Application de l'Élagage Alpha-Bêta au Jeu de Dames

Moncef Bouhabei, Paul Chambaz

Janvier 2023

Ce projet a été mené dans le cadre de l'Unité d'Enseignement *d'Intelligence Artificielle*, enseigné par *Elise Bonzon*, dans le cursus de licence 3 à *l'Université Paris Cité*. L'objectif principal de ce projet est de développer une intelligence artificielle capable de résoudre des jeux à deux joueurs, en proposant une implémenatation de l'algorithme alpha-bêta. Ce projet vise l'exploration de cet algorithme pour produire une intelligence artificielle capable de joueur contre des joueurs de différents niveaux. Le jeu des dames a été choisi en raison de sa complexité stratégique et de la varitété des méthodes d'évaluation de la position. En effet, malgré des règles simple, ce jeu permet l'élaboration de stratégies sophistiqués chez les joueurs de haut niveaux. Nous espérons que notre programme sera capable de reproduire certaines de ces stratégies, voire de les contrer. De plus, de nombreuses possibilités existent pour évaluer les positions, cela permet de développer une fonction d'évaluation complexe.

---

1. Description du jeu et du programme

1.1. Motivation

Dans ce projet, nous avons choisi le jeu de dames. Ce choix a été motivé par plusieurs facteurs. Tout d'abord, le jeu de dames est relativement simple à implémenter, ce qui nous permet de nous concentrer sur les aspects les plus importants de notre algorithme plutôt que sur la complexité du jeu lui-même. Cependant, malgré une apparence simple, le jeu de dames comporte une profondeur stratégique considérable. En effet, même si elle ne peut être comparée à celle de jeux classiques comme les échecs ou le go, il offre un nombre coup potentiels de 5×10^{20}. Il a fallu attendre 1994 pour qu'un algorithme de jeu de dames remporte le titre de champion du monde de ce jeu, et 2007 pour qu'une résolution complète soit atteinte.

Deuxièmement, et contrairement à d'autres projet envisagé, le jeu de dames nous apparu comme riche pour le nombre de méthode utilisée dans la fonction d'évaluation. En effet, en regardant une partie se dérouler, il à l'air facile de pouvoir juger de la force de la position de tel ou tel joueur. Bien sûr, si ce type de jugement est loin d'être suffisant pour l'analyse du jeu à très haut niveau, il reste cependant probable que la fonction d'évaluation nécessaire au fonctionnement de l'élagage alpha-bêta permette une solution plus créative et avec plus de résultat que d'autres jeux sur lesquel il est plus difficile de correctement juger d'une position.

Finalement, en faisant des recherches pour ce sujet, nous avons appris l'importance de ce jeu dans l'histoire de l'intelligence artificielle. Dès 1952, un premier algorithme fut écrit. En 1959, le premier document utilisant le terme de *Machine Learning* s'intitulais *Some Studies in Machine Learning Using the Game of Checkers*. De plus avec le programme *Chinook*, depuis 2007 le jeu de dames fait parti des jeu complètement résolu par l'intelligence artificielle. Ce projet, bien plus modeste ne s'interessera qu'à une application de l'élagage alpha-bêta.

1.2. Les règles du jeu de dames

Le jeu de dames se joue sur un damier de huit fois huit cases. Ce damier comporte deux type de cases, des cases blanches et des cases noires, les pions durant la partie ne pourront se déplacer que sur les cases noires. Ce jeu se joue avec deux joueurs, l'un prendra les pions blancs et les pions noirs.

Au début de la partie, chaque joueur reçoit exactement douze pions qu'il doit placer sur les

cases noires des trois premières lignes. À la fin du placement initial, seule les deux lignes du milieu ne devrait pas avoir de pion. Le joueur blanc peut alors commencer.

Un pion au dame n'a le droit d'avancer qu'en diagonale de une seule case vers l'avant. Un pion a donc deux cases potentiels sur lesquels il peut aller. Cependant, si un pion allié ou ennemi se situe sur une des cases sur lequel se trouve sur la case ou le pion veut se déplacer alors ce déplacement est interdit. Une fois que le pion a été déplacé, le tour passe à l'autre joueur qui peut effectuer à son tour un coup.

Un pion peut manger un autre pion. Pour ce faire, si un pion ennemi se situe en diagonale du pion, et que la case derrière en diagonale n'est pas occupée, alors le pion peut le manger. Si c'est le cas, alors le pion se déplacera sur la case libre de derrière et le pion ennemi sera retiré du damier – le jouer ne pourra plus l'utiliser. Il faut noter que du fait, un pion ne peut manger qu'en avant par rapport à son point de départ. De plus, si un pion peut manger alors un pion doit manger, si plusieurs choix sont disponible alors, le choix est laissé au joueur. Finalement, si un joueur a mangé un pion ennemi, alors il conserve son tour et peut effectuer un autre coup si et seulement si le même pion peut continuer à manger.

