BLAM11100105 - Martin BLANCKAERT

SIRT10080007 - Thomas SIRVENT

8INF878 - Intelligence Artificielle
Rapport TP2 - Échecs

Algorithme Minimax

Élagage alpha-bêta

Optimisations

UQAC

Trimestre d'automne 2022

Instructions d'exécution

Il existe deux manières d'exécuter notre programme :

- La manière la plus simple est de construire l'artefact du projet, permettant d'obtenir un fichier JAR, que l'on peut donner à Arena afin que l'on puisse jouer contre notre programme.

- Il est possible de l'utiliser dans un IDE comme un programme classique, mais il faut que l'humain qui lance le programme remplace le rôle d'Arena. Il doit donc rentrer à la main des instructions pour faire marcher le protocole UCI (cela sera expliqué plus tard).

  Au lancement, le programme attendra un input de la part de l'utilisateur. Afin de jouer une partie, on peut rentrer "*position startpos*" pour indiquer à notre moteur la position actuelle du plateau. Pour lui faire faire l'analyse, on peut ensuite entrer "*go*".

  En réponse, le programme retourne un coup sous la forme "bestmove [*mouvement*]". Pour continuer à jouer, on peut alors entrer la même instruction qu'au début, suivi de "*moves*" et tous les coups joués par l'humain et le moteur. On peut alors demander la réponse du moteur avec "*go*".

```
position startpos
go
bestmove g1f3
position startpos moves g1f3 e7e5
go
bestmove f3e5
```

Déroulement classique - Le coup e7e5 est décidé par l'humain, les autres par le moteur

Introduction

Pour ce travail, nous devons programmer un moteur d'échecs, cela veut dire créer un système expert qui, à partir d'une position donnée, saura indiquer le meilleur coup à jouer.

Plus qu'un simple algorithme, ce travail pratique présentera une association de techniques d'optimisation permettant de répondre aux contraintes données.

Notre moteur sera testé dans l'interface graphique *Arena*, qui utilise le protocole UCI afin d'envoyer et recevoir des informations avec notre programme.

Afin d'être compatible avec ce protocole, le projet sera programmé en Java, car cela nous permet d'utiliser deux librairies essentielles : chesslib[1] et UCI (Universal Chess Interface).

- La première, créée par Ben-Hur Carlos Vieira Langoni Junior, nous permet d'avoir l'architecture classique du jeu d'échecs déjà mise en place.

- La deuxième nous permettra d'avoir un ensemble de fonctions permettant la communication en UCI.

Ces librairies nous permettent de nous concentrer principalement sur l'algorithme de décision plutôt que sur les règles de jeu et les opérations protocolaires.

[1] https://github.com/bhlangonijr/chesslib

Structure des classes

Classe des librairies

Nous n'allons pas rentrer dans les détails des classes qui ont été fournies par nos librairies, mais nous pouvons quand même les mentionner afin de comprendre les méthodes utilisées dans nos propres classes.

La librairie chesslib nous fournit un éventail de fonctionnalités et de classes très large.
En interne, elle utilise la formalisation du jeu d'échecs sous forme de **Bitboard**, une liste de bit, permettant de traiter de l'information rapidement à travers des opérations bit à bit.

Nous pourrons donc utiliser la classe **Board**, qui est donc une interface pour ces opérations.
Au niveau des cases, nous avons les classes **Square** et **Piece** afin de gérer les pièces du jeu et les cases qui les contiennent.

La librairie UCI nous fournit un ensemble de méthodes qui ont été encapsulées dans la classe **UCI**. Ces méthodes sont les suivantes :

UCICommunication - inputUCI - inputIsReady - inputUCINewGame

Ces fonctions ont été inchangées de par leur nature protocolaire, mais les suivantes ont été modifiées car elles concernent l'aspect décisionnel du programme :

inputPosition - inputGo

Classe personnalisées

Ces classes sont toutes regroupées dans le dossier "algorithmes" du projet.

Minimax

Classe qui encadre le fonctionnement et le lancement du Minimax, il contiendra toutes les variables globales au contexte du jeu (poids des pièces, tables PST,) et de nos algorithmes (taille de la table de transposition, marge de *futility*). Les concepts mentionnés sont détaillés plus tard dans ce rapport.

Node

Lorsque nous étudions les algorithmes comme celui du Minimax, on apprend comment la réponse nous est donnée, mais nous sommes souvent libre de l'implémentation qui nous permettra de faire correspondre une valeur, réponse de Minimax, à un coup sur le plateau.

Une classe Node est donc utilisée. Elle contient un score et un coup. Ainsi, quand Minimax opère des changements de valeurs, nous passons par des Nodes afin de toujours retenir les coups correspondants.

