Rapport de projet

Jeu d'échec et intelligence artificielle

Félix Sabatié, Simon Marchand, Mathieu Regnard

Rapport présentant les algorithmes utilisés pour la réalisation de l'intelligence artificelle

3A Informatique
Polytech Lyon

1 Introduction

Lors de ce projet, nous avons choisi au départ d'implementer l'intelligence artificielle (IA) du jeu d'échec avec des techniques de *Machine Learning*. Seulement, par manque de temps nous nous sommes rabattus sur des algorithmes plus classiques.

L'ensemble de notre démarche sera présenté plus amplement dans la suite.

2 Choix du modèle

Pour nous permettre de nous concentrer plus amplement sur l'IA, nous avons décidé de choisir un modèle de jeu d'échec open source. Dans un premier temps, nous avons choisi le langage C++ et avons sélectionné un modèle crée par le site internet cplusplus.com.

Il s'est avéré que ce modèle était fait pour présenter les fonctionnalités avancées du C++. Nous avons donc décidé de continuer avec et d'apprendre plus en profondeur ce langage.

Comme certaines règles n'avait pas été gérées par les créateurs, nous avons du rajouter le roque, et la promotion de pion. Ce code nous a permis de nous familiarisé avec les expressions lambda en C++ ainsi que de découvrir les librairies standard ou STL plus en détails. Le code est orienté objet et chaque pièce héritait d'une fonction virtuelle pure pour la gestion du mouvement. De ce fait, chaque pièce pouvait réimplémenter son mouvement comme il se doit.
Les pièces étaient stockés dans le plateau à l'aide d'un set et étaient donc parcourues avec des itérateurs.

Par la suite, nous avons considéré ce modèle trop complexe et légèrement *lourd* pour un projet de ce genre. De ce fait, nous avons décidé de nous rabattre sur un langage plus familier, le Java. Nous avons trouvé un autre modèle en java, disponible en open source sur Github. Le lien du github se trouvera en annexe. Ce code contenait un package ai avec une intelligence artificielle crée par le créateur. Nous avons donc rajouté notre classe dans ce package et supprimé celle présente originellement.

3 Intelligence Artificielle

3.1 Approche avec du *Machine Learning*

En *Machine Learning* (ML) il existe plusieurs types d'apprentissages : l'apprentissage supervisé (supervised learning), l'apprentissage non supervisé (unsupervised learning) ou encore l'apprentissage par renforcement (reinforcement learning).
Le premier consiste a faire apprendre à notre algorithme à l'aide de données. Par exemple, pour faire un algorithme capable de reconnaître des chiffres écrits à la main, nous lui fournirons une base de données de chiffres écrits, avec le résultat correspondant.
Le second permet de trouver un schéma, un *pattern* dans un ensemble de données et n'est donc pas applicable à notre cas.
Le troisième va apprendre à l'aide d'essais et d'erreurs, appelé en anglais *trial and error*.

Comme la disposition du plateau n'est pas une disposition classique, trouver des données de parties d'échecs aurait été compliqué. Nous somme donc partis pour du reinforcement learning.

3.1.1 Reinforcement Learning

Dans le reinforcement learning, il existe plusieurs types d'algorithmes. nous avons choisi d'utiliser le TD-learning.

Temporal difference Learning

Le temporal difference learning (TD-learning) est une méthode qui mêle la programmation dynamique et les méthodes de Monte Carlo. La programmation dynamique consiste à diviser le problème en sous-problèmes et rend la tâche ainsi plus facile. Les méthodes de Monte Carlo utilisent des échantillonnages aléatoires pour déduire des résultats sur l'ensemble de la population.

Le TD-learning apprend en échantillonant l'environnement selon une certaine politique (*policy*) et approxime l'estimation courrante avec les précédentes estimations. Dans notre cas, l'algorithme apprend en fonction de ces parties précédentes, et devient meilleur avec le temps.

Pour utiliser cet algorithme, on doit disposer de ce que l'on appelle un processus de décision markovien (MDP).
Ce MDP possède les éléments suivants :

- Un ensemble d'états S (fini, dénombrable ou continu). Dans notre cas, un état sera défini par une position du plateau.
- Un ensemble d'actions A (fini, dénombrable ou continu). Ici, ce sera les pièces et la où elles peuvent se diriger.
- Une fonction de transition qui donne la probabilité pour une action a, de passer d'un état s à un état s'. Dans notre cas la fonction de transition sera constante car si l'agent effectue a, il passera nécessairement de l'état s à l'état s'.
- Une fonction de récompense, qui donne une récompense pour la transition d'un état s à s'. La récompense sera développé plus loin.

