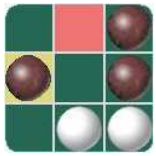
LINES OF ACTION

8 octobre 2004

Présentation du jeu, De son implémentation Et du moteur d'intelligence artificielle

---



Charles Clément & Julien Blanchard

Lines Of Action

Table des matières

Table des figures

Lines Of Action

1 Présentation du jeu "Lines of Action" :

"Lines of action" (soit "Lignes d'action" en français) est un jeu de stratégie à deux joueurs. Dans la suite de ce dossier, l'abréviation "LOA" sera utilisée pour le désigner.

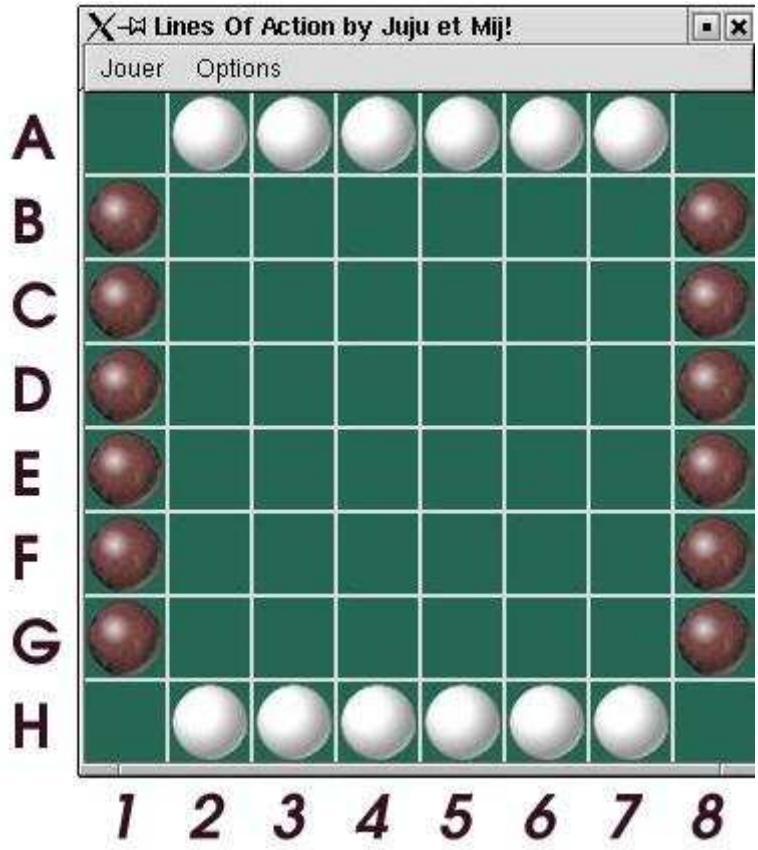
1.1 Origines et particularités de LOA :

LOA a été inventé par Claude Soucie dans les années 60. Il s'agit d'un jeu à information complète dans lequel le facteur hasard n'entre pas en compte. Son originalité vis à vis des jeux du même type -plus courants- réside en son but. En effet LOA est un jeu de "connexion" c'est à dire qu'il faut regrouper tous ses pions. Enfin, en ce qui concerne le matériel utilisé, il est standard puisqu'il se compose d'un plateau de cases (8*8) et de 12 pions de chaque couleur.

1.2 Règles du jeu :

1.2.1 Situation initiale

FIG. 1 – Position initiale des pions sur le plateau



Chaque joueur possède tous les pions d'une couleur. Ce sont toujours les noirs qui commencent à jouer.

1.2.2 But

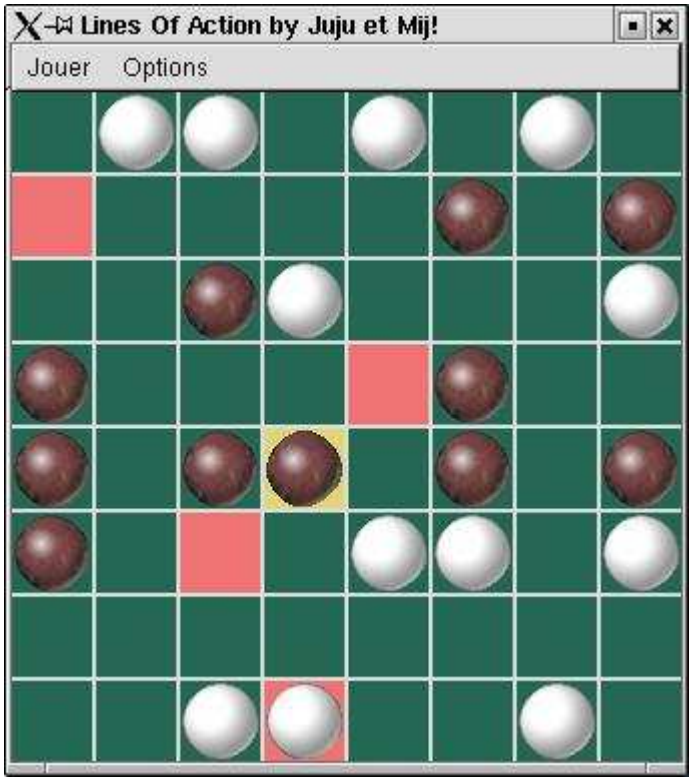
Comme nous l'avons énoncé précédemment, LOA est un jeu de "connexion", aussi chaque joueur tente de réunir tous ses pions. Deux pions sont déclarés connexes si ils se trouvent sur deux cases adjacentes, les 8 directions possibles étant autorisées(horizontales, verticales ou diagonales).

