Projet : Intelligence Artificielle de jeu d'échecs

Manhes Benoît, Pattier Lilian, Scalzo Pierre

Manb22079601, Patl21039604, Scap15089506

18 décembre 2018

Université du Québec à Chicoutimi - Maîtrise informatique

Cours 8INF846 : Intelligence Artificielle

Table des matières

Introduction

Ce rapport présente le fonctionnement d'un agent intelligent capable de jouer aux échecs, réalisé dans le cadre du cours d'Intelligence Artificielle 8INF846 de l'Université du Québec à Chicoutimi.

Première partie

Analyse du problème

1 Description du problème

1.1 Jeu d'échecs

Le jeu d'échecs oppose deux joueurs de part et d'autre d'un échiquier composé de soixante-quatre cases blanches et noires. Les joueurs jouent à tour de rôle en déplaçant l'une de leurs seize pièces (blanches pour un joueur, noires pour l'autre). Chaque joueur possède au départ un roi, une dame, deux tours, deux fous, deux cavaliers et huit pions. Le but du jeu est d'infliger à son adversaire un échec et mat, une situation dans laquelle le roi d'un joueur est en prise sans qu'il soit possible d'y remédier.

Nous laissons au lecteur le soin d'aller voir le reste des règles concernant les déplacements, les captures et les conditions de victoires. Nous rappelons cependant les 3 coups spéciaux qui sont parfois omis :

- **Le roque** : Il s'agit d'un déplacement spécial du roi et d'une des tours. Il permet, en un seul coup, de mettre le roi à l'abri tout en centralisant une tour. On distingue le petit roque et le grand roque (voir figure 1) Les conditions de légalité du roque sont les suivantes :
  - Les cases qui séparent le roi et la tour doivent être inoccupées
  - Ni le roi ni la tour ne doivent avoir quitté leur position initiale au cours de la partie
  - Aucune des cases par lesquelles transite le roi ne doit être contrôlée par une pièce adverse

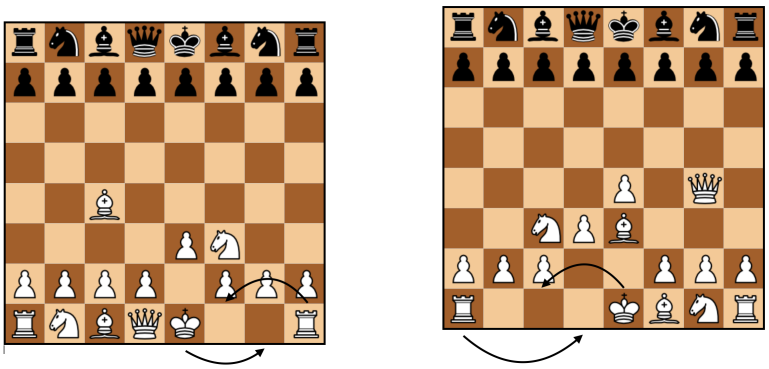

FIGURE 1 – Illustration des petit roque et grand roque

- **La prise en passant** : Il s'agit d'une condition particulière pour capturer un pion. Lorsqu'un pion se trouve sur la cinquième rangée et que l'adversaire avance de deux cases un pion d'une colonne voisine (les deux pions se retrouvent alors sur des cases adjacentes), le premier pion peut prendre le second : le joueur avance son pion en diagonale sur la sixième rangée et la colonne du pion adverse, et ôte ce dernier de l'échiquier. (Ce coup doit être réalisé immédiatement après que l'adversaire ait avancé son pion).
- **La promotion** : Lorsqu'un pion atteint la dernière rangée (adverse), il doit être changé (promu) en une pièce de sa couleur qui peut être au choix une dame, une tour, un fou ou un cavalier.

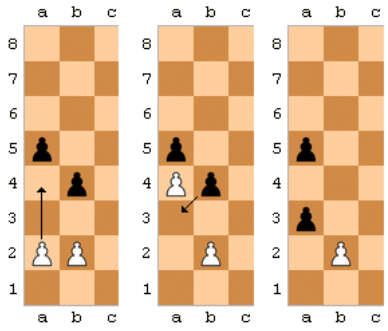

FIGURE 2 – Illustration de la prise en passant

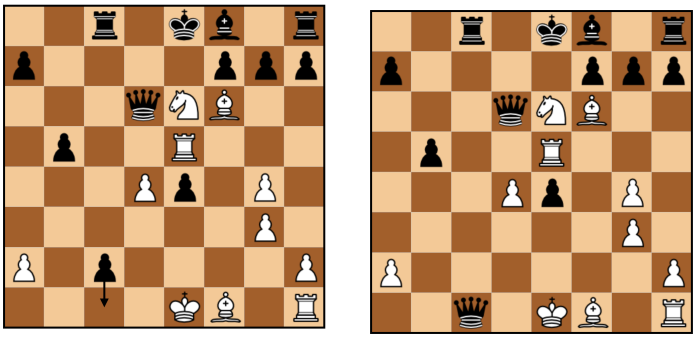

FIGURE 3 – Illustration de la promotion d'un pion en reine

1.2 Objectif du problème

L'objectif de ce projet est la conception d'un agent intelligent capable de jouer aux échecs contre un adversaire (un autre agent ou un humain). Il devra être capable d'observer son environnement et de prendre une décision sur le coup à jouer en fonction de ce dernier.