Le jeu d'échec peut donc être considéré comme un MDP.

Il existe plusieurs type d'algorithmes de TD-learning : Le TD-lambda, SARSA, Q-learning etc..
Nous nous concentrerons ici sur le Q-learning.

Q-learning

Le Q-learning est utilisé pour déterminer la meilleur action selon un état. C'est donc exactement ce qui correspond a notre problème : trouver le meilleur coup possible selon la position actuelle du plateau. L'algorithme va donc apprendre la *policy* de sélection.

Pour cela, l'algorithme (de manière simple) utilise des tables qui associe un couple (état/action) noté $Q(s_t, a_t)$ à une valeur que l'on appellera *q-value*. L'agent observe l'environnement et sélectionne une action (la sélection sera expliqué ensuite). L'action est effectué et l'agent va mettre à jours ses tables selon la formule suivante :

$$Q(s_t, a_t) = Q(s_t, a_t) + \alpha_t \left(\underbrace{r_{t+1} + \gamma \cdot \max_a Q(s_t, a_t)}_{\text{Valeur apprise}} - Q(s_t, a_t) \right)$$

Où α est le taux d'apprentissage, avec $0 < \alpha \leq 1$ (plus la valeur est proche de 1, plus l'apprentissage est conséquent), γ est le facteur d'actualisation, avec $0 \leq \gamma \leq 1$, qui permet de déterminer l'importance de la récompense.

La séléction de l'action se fait avec l'algorithme ϵ-greedy. Cela consiste à alterner entre le choix d'une action correcte, et le choix d'une action au hasard. Cela va permettre d'améliorer certains coups avec le premier choix, et aussi d'explorer de nouveaux coups avec le deuxième choix. Ce choix est fait avec une probabilité ϵ qui change avec le temps. Le choix d'une action correcte est fait en choisissant l'action avec la meilleur q-value.

Lorsque le nombre d'états et d'actions augmente, les tables ne sont plus forcément les structures les mieux adaptées. On peut utiliser par exemple un réseau de neurones artificiels, et comme les échecs possèdent beaucoup d'états, et l'arbre des possibilités à une complexité d'environ 10^{123} donc le réseau neuronal (artificial neural network ou ANN) est approprié dans notre cas.

Deep Q Learning

Le Deep Q Learning est un algorithme utilisé par DeepMind et qui à permis à une machine d'apprendre à jouer à des jeux Atari. Cet algorithme est une variante adaptée aux réseaux neuronaux du Q-learning.

Tout d'abord, définissons un ANN. Un ANN est une structure visant à ressembler aux neurones biologiques. Dans notre cerveau, nous avons des neurones, et lorsque ces neurones sont activés, ils envoie un signal électrique aux synapses (ce qui connecte plusieurs neurones entre eux). Ces synapses vont alors faire une sommation des signaux envoyé et transmettre ce nouveau signal obtenu à d'autres neurones.

Pour un ANN, le fonctionnement va être très proche. Un ANN sera composé au minimum de plusieurs couches, une couche d'*input* qui va représenter les valeurs d'entrées, et une couche d'*output* qui va représenter le résultat obtenue à la fin de l'exécution.

Chaque couche va contenir plusieurs neurones. Ces neurones auront une valeur. Ensuite chaque neurone va transmettre sa valeur à la couche suivante, les neurones de cette dernière vont alors effectuer une sommation des

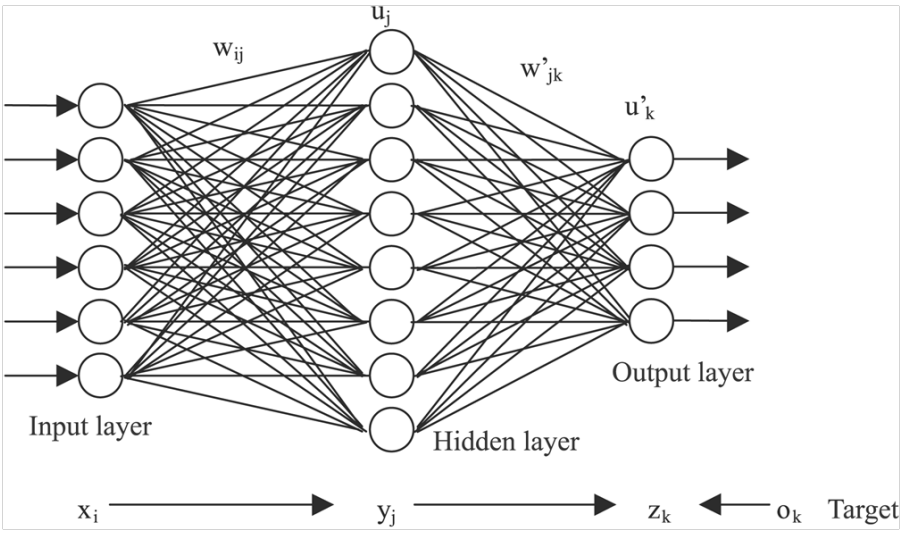
valeurs transmises. Pour transmettre sa valeur, un neurone va envoyer au neurone suivant la valeur multipliée par un poids. Cette nouvelle valeur sera ensuite passé à une fonction d'activation. Le changement de ces poids va donc changer le résultat. Donc l'apprentissage se fera sur les poids.