1.2.3 Mouvements autorisés

Les pions peuvent également se déplacer dans les 8 directions possibles. Cependant le pion doit avancer du nombre de cases correspondant au nombre total de pions (lui compris) étant sur la ligne dans laquelle s'effectue le mouvement (d'une extrémité à l'autre du plateau de jeu).

Un pion peut passer par dessus les cases vides et ses propres pions mais pas par dessus ceux de son adversaire. En revanche, la case d'arrivée doit être soit vide, soit occupée par un pion adverse auquel cas celui-ci sera "pris" et retiré définitivement du plateau de jeu.

FIG. 2 – Différents types de mouvement :
En considérant les abscisses de 1 à 8 et les ordonnées de A à H, le pion souligné de jaune est en E4 et peut aller en F3, D5, B1 et manger le pion blanc qui est en H4.



1.2.4 Fin de partie

La partie se termine lorsque l'un des deux joueurs n'a plus qu'un groupe de pions connexes.

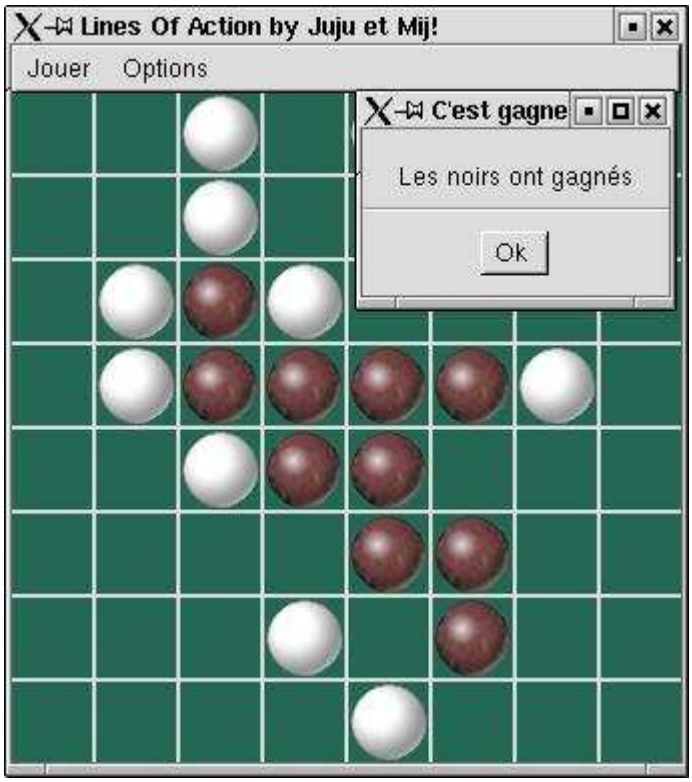
Si un joueur n'a plus qu'une seule pièce, il a gagné.

Si la fin du jeu est atteinte simultanément par les deux joueurs (les deux ont connectés tous leurs pions), c'est le joueur ayant joué le dernier coup qui est déclaré vainqueur.

Si un joueur ne peut pas jouer, il a perdu.

FIG. 3 – Position finale.

Les noirs ont gagné !



1.2.5 Prise de pions adverses

Contrairement à beaucoup de jeu la prise de pions n'est pas un objectif. Cela peut même devenir un handicap si jamais une prise réduit la connexité adverse. De plus il est parfois plus simple de connecter des pions si ils sont peu nombreux. Cependant les choses ne sont pas aussi simples que cela puisque la prise d'un pion peut au contraire casser un groupe connexe de l'adversaire. Nous étudierons ce problème original plus en détail dans la partie traitant des différentes heuristiques.

1.2.6 Complexité

Le jeu contient au maximum 24 pions et au minimum 3 pions sur un plateau de 64 cases. Aussi, on peut connaitre le nombre total de positions possibles.

Pour un nombre de pions blancs pB et un nombre pN de pions noirs, on a le nombre total de positions possibles nb_pos_possible tel que

$$nb_{pos_{possibles}} = (\sum_{pB=1}^{12} \sum_{pN=1}^{12} \binom{64}{pB} \binom{64-pB}{pN}) - \binom{64}{1} \binom{64-pB}{1}$$

Soit un total de plus de $1,3 * 10^{24}$positions possibles (ce nombre doit être légèrement minoré car à cause des règles, certaines positions ne peuvent jamais être atteintes).

En considérant que l'on peut effectuer, à l'heure actuelle, environ 2 milliards d'opérations élémentaires par seconde. En conservant les ordres de grandeurs, même si l'on peut examiner 1 milliard de positions par seconde (ceci est largement surévalué), il faudrait pour examiner tout le jeu à peu près $\frac{10^{24}}{10^9} = 10^{15}$ secondes. Sachant qu'un an compte : $3600 * 24 * 365 \simeq 31,5 * 10^6$ secondes, il faudrait donc $\frac{10^{15}}{10^7} = 10^8$ années .

Cet ordre de grandeur (100 millions d'années pour examiner toutes les positions posssibles du jeu) montre bien que l'on ne peut pas prétendre résoudre facilement LOA. Il n'existe pas non plus d'algorithme (connu) assurant la victoire depuis la position de départ.

Pour approcher réellement la difficulté à résoudre le jeu, nous pouvons également évaluer la complexité de l'arbre de recherche (cf chapitre). Le "facteur de branchement" de LOA est de 30 environ. Lorsque l'on effectue une recherche en profondeur p, le nombre de positions parcourues est donc 30^p. Le longueur moyenne d'un jeu est de 38 coups. Ceci implique donc un arbre de $30^{38} \simeq 10^{56}$ noeuds ce qui est (encore plus) impossible à parcourir à l'échelle de temps humaine.