Un pion peut devenir une reine, d'où le jeu tire son nom. Pour ce faire, il faut alors qu'il arrive à l'autre bout du plateau. Une fois cette destination atteinte, le pion est designé comme reine (souvent en placant un autre pion par dessus le premier). Une reine contrairement à un pion peut se déplacer dans les deux sens et sans restriction de case. Une dame peut donc aller d'un côté à un autre du damier. Une dame peut toujours manger mais elle ne peut le faire que si le pion ennemi est immédiatement à côté d'elle. Autrement dit, une dame ne peut pas parcourir deux cases, puis manger. En revanche, une dame n'a pas de limite sur la quantité de pièce.

Le jeu se termine dans deux cas. Premièrement, si un des joueurs n'a plus de pièces, alors il perd la partie. Deuxièmement, si un des joueurs n'a plus de coups valable, alors il perd aussi la partie.

1.3. Description du programme

Pour ce projet, nous avons fait le choix d'utiliser le langage *lisp*. Ce langage ayant été initialement créé pour résoudre des problèmes d'intelligence artificielle, nous nous sommes dit que ce choix serait adapté. De plus, ce langage, souvent décrit comme *méta-programmable*, permet de modifier la syntaxe du programme durant l'execution du programme. Cela permet de modéliser des cas mathématiques abstraits ou pour notre problème, des états et actions abstraits. Dans notre cas, nous utilisons le *common-lisp*, avec la librairie *cl-sdl2* pour l'implémentation graphique du logiciel. Les algorithmes d'intelligence artificielle ont été implémenté. Finalement, nous avons aussi choisi *lisp* pour une raison, même si ce programme est un programme qui est souvent interprété, et qu'il possède un garbage-collector, il est généralement considéré comme rapide. Nous espèrons que ce choix permettra de pouvoir explorer plus de possibilité dans l'arbre de recherche.

Le programme se découpe en plusieurs fichiers:

- *packages.lisp* contient les informations relative au paquet contenant l'application.
- *main.lisp* contient la mise en place du *main loop* de l'application. Il met en place *sdl2*, mais ne contient pas d'implémentation complexe qui sont laissé aux différents paquets.
- *constants.lisp* contient une liste des constantes requises pour le programme. La plupart des constantes sont utilisées pour l'interface graphique. D'autres sont lié à la logique du jeu et d'autres encore sont liées à l'intelligence artificielle.
- *interface.lisp* contient toutes les fonctions requises pour l'interface graphique. Dans ce fichier est géré à la fois l'affichage et l'interaction avec le joueur.
- *logic.lisp* contient tout le code permettant le bon déroulement du jeu. C'est dans ce paquet qu'est défini formellement des concepts comme l'état du jeu, une action ou l'ensemble des actions permisent à partir d'un état.
- *intelligence.lisp* contient tout le code de l'implémentation alpha-bêta et les fonctions nécessaire pour le bon fonctionnement de l'intelligence artificielle.

2. Description de l'intelligence articielle

2.1. Chinook

Avant de commencer la description du programme de ce projet, nous nous interesseront au programme Chinook. Ce programme est connu pour avoir battu le record du monde en 1994 – sans avoir battu *Marion Tinsley* qui était considéré comme de loin le meilleur joueur du monde, détenant le titre pendant plus de 40 ans. Le joueur avait pris sa retraite suite à des problèmes de santé et décedera l'année suivante invaincu ! En 2007, cependant, les chercheurs de *l'Université d'Alberta* réussirent à résoudre le problèmes du jeu de dames. Si le programme que nous proposons est loin d'égaler cette performance, il est important d'étudier le fonctionnement de *Chinook*. En effet, ce programme n'est pas un programme du type *réseau de neuronnes profonds*, comme le seront plus tard des programmes révolutionnaires comme *Deep Blue* qui vaincu *Garry Kasparov*, le champion du monde des échecs. *Chinook* est bien plus propre du fonctionnement de l'algorithme utilisé dans ce projet. Le programme fonctionne en trois phases distinctes, la première phase est résolu avec un base de donnée contenant tous les coups classiques utilisés par les joueurs professionels. La seconde phase qui nous intéresse particulièrement, est résolu par un algorithme de recherche profonde. Ce terme désigne les algorithmes basé sur le *depth-first search* qui explore des graphes. Ce type d'algorithme est très similaire à l'algorithme alpha-bêta utilisé dans ce projet. De plus l'algorithme utilisé par *Chinook* utilise une fonction d'évaluation pour juger d'une certaine position. Finalement, sur plusieurs années, les chercheurs ont compilé une base de coups pour effectuer des algorithmes pour la dernière partie du jeu en calculant pendant plusieurs années, pour obtenir la liste des actions parfaites pour les huit dernier coups. On notera qu'il s'agit probablement d'un algorithme *minimax* que l'on laisse tourner sur toutes les configurations légale de fin du jeu.