Cependant, ces deux couches ne sont pas suffisantes, autrement le problème serait un problème de régression linéaire et il suffirait d'étudier les poids pour trouver une valeur optimale. Pour palier à ce problème et permettre au réseau neuronal d'effectuer de grosses tâches, on rajoute entre les deux, des couches appelées *hidden*. C'est ce qui donne le nom de *deep learning* car souvent le nombre de couches *hidden* est très grand.

Pour entraîner un réseau neuronal, et donc déterminer des poids, on utilise un algorithme très simple :
On envoie les valeurs d'input, et on les fait passer aux couches suivantes avec la méthode expliqué plus haut. Cette phase est généralement appelée *feedforward*.
Ensuite, les poids sont changés avec une méthode appelée la *backpropagation*. Il existe plusieurs algorithmes de *backpropagation* mais leur fonctionnement est globalement le même. D'abord on regarde la valeur obtenue et on va renvoyer dans l'autre sens l'erreur. L'erreur est souvent utilisée ensuite avec une méthode appelée la descente de gradient stochastique. Ensuite chaque poids est changée selon l'erreur qu'il cause. On répète ces étapes jusqu'à obtenir une erreur convenable.

En principe, les poids changent beaucoup au début puis font des changements de moins en moins gros au fur et à mesure de l'apprentissage.

Voici un schéma résumant la structure d'un ANN.



L'algorithme de Deep Q Learning est le suivant : On *feedforward* l'état courant s pour obtenir une estimation des Q-values de chaque action. On *feedforward* ensuite l'état suivant s' afin d'obtenir les Q-values suivantes et on en calcul le maximum. On met ensuite pour la valeur cible, $r + \max_{a'} \underbrace{Q(s', a')}_{\text{Calculé précédemment}}$ et les autres Q-values restent les mêmes. Ensuite on passe à la *backpropagation* avec comme erreur $L = \frac{1}{2}\left(r + \max_{a'} Q(s', a') - Q(s, a)\right)^2$

Nous souhaitions créer le réseau neuronal avec comme input les 64 cases du plateau avec la pièce qu'elle contient, la couleur du joueur qui joue actuellement, les déplacements possibles pour les pièces principales, et peut être encore d'autres informations.

Comme le temps d'entraîner le réseau neuronal était plutôt conséquent, nous avons décidé de partir sur un algorithme plus classique, le negamax.

3.2 Negamax

Dans les échecs, la victoire d'un joueur implique la défaite de l'autre, les échecs font donc partie de ce que l'on appelle les jeux à somme nulle. Il existe une variante de l'algorithme du minimax pour les jeux à somme nulle. Cet algorithme s'appelle le Negamax. En voici un pseudo code :

```
function negamax(node, depth, color)
    if depth = 0 or node is a terminal node
        return color * the heuristic value of node

    bestValue = -infinity
    foreach child of node
        v = -negamax(child, depth - 1, -color)
        bestValue = max( bestValue, v )
    return bestValue
```

Cette algorithme reprend le principe du minimax, où on calcule à chaque fois ce que le joueur adversaire ferait, mais il ne sépare pas l'algorithme en deux parties "Si joueur est opposé (min)" et "Si joueur n'est pas opposé (max)" car il réside sur le fait que pour ce type de jeu, $\max(a, b) = -\min(-a, -b)$ ce qui simplifie légèrement l'algorithme.

Le principe est donc d'estimer le coup de l'adversaire en supposant que celui ci choisisse le coup qui nous désavantage le plus, et de continuer ainsi à alterner. Plus on va en profondeur, plus le joueur artificiel aura des "coups d'avance".