2 Représentation informatique de LOA :

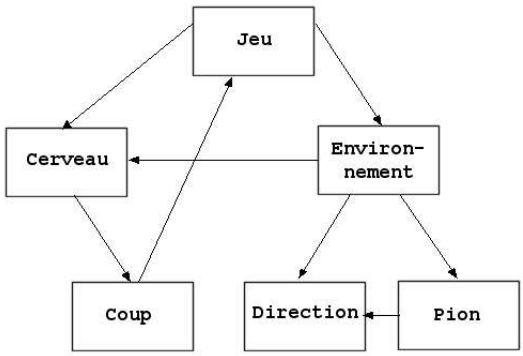
Le langage choisi afin d'implémenter ce jeu est C++. Nous aborderons dans cette partie la façon dont nous avons programmé le jeu lui-même (respect des règles ...).

2.1 Types de données :

2.1.1 Programmation orientée objet

Puisque nous utilisons un langage orienté objet, nous avons décider de représenter le jeu par différentes classes. Schématiquement, nous avons trois grandes entités ayant les intéractions suivantes. Un JEU met en relation deux CERVEAUX et un ENVIRONNEMENT de jeu. Un CERVEAU choisi de jouer un coup en fonction de l'ENVIRONNEMENT. Celui-ci se modifie au fur et à mesure du jeu. Des classes annexes existent bien sûr, elles sont décrites dans le schéma du fonctionnement global suivant.

FIG. 4 – Fonctionnement global



Comme nous l'avons dit le JEU met en relation deux CERVEAUX et un ENVIRONNEMENT. Le CERVEAU choisit un COUP à jouer. Le JEU demande alors à l'ENVIRONNEMENT de se mettre à jour. Pour ceci, l'ENVIRONNEMENT modifie notemment ses PIONS suivant les DIRECTIONS nécessaires.

Pour avoir une vue plus détaillée des classes, nous avons fourni les fichiers d'entêtes (.h) en annexe.

2.2 Calcul de la connexité

2.2.1 Différentes méthodes et comparaisons

Une des difficultés de LOA est de réussir à déterminer à quel moment le jeu est terminé, c'est à dire trouver les groupes connexes de chaque couleur. Il existe au moins trois manières différentes de vérifier la propriété de connexité.

La première méthode

Elle consiste à compter le nombre de groupes de pions pour chaque couleur, si ce nombre est supérieur à 1, ce n'est pas gagné.

Pour ceci, il faut créer pour chaque couleur un tableau de 64 cases ayant des 1 pour les coordonnées correspondant à un pion de la couleur et des 0 pour les autres. On parcourt ce

tableau, lorsque l'on trouve un 1, on appelle une fonction f récursive qui met à 0 la case et regarde dans les 8 directions autour de la case courante. Pour chaque case à 1, on appelle de nouveau f. A la sortie de la récursion, on continue le parcours du tableau jusqu'à trouver une autre case à 1, auquel cas on réitère la même opération.

La deuxième méthode

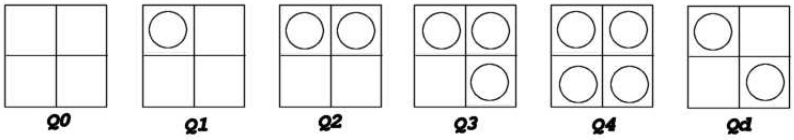
Il s'agit d'une optimisation de la première, on décide simplement de compter le nombre de pions présent dans le premier groupe et de le comparer au nombre total de pions de la couleur.

Enfin il existe la *méthode d'Euler* que nous décrivons dans la partie suivante.

2.2.2 Description de la *méthode d'Euler*

Cette heuristique permet de déterminer dans 99% des cas si la position est finale ou non. Pour ceci, il faut examiner le plateau (pour chaque couleur) en représentant le jeu par 81 carrés. Les carrés peuvent avoir les formes suivantes (avec les rotations).

FIG. 5 – Différents types de carrés



Le nombre d'Euler E est le suivant : $E = \frac{\sum Q1 - \sum Q3 - 2*\sum Qd}{4}$.

Il représente le nombre de groupes connexes moins le nombre de trous, les trous étant des cases entourées par des pions connexes (cycle) d'une meme couleur. Aussi, cette méthode est elle valable, pour nous, tant que le nombre d'Euler est supérieur (strictement) à 1.

Si E=1 alors, on utilise la *méthode 2* (décrite ci-dessus) afin de vérifier si le jeu est fini ou bien si un ou des trous existent. Heureusement les trous sont très rares dans LOA, donc cette méthode est très utile. En effet, on trouve rapidement la connexité et le nombre de groupes (contrairement à la méthode 2) ce qui sert à chaque fois que l'on joue un coup (comme nous l'avons vu), mais également dans la fonction d'évaluation (cf chapitre 4).

2.2.3 Implémentation

Comme nous le verrons plus tard l'ENVIRONNEMENT stocke les 2 tableaux de 81 carrés et les met à jour lorsqu'un coup est joué. Il y a également les attributs Q1, Q3 et Qd qui sont stockés et mis à jours.