L'environnement ici est l'échiquier ainsi que les différentes pièces qui le composent. Afin de guider l'agent sur les choix qu'il devra faire, une fonction d'évaluation des coups possible sera mise en place.

Le programme devra être capable de pouvoir communiquer et d'être implémenté sur une interface graphique (GUI) open-source : ARENA. Les frontières sont les suivantes : Le programme prend la forme d'un moteur de jeu qui prend en entrée l'état de l'échiquier au moment de jouer (qui sera communiqué par le GUI), évalue en interne le coup le plus favorable, et enfin transmet au GUI en sortie, le coup qu'il décide de jouer.

Il est à noter que le GUI se contente d'effectuer les mouvements que le moteur lui transmet. C'est donc au moteur de s'assurer qu'il connaît l'état réel de l'échiquier, et de s'assurer que les mouvements qu'il envoie sont des mouvements légaux. Il est donc théoriquement possible d'envoyer des mouvements illégaux, le cas échéant, ARENA le détecte et renvoie une exception "illegal move". Dans ce cas, la partie est perdue pour celui qui a effectué le mouvement illicite.

1.3 Formulation du problème

On peut formuler le problème de la façon suivante :

- **État initial** : l'état actuel de l'échiquier ;
- **Fonction de succession** : l'agent peut effectuer un mouvement d'une de ses pièces encore en jeu : le mouvement doit être légal.
- **Test d'état terminal** : Un joueur est en échec et mat : plus de mouvement possible.
- **Fonction d'utilité** : on évalue les états en attribuant un score plus ou moins élevé par une fonction d'évaluation, traduisant ainsi les configurations favorables ou non-favorables de l'échiquier.
- **Arbre de jeu** : arbre d'une profondeur donnée, avec pour noeud les différents états successifs de l'échiquier.

Nous devons également prendre en compte les contraintes suivantes :

- L'IA doit être programmée en langage Java
- L'IA ne doit pas dépasser 20Go d'espace mémoire
- L'IA doit choisir son coup en moins d'une seconde

2 Solution proposée

Notre solution s'est tournée sur l'implémentation d'un agent intelligent qui évalue les configurations de l'échiquier pour déterminer le mouvement le plus favorable qu'il peut jouer.

Nous utilisons pour cela, un algorithme MinMax, avec une profondeur fixée et implémenté avec un élagage alpha-beta de façon à réduire le nombre de noeuds évalués.

L'agent simule ainsi un certain nombre de configurations à partir de l'échiquier courant, et suivant une profondeur fixée, évalue les configurations finales possibles puis choisit le mouvement le plus favorable pour lui en fonction des évaluations réalisées.

2.1 L'agent

L'agent est basé sur l'utilité. Il prend en compte son environnement pour évaluer quelle prochaine action est la meilleure pour satisfaire son but. Ce dernier se traduit ici par un score, attribué en fonction des pièces présentes sur le plateau et de leur position. Nous verrons plus en détail les heuristiques dans la partie "Évaluation des mouvements".

Notre agent a donc plusieurs caractéristiques inhérentes :

- **Autonomie** : aucune action humaine n'est nécessaire à son fonctionnement, il contrôle ses actions seul en fonction de son but ;
- **Habileté sociale** : pas d'interaction avec d'autres IA mais l'agent communique avec son environnement grâce au protocole UCI ;
- **Réaction** : ce même protocole lui permet de lire les informations du plateau avant chacun de ses tours ;
- **Pro-action** : les actions de notre agent sont basées sur un but.

Nous décrirons le protocole UCI plus en détail par la suite.

2.2 L'environnement

2.2.1 Élements constituants l'environnement

L'environnement a ici la particularité d'être pris en charge par Arena, une entité tierce servant d'interface graphique. Grâce au protocole UCI nous sommes néanmoins capables de le capter et d'assumer des actions.