Comme les possibilités du jeu d'échecs sont très grandes, il existe deux possibilités lorsque l'algorithme est arrivé à la profondeur maximum allouée. Soit la position obtenue est une position finale, dans ce cas on regarde quel joueur à gagné, soit la position obtenue est en pleine partie. Dans ce dernier cas, il faut alors évaluer la position obtenue à l'aide d'une fonction d'évaluation.

Fonction d'évaluation

Dans notre fonction d'évaluation, nous avons choisi de prendre en compte quelques éléments. Le premier, le plus simple, est de donner une valeur à chaque pièces, et de sommer les pièces actuelles en soustrayant les pièces adverses, ce qui donne un estimateur permettant de déterminer si l'adversaire a plus, ou de meilleurs pièces que nous.

Ensuite nous regardons si le roi est en danger. Pour cela, nous regardons son emplacement sur le plateau, si des pièces l'entoure, si une pièce adversaire le menace.

Enfin, nous regardons la mobilité des pièces, c'est à dire est ce que les pièces dont nous disposons peuvent bouger ou sont elles bloquées ?

Élagage $\alpha - \beta$

Comme l'arbre des possibilités est très grand, on utilise des méthode pour optimiser le parcours de l'arbre. C'est ici qu'intervient l'élagage $\alpha - \beta$. Cette méthode va permettre de ne pas parcourir des branches selon certaines valeurs. Comme le joueur essaye de minimiser les gains de l'adversaire, et donc maximiser les siens, il y a deux coupures possibles. La coupure α et la coupure β. Lorsque l'arbre est sur un noeud min, celui ci va donc choisir le noeud suivant avec la valeur la plus petite. Donc si on a par exemple un noeud qui obtient la valeur 5, et un autre avec la valeur 6, comme l'algorithme choisira la valeur 5 (min) alors il n'est pas nécessaire d'explorer la branche 6. De même pour le cas inverse. Ainsi, si on admet que le nombre de noeuds à chaque branche est constant, et que l'on note celui ci b, et que l'on a une profondeur d, la complexité de parcours est $\mathcal{O}(\underbrace{b * b * b * b ... * b}_{\text{d fois}}) = O(b^d)$. Seulement, dans le meilleur des cas avec l'élagage $\alpha - \beta$, la complexité est seulement de $\mathcal{O}\left(b^{\frac{d}{2}}\right) = \mathcal{O}(\sqrt{b^d})$.

Cependant, pour être dans le *meilleur des cas* il faut que les meilleurs coups soit testés en premier. C'est pour cela que nous avons également rajouté une heuristique appelée *killer moves*. Cela permet de placer les coups à évaluer en premier et donc permettre un élagage plus grand. Nous avons utilisé ce qu'on appelle la *iterative deepening* pour ordonner les coups.

Comme l'algorithme du Negamax est légèrement diffèrent de celui du Minimax, l'appel récursif se fera de la forme :
-negamax(child, depth - 1, $-\beta$, $-\alpha$, color) en inversant bien α et β.

Iterative Deepening

Le *iterative deepening* nous aide à améliorer l'ordonnancement des coups et possède d'autres avantages.

Le principe est le suivant, on appelle le negamax avec une profondeur de 1, puis de 2, puis de 3 jusqu'à la profondeur maximale désirée. Cela permet d'une part d'ajouter un timer et de limiter le temps de jeu, en effet si le temps imparti est écoulé, alors l'algorithme prend le meilleur coup actuel, sinon il va plus loin en profondeur. Comme des informations sont stockées à chaque itérations, cela va permettre au fur et à mesure de placer l' $\alpha - \beta$ dans une position optimale.

Lors du premier appel récursif, les coups sont ordonnés en plaçant en premier ceux qui permettent de capturer une pièce ou d'effectuer une promotion.

Tables de Transpositions

Aux échecs, il existe plusieurs moyens de se retrouver dans une même configuration. Pour éviter d'explorer plusieurs fois le même plateau et donc de faire perdre du temps à l'algorithme, il existe ce que l'on appelle des tables de transpositions. Ces tables vont permettre de stocker les positions explorées dans une hashmap (structure aux temps d'accès et d'insertion de $\mathcal{O}(1)$) et leur associer un meilleur coup. Cela va aussi permettre de stocker d'autres informations utiles pour l'ordonnancement des mouvements. Comme les hashmaps nécessitent un hashage permettant de distinguer les plateaux, nous utilisons le hash de Zobrist

Hash de Zobrist

Pour obtenir un identifiant unique pour chacun des plateaux possibles, il existe un type de hash inventé par Albert Lindsey Zobrist en 1970. Celui-ci est dans les faits assez simple et permet d'obtenir de façon optimisé (très économe en calcule) un identifiant unique pour un plateau donné.