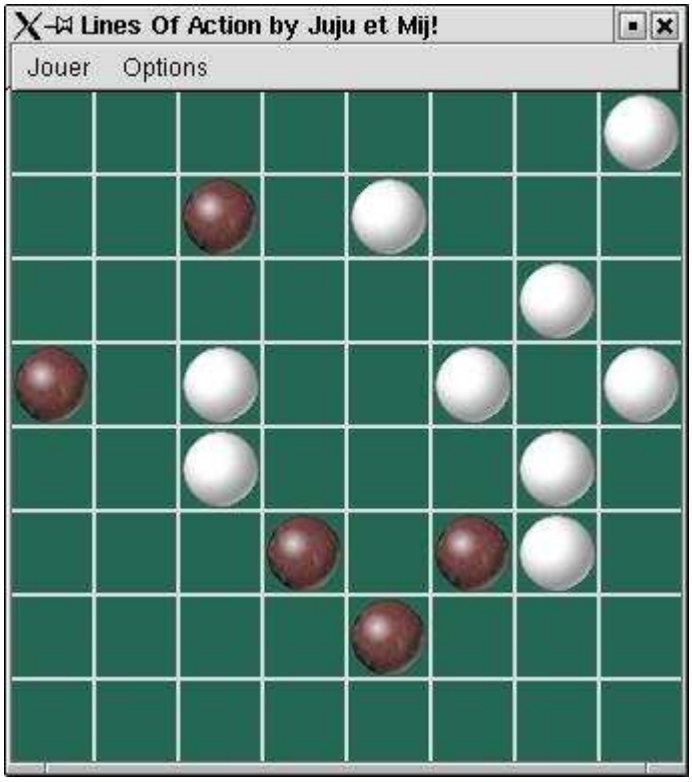
Le principe de l'implémentation est le suivant. Chaque carré est repéré par son coin inférieur droit et est représenté par un entier. Il a des coefficients pour chacune de ses quatres cases qui sont additionnés pour donner sa valeur.

Un "Switch" permet de savoir à quel type de carré correspond l'entier (Ex : 1001 est un Qd).

Si l'on reprend les carrés de la figure 5, on obtient dans l'ordre : 0, 1, 11, 1011, 1111, 1001.

Lors d'une mise à jour on a de 8 à 12 carrés (lorsqu'un pion est pris) à changer. Il n'y a alors qu'à ajouter ou retirer les coefficients correspondant (1, 10, 100 ou 1000)

Cas où la fonction d'Euler renvoie une réponse erronnée



(a)

2.3 Gestion dynamique du jeu :

2.3.1 Principe

Nous avons décidé d'implémenter pour des raisons de performances, ce que nous appelons la gestion dynamique du jeu. Il s'agit en fait de connaitre à tout instant tous les coups possibles, l'état du jeu (fini ou non ...), sans avoir à parcourir de nouveau tout le plateau de jeu en effectuant de nombreux calculs.

Ceci implique de savoir modifier, lorsque l'on joue un coup, uniquement les informations qui sont susceptibles d'avoir été changées.

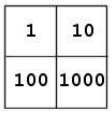
Ex : lorsque l'on joue un pion d'une case A sur une autre B, les seuls coups à mettre à jour sont ceux concernant les pions situés sur la ligne, la colonne et les 2 diagonales passant par la case A (de même pour B).

2.3.2 Intérêts $\leftrightarrow$ inconvénients

Bien sûr d'un point de vue de la programmation cette gestion est très lourde.

Il serait beaucoup plus simple d'avoir une fonction calculant, dans une grille donnée, tous les coups possibles pour un pion. Dès qu'un pion est joué, on applique cette fonction à tous les pions de la grille et on obtient tous les coups possibles. De même, dès qu'un coup est joué, on parcourt la grille afin de déterminer si la partie est finie ou non. Un autre

FIG. 6 – Codage des différents carrés



défaut de cette façon de programmer est la surchage en mémoire qu'elle implique. En effet, le nombre d'informations à stocker est beaucoup plus important puique l'on doit avoir une connaissance maximale de la situation courante pour en déduire le plus simplement possible la situation suivante.

Cependant l'avantage sous-jacent d'avoir un calcul uniquement de ce qui est nécessaire est évident. Comme nous le verrons plus tard dans la partie traitant des algorithmes de recherche, l'ordinateur descend en profondeur dans le jeu afin de déterminer la pertinence des coups à jouer. Si il peut gagner en rapidité à chaque itération (quand un coup est joué), il pourra descendre plus profond en un même temps et ainsi avoir une meilleure appréhension du jeu.

Avant d'estimer le gain de cette technique, nous tenons à préciser que le problème de place mémoire pourrait se poser dans notre cas. Nous implémentons les méthodes permettant de passer l'ENVIRONNEMENT d'un état initial (A) à un état résultat (B) d'un coup joué. Cependant nous ne nous occupons pas du passage inverse (de B à A). Or dans la recherche de l'ordinateur ceci est nécessaire puisqu'à chaque profondeur, celui-ci joue tous les coups qu'il peut, les évalue et les "dé-joue". Nous éviterons de "dé-jouer" en sauvegardant l'ENVIRONNEMENT initial et en le restaurant à la fin. Heureusement les profondeurs auxquelles le programme descend ne sont pas énormes et il n'y aura jamais plus de dix environnements sauvegardés simultanément.