L'environnement est un échiquier classique constitué de 64 cases et de 32 pièces, 16 blanches et 16 noires de plusieurs types et avec des spécificités particulières :

- **Les pions** (8 pièces par équipe) : ils ne se déplacent qu'en avant et d'une unique case. Leur premier mouvement peut les faire avancer de deux cases. Leur attaque consiste en un déplacement en diagonale, prenant la place d'une autre pièce alors ôtée. Ils peuvent également faire l'objet de coup spéciaux comme la prise en passant ou la promotion détaillées en première partie.
- **Les fous** (2 pièces par équipe) : leur déplacement est limité à une direction diagonale, mais n'est pas limité en nombre de cases ;
- **Les cavaliers** (2 pièces par équipe) : leur déplacement est constitué d'un déplacement d'une case suivie d'un de deux cases dans une direction perpendiculaire (ou l'inverse). Ils sont les seuls à pouvoir passer au dessus d'autres pièces ;
- **Les tours** (2 pièces par équipe) : leur déplacement est limité à une direction orthogonale, mais n'est pas limité en nombre de cases ;

- **Les reines** (1 pièce par équipe) : leur déplacement s'effectue en ligne droite orthogonale ou diagonale, sans limite de case ;
- **Les rois** (1 pièce par équipe) : leur déplacement est permis dans toutes les directions mais limité à une case. Il s'agit de la pièce à protéger. Si elle est en danger de se faire ôter par l'adversaire, il est dit "en échec", laissant alors un tour pour le protéger. Si il est impossible de le sauver, alors il y a "échec et mat", la partie se termine et accorde la victoire à l'adversaire.

Bien que cette propriété ne soit pas utile pour notre agent ou pour le fonctionnement général du jeu, un plateau d'échecs présente le plus souvent une alternance de couleur entre chaque case, aidant à distinguer plus rapidement les mouvements admis pour chaque pièce.

Si la couleur du plateau a peu d'importance, la distinction des cases en a. Celles-ci sont alors décrites par un numéro en ordonnées et une lettre en abscisses.

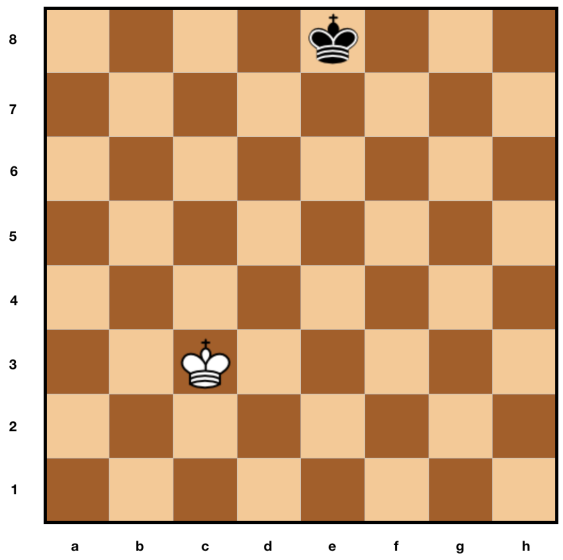

FIGURE 4 – Exemple de distinction

Par exemple sur la figure précédente, le roi blanc se situe à la case c3 et le noir en e8.

De plus, l'environnement admet une limite de temps, déterminée dans ARENA. Ici nous imposons une réévaluation du plateau toutes les secondes, un mouvement devant être effectué par un des moteurs en jeu à tour de rôle dans ce temps imparti.

2.2.2 Propriétés

L'environnement dans lequel évoluera notre agent a certaines propriétés :

- **Complètement observable** : c'est-à-dire qu'avant chaque action, l'agent sera capable d'observer l'ensemble du plateau pour décider de son prochain mouvement ;
- **Stochastique** : le prochain état de l'environnement n'est pas seulement déterminé par son état courant et l'action de l'agent. C'est le cas pour le tour de l'agent, mais ce ne l'est plus lorsque l'adversaire joue, son choix étant imprévisible de façon certaine ;
- **Séquentiel** : tout le but de l'agent est d'évaluer les prochains coups de l'adversaire et de trouver un enchaînement d'actions qui lui seront favorable ;
- **Statique** : l'environnement n'est pas modifié lors de la délibération de l'agent. (Cela serait possible mais notre agent délibère uniquement pendant son tour)
- **Discret** : le jeu se déroule par tours successifs, nombre fini d'actions et de perceptions... ;
- **Multi-agents** : il y a bien entendu deux joueurs, donc deux agents jouant sur le même plateau, donc dans le même environnement.

2.2.3 Evaluation

Afin que l'agent exécute des mouvements favorables à sa victoire, celui-ci est guidé par un but qui se traduit par un score. Ce score est calculé en fonction des pièces restantes sur le plateau et de leur position. L'intérêt est d'accorder plus d'importance à une reine qu'à un pion par exemple, le but ultime étant bien entendu la mise en échec du roi adverse. Le système de points associé à une recherche adversariale est très pratique pour le jeu d'échecs puisqu'ils permettent une évaluation aussi bien pour l'attaque des pièces adverses que pour la défense de ses propres pièces.

Nous avons alors réparti les points de cette façon :

- Pion : 10 points ;
- Cavalier : 30 points ;
- Fou : 30 points ;
- Tour : 50 points ;
- Dame : 90 points ;
- Roi : 1000 points ;

Ainsi notre agent cherchera à attaquer les pièces avec le plus de points mais également défendre ses pièces pendant le tour de l'adversaire.