HashEntry

Cette classe servira de structure de données afin d'être utilisée pour les tables de transposition. Les valeurs qu'elle contient sont détaillées plus tard dans la section Tables de transposition.

StartTree

Cette classe contient toutes les informations concernant les openings connues que notre moteur peut utiliser. Afin de ne pas charger notre programme avec des librairies supplémentaires et des méthodes de parsing de fichier PGN, les données ont été rentrées en brut, permettant d'utiliser l'arbre plus rapidement et simplement

treeNode

Associé à *StartTree*, il permet de construire la structure de l'arbre. Un treeNode contient un coup sous forme de String et une liste d'enfants (qui sont des treeNode également).

Communication UCI

UCI, ou Universal Chess Interface, est un protocole de communications permettant de communiquer de manière standardisée avec des GUI comme Arena. Il utilise les outputs en console des programmes comme des instructions.

Dans notre projet, les instructions se déroulent de cette manière :

On prépare un objet *Scanner* qui recevra les inputs, provenant d'Arena, et outputs, provenant de notre programme.

```
Scanner input = new Scanner(System.in);
```

Initialisation du protocole

1. *inputUCI* : Réponse à un input "uci". Envoie des informations sur l'expéditeur (notre IA) afin de légitimer les commandes.

```
System.out.println("id name " + ENGINENAME);
System.out.println("id author Thomartin");
System.out.println("uciok");
```

2. *inputIsReady* : Réponse à un input "isready". Prévient Arena que nous sommes prêt à échanger

```
System.out.println("readyok");
```

Initialisation de la partie

3. *inputUCINewGame* : Réponse à un input “ucinewgame”. On initialise le plateau du jeu, notre classe **Minimax** et notre arbre d’opening. Ce dernier sera expliqué plus tard.

```
board = new Board();
st = new StartTree();
new Minimax();
board.loadFromFen("rnbqkbnr/pppppppp/8/8/8/8/PPPPPPPP/RNBQKBNR w KQkq - 0 1");
```

Le plateau démarre à partir d’un FEN (ou Forsyth–Edwards Notation) d’un plateau de départ.

4. *inputIsReady* : On répète cette instructions pour s’assurer que l’initialisation du jeu s’est faite sans erreurs.

Envoi des coups joués

5. *inputPosition* : Réponse à un input type “position startpos”, il est suivi d’une liste des coups joués “moves [*mouvement 1*] [2]...” si nous sommes en cours de partie. Cette notation correspond à la notation dite SAN (ou Standard Algebric Notation) d’une situation de jeu.

   Cette fonction prépare le plateau pour l’analyse et l’évaluation. Les librairies que nous utilisons partent toujours du plateau initial, et construisent la situation actuelle grâce à la liste des coups effectués.

6. *inputGo* : Réponse à un input “go”, peut être suivi d’options. Cette fonction lance l’analyse du moteur, donc notre Minimax.

   Il répondra par l’output “bestmove [*mouvement*]” qui sera joué par Arena.

Fin du programme

7. *inputQuit* : Envoie un *System.exit(0)* permettant de clore le programme correctement.

Implémentation

Phases de la partie

Afin de faciliter l'implémentation de certaines fonctions, nous avons déterminé quatre phases à une partie.

Gamephase 0 - Opening phase

Tout d'abord, la phase d'ouverture. Lors de cette phase, nous n'utilisons pas le MiniMax, mais le *StartTree*. Il s'agit d'un arbre recensant les coups les plus communs et la réponse que nous souhaitons effectuer.

Gamephase 1 -Midgame

Cette phase se déclenche lorsque le *StartTree* n'a plus de coups à répondre. Elle dure la majorité de la partie, et est sujette à toutes les fonctions d'amélioration.

Gamephase 2 -Endgame

L'*endgame* se déclenche lorsqu'il reste 6 pièces majeures (excepté les rois). A partir de ce moment, certains élagages sont enlevés, et la fonction d'évaluation change.
Cette dernière est très concentrée sur le matériel, et si notre moteur à un avantage à ce stade, il va utiliser cet avantage pour petit à petit réduire les pièces de l'adversaire jusqu'à ce qu'il n'en reste plus qu'une : le Roi.

Gamephase 3 -Solitary King

C'est pour cette raison que nous avons déclaré cette dernière phase, où il ne reste qu'un Roi d'un côté. Nous modifions la fonction d'évaluation à ce stade, afin de favoriser l'obtention d'un échec et mat. Tous les élagages "incertains" sont enlevés afin d'être sûr de ne pas rater un mat, et nous lançons le MiniMax à profondeur 7 à ce stade. De plus, si il reste 4 pièces ou moins à la couleur opposée, nous lançons le MiniMax avec une profondeur de 9.

