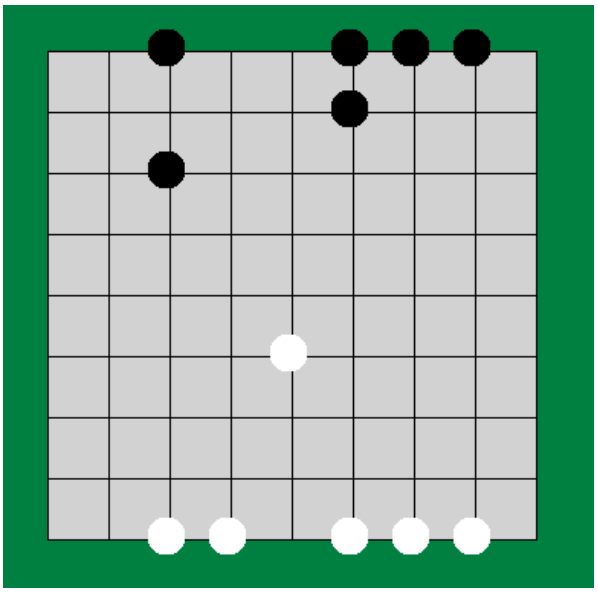

Les cavaliers chinois

Introduction à l'intelligence artificielle

Rémi BUTET et LEGUET

Sommaire

Introduction

Les cavaliers chinois sont un jeu qui se joue sur une grille de 9 fois 9 lignes. Chaque ligne verticale est référencée par une lettre de A à I. Chaque ligne horizontale est référencée par un chiffre entre 1 et 9. Chaque intersection peut donc être référencée par un couple composé d'une lettre et d'un chiffre.

Chaque joueur est identifié par une couleur et possède au départ 6 cavaliers. La configuration initiale est la suivant : les cavaliers blancs sont situés sur la ligne numéro 9, les cavaliers noirs sur la ligne 1, la position centrale et les deux extrémités de la ligne étant laissées libre. Le but du jeu est de faire traverser le plateau à trois de ses cavaliers, ou de faire traverser plus de cavaliers que l'adversaire.

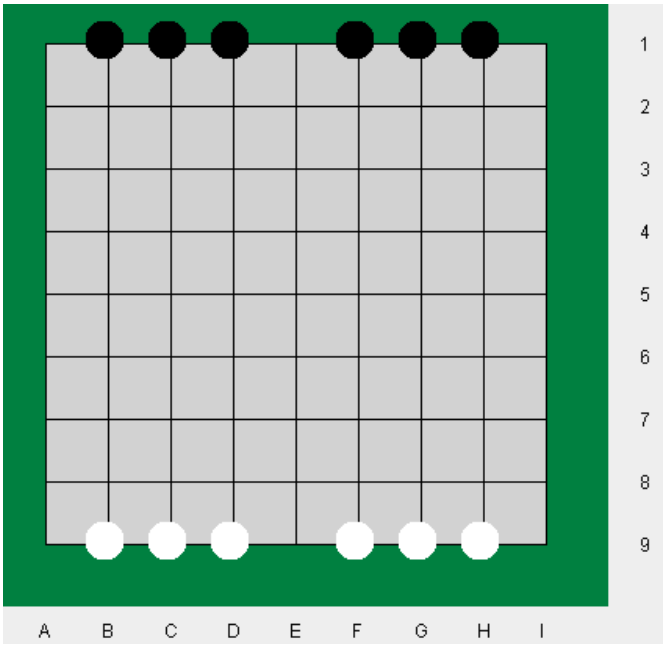

Figure 1 Représentation du plateau de l'énoncé

La représentation du jeu et ces règles sont simples mais intelligence artificielle cela peut rapidement devenir compliqué. Sachant qu'il y au maximum 24 coups possibles par tour et qu'il peut y avoir 96 tours, cela signifie qu'il y a un trop grand nombre de possibilité pour réussir à jouer parfaitement bien.

