RAPPORT FINAL

Jeu d'échecs

L'intégralité du code est disponible à l'adresse :
https://github.com/stressGC/Java-Chess-Engine

Par

GEORGES COSSON
AUGUSTIN DE LA TAILLE
ANTOINE DEMON

TABLE DES MATIÈRES

I. REPRÉSENTATION DES DONNÉES

a. Bitboards

Les bitboards sont des objets représentants les positions des pièces et des mouvements possibles dans les jeux de plateau. Ce format est particulièrement utile aux échecs qui contiennent 64 cases, comme la taille de l'architecture des processeurs modernes, ainsi aucune opération inutile n'est à faire et chaque bitboard rentre exactement dans une place mémoire.

Pour représenter le jeu d'échec, nous utilisons un bitboard pour chaque type de pièce de chacun des joueurs. Un bit "1" dans un bitboard implique la présence d'une pièce sur une certaine case, connue par la position de celui-ci.

Ces bitboards sont également utilisés pour représenter les mouvements possibles, comme les attaques ou encore les défenses.

Le choix de la représentation par bitboard s'explique par leur haute densité d'information, qui implique le peu d'espace mémoire qu'ils utilisent. De plus, en Java, le type natif "long" permet une utilisation plus simple de ceux-ci. Finalement, cette représentation permet de faire des opérations sur tous les bits à la fois avec des opérations binaires, notamment le "shifting" pour obtenir les mouvements possibles.

b. Calcul des déplacements possibles

Nous nous sommes basés sur les tutoriels de Crazy Logic Chess pour calculer les déplacements possibles pour chaque type de pièce.

https://www.youtube.com/channel/UCmMjMHTeUEBJJZhxix-N-yg

II. ALGORITHMES

La majorité des problèmes d'intelligence artificielle appliqués aux jeux où l'ont joue contre un adversaire sont résolus par l'algorithme Minimax où l'un de ses dérivés.

a. Minimax

Le Minimax est un algorithme de recherche permettant de connaitre le prochain coup qui minimisera la perte maximum. Il passe en revue toutes les possibilités du jeu et leur assigne un score calculé grâce à une heuristique. Ce score prend en compte les bénéfices du coup pour le joueur et son adversaire. Le meilleur choix est alors celui qui minimise les pertes du joueur tout en supposant que l'adversaire cherche au contraire à les maximiser.

b. Elagage Alpha-Beta

L'élagage alpha-beta permet d'optimiser grandement l'algorithme Minimax sans en modifier le résultat. Pour cela, il ne réalise qu'une exploration partielle de l'arbre. Lors de l'exploration, il n'est pas nécessaire d'examiner les sous-arbres qui conduisent à des configurations dont la valeur ne contribuera pas au calcul du gain à la racine de l'arbre. Dit autrement, l'élagage alpha-beta n'évalue pas des nœuds dont on est sûr que leur qualité sera inférieure à un nœud déjà évalué. Cet élagage permet de conserver plus de place mémoire et donc de chercher plus en profondeur dans l'arbre.

c. Principal Variation Search

Nous avons utilisé l'algorithme Principal Variation Search (ou NegaScout) dans notre moteur d'échecs. C'est un algorithme de la famille de Negamax, basé sur l'algorithme Minimax. Son avantages est de trier les noeuds et de partir du principe que le premier noeud est le meilleur (donc dans la variation principale). Nous avons trié les noeuds en fonction de leur heuristique, dans l'ordre croissant si c'est un noeud minimum et dans l'ordre décroissant si c'est un noeud maximum.

PVS recherche ensuite le score estimé des noeuds restants grâce à l'algorithme **Zero Window Search**, qui est une version plus rapide de l'élagage alpha-beta car alpha et beta y sont égaux. Si la recherche échoue, alors le premier noeud n'était pas dans la variation principale et la recherche continue comme un algorithme alpha-beta classique.

III. Heuristique

Nous avons pour l'instant implémenté une heuristique simple. Elle consiste à compter le nombre de pièces présentes sur le plateau pour chaque joueur, qu'elle pondère ensuite avec un poids défini pour chaque type de pièce:

- pion = 100
- cavalier = 325
- fou = 330
- tour = 500
- reine = 900
- roi = +inf

Nous faisons ensuite la différence de score entre les joueurs, qui donne le score estimé d'un certain état du jeu, variant de -inf pour "être en échec et mat" à +inf pour "l'adversaire est en échec et mat".

IV. AMÉLIORATIONS POSSIBLES

a. Propreté du code

Le code source du moteur d'échecs est disponible sur GitHub à l'adresse suivante : https://github.com/stressGC/Java-Chess-Engine. Le code peut être optimiser et réécrit de manière à être plus rapide, réutilisable et plus compréhensif. La majorité de celui-ci est cependant commentée et expliquée.

b. Iterative Deepening

Le temps est une des composantes principales des échecs. En effet, bon nombre de compétitions imposent une limite de temps, comme dans notre compétition de fin de semestre.

L'algorithme Iterative Deepening est une bonne solution pour palier ce facteur. Les facteurs limitants de l'algorithme PVS sont la mémoire et le temps de calcul qui est exponentiel. Ainsi, à chaque fois que non allons plus profond dans l'arbre, nous démultiplions le temps de calcul. Iterative Deepening Search permet de lancer l'algorithme PVS tant que le n'est pas révolu tout en augmentant la profondeur de recherche. Ainsi, PVS utilisera le temps maximal aloué pour ses calculs.

c. Heuristiques

Une amélioration possible de notre heuristique serait de la pondérer en fonction de l'emplacement des pièces. Notre IA joue de manière trop agressive, et a tendance à mettre en danger des pions importants comme la reine, en les avançant trop sur le plateau par exemple.

d. Ajout de librairie d'ouvertures / fermetures

Puisque notre IA va rarement plus loin que 4 de profondeur, elle ne calcule pas toujours les coups jusqu'au moment où les premiers pions seront mangés. Ainsi elle joue de manière quasi-aléatoire en début de partie.

Nous pourrions utiliser des librairies d'ouvertures et de fermetures afin d'améliorer ses performances, puisque toutes celles-ci ont été calculées et l'on connait leur valeur stratégiques.