Fonction d'évaluation

Afin de pouvoir prendre des décisions sur les situations de jeu, nous devons être capable de les juger, nous avons donc besoin d'une fonction d'évaluation. Nous avons fait plusieurs itérations de cette fonction car il existe plusieurs niveaux de complexité, prenant chacun en compte différentes variables du jeu.

Tapered Evaluation

Pour notre fonction d'évaluation, nous nous sommes inspirés du principe de *Tapered Evaluation*, qui consiste à modifier l'évaluation **en fonction de l'avancement** de la partie. Nous avons donc appliqué des poids aux trois types d'évaluations que nous implémentons en fonction de la phase de la partie où nous sommes.

Evaluation matérielle

Valeur des pièces

Cette version de l'évaluation repose sur le "matériel" de la partie, c'est-à-dire l'état des pièces (éliminées ou non).

Une première approche peut se baser uniquement sur une différence des pièces éliminées, cela veut dire que le joueur *Max* aura une meilleure position s' il a éliminé plus de pièces que *Min*. Cependant, on remarque rapidement que cette approche est trop naïve, en effet, avec cette méthode, *Max* aura l'avantage avec 3 pions éliminés alors que *Min* a éliminé la reine. Il faudrait donc pondérer la valeur des pièces car une reine est une perte bien plus lourde que 3 pions.

La deuxième approche intégrera donc les coefficients suivants :

Roi	Reine	Tour	Cavalier	Fou	Pion
20000	900	500	330	320	120

Ces valeurs viennent du principe *Centipawn*, qui consiste à "gonfler" les valeurs traditionnelles attribuées aux pièces aux échecs afin de régler d'autres coefficients de façon plus efficace. Habituellement, ces valeurs sont de l'ordre de *+∞*, 9, 5, 3 et 1 respectivement, mais avec ce principe, on peut plus facilement situer des appréciations comme "cela vaut un peu moins qu'un pion" ou "cela vaut presque 2 pions" sans avoir à utiliser de fractions.

Si nous reprenons l'exemple précédent, *Max* aura un score de $3 * 120 = 360$ et *Min* $1 * 900 = 900$. L'évaluation est alors plus réaliste.

Piece Square Tables (PST)

Afin de **pondérer** les valeurs associées aux pièces, nous avons utilisé des *PST* pour chacune des pièces. Ce sont des tableaux contenant 64 entrées, une pour chaque case de l'échiquier. Pour chaque case est donc défini une valeur - bonus ou malus - qui va venir s'ajouter à la valeur de la pièce. Afin d'être le plus exhaustif possible, nous avons utilisé deux PST par pièce, une pour le *midgame* et une pour l'*endgame*.

On peut donc voir sur la figure ci-contre, représentant la PST du cavalier pour le *midgame*, que l'on valorise les positions centrées sur l'échiquier, qui contrôlent ainsi plus de cases, et que l'on pénalise les positions très excentrées.

```
public static List<Integer> knightMiddleTable = Arrays.asList(
        -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, -30, -30, -30, -30, -40, -50);
```

Evaluation des pions

En addition à l'évaluation matérielle, nous souhaitions également préciser la position des pions, afin qu'à matériel égal certaines positions soient favorisées. Ainsi, nous **pénalisons** légèrement trois types de pions :

- les pions doublés, c'est-à-dire les pions qui se suivent sur la même colonne.
- les pions bloqués, c'est-à-dire les pions qui ont une pièce qui les empêche d'avancer.
- les pions isolés, c'est-à-dire les pions qui n'ont pas d'autres pions sur une case adjacente à la leur.

De cette façon, nous ne modifions pas suffisamment l'évaluation pour influencer le matériel, mais à matériel égal nous favorisons des positions plus avantageuses.

Evaluation de contrôle

Bien que l'évaluation matérielle représente le cœur de l'évaluation, afin de préciser et de nuancer des positions qui contiendrait le même matériel, nous effectuons également une évaluation de contrôle. Cela consiste à vérifier quelle couleur "contrôle" chaque case de l'échiquier : si les blancs ont plus de pièces pouvant aller sur cette case, on considérera que les blancs contrôlent cette case. Il s'agit en quelque sorte d'un *Static Exchange Evaluator*[2], modifié pour concerner toutes les cases.

Lors du *midgame*, nous appliquons un poids de 10 à cette évaluation. Concrètement, si une couleur contrôle 12 cases de plus que l'autre (par exemple les blancs contrôlent 38 cases et les noirs en contrôlent 26), cet avantage correspond à la valeur d'un pion. Les noirs seraient donc prêts à sacrifier un pion pour rééquilibrer le contrôle des cases à ce stade de la partie.