1. Plateau

Pour représenter un plateau il existe un grand nombre possibilité :

- Tableau à 2 dimensions de Char : char[][]
- Une Map avec tableau de Char : HashMap<Char, Char[]>
- Une Map de String et de Char : HashMap<String, Char>

Cependant nous avons préféré choisir un tableau à 2 dimensions de « Char » car c'est la méthode la plus simple et la plus rapide lors d'une recherche.

Peu de temps après nous avons rencontré un problème de compatibilité, avec les coordonnées transmises par l'arbitre. Un plateau est normalement continué de ligne verticale référencée par une lettre d'A à I et de ligne horizontale référencée par un chiffre entre 1 et 9. Mais comme nous utilisons un tableau à 2 dimensions, il nous faut donc 2 nombres. Pour cela nous avons mis en place un classe « CoupCavaliers » qui permet de faire la compatibilité :

- `public CoupCavaliers(String move);`
- `public CoupCavaliers(int init_l, int init_c,int finale_l, int finale_c)`
- `public int getInitialeLigne();`
- `public int getInitialeColonne();`
- `public int getFinaleLigne();`
- `public int getFinaleColonne();`
- `public boolean isPasse();`
- `public String parse();`

Une fois la représentation du tableau et les coordonnées choisissent, il a fallu choisir valeur d'une pour les cases. Une case peut être vide, blanc ou noir. Nous avons choisi de définir 3 variables statiques et d'enregistrer les mêmes valeurs que l'exemple donné l'énoncé :

- Case vide : **+**
- Cavalier blanc : **b**
- Cavalier noir : **n**

	0	1	2	3	4	5	6	7	8
0	+	n	n	n	+	n	n	n	+
1	+	+	+	+	+	+	+	+	+
2	+	+	+	+	+	+	+	+	+
3	+	+	+	+	+	+	+	+	+
4	+	+	+	+	+	+	+	+	+
5	+	+	+	+	+	+	+	+	+
6	+	+	+	+	+	+	+	+	+
7	+	+	+	+	+	+	+	+	+
8	+	b	b	b	+	b	b	b	+

Figure 2 Notre représentation du plateau

2. Heuristique

Tous au long du développement de l'IA nous avons fait évoluer l'heuristique, mais nous avons toujours respecté les critères que nous avions définis dès le début :

- Le nombre de cavaliers encore présents sur le plateau : il faut avoir un maximum de cavalier
- Le nombre de cavaliers arrivés chez l'adversaire : il faut avoir un maximum de cavalier dans le camp ennemi
- La position des cavaliers sur le plateau : la position du cavalier est proche du camp ennemi plus on a de chance de gagné

a. Première heuristique

Dans un premier temps nous avons programmé l'heuristique en appliquent simplement les règles :

- Si j'ai 3 cavaliers chez l'adversaire j'ai gagné
- Si l'adversaire à 3 cavaliers chez moi j'ai perdu
- Sinon on additionne :
  - Le nombre de cavaliers blancs arrivés – Le nombre de cavaliers noirs arrivés
  - Le nombre de cavaliers blancs sur le plateau – Le nombre de cavaliers noirs sur le plateau
  - Le nombre de points du cavalier blancs – nombre de points du cavalier noir (Pour calculer les points on regarde s'il est proche du camp ennemi et plus il est plus il rapporte de point : Figure 3)

	0	1	2	3	4	5	6	7	8	
0					n					0
1										1
2							n			2
3										3
4										4
5			n							5
6						n				6
7										7
8					n			n		8

Point Noir : $0 + 2 + 5 + 6 + 8 + 8 = 29$

Figure 3 Calculer les points de l'Heuristique 1

b. Deuxième heuristique

Dans un second temps nous avons fait évoluer l'heuristique pour favoriser l'avancement des cavaliers proches du camp ennemi. Car dans l'ancienne heuristique, avancer un cavalier de la ligne 0 à ligne 2 rapporté autant de points qu'avancer un cavalier de la ligne 5 à 7. Voici à quoi ressemble la nouvelle heuristique :

