonne ou non). Une manière plus récente de faire est d'utiliser des réseaux de neurones pour que l'IA trouve "elle-même" ses bons paramètres. Cette méthode s'éloignant trop de ce qui m'intéressait réellement, à savoir étudier les manières d'optimiser l'algorithme *alpha-beta* plutôt que la fonction d'évaluation seule, je me suis contenté de mon évaluation simpliste (c'est donc un axe certain d'amélioration pour mon IA).

4.2 Mieux apréhender les différentes phases de jeu

Une autre difficulté que rencontre l'algorithme est sa gestion des premiers et derniers coups d'une partie. Durant l'ouverture, peu de pièces peuvent être mangées en apportant un gain notable (puisque souvent protégées) et il faut comprendre où déplacer ces pièces pour avoir une position équilibrée en milieu de jeu. La fonction d'évaluation permet de conduire l'IA, par exemple, à développer ses pions centraux, ses pièces mineures (cavalier, fou), roquer (voir la fonction d'évaluation et les `POS_SCORE` dans `init.py`), mais souvent cela ne suffit pas. C'est pourquoi j'ai ajouté une petite table d'ouverture afin de faciliter le début de jeu (voir `book.txt`). je ne me suis pas penché sur une table d'ouverture approfondie car cela sort de mon objectif, je voulais simplement aider l'IA à obtenir une position correcte pour pouvoir ensuite utiliser le potentiel de l'algorithme.

Pour les finales, le problème est la très grande difficulté de trouver un mat, même avec un grand avantage, car souvent les combinaisons nécessitent un grand nombre de coups qui, bien que logiques pour un homme, ne sont pas calculables par l'IA (profondeur trop grande). J'ai alors modifié les paramètres de la fonction d'évaluation en finale afin d'inciter le roi à venir au centre et à essayer d'amener le roi adverse sur les bords, permettant par exemple à l'intelligence artificielle de réussir un mat avec uniquement un roi et une tour (cf. Figure 9).

4.3 Eviter l'effet d'horizon grâce à la *quiescence search*

Il subsiste un problème non négligeable dans l'algorithme *alpha-beta*, que se passe-t-il si la position calculée à la profondeur maximale est considérée comme "bonne" car la dame vient de capturer un pion ?

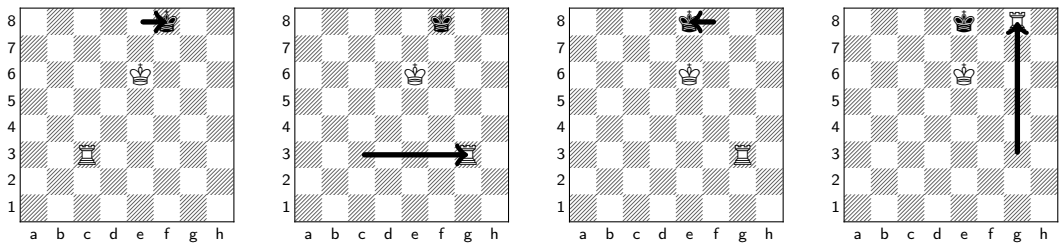


FIGURE 9 – Mat avec un roi et une tour

L'algorithme va choisir le coup menant à cette position, qu'il considère "bonne", sans voir qu'au coup d'après la dame sera reprise par un autre pion (rendant donc le coup précédent très mauvais) : c'est l'effet d'horizon. Comment l'éviter ? Une fois arrivé à la profondeur maximale de recherche, on effectue une recherche "tranquille" (*quiescence search*). La recherche "tranquille" (veuillez excuser ma traduction approximative) est un algorithme *alpha-beta* simplifié, qui ne regarde que les prises et s'arrête quand il n'en trouve plus (voir `quiescence_search` ligne 234). Dans la même optique, on choisit d'augmenter la profondeur de recherche lorsque l'on analyse une position où il y a échec. On le peut car les positions d'échecs ont l'avantage de n'octroyer que quelques coups possibles ensuite. Cela permet notamment de voir des mats que l'IA ne pourrait pas voir sinon.

Toutes ces améliorations ont tendance à augmenter considérablement le temps de calcul car l'ordinateur analyse beaucoup plus de noeuds et prend plus de temps pour traiter chaque position terminale (en raison de la fonction d'évaluation plus complexe). Il faut donc s'attacher maintenant à réduire ce temps de calcul.