[2] https://www.chessprogramming.org/Static_Exchange_Evaluation

Evaluation de fin de partie

En fin de partie, c'est-à-dire lors des phases 2 et 3, l'évaluation de contrôle est baissée car elle importe désormais peu.

De plus, lors de la phase 3 où il ne reste plus qu'un Roi d'un côté, nous oublions les évaluations de contrôle et de pion. L'évaluation matérielle est également modifiée en appliquant une PST spéciale au Roi solitaire. Celle-ci pénalise ce Roi pour les positions de bord d'échiquier, de façon à ce que ce dernier soit poussé vers ces positions délicates par le côté qui domine. De cette façon, couplée à l'augmentation de la profondeur du MiniMax, il y a plus de chances que notre moteur voit une position de mat et ainsi clôture la partie.

Lazy Evaluation

Bien que nous ayons accordé beaucoup d'importance à une évaluation précise des situations, les précisions que nous apportons à l'évaluation matérielle sont lourdes en termes de calcul. Ainsi, lors d'évaluations pour lesquelles nous souhaitons avoir une idée de la valeur d'une position et non la valeur exacte, nous utilisons uniquement l'évaluation matérielle.

De plus, même pour les positions qui nécessitent une valeur exacte, si nous voyons que la valeur de l'évaluation est loin de la valeur d'alpha (ou de bêta respectivement pour max et min), étant donné que les précisions ne vont pas modifier drastiquement la valeur de l'évaluation, nous n'effectuons pas ces précisions.

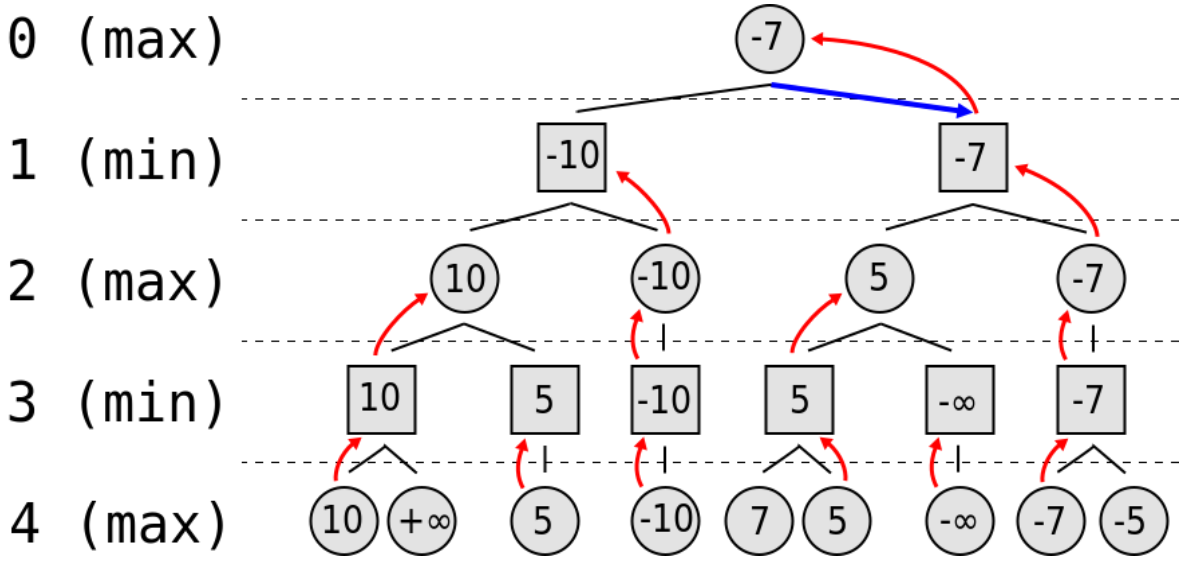
Recherche

Minimax

Minimax est un algorithme d'**exploration adverse** très utilisé dans la théorie des jeux. Il sert à minimiser la perte ou la dégradation d'une situation de jeu. Il repose sur le fait de faire évoluer "virtuellement" une situation de départ et imaginer toutes les réponses possibles du joueur adverse sur une profondeur donnée.

Un joueur retiendra la plus grande valeur (le joueur *Max*) et l'autre la plus petite (*Min*), grâce à cela, chaque joueur remonte la situation qui l'avantage le plus. La meilleure est ensuite remontée jusqu'à la racine, et cela nous indiquera le prochain coup à jouer.

De par ce système de perspective, il est implicite que l'algorithme considère que l'adversaire joue toujours le meilleur coup. Cela peut dégrader la performance du moteur s'il joue contre des humains, qui ont tendance à ne pas jouer les meilleurs coups.