2.3.3 Evaluation du gain

Nous utiliserons les variables suivantes :
- nb_{case}, nombre de cases du plateau de jeu (nb_{case}= 64)
- $nb_{direction}$, nombre total de directions ($nb_{direction}$= 8)
- nb_{pion} : Nombre de pions (nb_{pion}= 24)

Calcul en absolu :
- Coût de calcul C_{C1} du nombre de pions suivant chaque droites.
  Sur chaque ligne ou colonne, il y a 8 cases ; sur chaque diagonale de 1 à 8 cases. On a donc $C_{C1} = 64 * 4 = 256$.
- Coût de calcul C_{C2} de tous les coups possibles.
  Pour chacune des 64 cases, si la case contient un pion, on regarde dans les 8 directions si il y a un coup possible.
  $C_{C2} = nb_{case} + C_{C1} + nb_{pion} * nb_{direction} * nb_{pions_{sur-la-droite}} * k_{test}$
  k_test représentant le coup en opérations élémentaires des tests. Nous le ramènerons par la suite à 1.
  Il y a 8 lignes, 8 colonnes et 30 diagonales, soit 46 droites. Ainsi, même avec les 24 pions, on a au maximum $nb_{pions_{sur-la-droite}} = \frac{46}{24} = 2$
  donc $C_{C2} = 64 + 64 + 12 * 8 * 2 * k_{test} = 128 + 196 * k_{test}$
- Coût de calcul C_{C3} de la fin du jeu (ou non) (cf partie 2.2.2).
  La complexité de l'algorithme est $C_{C3} = nb_{direction} * nb_{pion}$, soit au maximum $C_{C3} = 2 * 8 * 12 = 192$ (dans le pire des cas) et $C_{C3} = 96$ en moyenne.
- Il n'y a pas de mise à jour effectuée.

Le coût total d'opérations est donc

$$\begin{aligned} C_{Tot_1} &= (C_{C2} + C_{C3}) * k \\ C_{Tot_1} &= (256 + 128 + 196 * k_{test} + 96) * k \\ C_{Tot_1} &= (480 + 196 * k_{test}) * k \\ C_{Tot_1} &\simeq 650 * k \end{aligned}$$

Où k est le coefficient représentant un nombre moyen d'opérations élémentaires pour chaque opération décrite ci-dessus.

Calcul en relatif :

- Coût de calcul C_{C1} du nombre de pions suivant chaque droites.
  C'est une lecture dans un tableau pour chaque direction donc $C_{C1} = 4$
- Coût de calcul C_{C2} de tous les coups possibles.
  C'est également une lecture, chaque pion contenant un tableau des coups pour chaque direction $C_{C2} = \frac{nb_{pion}}{2} * 8 = 96$
- Coût de calcul C_{C3} de la fin du jeu.
  C'est le *calcul d'Euler* et $C_{C3} = 1$
- La complexité se trouve ici lors des mises à jour (pion déplacé d'un endroit à un autre).
- $C_{maj_1} = 8$ (une opération pour chaque direction)
- C_{maj_2} : On ne remet à jour que les pions susceptibles d'avoir un changement et que sur les directions concernées (2) et ceci suivant 7 droites.
  $C_{maj_2} = nb_{pions-concernes} * 2 * nb_{pions_{sur-la-droite}}$, on a alors $C_{maj_2} = 24 * 2 * 2 * k_{test} = 96 * k_{test}$
  Ceci correspond à la complexité maximale, en réalité, on peut considérer que :
  $nb_{pions-concernes} = \frac{46}{24} * 7 = 13$ et $nb_{pions_{sur-la-droite}} = 2$, d'où $C_{maj_2} = 13 * 2 * 2 * k_{test} = 52 * k_{test}$
- C_{maj_3} : On remet à jour 8 carrés de la couleur qui joue et eventuellement un carré de la couleur adverse (lors d'une prise) donc au pire $C_{maj_3} = 12$

Le coût total d'opérations est donc

$$\begin{aligned} C_{Tot_2} &= (C_{C2} + C_{C3} + C_{maj_1} + C_{maj_2} + C_{maj_3}) * k \\ C_{Tot_2} &= (96 + 1 + 8 + 52 * k_{test} + 12) * k \\ C_{Tot_2} &\simeq 150 * k \end{aligned}$$

Où k est le coefficient représentant un nombre moyen d'opérations élémentaires pour chaque opération décrite ci-dessus

On peut également dans le cas du calcul en relatif calculer le nombre d'opérations effectuées lors de la sauvegarde de l'environnement :

Il faut sauver :

- un plateau de 64 cases (nécessaire aussi dans le calcul en absolu)
- deux tableaux des carrés de connexité de 81 cases 24 pions de 19 entiers d'où, à chaque étape 81 * 2 + 24 * 19 = 618 opérations élémentaires supplémentaires (au maximum)

On obtient donc en considérant qu'il y a toujours 24 pions (on est donc dans un cas maximum) $C_{Tot_1} = 650 * k$ et $C_{Tot_2} = 150 * k + 618$

En prenant k=3, $C_{Tot_1} = 1950$ et $C_{Tot_2} = 1068$. Le calcul en relatif sera alors $\frac{C_{Tot_1}}{C_{Tot_2}} = 1,8$ fois plus rapide.

Ainsi en imaginant une moyenne de 3 opérations élémentaires par fonctions (ce qui est très peu), théoriquement, nous doublons les performances. Cependant, il faut rester prudent car nous avons fait beaucoup d'approximations.

3 Algorithmes de recherche :

Dans cette partie, nous ne développerons pas les différents algorithmes existants. En revanche nous y expliquons brièvement la façon dont nous implémentons l'algorithme choisi.

Il s'agit d'un alpha-bêta amélioré. Il utilise les tables de transposition, c'est un alpha-bêta à mémoire. Pour profiter pleinement des avantages des algorithmes à mémoire, on appliquera une technique de fenêtrage s'inspirant de SCOUT.

Rappelons le principe de cette technique. Le programme appelle autant de fois que nécessaire l'alpha-bêta afin de trouver un encadrement de la valeur associée à la position. Lorsque les deux bornes (sup et inf) se rencontrent, on a trouvé la valeur. En fait, on utilise une fenêtre de taille 0, ce qui entraîne un élagage important de l'arbre. Grâce aux tables de transposition, les positions atteintes plusieurs fois ne sont pas ré-étudiées. Evidemment cet algorithme est d'autant plus efficace que la valeur initiale est bien choisie. Pour ceci on lance d'abord l'alpha-bêta "simple" afin d'avoir un ordre d'idée correct.