- Si j'ai 3 cavaliers chez l'adversaire j'ai gagné
- Si l'adversaire à 3 cavaliers chez moi j'ai perdu
- Sinon :
  - Le nombre de points du cavalier blanc – nombre de point du cavalier noir (Pour calculer les points on regarde s'il est proche du camp ennemi et plus il est plus il rapporte de point mais cette fois on utilise exponentielle : Figure 4)

	0	1	2	3	4	5	6	7	8	
0					n					e^0
1										e^1
2							n			e^2
3										e^3
4										e^4
5			n							e^5
6						n				e^6
7										e^7
8					n			n		e^8

Point Noir : $e^0 + e^2 + e^5 + e^6 + e^8 + e^8 = 2981$

Figure 4 Calculer les points de l'Heuristique 2

L'utilisation d'exponentiel permet d'enlever 2 paramètres par rapport à l'heuristique 1 car il n'y a plus de besoin de regarder combien de cavalier chez l'ennemie car il rapporte déjà e^8 et aussi de regarder combien de cavalier il y a sur le plateau car il y aura toujours une perte d'au moins e^0.

3. Algorithme de recherche

a. Algorithme

Dans un premier temps nous avons réutilisé l'algorithme AlphaBeta que nous avions déjà implémenté dans un TP. Cependant pour augmenter la vitesse de calcul nous avons appliqué l'algorithme NegEchecAlphaBeta que nous avions étudié pendant le cours.

Le grand avantage de NegEchecAlphaBeta c'est qu'il permet de sauvegarder dans une table de transposition les éléments déjà calculés. Donc lorsque l'on déroule l'algorithme et que l'on arrive à l'état d'un plateau que l'on a déjà le calculé, il suffit de récupérer les données.

Cependant il est beaucoup plus difficile à programmer et consomme plus de mémoire car il faut sauvegarder les plateaux déjà calculés.

b. Table de transposition

Nous avons donc mis une place une table transposition qui est en réaliser un HashSet<EntreeT>. Un « HashSet » permet d'avoir seulement une fois un même élément donc un seul plateau identique et « EntreeT » contient toutes les données d'un plateau.

Pour optimiser au maximum l'utilisation de la mémoire nous avons utilisé dans le type valeur avec le moins de bit possible :

- La position est l'état du plateau, il est sauvegardé sur 32 bits
- La profondeur est sauvegardée sur 8 bits car il ne peut pas être supérieur à 127
- La valeur est sauvegardée sur 16 bits car le maximum de l'heuristique est $(e^8 * 6) < 32\,767$.
- Le meilleur coup est constitué de 2 nombres de 8 bits :
  - Les coordonnées initiales : numérotés de 0 à 80
  - Les coordonnées finales : numérotés de 0 à 80
- Les flags sont constitués de 2 booléens de 1 bit :
  - La valeur est exacte : Vrai ou Faux
  - La borne : Borne inférieur (Faux) ou Borne supérieur (Vrai)

Le plateau est donc au final sauvegardé sur seulement 74 bits.

Position : Boolean[] (32-bit)	Profondeur : Byte (8-bit)	Valeur : Short (16-bit)	Meilleur Coup		Flag	
			Initiale : Byte (8-bit)	Finale : Byte (8-bit)	Exact : Boolean (1-bit)	Borne : Boolean (1-bit)
01101011001011010110001 1…	11	5	56	75	False	False
10100010100001000110111 1…	9	5	2	9	False	False
11111001000001110110110 0…	10	0	25	8	True	False
00001110100010111010110 1…	14	2545	25	8	False	False
01111011011110110001010 1…	1	-48	1	20	False	True
01101000010011111101101 0…	14	660	25	14	False	False
01111000111110001100001 0…	13	-1	61	80	False	False
00100010110011001011101 0…	14	1980	25	8	False	False

Figure 5 Table de Transposition

c. Dictionnaires

Pour calculer l'état du plateau il a fallu mettre en place un dictionnaire, il est constitué de la couleur du cavalier, l'emplacement du cavalier et la « HashValue ». Le dictionnaire est recréé aléatoirement à chaque partie.