Source de l'image : https://en.wikipedia.org/wiki/Minimax

On peut également comprendre que dans le cas du jeu d'échecs, l'arbre grossit rapidement à cause de toutes les possibilités que l'on peut avoir chaque tour. La majorité du travail de ce projet va donc être de chercher à améliorer le Minimax afin de réduire le champ de recherche et le temps d'exécution.

Quiescence

La recherche par Quiescence, ou *Quiescence Search*, est une méthode d'exploration qui permet de résoudre le problème de l'**effet d'horizon**.

Ce problème est lié à la grande complexité des arbres de décisions des jeux comme les échecs ou le go. Nous ne pouvons pas parcourir un arbre de jeu dans son intégralité, à cause de sa largeur et profondeur importante, nous devons donc limiter notre exploration. Cependant, un désavantage apparaît, brider notre profondeur d'analyse nous rend aveugle aux réponses que nous aurions pu étudier à l'itération d'après. Une manière de remédier à cela est de détecter si une feuille de l'arbre est dite "instable", et lancer une recherche supplémentaire pour décider de la valeur réelle de cette feuille.

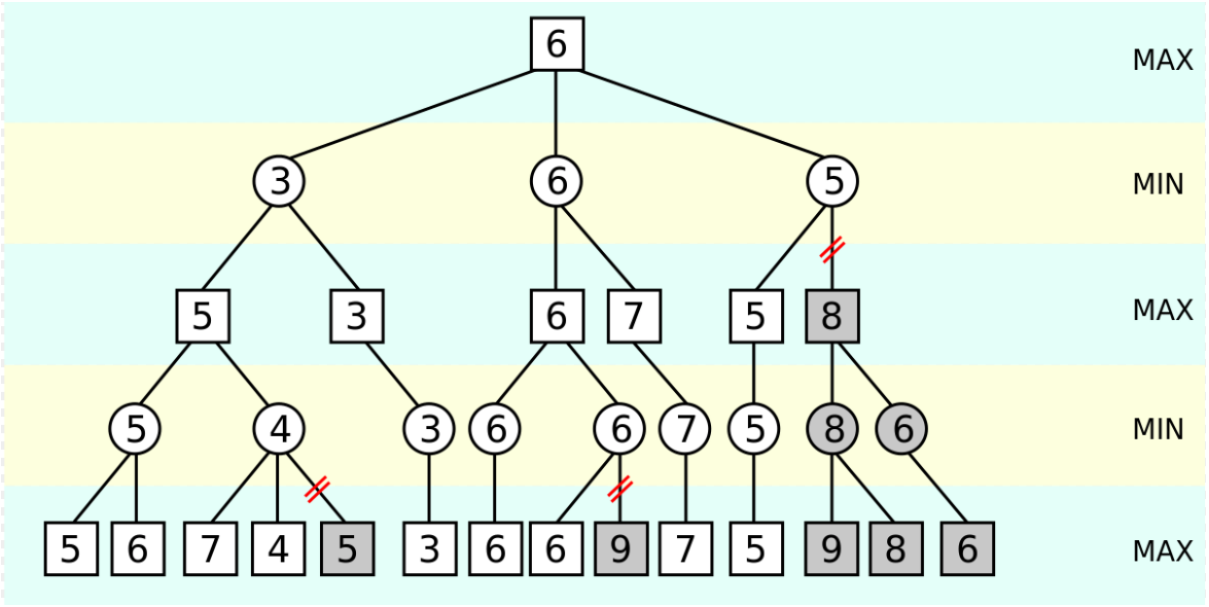
Dans notre cas, une feuille instable est une feuille dont le coup précédent était une capture. Sur ces feuilles, nous lançons une *Quiescence Search*, et nous explorons donc encore un peu cette branche de l'arbre - en ne considérant uniquement les captures afin de limiter le facteur de branchement. Concrètement, si nous finissons l'exploration de l'arbre MiniMax par une capture à l'aide d'une Dame, cette recherche supplémentaire permet de s'assurer que cette Dame ne va pas être capturée juste après par une pièce couvrant celle que nous avons capturée.

Comme on peut le deviner, cette méthode est très gourmande en ressources, elle sert donc principalement à améliorer la qualité des décisions de notre moteur. Afin de limiter cette recherche (qui était jusque 10 fois plus coûteuse que le MiniMax en profondeur 5 lorsque débridée), nous lui imposons une profondeur maximale de 3.

Optimisation

Élagage alpha-bêta

Cette optimisation est généralement toujours implémentée avec le Minimax de base. Cela est dû à la facilité d'implémentation de l'algorithme, et du gain significatif en performance qu'il apporte. Il repose sur une logique mathématique de comparaison. On fait en sorte d'élaguer les branches de l'arbre qui n'améliore pas la valeur[3] du gain pour le joueur.