D'autre part, des plateaux de points ont été créés dans la classe `Evaluation` permettant d'assigner pour chaque pièce les positions les plus favorables.

2.3 Stratégie de recherche

Nous avons choisi d'implémenter un algorithme Minimax pour la recherche du coup optimal à jouer. Nous avons fixé par défaut la profondeur de l'arbre de recherche à 4, offrant ainsi des performances correctes et permettant de respecter la contrainte des 1 seconde de temps de jeu.

Nous avons également implémenté un élagage alpha-beta de façon à améliorer la complexité et accélérer le processus de sélection du meilleur coup à jouer.

Nous détaillerons l'algorithme plus précisément en deuxième partie.

Deuxième partie

Présentation du programme

1 La communication entre le moteur et le GUI

L'implémentation proposée présente deux acteurs majeurs : le moteur et l'interface graphique nommée Arena. Une partie de notre travail a consisté en la communication entre ces deux entités. Nous avons utilisé pour ce faire le protocole UCI (Universal Communication Interface).

Cette communication se fait via la classe `UCI` du package `communication` de notre projet. Plusieurs étapes sont déterminantes dans son implémentation. Premièrement la connexion doit s'établir entre le moteur et le GUI. Cela s'exécute par un ensemble de "Question-réponse" que nous avons satisfait par des méthodes publiques renvoyant des String au système, comme nous pouvons le voir sur la figure suivante :

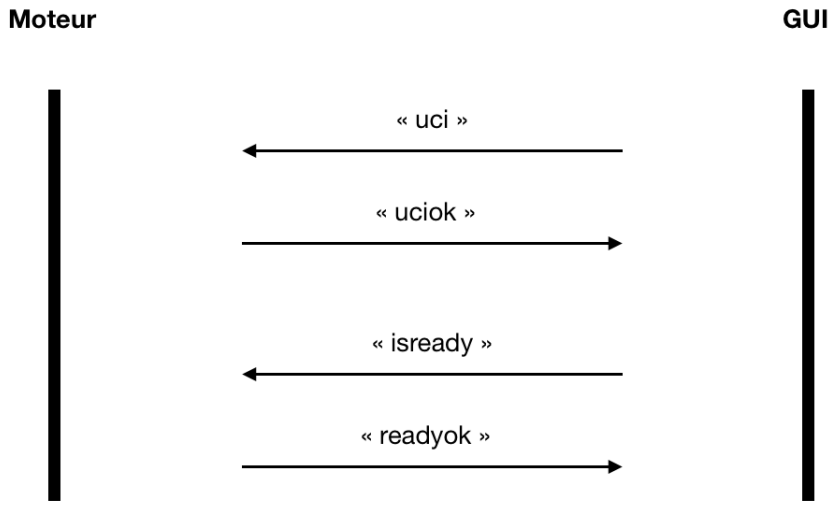

Figure 5 – Protocole de connexion UCI

Notons que la commande `uci` s'exécute une fois en initialisation de la communication, contrairement à `isready` qui peut être appelé autant de fois que le GUI le souhaite.

Viennent ensuite les informations de jeu. Pour cela le GUI envoie la commande `position`, suivi de ses intentions et de l'état du plateau de jeu. Afin d'initialiser ce dernier l'entrée UCI est `startpos`. Elle peut être seule (commande complète étant alors `position startpos`) indiquant au moteur que nous commençons une partie, ou suivie de mouvements avec l'indication `moves`. Ainsi le déplacement d'un pion blanc de la case b2 à b3 sera précisé par la commande complète `position startpos moves b2b3`. Si le coup d'après le joueur noir décide de bouger son cavalier la commande deviendra alors : `position startpos moves b2b3 b8c6`, le GUI renvoyant donc à chaque tour l'ensemble des mouvements depuis le début de la partie. Le moteur a accès à ces données via la méthode `inputPosition(String input)` et applique ainsi les mouvements effectués à notre instance de l'échiquier.

Plusieurs remarques peuvent être faites par rapport aux décisions que nous avons prises à ce point. Tout d'abord, il faut savoir que Arena est robuste aux entrées mal formées ou à la non-prise en compte d'éléments du protocole. Nous avons décidé de lire les informations d'Arena que par la commande `moves`, et donc de rejouer dans notre moteur chaque mouvement à chaque tour. Une autre solution aurait été de lire la commande `importfen` renvoyant l'état du plateau. Ce choix n'a néanmoins pas d'impact sur le jeu ou la complexité, l'environnement étant statique entre chaque tour et la formation du plateau par fen correspondant au final au parcours complet de celui-ci. Plusieurs autres commandes n'ont également pas été prises en compte comme par exemple `options` ou `ponderhit`. Ce dernier aurait permis de transmettre à Arena un pré-choix de mouvement afin d'éviter d'être à court de temps dans notre exploration.