Pour cela il faut en premier lieu générer des valeurs uniques et pseudo-aléatoires pour chacune des pièces (pion noir, pion blanc, cavalier noir, cavalier blanc...) à chacune des positions de l'échiquier soit : couleurs * nombre de pièces * nombre de positions = 2 * 5 * 64 = 640

Ainsi que deux valeurs uniques et pseudo-aléatoires pour indiquer le roque de chaque couleur et une valeur indiquant quelle couleur doit jouer. On obtient alors un ensemble de 643 valeurs toutes différentes et aléatoirement décidées.

Puis dans un second temps à l'aide de la fonction XOR (ou exclusif, noté $\oplus$) et de ses particularités mathématiques très intéressantes, il est possible de générer un identifiant unique de chacun des plateaux possible en réalisation un XOR de chacune des valeurs de chacune des pièces présente sur le plateau à leur position précise ainsi que le roque si il est réalisé et l'information sur la couleur du prochain joueur ayant la main (64 + 3 - 1 XOR) :

$IdPlateau = pion1Noir \oplus pion2Noir \oplus pion3Noir... \oplus roiNoir \oplus pion1Blanc... \oplus roiBlanc...$

Ainsi, grâce à cette technique de hash on s'assure d'obtenir (en utilisant des valeurs de tailles suffisantes, 64 bits) un identifiant unique pour chaque disposition de plateau.

3.3 Améliorations

Cette section comporte une liste des choses auxquelles nous avons pensé pour améliorer notre IA mais que nous n'avons pas mis en place.

Dans un premier temps, il est possible de ne pas recalculer le hash de Zobrist à chaque fois. Il est important de noter que le XOR possède une particularité très intéressante qui permet d'apporter une optimisation plutôt conséquente. En effet si l'on réalise un calcule de XOR avec un nombre x1 et x2 et qu'on réalise sur le résultat un XOR x2 on obtiendra directement x1 : $x1 \oplus x2 \oplus x2 = x1$. Par ce procédé de calcul, il peut être intéressant d'optimiser la génération des identifiants des plateaux en calculant chacun des plateaux à partir du plateau précédent le nouveau coup.

Ainsi on peut réaliser le calcule d'un nouveau plateau suite à un mouvement d'un pion mangeant une tour de la sorte :

'tour sur son ancien emplacement' (suppression par XOR de la tour)

'pion sur son nouvel emplacement' (ajout par XOR du pion à la place de la tour)

'pion sur son ancien emplacement' (suppression par XOR du pion à son ancienne place)

De ce fait, en seulement 3 opérations nous pouvons calculer l'identifiant d'un nouveau plateau et cela économise donc 63 opérations par rapport au fait de calculer chacun des plateaux à partir de ses pièces.

Ensuite, il existe dans la théorie des jeux, et surtout dans le domaine de l'IA, un effet appelé l'effet d'horizon. Il est possible que l'algorithme trouve un certain coup correct jusqu'à une certaine profondeur, cependant ce coup peut s'avérer très mauvais si l'algorithme va une profondeur plus loin. Une solution naïve serait de se

dire "Je rajoute une profondeur à mon algorithme et le problème est réglé", sauf que d'une part, la complexité augmente exponentiellement à chaque profondeur, et d'autre part cela ne résout pas le problème, car qu'elle que soit la profondeur, il peut y avoir une profondeur de plus qui détermine le coup choisi comme mauvais.

On peut donc utiliser une méthode dite de recherche tranquille (quiescence search), qui va permettre d'explorer plus amplement les coups intéressants, et moins les coups non intéressants.

Nous voulions aussi effectuer la méthode de Monte Carlo Tree Search, ou MCTS. Cette méthode est constituée de plusieurs étapes, la *selection* où un noeud est choisi, l'*expansion* où l'on rajoute des noeuds à celui choisi, la *simulation* où l'on simule des coups aux hasards jusqu'à la fin de la partie, et une phase de *backpropagation* pour faire remonter l'état final et ainsi mettre à jour les informations sur le noeud sélectionné.

4 Références

— `https://github.com/skeeto/october-chess-engine` (Modèle utilisé)
— `https://chessprogramming.wikispaces.com/` (Wiki de chessprogramming)
— `http://people.inf.elte.hu/lorincz/Files/RL_2006/SuttonBook.pdf` (Reinforcement Learning)
— `https://arxiv.org/pdf/1509.01549.pdf` (Deep Learning to play chess)
— `https://en.wikipedia.org/wiki/Zobrist_hashing`
— `https://tel.archives-ouvertes.fr/tel-01234642/document` (MCTS)