5 Alpha-beta : optimisation de la vitesse d'exécution

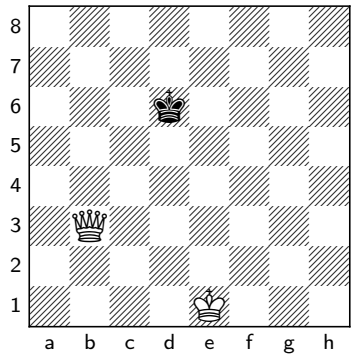
5.1 Trier les coups

Comme on l'a vu, l'élagage *alpha-beta* est d'autant plus efficace que les coups sont triés dans le bon ordre, c'est-à-dire dans l'ordre des meilleurs coups probables en premier. J'utilise pour cela une fonction de tri, appelée `tri_move`, utilisant la fonction `score_move` (ligne 270) qui trie dans le bon ordre les coups possibles à partir d'une position donnée. Plus le score d'un coup est élevé, plus la probabilité qu'il soit le meilleur coup est haute. Les coups déjà calculés comme étant les meilleurs précédemment (ceux de la variation principale) ont de grande chance d'être toujours les meilleurs. De plus une capture est souvent un coup intéressant à regarder ensuite, car plus succeptible de changer drastiquement une position. On les regarde alors ensuite en les triant eux-mêmes avec la table `MVV_LVA` (voir la figure 10). Viennent ensuite les *killer moves*. Ces coups sont des coups "trop bons", i.e. des coups ayant un score superieur à β quand calculés. Cela signifie qu'il existe pour l'autre joueur une manière d'éviter ce "trop bon coup" (il est réfutable). Pourtant, un coup étant "trop bon", a des chances de devenir un "bon coup" un peu plus tard, la position ayant été légèrement modifiée. Supposons qu'il existe un très bon coup mais réfutable par la mise en échec de notre roi, mettre à l'abri notre roi rendra alors ce coup réellement bon. Enfin, pour terminer le tri, on utilise l'*history heuristic*. Les coups discrets (n'étant pas des captures) obtiennent un petit score s'ils sont assez bons, i.e. si durant l'agorithme *alpha-beta* ils font augmenter α, si ce sont des variantes intéressantes. [4]

Un tri correct ayant été appliqué aux coups, il peut maintenant être intéressant de sauvegarder les scores des positions déjà calculées afin de ne pas avoir à les recalculer si l'on retombe sur la même position. Je vais donc implémenter une table de transposition.

Attaquant/Victime	Pion	Cavalier	Fou	Tour	Dame	Roi
Pion	105	205	305	405	505	605
Cavalier	104	204	304	404	504	604
Fou	103	203	303	403	503	603
Tour	102	202	302	402	502	602
Dame	101	201	301	401	501	601
Roi	100	200	300	400	500	600

FIGURE 10 – Most Valuable Victim, Least Valuable Attacker



Roi blanc en e1	1000111010011010	$\oplus$
Roi noir en d6	1110001110000111	$\oplus$
Dame blanche en b3	1101010111001011	$\oplus$
Trait aux blancs	0010010010110000	$=$
	1001110011100110	

FIGURE 11 – *Hashing* de Zobrist d'une position simple

5.2 La table de transposition

L'idée de la table de transposition est de stocker les positions déjà calculées pour réutiliser le résultat trouvé. L'on y enregistre la profondeur à laquelle on a calculé la position, la valeur de α ainsi qu'un drapeau nous indiquant si le coup a été réellement calculé ou s'il a été coupé car trop mauvais ou trop bon. Lorsque l'on retombera plus tard sur ce noeud, si la profondeur à laquelle on le découvre est plus grande que celle enregistrée on récupère la valeur enregistrée, au lieu de le recalculer, car on sait déjà que l'on connait sa valeur plus précisément (voir `alphabeta` ligne 120 et `get_transpo` ligne 350).