Source : https://www.ericdrosado.com/2018/01/02/alpha-beta-pruning-in-minimax.html

En prenant l'exemple présent sur l'image, nous pouvons voir que cet algorithme est capable de couper des sous-arbre très important, cela explique le gain important en performance.

[3] Il est implicitement considéré que le terme "améliorer" est relatif à la perspective du joueur.

Tri des coups

L'alpha bêta se repose sur des comparaisons, il est donc possible d'augmenter le nombre de branches élaguées (et donc la performance) en faisant en sorte d'évaluer **le plus tôt possible** les positions prometteuses. De cette façon, les seuils alpha et bêta auront pour référentiel un coup intéressant et les positions suivantes, moins intéressantes car nous avons ordonné notre liste de coup, seront éliminées.

Bien que nous utilisons des valeurs, elles sont uniquement utilisées pour trier les coups et n'ont aucune incidence sur l'évaluation d'une position : un coup ayant une valeur de 12 sera exploré avant un coup ayant une valeur de 4, qui sera lui-même exploré avant un coup ayant une valeur de -1.

Most Valuable Victim/Least Valuable Attacker (MVV/LVA)

Les coups capturants sont analysés à l'aide du tableau *MVV/LVA*, qui renvoie une valeur en fonction de la pièce qui prend et de celle qui est capturée. Ainsi, si la victime est la Dame, la valeur renvoyée sera plus élevée que si la victime est un pion.
Ensuite, une fois que nous avons géré la victime, nous allons également trier en fonction de l'attaquant. En effet,il vaut mieux prendre une pièce à l'aide d'un Pion plutôt que d'une Tour, car souvent, on met en danger la pièce qui prend.

```
public static int[][] vic_atk_val = {
        {60, 61, 62, 63, 64, 65, 0}, // victim K, attacker K, Q, R, B, N, P, None
        {50, 51, 52, 53, 54, 55, 0}, // victim Q, attacker K, Q, R, B, N, P, None
        {40, 41, 42, 43, 44, 45, 0}, // victim R, attacker K, Q, R, B, N, P, None
        {30, 31, 32, 33, 34, 35, 0}, // victim B, attacker K, Q, R, B, N, P, None
        {20, 21, 22, 23, 24, 25, 0}, // victim N, attacker K, Q, R, B, N, P, None
        {10, 11, 12, 13, 14, 15, 0}, // victim P, attacker K, Q, R, B, N, P, None
        {0, 0, 0, 0, 0, 0, 0},       // no victim
};
```

Ainsi, ce tableau MVV/LVA prend en compte toutes ces considérations afin de renvoyer une valeur entre 10 (Pion pris par Roi) et 55 (Dame prise par Pion).
Bien que ces valeurs puissent paraître anodines, le minimum à 10 permet de laisser une marge afin de trier également les coups non capturants, tout en les classant moins bien que les capturants.

Promotions

Bien que nous ayons dit que les coups capturants étaient les mieux classés, ce n'est pas tout le temps vrai : les promotions renvoient la valeur de la pièce promue. Ainsi, une promotion en Dame sera explorée avant une promotion en Cavalier, qui sera tout de même exploré avant tous les autres coups.

Rock

En troisième position après les promotions et les coups capturants se trouvent les rocks. Ce sont des coups que nous avons jugé nécessaire de mettre en avant car notre MiniMax explorait jusqu'à 7 fois plus de nœuds lorsque c'était le meilleur coup disponible, nous leur avons donc attribué une valeur de 4.

Killer Heuristic

La *Killer Heuristic* se base sur le concept de *Killer Move*, qui est un coup qui a causé un élagage. Concrètement, si un coup joué menace une Dame, ce coup va causer beaucoup d'élagage dans l'arbre : tous les coups qui ne bougent pas la Dame menacée ou ne la protègent pas. Ainsi, pour chaque nouvel appel d'un MiniMax, on en stocke deux par profondeur, et dans le tri des coups, on renvoie une valeur de 3 si un coup est un *Killer Move*.

History Moves Heuristic

Cette méthode est celle qui a été la plus efficace pour trier nos coups non-capturants, tout simplement car elle prend en compte les coups qui ne sont pas "exceptionnels" comme les autres. Simplement, lors de l'exploration du MiniMax, si un coup devient le meilleur coup disponible à n'importe quel moment de l'arbre, la valeur associée à ce coup est incrémentée de 1. Ainsi, les meilleurs coups sont automatiquement mis en avant dans le tri des coups.