Couleur : Boolean (1-bit)	Case : Byte (8-bit)	HashValue : Boolean[] (32-bit)
False : Blanc	0	00001100101011101110100100011001
False : Blanc	1	11111011110010111111101000001010
False : Blanc	…	01110100001111100011001111011110
False : Blanc	80	10100100000001111100101101011001
True : Noir	0	10011111101100000101101001101110
True : Noir	…	01111100111100010101001000001011
True : Noir	79	10011110000111001101100011110111
True : Noir	80	10010100011010011101001000111100

Figure 6 Représentation d'un dictionnaire

d. Plateau

Pour créer un état d'un plateau il suffit d'appliquer un XOR sur tous les « HashValue » des pièces sur le plateau. Lors de la création du plateau on initialise l'état avec les positions par défaut. Puis à chaque tour on applique les changements.

- On enlève la pièce de l'état du plateau :

Plateau actuelle	00001100101011101110100100011001
Position Initiale de la pièce	11111011110010111111101000001010
XOR : Plateau sans la pièce	11110111011001010001001100010011

- Ensuite on rajoute la pièce a la nouvelle position :

Plateau sans la pièce	11110111011001010001001100010011
Position Finale de la pièce	01110100001111100011001111011110
XOR : Nouveau Plateau	10000011010110110010000011001101

Puis la valeur « Nouveau Plateau » est sauvegardée dans la table de transposition.

e. Gestion du temps

Comme nous avons une contrainte de temps nouveau avons mis un place un système qui permet d'aller calculer plus en profondeur si le temps le permet. Nous avons défini que l'algorithme ne pouvez être appelé que 48 fois car il y a 6 cavaliers et ils sont obligé d'avancé d'une ligne à chaque fois.

Donc à chaque fois nous calculons combien de coup on a joué et en combien de temps, ce qui nous permet de savoir pendant combien de temps on peut appliquer l'algorithme :

Temps restant / Coups restant

De plus nous avons mis en place un système qui permet d'aller plus en profondeur seulement le nombre mouvement déjà réalisé. Car nous nous sommes rendu compte qu'au début il était inutile calculer en profondeur 6 car il y était impossible de voir si l'on avait gagné ou perdu.

Mouvement total	Profondeur Min	Profondeur Max
< 6	**3**	**3**
< 20	**3**	**5**
>= 20	**3**	**6**

Figure 7 Profondeur selon le nombre de mouvement déjà réalisé et le temps restant

4. Analyse des performances

Nous avons réalisé de très nombres tests de performance pour mettre en place le système de calcul de profondeur seulement nombre de mouvement. Car il est très intéressant de calculer à profondeur 6 mais impossible de l'appliquer à chaque fois. Donc nous avons préféré l'utilisé uniquement quand on a une chance trouvé un chemin gagnant.

Profondeur	Par recherche	Total
4	1 seconde	1 minutes
5	10 secondes	4 minutes
6	25 secondes	10 minutes

Figure 8 Le temps selon la profondeur

Conclusion

Nous avons appris beaucoup de choses sur l'algorithme NegEchecAlphaBeta, cependant nous n'avons pas eu assez de temps pour tester découvrir. L'algorithme NegEchecAlphaBeta est très complexe ce qui rend extrêmement compliquer débugger et l'améliorer.

Nous avons aussi découvert beaucoup de choses sur le fonctionnement des heuristiques et de l'importance et son optimisation.

Cependant nous avons rencontré de nombreuses difficultés sur la gestion du temps et du calcul en profondeur.

D'un point général l'introduction à l'intelligence artificielle a été très instructif et nous a permis de découvrir plein de nombreux algorithme et nous ouvert les portes du monde de l'intelligence artificielle.