2.2. PEAS du programme

2.3. Définition formelle d'un état

Il nous faut tout d'abord définir formellement l'état. Tout d'abord, il faut noter que la définition d'un état doit contenir toute les informations nécessaire pour pouvoir calculer indépendamment les décisions sur le futurs du jeu. Notons aussi que c'est cet état qui sera passé à l'arbre. Le plus compacte nous somme donc capable de représenter cet état, le plus petit le coup mémoire associé sera – pas au sens de la compléxité, mais simplement au sens de la vitesse d'éxecution en cycle de processeur. De la notion de *cache-locality*, on espère que cela aura aussi un impact sur la vitesse d'execution du programme. Finalement, pour mieux utiliser la notion de récursivité, il serait intéressant de ne jamais vraiment stocker l'état, mais simplement de le transformer en le faisant passer. On n'aurait alors jamais deux copies : cela permettra de réduire considérablement l'empreinte mémoire du programme. Il faut donc que l'état soit facilement réversible. Il faut qu'une la fonction d'action puisse être une *bijection* entre un état et un autre.

Nous avons donc choisi de représenter l'état comme une liste de troix élements:

- Le tableau du damier, comportant un total de 8×8 cases. A chaque case, un entier est attribué, 0 pour une case vide, 1 pour un pion noir, 2 pour un pion blanc, 3, pour une dame noire et 4 pour une dame blanche. On fait le choix de stocker les cases blances : en effet, on espère que cela permette à certains algorithme d'être plus rapide et la taille total de la représentation de la liste ne sera pas copié, ce qui devrait ne pas couter plus.
- Le numero du joueur, 0 pour les noirs et 1 pour les blancs.
- Un argument optionel si un pion doit continuer à manger. Ce genre de configuration n'arrive que après qu'un pion ait mangé et qu'il peut encore manger après. Comme le pion peut faire un choix, nous avons choisi de laisser le programme chercher et considerer cela comme un autre tour. Durant ce tour, le rôle du joueur ne changera pas, mais il faut indiquer à l'état que seul ce pion peut jouer pour obtenir la liste correcte des actions légales. Si on est dans cette situation, un entier désignant le numéro de la pièce par ordre d'apparition sur le damier du joueur.

Si on utilise un octet pour chacun de ces trois paramètres constituant l'état, alors stocker l'état revient à stocker 66 octets.

D'un point de vue plus formel, on note l'état:

$$E_n \in \mathbb{E}$$

Ou $\mathbb{E}$ est l'ensemble de tous les états possibles de position légales sur le damier. On ajoute aussi E_n afin de pouvoir donner un sens à la direction du jeu.

2.4. Définition formelle d'une action

Une action dans ce programme doit représenter le déplacement d'un pion à un autre pion, il doit aussi donner des informations sur le joueur qui vient de jouer afin d'être completement bijectif. Une action est donc:

- Un premier entier signifiant la position de départ du pion.
- Un second entier signifiant la position d'arrivée du pion.
- Un entier qui signifie le joueur qui a fait l'action.

D'un point de vue plus formel, on note une action:

$$a \in \mathbb{A}$$

$$u : \mathbb{A} \times \mathbb{E} \to \mathbb{A} \times \mathbb{E}, a, E_n \mapsto a, E_{n+1}$$

$$u^{-1}(u(a, E_n)) = a, E_n$$

$$\mathcal{A} \subset \mathbb{A}, m \in \mathbb{N}$$

$$v : \mathbb{E} \to \mathcal{A}, E_n \mapsto \{a_1, ..., a_m\}$$

Ici $\mathbb{A}$, est l'ensemble des actions légales et u est la fonction qui transforme un état en un autre à partir d'une action en allant vers l'avant. On renvoie aussi l'action afin de pouvoir avoir une fonction bijective.

Finalement, v est la fonction qui génère tous les coups légaux. Elle renvoie une liste d'action légale. $\mathcal{A}$ est un sous ensemble de $\mathbb{A}$ et m est le cardinal de $\mathcal{A}$.

Nous utiliserons toutes ces fonctions pour traverser le graphe de l'ensemble des actions et générer l'arbre de recherche. On utilise v pour générer l'ensemble des actions légale, puis pour chaque membre de la liste, on avance dans le graphe avec u. Pour remonter, il suffit d'utiliser u^{-1}.