Enfin, la dernière commande prise en compte est `go` et demande au moteur son prochain mouvement. Dans cette méthode nous mettons à jour la variable `move`, renvoyée à travers `go` par `"bestmove "+move`. La

méthode employée est la méthode `inputGo()` de la classe `UCI`. C'est dans cette méthode que notre moteur utilise l'algorithme de Minimax pour envoyer le meilleur mouvement.

2 Représentation de l'environnement

Notre environnement et constitué de l'échiquier et de ses pièces. A cette fin, nous avons défini une classe `Environment` dans le package `environnement` dans laquelle nous allons pouvoir représenter les éléments.

L'échiquier est représenté par l'attribut `chessBoard` de type `String[][]` et de dimension 8×8. Chaque case de ce tableau représente une case de l'échiquier. Suivant la valeur de la case, on peut définir la présence ou non d'une pièce ainsi que son type :

- Les pièces blanches sont représentées par un caractère en majuscule : R,N,K,B,Q,P respectivement pour : Tour, Cavalier, Roi, Fou, Reine et Pion.
- Les pièces noires sont représentées par un caractère en minuscule : r,n,k,b,q,p respectivement pour : Tour, Cavalier, Roi, Fou, Reine et Pion.
- L'absence de pièce est représentée par un espace : " ".

Ainsi l'échiquier de base en début de partie est représenté par le tableau :

```
chessBoard = new String[][]{
                        {"r","n","b","q","k","b","n","r"},
                        {"p","p","p","p","p","p","p","p"},
                        {" "," "," "," "," "," "," "," "},
                        {" "," "," "," "," "," "," "," "},
                        {" "," "," "," "," "," "," "," "},
                        {" "," "," "," "," "," "," "," "},
                        {"P","P","P","P","P","P","P","P"},
                        {"R","N","B","Q","K","B","N","R"}};
```

On dispose justement d'une méthode `initialize()` permettant d'initialiser le plateau et appelée dans le constructeur de l'environnement.

On dispose également de plusieurs méthodes prenant en argument un mouvement et l'appliquant sur l'échiquier :

- La méthode `applyMoveFromArena(String move)` prend en argument un mouvement lu dans Arena pour l'exécuter sur l'échiquier.
- Les méthodes `move(String move)` et `undoMove(String move)` sont utilisées dans l'algorithme Minimax pour simuler des états du plateau. Elle permettent respectivement d'effectuer et d'annuler un mouvement sur l'échiquier.

Nous détaillerons ces méthodes plus précisément par la suite.

Enfin la méthode `gameOver()` renvoie un booléen qui spécifie selon le plateau si l'un des joueurs est en situation d'échec et mat.

3 Mouvements

3.1 Représentation algébrique des mouvements

Comme nous l'avons spécifié précédemment, nous devons être en mesure de lire les mouvements déjà effectués et envoyés par Arena à notre agent (via la commande `position startpos moves ...`). Nous devons également être en mesure d'envoyer le mouvement à Arena via la commande `bestmove ...`.

Les mouvements traités par Arena sont des `String` de longueur 4 (ou 5 s'il s'agit d'une promotion). Ils sont écrits en notation algébriques : les 8 colonnes sont « numérotées » de gauche à droite par les lettres minuscules

allant de a à h et les 8 rangées numérotées de 1 à 8. Une case est donc représentée par l'intersection d'un chiffre d'une lettre. Un mouvement est représenté par la case de départ et la case d'arrivée de la pièce.

Ainsi par exemple, le mouvement de deux cases du second pion blanc en partant de la gauche en début de partie est noté : `b2b4`. Dans le cas d'une promotion on envoie (ou on reçoit) d'ARENA un mouvement de la forme `case départ - case arrivée - pièce promu`. Par exemple : `e7e8q`.

3.2 Représentation interne des mouvements

De façon à traiter de manière simple les mouvements sur notre instance du plateau (qui est un tableau de String), nous avons pris la décision d'utiliser notre propre notation. Un mouvement et donc représenté pour notre IA sous la forme d'un string de taille fixe de 6 caractères qui sont les suivants :

`xDepart yDepart xArrivee yArrivee pieceCapturee piecePromue`.

Les coordonnées sont les coordonnées réelles du tableau de String. Par exemple la case `b2` en coordonnées algébriques correspond à la case `61` dans notre notation.

Les caractères `pieceCapturee` et `piecePromue` peuvent être éventuellement impertinent (dans le cadre d'un mouvement classique), dans ce cas il sont égaux à un espace de façon à toujours respecter la taille de 6 caractères.

S'il s'agit d'une capture, le caractère `pieceCapturee` est égal à la pièce capturée (N,B,q,p,...) en majuscule ou minuscule suivant la couleur de la pièce capturée.

S'il s'agit d'une promotion, le caractère `piecePromue` est égal à la pièce promu (R,N,Q,B) ou (r,n,q,b) suivant la couleur du pion initial.