La question est maintenant de comprendre comment sauvegarder une position de manière efficace dans le dictionnaire représentant la table. Il faut une manière de générer des clés, si possible uniques pour chaque position à enregistrer. C'est ici qu'intervient le *hashing* de *Zobrist*. Le principe est d'attribuer, pour chaque pièce sur chaque case, un nombre aléatoire unique, puis de les additionner (avec un **ou exclusif**) ensemble en fonction des pièces présentes dans la position pour obtenir un identifiant, ou *hash* (cf. Figure 11). On utilise des entiers de 64 bits pour coder ces *hash*. Le nombre de positions aux échecs étant bien supérieur à 2^{64}, la probabilité qu'il y ait collision, i.e. que deux positions différentes aient la même clé, n'est absolument pas négligeable. Heureusement, cette méthode de *hashing* a l'avantage d'attribuer des clés très lointaines (au sens de la distance de *Hamming*) pour des positions proches, si bien que si collision il y a, les positions sont assez différentes pour ne pas se retrouver dans une même recherche. Il est alors important de bien choisir les nombres aléatoires initiaux afin de posséder cette propriété. Le choix de nombres aléatoires optimaux pour le *hashing* de Zobrist est une question complexe que je n'ai pas eu le temps de bien étudier. C'est un autre axe d'amélioration potentiel de mon IA. [4] [2]

Cette technique a, de plus, l'avantage d'être très rapide puisque, une fois le *hash* de la position initiale généré, le calcul des *hash* des positions filles est très rapide. En effet, notons `h:int` le *hash* de la position actuelle, `Nz:int array` la liste de tous les *Zobrist numbers* du cavalier blanc pour chaque cases et `side_z:int` le *Zobrist number* pour coder le trait aux blancs ou aux noirs. On suppose qu'on déplace le cavalier blanc de **f3** à **e5** (le trait et donc aux blancs). Remarquons que $\forall$ `x` entier, `x`$\oplus$`x=0` (on rappelle $\oplus$ est le **ou exclusif**). On a donc comme nouvelle position :

$$\texttt{h'} \leftarrow \texttt{h} \oplus \texttt{Nz["f3"]} \oplus \texttt{Nz["e5"]} \oplus \texttt{side\_z}$$

5.3 Autres améliorations

Je souhaite terminer ce rapport par une présentation succinte de plusieurs améliorations significatives de l'algorithme *apha-beta* en lui même. Elles ont toutes pour principe d'appliquer *alpha-beta* de manière approchée, en réduisant la profondeur de recherche et en utilisant une **fenêtre nulle de recherche**. Le principe de la fenêtre nulle de recherche (*null window*) est d'appliquer *alpha-beta* de manière booléenne, i.e. de ne regarder que s'il existe ou non un coup donnant un score meilleur que α (β n'importe plus). Cela donne donc une **approximation** du vrai résultat ! L'approximation ne nous intéressant pas, il est judicieux de l'utiliser dans le cas où l'on a de "bonnes raisons" de penser que tous les coups que l'on va analyser sont moins bons que ceux déjà calculés (si tous les scores sont inférieurs à α alors on a bien un résultat exact). Si ce n'est pas le cas, on s'est trompé et il faut recalculer tous les coups avec la bonne fenêtre : $[\alpha, \beta]$. Examinons plus en détails l'utilisation de cette fenêtre nulle.

La première utilisation que j'en ai faite est pour la *Principal Variation Search* (PVS). L'idée est qu'un coup, considéré préalablement comme appartenant à la variation principale (la meilleure séquence de coups

possibles pour les deux joueurs), a de grandes chances de rester le meilleur coup. On effectue donc une recherche avec fenêtre nulle pour les coups suivant à calculer. [4]

Vient ensuite la *Late Move Reduction* (LMR). L'idée ici est que les 2 ou 3 premiers coups analysés ont de grandes chances d'être les meilleurs, on teste alors simplement si c'est le cas avec les suivants grâce à la fenêtre nulle de recherche (et on diminue même la profondeur de recherche). Ces deux principes sont, en pratique, très efficace (gain d'environ un facteur 10 en temps de calcul) mais il faut alors avoir, préablement, bien trié les coups, comme vu précédemment. Dans le cas contraire, l'ajout de ces principes ralentirait grandement le programme puisque, les suppositions n'étant pas bonnes, il faudrait recalculer beaucoup plus de coups avec les paramètres corrects (les recherches avec fenêtre nulle seraient donc une totale perte de temps). [4]