Du point de vue de l'implémentation, il est très important que ces fonctions aient un empreinte mémoire *locale*, c'est à dire que l'on ne veut jamais stocker leur résultat, mais toujours le faire passer au reste du programe. Cela empechera de consommer de l'espace mémoire au cours du graphe. De plus, afin d'accélerer la vitesse d'éxecution du programme, il est important que ces fonctions soit rapide puisqu'elle seront effectuée pour chaque noeud de l'arbre au moment du déroulement.

2.5. La fonction d'évaluation

La fonction d'évaluation est très important dans ce programme:

$$f : \mathbb{E} \to \mathbb{R}, E_n \mapsto f(E_n)$$

$$f(E_n) = w_1 \times f_1(E_n) + ... + w_p \times f_p(E_n)$$

On voit bien que la fonction f est en fait composé de p fonctions f_p. Chacune de ces fonctions est en fait une fonction d'évaluation elle même qui est définie comme f. f est donc la composée de p fonctions d'évaluation elle même.

De façon plus concrête, chaque fonction f_p est une *évaluation de l'état*. Cela ne veut rien dire de plus, il n'y a donc, en particulier pas de notion de *bon état* et de *mauvais état*. Alors, comment fait-on pour discriminer dans f pour obtenir la notion désiré de bon état. C'est l'utilité des w_p.

Les w_p sont une liste de poids que l'on utilise pour attribuer à chaque fonction f_p un poids. Ainsi, si le poids est positif et grand, alors la fonction d'évaluation f_p est un particulièment bon moyen de juger de la bonne qualité de la position sur le damier. À l'inverse, si le poids est négatif et grand, alors la fonction d'évaluation f_p est un particulièrement bon moyen de juger de la mauvaise qualité de la position sur le damier. En revanche si le poids est proche de 0, alors la mesure n'est pas vraiment un bon moyen de juger de la qualité d'une position sur le damier.

Pour ce projet, nous avons fait le pari que nous ne savions pas vraiment ce qui était bon pour chaque fonction d'évaluation, dans la prochaine section, nous verrons comment nous avons pu trouver ces poids.

Pour la liste des potentiels fonctions d'évaluations f_p, nous avons chercher une liste de fonctions qui pourraient être une bonne mesure. En effet, si le poids venait à être autour de 0, alors c'est possible que cette fonction soit plus un ralenti qu'une aide. En effet, cette fonction est executé pour chaque noeud de l'arbre, en la rendant plus rapide, cela permet d'explorer

plus de noeud. Nous avions donc l'objectif suivant, dans un premier temps énumerer une liste de fonctions d'évaluations qui nous paraissaient prometteuse. Dans un second temps, nous ajustions les poids. Dans un troisième temps, nous retirions les fonctions d'évaluations qui n'avaient pas fait leur preuve et ne gardions que les fonctions qui étaient suiffisement efficace.

Au départ, nous avons identifié fonctions d'évaluations:

1. Le nombre de pièces du joueur
2. Le nombre de pièces du joueur adverse
3. Le nombre de reine du joueur
4. Le nombre de reine du joueur adverse

2.6. Un tournoi pour "l'entraînement" de la fonction d'évaluation

Pour trouver la valeur exacte de chaque poids, nous somme parti du principe que nous ne pourrions pas trouver en raisonnant. Bien sûr, nous pourrions mettre des poids arbitraires, mais comment s'assurer que nous avions raison et qu'une évaluation que nous pensions bonne était en fait pas vraiment utile, voir pire, mauvaise. Du fait nous avons eu l'idée de faire une sorte de tournoi pour juger de la qualité des poids.

Au départ, chaque poids est mis à zero, une suite de dix tirages entre -1 et 1 sont ensuite réalisé et additioné pour produire le poids initial de chaque fonction d'évaluation. Nous générons ensuite un nombre n de fonctions d'évaluations différentes f. Nous réalisons ensuite un genre de tournoi de tennis, ou chaque ia en affronte une autre, puis le vainqueur d'une autre affrontement jusqu'a ce que la meilleure soit trouvé. Chaque affrontement comporte 5 parties, si il y a match nul à la fin de ces 5 parties, alors une des ias est choisies au hasard.

Cela produit ensuite une base sur laquelle on peut ensuite créer notre prochain tournoi. Sur la base des poids du gagnant, on peut refaire les dix tirages entre -1 et 1 pour recréer n ias et refaire un second tournoi. On mesure ensuite la distance entre les poids de la génération 1 et les poids de la génération 2, si ces poids sont inferieurs à une certaine valeur, alors on arrête l'entrainement, en revanche, si ce n'est pas encore le cas, on continue à regenerer des nouvelles ia. Plus précisement, nous avons arrêter lorsque l'on observait une stabilisation de la différence.

Ce type de méthode s'apparent vaguement à un algorithme génétique, mais on prend toujours uniquement le meilleur de chaque génération et on ne croise pas les résultats, du fait il n'en est pas vraiment un.

2.7. Notion de difficulté

3. Bilan du projet