Coup avançant

Enfin, si un coup avance (c'est-à-dire monte pour les blancs et descend pour les noirs), il va être attribué la valeur de 1, la plus petite valeur possible. C'est une petite amélioration, mais permet de différencier des coups qui ne sont concernés par aucun cas précédent, et ainsi de mettre en avant par exemple les pions qui avancent.

Mouvement du Roi

Afin d'éviter que le Roi ne bouge alors que ce n'est pas nécessaire, nous avons également attribué une valeur de -1 à tout déplacement du Roi qui n'est pas un rock, afin que ces mouvements soient explorés en dernier.

Tables de transposition

Principe général

La table de transposition est une amélioration qui n'est pas remise à zéro entre les appels de MiniMax. Concrètement, si une position est explorée, nous stockons une représentation de cette position - appelée Zobrist Hash, qui permet de stocker une position et ses informations de façon efficace - et la valeur qui lui est associée. De cette façon, pour chaque position explorée lors du MiniMax, nous vérifions avant si nous ne connaissons pas déjà la position et donc sa valeur, et ainsi s'éviter une exploration coûteuse.

Flags

Il existe trois types de noeuds dans une table de transposition :

- Tout d'abord, les nœuds ayant été explorés entièrement et donc on peut stocker la valeur **exacte**.
- Ensuite, les nœuds dont on ne connaît pas la valeur exacte car ils ont subi un élagage *alpha*. On les stocke donc avec un *flag* alpha, afin de signaler qu'on sait que ce nœud vaut **au moins** la valeur qu'on lui a associé. On vérifie donc que cette valeur est inférieure à l'alpha actuel avant de la réutiliser.
- Enfin, les nœuds dont on ne connaît pas la valeur exacte car ils ont subi un élagage *beta*. On les stocke donc avec un *flag* beta, afin de signaler qu'on sait que ce nœud vaut **au plus** la valeur qu'on lui a associé. On vérifie donc que cette valeur est supérieure au beta actuel avant de la réutiliser.

Gestion de la profondeur

En plus de vérifier le *flag* avant de réutiliser la valeur stockée, il faut également gérer la profondeur : il faut que la profondeur de la recherche actuelle soit **inférieure ou égale** à la profondeur stockée pour qu'elle soit utilisable. Cette vérification est nécessaire car sinon on pourrait réutiliser des valeurs non représentatives car issues d'une exploration moins profonde.

Futility pruning

Le *futility pruning* est appliqué lorsqu'il reste une profondeur d'au plus 3 à la recherche et permet d'élaguer les positions qui n'ont **aucun potentiel**. Pour chaque profondeur est associée une valeur, et si l'évaluation de la position ajoutée à cette valeur est inférieure à l'alpha - ou bien si l'évaluation de la position retirée à cette valeur est supérieure au beta, on considère que cette position ne peut pas améliorer le meilleur coup : on peut ainsi l'élaguer.

Null move pruning

Le *null move pruning* se base sur un concept assez amusant et humiliant : si on joue un coup "vide" - c'est-à-dire qu'on ne joue pas et qu'on laisse donc virtuellement deux coups à l'adversaire - et que ce dernier accuse tout de même d'un retard suffisamment important pour causer un élagage alpha-beta, alors on peut élaguer toute cette partie de l'arbre. Ce postulat est valable uniquement si l'on considère que jouer un coup permet d'améliorer sa position, il faut ainsi vérifier que l'on est pas dans une situation de *ZugZwang* avant de l'appliquer, c'est-à-dire une situation où ne pas jouer est bénéfique.

Delta pruning

Le *delta pruning* est appliqué uniquement à la *Quiescence Search*, il fonctionne grossièrement de la même façon que le *futility pruning*. On attribue à *delta* une valeur fixe, et si l'évaluation d'une position à laquelle on ajoute le delta ne permet pas d'améliorer la meilleure valeur, on considère que cette branche ne vaut pas le coup d'être explorée. Cela permet d'éviter d'explorer des branches **inutiles** de l'arbre, ce qui arrive souvent dans la *Quiescence Search*, qui a pour but d'explorer toutes les captures. Ainsi, avec le *delta pruning*, on n'explore pas les branches où par exemple une Dame prend un Pion et est immédiatement capturée car le déficit est trop important[4].

[4] Nous avons établi notre delta à 1500 (supérieur à la valeur d'une dame), cet exemple est donc purement explicatif.

Performance

Temps d'exécution

La contrainte établie de devoir jouer un coup en moins d'une seconde nous a poussé à beaucoup se concentrer sur l'élagage de l'arbre et la minimisation du nombre de nœuds explorés. Nous nous sommes également rendus compte que notre fonction d'évaluation complète était assez lourde, d'où l'implémentation de la *lazy evaluation* pour les cas où une évaluation précise n'est pas nécessaire.