Voici des exemples de mouvements dans notre notation : `"6344␣␣"` (mouvement d'un cavalier), `"4132b␣"` (mouvement avec capture d'un fou noir), `"1303␣Q"` (mouvement et promotion d'un pion blanc en reine)...

3.3 Conversion des types de mouvements

Nous avons ainsi implémenté des méthodes dans la classe `Environment` permettant de convertir les mouvements algébriques reçus et envoyés à ARENA en mouvements traités en interne par notre IA :

La méthode `parseAmoveToCBmove(String move)` prend en argument un mouvement au format algébrique (envoyé par ARENA) et le convertit en mouvement au format interne.

La méthode `parseCBmoveToAmove(String move)` fait l'opération inverse et renvoie le string au format algébrique.

Les méthodes `move(String move)` et `undoMove(String move)` cités dans la partie précédente prennent en argument un mouvement au format interne et l'effectue (ou l'annule) sur l'attribut `chessBoard`. Elles sont utilisées par Minimax pour simuler des configurations de l'échiquier.

La méthode `applyMoveFromArena(String move)` prend en argument un mouvement algébrique et l'effectue sur l'échiquier. (C'est cette méthode qui est utilisée pour initialiser l'échiquier en appliquant tous les mouvements effectués jusqu'à là et envoyés par ARENA à chaque tour) Cette méthode prend également en compte la possibilité de lire et d'appliquer un roque au `chessBoard`. Or le roque est le seul coup spécial aux échecs qui implique le mouvement de deux pièces. Il doit donc être traité différemment. Par exemple le petit roque blanc est codé par le mouvement `e1g1`. Cela correspond au vu de ce qu'on a dit plus haut au mouvement du roi de deux cases vers la droite... mais dans le cadre du roque, il faut également changer la position de la tour. A cette fin, nous avons créé deux méthodes :

La méthode `isCastlingMove(String move)` renvoie un booléen égal à `true` si le mouvement au format interne qu'elle prend en argument correspond à un mouvement de roque.

La méthode `makeCastlingMove(String move)` effectue le mouvement de roque au format interne qu'elle prend en argument sur le `chessBoard`.

Récapitulons le fonctionnement de la méthode `applyMoveFromArena(String Move)` :

1. La méthode prend en argument un mouvement au format algébrique.
2. Elle le convertit au format interne grâce à la méthode `parseAmoveToCBmove(String move)`
3. Elle vérifie s'il s'agit d'un mouvement de roque grâce à la méthode `isCastlingMove(String move)`
4. Si c'est le cas elle l'effectue grâce à la méthode `makeCastlingMove(String move)`
5. Sinon il s'agit d'un mouvement classique qu'elle effectue également sur le `chessBoard`

3.4 Mouvements légaux

Comme mentionné précédemment, ARENA se contente d'appliquer les mouvements que notre moteur envoie. C'est donc à notre IA de savoir, en fonction de l'état de l'échiquier, quels mouvements sont légaux et peuvent donc être envoyés à ARENA. Le traitement des mouvements possibles se fait dans la classe `Moves`.

La méthode principale est la méthode `legalMoves(Environnement board, boolean isWhite)`. Cette méthode prend en argument l'environnement `board` (un état donné de l'échiquier) et un booléen `isWhite` caractérisant la couleur du joueur. Elle renvoie la liste des mouvements légaux pour le joueur correspondant, à partir de la position de l'échiquier donnée en argument.

Pour cela, la méthode `legalMoves` a recours à 6 sous-fonctions :

- `legalMovesPawns(board, isWhite, i, j)`
- `legalMovesRook(board, isWhite, i, j)`
- `legalMovesKnight(board, isWhite, i, j)`
- `legalMovesBishop(board, isWhite, i, j)`
- `legalMovesQueen(board, isWhite, i, j)`
- `legalMovesKing(board, isWhite, i, j)`

Ces méthodes renvoient la liste des mouvements possibles d'un certain type de pièce aux coordonnées `i,j` pour le joueur correspondant. `legalMoves` parcourt alors l'échiquier `chessBoard` en testant la valeur de chaque case et en appelant l'une des 6 méthodes correspondante, ajoutant ainsi les mouvements possibles pour le type de pièce donné à la liste des mouvements légaux.

Détaillons maintenant le fonctionnement interne de l'une des 6 sous méthode. Bien qu'il soit aisé de coder les mouvements possibles en fonction du type de pièce (ce qui est implémenté dans chacune de ses 6 méthodes), il faut cependant prendre garde à la mise en échec. Il faut notamment s'assurer :

1. Que le déplacement d'une pièce ne met pas notre propre roi en échec
2. Que le déplacement de notre roi lui-même ne le met pas en échec

Pour cela, les 6 sous méthodes fonctionnent de la manière suivante :

1. Établit un mouvement possible en fonction du type de pièce
2. Effectue le mouvement sur le plateau
3. Vérifie si on est en situation d'échec
4. Si oui, le mouvement n'est pas ajouté à la liste des mouvements possibles
5. Si non, le mouvement est légal : il est ajouté à la liste des mouvements possibles.
6. Annule le mouvement effectué sur le plateau
7. Établit un nouveau mouvement possible... etc

Ainsi, pour chaque coup, on simule le mouvement sur le plateau et on vérifie que le mouvement n'induit pas un échec pour notre roi.