3.1 Tables de transposition

3.1.1 Principe

On se sert d'une table de hachage pour les stocker. On désire associer une position du plateau de jeu à une valeur.

Pour ceci nous utilisons la technique suivante : On associe à chaque case du jeu deux valeurs choisies aléatoirement (une pour chaque couleur). Nous avons donc 128 nombres pris au hasard. Pour obtenir la valeur associée à une position on effectue l'opération XOR sur l'ensemble des nombres correspondants aux coordonnées des pions. Cette méthode est appelée méthode de Zobrist-Hashing (Zobrist, 1970). Son principal intérêt est de pouvoir obtenir incrémentalement les positions qui suivent par la formule :

Valeur (nouvelle pos) = Valeur (ancienne pos) XOR Valeur (case de départ du pion) XOR Valeur (case d'arrivée du pion).

Lorsqu'il y a prise d'un pion, on ajoute XOR Valeur (case d'arrivée du pion, couleur adverse).

3.1.2 Valeurs choisies

Maintenant, nous allons définir les caractéristiques de ces tables de transposition. Ces valeurs sont susceptibles d'être modifiées après les premiers tests. Nous prenons les valeurs de hachage sur 32 bits (ou 64 bits voir ci-dessous), mais comme $2^{32} > 4 * 10^9$, la table n'est pas stockable sur une machine, ainsi nous décidons de garder uniquement les 16 bits de poids fort comme index de la table et il faudra donc gérer les conflits entre deux positions différentes ayant le même début de valeur de hachage).

Dans la table de transposition nous stockons les composants suivants :

- clef : On place ici les 16 bits de poids faible de la valeur de hachage
- meilleur_coup : Il s'agit des coordonnées du meilleur coup (2*3 bits pour abscisses + 2*3 bits pour ordonnées) soit 12 bits
- haut : borne supérieure sur 16 bits
- bas : borne inférieure sur 16 bits
- profondeur : profondeur du sous arbre sur 4 bits

Les composants font alors 64 bits chacun.

3.1.3 Utilisation

L'utilisation des tables de transposition se fait de la façon suivante :

Le cas idéal : la profondeur à laquelle on cherche est inférieure ou égale à la profondeur stockée dans la table. A ce moment là, si bas >= bêta retourne bas et si haut <= alpha, on retourne haut sinon alpha = max (alpha, bas) et bêta = min (bêta, haut)

Second cas : si la profondeur à laquelle on cherche est supérieure à ce que nous avons dans la table, on peut s'en servir uniquement pour ordonner les coups.

3.2 Gestion des erreurs

Nous choisissons (dans un premier temps) une gestion des collisions élémentaire : c'est le dernier élément entrant dans la table qui y reste.

Le véritable problème provient d'un deuxième type d'erreurs. Si deux positions ont exactement la même valeur, on peut arriver à des erreurs d'évaluation (faire notemment attention au meilleur coup stocké dans la table).

Evaluons la probabilité d'une telle erreur. Avec N le nombre de positions distinctes possibles et M le nombre de positions à sauver dans la table, la probabilité de non erreur est la suivante : $P = e^{-\frac{M^2}{2N}}$

Sachant que nous avons codé la valeur de hachage sur 32 bits $N = 2^{32}$ et nous avons vu que pour jouer un coup, il fallait au minimum $\frac{1000}{10^9} = 10^{-6}$ secondes. Nous allons supposer que l'on peut examiner jusqu'à 100000 noeuds par seconde.

A ce moment là pour une recherche de 1 minute, $M = 60 * 10^5 = 6 * 10^6$ et, la probabilité qu'il y ait au moins une erreur est :

$P = 1 - e^{-\frac{(6*10^6)^2}{2*2^{32}}} = 1$. Pour une recherche de 1 s, P = 0,7.

Il y aura donc au moins une erreur par recherche. Pour réduire largement ce risque, on peut passer à une valeur de hachage sur 64 bits par exemple. A ce moment la probabilité d'avoir une erreur au moins sera :

$P = 1 - e^{-\frac{(6*10^6)^2}{2*2^{64}}} = 10^{-6}$

La probabilité est donc quasiment nulle. Cependant dans ce cas, il faut stocker 48 bits dans la clé de la table de transposition. Un choix sera à faire entre ces solutions. Mais il semble peut être plus prudent d'éviter ce type d'erreur qu'on ne peut pas maîtriser ni repérer. Bien sûr, en fonction du temps disponible, nous tenterons d'améliorer notre algorithme (gestion plus fine des collisions...).

4 Heuristiques :

4.1 Présentation de différentes heuristiques :

Avant de présenter les différentes heuristiques choisies, voici nos deux critères principaux de sélection. Nous avons voulu créer des heuristiques pouvant allier une bonne évaluation du jeu tout en ne nécessitant pas de lourds calculs.

4.1.1 La connexité

Il s'agit de l'heuristique la plus évidente. Le but du jeu étant de connecter tous ses pions, il parait indispensable de calculer la différence entre les nombres de groupes de chaque couleur. Cette heuristique est à la fois très facile à implémenter et très rapide à exécuter car il suffit de faire la différence des deux nombres d'Euler dans l'ENVIRONNEMENT.

En fait pour cette première fonction d'évaluation tout à déjà été fait. Ainsi elle parait très performante puisqu'elle "colle" au jeu et ne demande aucune ressource supplémentaire, de plus elle légitimise l'utilisation du *"nombre d'Euler"*.