Cela nous permet d'utiliser un MiniMax de profondeur 5 (augmentée jusqu'à 9 en fin de partie), auquel s'ajoute une *Quiescence Search* de profondeur 3, ainsi qu'une fonction d'évaluation utilisant non seulement une évaluation matérielle mais également une analyse des pions et du contrôle - le tout en moins d'une seconde.

Exploration de noeuds

Évidemment, l'exploration des nœuds varie énormément en fonction de la position. Cependant, à l'exception de positions très compliquées où les élagages sont minimes - car il n'y a ni captures intéressantes possibles, ni des coups donnant un avantage stratégique net - notre moteur n'explore rarement plus de 200 000 nœuds. Afin de conserver un certain contrôle sur le nombre de nœuds explorés (et ainsi sur le temps d'exécution), nous avons défini deux paliers :

- A 400 000 nœuds explorés, nous réduisons à 2 la profondeur de la *Quiescence Search*, et nous augmentons également les valeurs associées au *futility pruning* afin d'élaguer plus agressivement l'arbre.
- A 800 000 nœuds explorés, nous attribuons à nos deux recherches une profondeur de 0, ce qui a pour but d'arrêter l'exploration de l'arbre, qui remonte ainsi les valeurs avec ce qu'il a eu le temps d'explorer jusque là.

Niveau du moteur

Il est compliqué d'estimer le réel niveau de notre moteur. Nous avons effectué des parties contre des ordinateurs sur des sites d'entraînement d'échecs, mais cela n'est pas représentatif d'un vrai joueur. Notre moteur voyant à 5 coups de profondeur - et 8 pour des coups capturants - il est tout de même suffisamment performant pour battre ses créateurs, qui restent tout de même de piètres joueurs manquant cruellement de pratique.

Possibilités d'amélioration

Fin de partie

Notre moteur manque de performance dans les fins de parties, car malgré la stratégie que nous lui avons insufflé à travers la fonction évaluation (capturer toutes les pièces adverses sans en perdre puis forcer le Roi adverse dans un coin), si un mat assuré n'est pas disponible dans son horizon de recherche (soit dans les 9 prochains coups), il n'ira pas dans cette direction. Il faut donc jouer de réussite pour que ce cas se produise.

A part cette grosse faiblesse évidente, le moteur est **complet**, et permet de jouer à un bon niveau toute la partie tout en étant rapide. En conservant l'architecture du MiniMax, et avec des améliorations "standard", nous ne pouvons pas prétendre à plus de profondeur. Ainsi, nous sommes très satisfaits et fiers du travail que nous avons produit, car nous avons le sentiment qu'il est **abouti**.

Negamax, Principal Variation Search

Une amélioration que l'on pourrait apporter serait de s'éloigner du MiniMax pour favoriser un autre algorithme de recherche adverse. Le Negamax est une alternative au MiniMax très populaire, qui consiste à se placer dans la perspective d'un seul joueur au lieu d'alterner.
Pour aller plus loin, le *Principal Variation Search* est un Negamax qui est souvent plus efficace que l'élagage alpha-beta, principalement car il n'examine pas les nœuds élagués. De plus, il fonctionne mieux lorsqu'un bon tri de coups est implémenté, il serait donc judicieux d'appliquer cela à notre moteur.

Parallélisation

Une dernière amélioration très efficace serait de paralléliser notre arbre de recherche. En récupérant la valeur à l'extrême gauche de l'arbre, on peut ensuite répartir le reste de la recherche sur des nœuds parallèles, ce qui permet d'augmenter drastiquement la vitesse de la recherche.

Conclusion

Ce devoir nous a permis de découvrir le principe du Minimax et de son optimisation alpha-bêta, qui sont populaires pour l'exploration adverse. Même si nous l'avions déjà implémenté durant notre parcours universitaire, cela a été très formateur de partir de cette base déjà connue et de la pousser afin qu'elle atteigne son plein potentiel.

Nous avons donc exploré plusieurs techniques d'optimisation, que cela soit en algorithmique ou même en programmation pure (usage modéré de boucles, nombreuses relectures du code pour éviter de dupliquer des actions, etc).

Ce travail, certes complexe mais gratifiant, nous a poussés à aller au-delà du travail demandé pour créer un moteur que nous sommes très fiers d'avoir conçu et qui représente pour nous une belle réussite. Nous souhaitons remercier Monsieur Kevin Bouchard pour son implication dans son cours et sa façon ludique d'amener les choses, notamment avec le concept de compétition qui a su nous motiver pour la durée du projet.