Cette partie simulation se fait à l'aide de la méthode `simulateMoveForKingCheck(Environment board, boolean isWhite, int i, int j, int ni, int nj)` qui simule un mouvement de la pièce de coordonnées (i, j) à la case (n_i, n_j) et vérifie si le mouvement de cette pièce induit un échec en renvoyant `false` le cas échéant.

Pour ce faire, cette méthode fait elle même appel à une autre méthode `kingCheck(Environment board, boolean isWhite)` qui en fonction de la position du roi, va respectivement regarder les possibilité de mise en échec :

- Par les pièces à mouvements droits (Reine, Tours)
- Par les pièces à mouvements diagonaux (Reine, Fous)
- Par les cavaliers
- Par les pions

Ces 4 conditions sont testées par les méthodes :

```
checkByDiagonallyMovingPieces(board, isWhite, kingPosI, kingPosJ)
checkcheckByAlongRankMovingPieces(board, isWhite, kingPosI, kingPosJ)
checkByKnights(board, isWhite, kingPosI, kingPosJ)
checkByPawns(board, isWhite, kingPosI, kingPosJ)
```

qui renvoient des booléens a `true` en cas de situations d'échec.

4 Évaluation des mouvements

De façon à permettre à l'algorithme de recherche du meilleur mouvement de fonctionner adéquatement, nous avons mis en place une fonction d'évaluation d'un échiquier dans une configuration donnée. Cette méthode `evaluate(Environment board, boolean isWhite)` est située dans la classe `Evaluation`. Cette évaluation est utilisée par notre algorithme MinMax afin d'étiqueter les noeuds de façon à choisir le prochain mouvement optimal.

Un score total par joueur est calculé selon deux critères :

1. Les points associés aux nombres de pièces du joueur
2. Les points associés à la position des pièces

Notons l'ensemble : $Pieces = \{tour, fou, cavalier, reine, pions, roi\}$

Le point numéro 1 est calculé suivant la formule suivante :

$$PointNbPieces = \sum_{k \in Pieces} nombreDePiecesDeType(k) * valeurDeLaPiece(k)$$

Les valeurs de chaque type de pièces étant spécifiées à la section 2.2.3.

Le point numéro 2 est calculé suivant la formule suivante :

$$PointPositionPieces = \sum_{k \in Pieces} M_k(coordonneePiece)$$

Avec M_k la matrice de position de la pièce de type k. (Voir annexe). Ces matrices de positions M_k permettent d'accorder des points en fonction de la position de la pièce sur l'échiquier. Si par exemple la pièce est positionnée en (i, j), on lui attribut la valeur du coefficient (i, j) de sa matrice de position. Un coefficient très négatif traduit une position défavorable pour la pièce en question tandis qu'un coefficient fortement positif indique une position à prioriser.

Ces scores sont attribués à l'aide des méthodes :

```
calculatePointsPiecesWhite(analyse)
calculateBlackPiecesPoints(analyse)
calculateWhitePositionPoints(analyse)
calculateBlackPositionPoints(analyse)
```

Et des matrices de positions

`kingPointsAccordingToPos`
`queenPointsAccordingToPos`
`rookPointsAccordingToPos`
`bishopPointsAccordingToPos`
`knightPointsAccordingToPos`
`pawnointsAccordingToPos`

On dispose ainsi d'une heuristique permettant de hiérarchiser les pièces en fonction de leurs valeurs et de leurs positions.

Chaque joueur dispose ainsi d'un score :

$$ScoreBlanc = PointNbPiecesBlanc + PointPositionPiecesBlanc$$

$$ScoreNoir = PointNbPiecesNoir + PointPositionPiecesNoir$$

On peut donc calculer la valeur relative du score pour chaque joueur :

$$ScoreRelatifBlanc = ScoreBlanc - ScoreNoir$$

$$ScoreRelatifNoir = ScoreNoir - ScoreBlanc$$

5 Algorithme de recherche

L'algorithme de recherche du meilleur coup possible est un MinMax avec élagage alpha-beta. La profondeur est fixé à 4 mais peut-être modifiée en fonction des besoins. Il est situé dans la classe `MinMax` du package `algorithm` et est appelé dans la fonction `inputGo()` de l'UCI pour envoyer le meilleur mouvement.

De façon à toujours pouvoir fournir un mouvement à l'UCI même en cas de non-terminaison de l'algorithme dans les temps, nous stockons le meilleur mouvement courant dans la variable statique. `bestMove`.

Troisième partie

Bilan

1 Améliorations possibles

Une première amélioration tiendrait en la prise en compte des mouvements spéciaux du petit-roque et du grand-roque. Nous arrivons actuellement à les lire, mais ne les avons implémentés pour que l'agent puisse les jouer. De même que pour la prise en passant.