L'avant dernière approche que j'ai implémentée est appelée l'élagage par coup nul (*Null Move Pruning*). Le principe est, avant de calculer nos coups possibles, de supposer que l'adversaire a encore le trait et donc de lui laisser jouer un coup supplémentaire. S'il ne trouve aucun coup "assez bon", i.e. $\geq -\beta$ (on voit ici la condition booléenne de la fenêtre nulle), on peut facilement supposer qu'il existe pour nous un coup "trop bon" (i.e. $\geq \beta$). Je trouve cette approche intéressante car ressemblant fortement à la manière de penser que peut avoir un joueur humain : il n'est pas rare, pour nous, de comprendre qu'une position est bonne car l'adversaire n'a aucun coup intéressant. Il faut tout de même faire attention aux cas de *zugzwang* (dans certaines positions, le fait d'être obligé de jouer peut dégrader la position que l'on avait). [4]

Enfin, la dernière amélioration, est une méthode intitulée *aspiration window*. Le principe est de calculer *alpha-beta* avec une profondeur augmentant itérativement mais en affinant les bornes α et β de départ. Pour cela, en posant $\epsilon = 1/2$ pion (valeur empirique classique) et `score` le score calculé à l'itération `p` précédente, on appelle `alphabeta(score-`ϵ`,score+`ϵ`,p+1)`. Si le score renvoyé est en dehors des bornes, on s'est trompé et on recommence à la même profondeur en posant cette fois $\alpha = -\infty$ et $\beta = +\infty$. En général, on reste dans l'intervalle et la recherche à chaque nouvelle profondeur est plus rapide. [4]

Voilà ci-dessous l'algorithme 4, *alpha-beta* final, implémentant ces procédés.

6 Conclusion

Après avoir implémenté toutes ces optimisations j'ai pu observer une amélioration de vitesse d'un facteur 100 environ. Par des méthodes variées, allant de l'optimisation de la structure de donnée utilisée jusqu'à des approches plus probabilistes, j'ai ainsi pu apréhender et manipuler de nombreux concepts me permettant de comprendre comment optimiser un algorithme de complexité exponentielle : l'algorithme *alpha-beta*.

Références

[1] Claude E Shannon. Xxii. programming a computer for playing chess. *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*, 41(314) :256–275, 1950.

[2] Jean-Marc Alliot, Thomas Schiex, Pascal Brisset, and Frédérick Garcia. *Intelligence artificielle et informatique théorique*. Cépaduès-éd., 1994.

[3] Sebastien Lague. Coding adventure : Chess ia, 2021.

[4] Wikimedia Foundation. Chess programming wiki.

[5] Code Monkey King. Bitboard chess engine in c, 2021.

Algorithme 4 : *AlphabetaFinal*

Données : P, α, β (initialement à $-\infty$ et $+\infty$),**table_transposition**,**killer_move**
Résultat : score $\in \mathbb{Z}$
si hash(P) *est dans* **table_transposition** *avec une profondeur inferieure* **alors**
  retourner table_transposition(hash(P)**)**
si P *est terminal* **alors**
  retourner *quiescence_search*(P)
;
; `/* élagage par coup nul */`
On donne le trait au joueur adverse;
$score \leftarrow -$**Alphabeta**$(P, -\beta, -\underline{\beta + 1})$;
si $score \geq \beta$ **alors**
  retourner β
; `/* fin élagage par coup nul */`
;
pour tous p_i *fils de* P *triés dans le bon ordre* **faire**
  si $i \leq k$; `/* k est le nombre de coups recherchés précisément */`
  alors
    $score \leftarrow -$**Alphabeta**$(p_i, -\beta, -\alpha)$;
  sinon
    $score \leftarrow -$**Alphabeta**$(p_i, -\underline{(\alpha + 1)}, -\alpha)$; `/* Fenêtre de recherche nulle */`
    si $score \geq \alpha$; `/* On s'est trompé, ce coup est meilleur */`
    alors
      $score \leftarrow -$**Alphabeta**$(p_i, -\beta, -\alpha)$; `/* On recalcule donc le coup */`
  si $score \geq \beta$ **alors**
    On enregistre le coup dans **killer_move** (si non capture) et p_i dans **table_transposition**;
    retourner β
  si $score \geq \alpha$; `/* On a trouvé un meilleur coup */`
  alors
    On le considère comme la nouvelle variation principale (on le regardera en premier aux prochains calculs);
    $\alpha \leftarrow score$
On enregistre P dans **table_transposition**;
retourner α