On peut également tirer parti des informations sur les types de "carrés" existants. Plus une couleur possède de nombreux carrés avec 3 ou 4 pions, plus elle a de chances d'arriver au but sans encombres. En effet, l'adversaire devra "prendre" plusieurs pions pour casser des groupes.

Enfin, pour traiter le problème récurrent à cette solution, lorsqu'un des nombres d'Euler est à 1, on ajoute 1 (ou 2, si l'on veut pénaliser la situation) car si la position est effectivement terminale, on part dans l'évaluation finale qui sera décrite plus tard.

Ceci reste une approximation, mais le cas d'un trou ne se présentant qu'une fois sur 100 au maximum, on préfère faire comme ceci plutôt que de calculer le nombre exact de groupes, ce qui prend beaucoup de temps.

Cependant, même si ces fonctions paraissent excellentes, elles ne peuvent suffir. En effet, elles ne sont pas idéales dans tous les cas. Par exemple au départ, n'importe quel coup valable augmentera la connexité. De plus, les meilleurs coups ne seront pas forcemment ceux qui limitent le nombre de groupes dès le départ. Dans ce cas, il parait plus opportun de regrouper, rapprocher les pions.

D'autre part, une couleur peut avoir plusieurs carrés Q3 ou Q4, mais, si ceux-ci sont dispersés, la situation n'est pas particulièrement avantageuse. Pour combler ces problèmes, une méthode de mesure de la distance entre les pions est exposée dans le paragraphe suivant.

4.1.2 Le centre de gravité

Pour cette heuristique, on calcule le centre de gravité des pions (pour chaque couleur). Ensuite, on mesure la distance de chaque pion à ce centre. Cette méthode permet d'obtenir différentes informations.

On peut tout d'abord faire la somme S des distances (signées), dans ce cas une position est bonne si la différence de la valeur absolue de S adverse et la valeur absolue de S amie est positive. La lacune de cette technique se fait sentir par exemple lorsque tous les pions sont regroupés sauf 1 pour une couleur. La somme S de cette couleur peut alors être très faible, malgré la grande distance d'un pion au centre de gravité de sa couleur. Une solution simple pour limiter ce problème est d'effectuer parallèlement la somme des distances non signées, ou même de ne faire que cette somme. En terme de calculs cette heuristique demande de parcourir deux fois les listes de pions (48 itérations au maximum). Cette fonction semble

elle aussi être bonne car même si elle demande un peu plus de calcul, elle reste très proche de la “philosophie” du jeu et correspond bien à la vision humaine que l’on peut avoir. En effet lorsque l’on joue on tente dans un premier temps de regroupé tous ses pions dans un même endroit.

4.1.3 Le nombre de pions "prenables"

Nous avons déjà dit que le but même de LOA étant d’arriver à un groupe connexe, le fait de prendre des pions adverses n’est pas forcemment une bonne chose. Cependant, il peut être très utile, lorsque l’adversaire est sur le point de gagner, de lui manger un pion qui brise un groupe. Ou bien encore de placer un pion à un endroit stratégique en prenant un pion adverse. Pour ceci, il est intéressant de faire la différence entre les pions que l’on peut prendre et ceux que l’adversaire peut nous attaquer. Pour calculer ce nombre, on parcourt les listes de pions qui contiennent eux-même les coups possibles. Pour chaque coup, on regarde si l’on mange un pion adverse ou non. Cette heuristique paraît difficilement utilisable seule, pourtant, elle peut être très utile. Cette heuristique coûte aussi assez cher bien qu’une fois de plus, le fait d’avoir tout gardé dans l’ENVIRONNEMENT nous aide beaucoup. On doit parcours les deux listes de pions et dans chacun les coups possibles, soit au maximum 24 * 8 = 192 itérations.

4.1.4 Le nombre de "degrés de liberté"

Un autre facteur important dans le jeu (comme le nombre de pions "prenables") est le nombre de coups que l’on peut jouer avec chacun des pions. En effet une position très néfaste est le cas où un pion est totalement bloqué. De façon plus générale, moins un pion a de possibilités pour jouer moins la situation est bonne.

On peut également envisager de pondérer ceci par la distance du pion au centre de gravité.

On calcule ce nombre en même temps que le nombre de pions "prenables". Le coût calculatoire de cette fonction est par conséquent le même.

4.1.5 L’évaluation finale

Lorsque la position examinée est une position terminale, on l’évalue d’une manière un peu différente. On prend en compte le nombre de groupes de l’adverse, puis le nombre de coups qui ont été joués depuis le départ qui seront stockés soit dans le cerveau, soit dans l’environnement.

4.1.6 Récapitulation des différentes heuristiques retenues

- Nombre d’Euler
- Nombre de carrés Q3 et Q4 avec des pondérations sur la quantité de Q3 et de Q4
- Somme des éloignements absolus au centre de gravité. On testera des éloignements linéaires ou exponentiels (coefficient coeff)
  Ex : Un pion ayant pour distance sur l’abscisse 2 et 3 sur l’ordonnée pourra avoir un éloignement de $2^{coeff} + 3^{coeff}$
- Nombre de pions "prenables"
- Nombre de "degrés de liberté" avec des pondérations, afin d’augmenter les différences entre le nombre de coups possibles.

4.2 Système de calculs de coefficients

La principale difficulté est de connaître en quelles proportions ces différentes heuristiques doivent être prises en compte. Il est évident que certaines sont plus importantes que d'autres, mais il est très difficile de quantifier ceci.