Ensuite, un avantage majeur au niveau complexité en début de partie serait de créer un dictionnaire d'ouvertures, basculant ensuite sur notre algorithme de recherche. Cela permettrait de fixer une plus grande profondeur, quelques pièces du plateau étant déjà ôtées lors de l'ouverture, l'arbre de recherche étant ainsi diminué pour la suite.

Une optimisation de la complexité temporelle, au détriment de la complexité spatiale, pourrait se faire au niveau de l'évaluation en mémorisant les états de plateau, et donc en considérant la variation du score uniquement due au dernier mouvement.

Enfin, nous pourrions déterminer un changement de stratégie en cours de jeu. Par exemple, lorsqu'il y a peu de pièces restantes, le roi devrait préférer le centre du plateau plutôt que le bord comme au début du jeu. Pour cette amélioration un booléen `ENDGAME` pourrait être mis en place, avec une matrice de position pour le roi dans cette phase de jeu. Il faudrait alors estimer le meilleur moment pour passer le booléen à `true` pour basculer le mode de jeu.

2 Impressions

Nous avons éprouvé quelques difficultés au début pour la communication avec Arena par le protocole UCI. Une fois cette étape passée nous nous sommes particulièrement concentrés sur la modélisation de l'environnement par notre moteur, la lecture des informations d'Arena et le codage des mouvements légaux de chaque pièce.

Au final, nous n'avons pas réalisé un agent particulièrement puissant, sa profondeur de recherche étant limitée à 4 pour environ 1 seconde suivant la machine de support. Sa force est alors due à sa limitation de mouvements illégaux et à ses heuristiques.

Quatrième partie

Annexes

Matrices de positions

Les matrices de positions ont été choisies telles que suggérées sur le site ChessProgramming :

https://www.chessprogramming.org/Simplified_Evaluation_Function

```
        public static int[][] kingPointsAccordingToPos = {
                        {-30, -40, -40, -50, -50, -40, -40, -30},
                        {-30, -40, -40, -50, -50, -40, -40, -30},
                        {-30, -40, -40, -50, -50, -40, -40, -30},
                        {-30, -40, -40, -50, -50, -40, -40, -30},
                        {-20, -30, -30, -40, -40, -30, -30, -20},
                        {-10, -20, -20, -20, -20, -20, -20, -10},
                        { 20,  20,   0,   0,   0,   0,  20,  20},
                        { 20,  30,  10,   0,   0,  10,  30,  20}};

        public static int[][] queenPointsAccordingToPos = {
                        {-20, -10, -10, -5, -5, -10, -10, -20},
                        {-10,   0,   0,  0,  0,   0,   0, -10},
                        {-10,   0,   5,  5,  5,   5,   0, -10},
                        {- 5,   0,   5,  5,  5,   5,   0, - 5},
                        {  0,   0,   5,  5,  5,   5,   0, - 5},
                        {-10,   5,   5,  5,  5,   5,   0, -10},
                        {-10,   0,   5,  0,  0,   0,   0, -10},
                        {-20, -10, -10, -5, -5, -10, -10, -20}};

        public static int[][] rookPointsAccordingToPos  = {
                        { 0,  0,  0,  0,  0,  0,  0,  0},
                        { 5, 10, 10, 10, 10, 10, 10,  5},
                        {-5,  0,  0,  0,  0,  0,  0, -5},
                        {-5,  0,  0,  0,  0,  0,  0, -5},
                        {-5,  0,  0,  0,  0,  0,  0, -5},
                        {-5,  0,  0,  0,  0,  0,  0, -5},
                        {-5,  0,  0,  0,  0,  0,  0, -5},
                        { 0,  0,  0,  5,  5,  0,  0,  0}};

        public static int[][] bishopPointsAccordingToPos = {
                        {-20, -10, -10, -10, -10, -10, -10, -20},
                        {-10,   0,   0,   0,   0,   0,   0, -10},
                        {-10,   0,   5,  10,  10,   5,   0, -10},
                        {-10,   5,   5,  10,  10,   5,   5, -10},
                        {-10,   0,  10,  10,  10,  10,   0, -10},
                        {-10,  10,  10,  10,  10,  10,  10, -10},
                        {-10,   5,   0,   0,   0,   0,   5, -10},
                        {-20, -10, -10, -10, -10, -10, -10, -20}};

        public static int[][] knightPointsAccordingToPos = {
                        {-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}};
```

```
public static int[][] pawnPointsAccordingToPos = {
                {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,  25,  25,  10,  5,  5},
                {0,   0,   0,  20,  20,   0,  0,  0},
                {5,  -5, -10,   0,   0, -10, -5,  5},
                {5,  10,  10, -20, -20,  10, 10,  5},
                {0,   0,   0,   0,   0,   0,  0,  0}};
```