De plus l'importance des heuristiques peut évoluer durant la partie ; nous avons vu qu'au départ, l'heuristique calculant le nombre de composantes connexes n'était pas représentative, mais en fin de partie elle revêt au contraire un rôle primordial.

Cependant, nous n'implémenterons pas dans un premier temps l'évolution des coefficients au cours du jeu. En revanche, nous avons décidé, plutôt que de déterminer l'importance des heuristiques de façon purement intuitive (voire hasardeuse) de créer un système d'essais de coefficients. Ce système s'inspire un petit peu des algorithmes génétiques (de façon très simplifiée).

Nous aurons tout d'abord un petit programme créant des fichiers de coefficients au hasard, suivant des contraintes fixées elles aussi dans un fichier de paramètres. Dans la classe jeu, on ajoute une fonction f qui est une boucle lançant des parties de l'ordinateur contre l'ordinateur. A chaque initialisation, f choisit au hasard deux fichiers d'initialisation, puis crée deux cerveaux ayant chacun en argument un des deux fichiers (les deux cerveaux doivent être différents). La partie est lancée, lorsqu'elle se termine, f met à jour les 2 fichiers f_gagnant et f_perdant de la façon suivante (avec eval l'évaluation finale pour le gagnant) :

- coeff = 1 si un des deux fichiers n'a jamais servi
- coeff = max (1, $1+\frac{nombre-de-points-de-f_{perdant}}{nombre-de-parties-de-f_{perdant}} - \frac{nombre-de-points-de-f_{gagnant}}{nombre-de-parties-de-f_{gagnant}}$) . Ce coefficient permet de majorer une victoire d'un "faible" contre un "fort".
- Le nombre de parties jouées par $f_{gagnant}$ et $f_{perdant}$ est incrémenté
- nombre de points pour $f_{gagnant}$ = nombre de point $f_{gagnant}$ + eval * coeff
- nombre de points pour $f_{perdant}$ = nombre de point $f_{perdant}$ - eval * coeff

Lorsque un fichier a dépassé un certain nombre de parties (fixé à l'avance), différents cas se présentent :

- Son nombre de points est inférieur à un seuil minimal : il est supprimé, étant considéré comme mauvais.
- Son nombre de points est compris entre le seuil minimal et le seuil maximal : il ne se passe rien.
- Son nombre de points est supérieur au seuil maximal et il s'est battu avec un fichier dans la même situation : un croisement est effectué entre les deux fichiers pour en donner un troisième.
- Son nombre de points est supérieur au seuil maximal et il ne s'est pas battu avec un fichier dans la même situation : le fichier se dédouble avec des mutations.

Tout s'arrête lorsqu'il n'y a plus qu'un fichier qui se trouve être le meilleur. Nous espèrons ainsi trouver une bonne fonction d'évaluation.

5 L'application créée

5.1 Installation

Pour installer LOA, il suffit de désarchiver le fichier tar.gz qui créé un répertoire "loa". Il faut ensuite faire un "make". L'exécutable s'appelle *loa.*

5.2 Utilisation

L'application peut s'utiliser en mode texte ou en mode graphique.

5.2.1 Mode texte

Il y a trois mode principaux. Pour chacun, il faut taper *./loa* + un attribut.
- attribut "-c" lance l'application en mode caractère.
- attribut "-s" lance le nombre de parties voulues (demandé en intéractif) entre un cerveau ALPHA-BÊTA à profondeur 3 et un cerveau MTD à profondeur 3.
- attribut "-g" lance le pseudo algorithme génétique pour le nombre de cerveaux voulus (demandé en intéractif). Il reste un petit bogue dans cette partie. Nous renverrons l'archive correcte dès que celui-ci sera réglé.

5.2.2 Mode graphique

Il suffit de lancer avec ./loa. L'interface graphique est classique et réalisée en GTK. On peut choisir parmi 5 niveaux correspondant respectivement aux algorithmes suivants : MINIMAX, NEGAMAX, ALPHA-BÊTA, ALPHA-BÊTA à MÉMOIRE et MTD.

Une fonctionnalité très pratique de cette interface s'obtient en sélectionnant AIDE du menu OPTION. A ce moment là, lorsqu'un pion est cliqué, toutes les cases accessibles sont indiquées sur le plateau.

5.3 Différences entre ce rapport et l'application

- Nous avons codé les valeurs de hachage pour les tables de transposition sur 32 bits car il s'est avéré que le programme effectue plutôt 10 à 20000 opérations par secondes contrairement aux 100000 auxquelles nous nous attendions. Ceci limite donc considérablement l'erreur possible, nous avons donc préféré réduire l'espace mémoire utilisé plutôt que d'éviter à tout pris d'avoir un coup qui ne veut rien dire. Cependant, nous avons bien sûr détecté et traité cette erreur (en regardant simplement si le coup fait partie de la liste de coups possibles).
- Le pseudo algorithme génétique est codé, mais pas encore opérationnel, nous n'avons donc pas pu encore affiner notre fonction d'évaluation qui dépend de beaucoup de paramètres.

Voici donc les différentes stratégies que nous avons décidé d'implémenter pour ce jeu. Ce qui est particulièrement intéressant dans Lines of Actions, c'est le but original de connexité. La solution pour la traiter grâce au nombre d'Euler est inspirée de l'analyse que nous avons trouvé dans la thèse de M.H.M. Winands intitulée *Analysis and Implementation of Lines of Action* d'Août 2000. Nous nous sommes également inspirés pour l'algorithme MTD et l'ALPHA-BÊTA en mode graphique du châpitre "La programmation des jeux" du livre "Intelligence Artificielle et Informatique théorique" de Jean-Marc Alliot et Thomas Schiex.