Chapitre 6

Architecture d'un programme de *Lines of action*

6.1. Introduction

Nous présentons BING (BING *is not* GNU Chess), un programme de Lines of action écrit en réutilisant une grande partie du code de GNU Chess. L'algorithme de recherche *alpha-beta* a été gardé sans grande modification. La fonction d'évaluation a été réécrite ; elle est basée sur six composantes qui sont données en entrée à un petit réseau de neurones, dont l'apprentissage a été fait par l'algorithme des différences temporelles.

Notre programme utilise donc des méthodes relativement classiques. Sa particularité est d'avoir été développé rapidement (environ un moi et demi), et d'avoir malgré cela atteint un niveau proche des meilleurs programmes de Lines of action. Il a d'abord été développé pour participer à un tournoi interne à Paris 8. Il a fini premier *ex-æquo* de ce tournoi, avec un revers inattendu dans une de ses deux parties contre un programme de T. Goossens. Il a ensuite fini second sur 3 participants aux 8^{e} *Computer Olympiads* de 2003 à Graz, avec un score de 5/8. Il a réussi la performance de gagner une partie contre le vainqueur, MIA de M. Winands. Aux 9^{e} *Computer Olympiads* de 2004 à Ramat-Gan, il a encore fini 2^{e} sur 4 participants, avec un score de 8/12 ; derrière MIA mais devant YL de Y. Bjornsson, le vainqueur des 5^{e}, 6^{e} et 7^{e} *Computer Olympiads*.

Chapitre rédigé par Bernard HELMSTETTER et Tristan CAZENAVE.

6.2. Le jeu de Lines of action

Le jeu de Lines of action est relativement récent ; il s'est un peu développé depuis les années 1990, à la fois chez des joueurs humains et chez les programmeurs. Actuellement, les programmes ont un net avantage sur les joueurs humains.

Lines of action se joue entre deux joueurs sur un damier 8x8. La position initiale est celle de la figure 6.1. Les joueurs jouent à tour de rôle, en commençant par Noir. Un coup consiste à déplacer un pion de sa couleur sur une ligne horizontale, verticale ou diagonale, du nombre exact de pions qui se trouvent sur cette ligne. Un pion peut sauter par dessus des pions amis, mais pas par dessus des pions ennemis ; il ne peut pas atterrir sur un pion ami mais peut atterrir sur un pion ennemi, et dans ce cas ce pion est capturé (figure 6.2). Le but pour chaque joueur est de former une seule composante 8-connexe avec ses pions. La partie est nulle en cas de répétition de position, ou si les deux joueurs forment une seule composante en même temps.

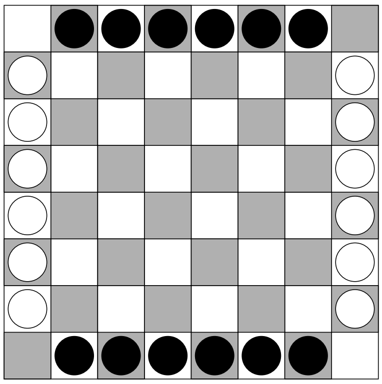

Figure 6.1. *Position initiale*

6.3. Réutilisation de GNU Chess

Nous avons réutilisé une grande partie du code de la version 5.07 de GNU Chess. GNU Chess n'est pas parmi les meilleurs programmes d'échecs. Même parmi les programmes libres, il est largement battu par des programmes comme Crafty ou Fruit. Son attrait pour nous est la simplicité du code, qui est bien commenté. Il consiste en 13 700 lignes de C, avec essentiellement un seul gros fichier d'entêtes. L'adaptation du code pour Lines of action n'a pas posé de difficulté, et a résulté en un gain de temps considérable.

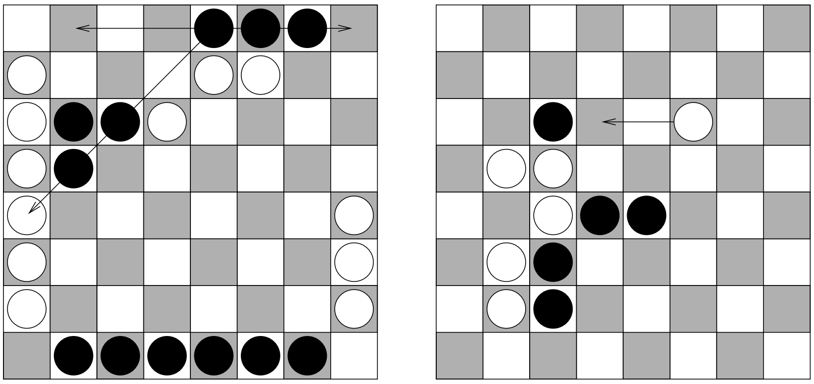

Figure 6.2. *Exemples de coups, et une position finale gagnée après le coup blanc*

L'interface de jeu a été gardée avec peu de modifications. L'essentiel du code de recherche a été gardé, c'est-à-dire : un *alpha-beta* avec negamax, approfondissement itératif, *principal variation search*, le coup nul, deux *killer moves*, l'*history heuristic* et une table de transposition [MAR 86]. Certaines optimisations auraient demandé du travail pour être adaptées à Lines of action et ont été supprimées :

– la recherche de quiescence,

– le livre d'ouvertures,

– les extensions de la profondeur de recherche : dans GNU Chess, la recherche est étendue en cas de menaces sur les rois, de certaines captures et de menaces de promotions de pions.

– le *razoring* et le *futility pruning* qui consistent, lorsque l'évaluation est particulièrement mauvaise et que la profondeur de recherche restante est faible, à diminuer la profondeur de recherche ou à supprimer certains types de coups.

Au niveau des optimisations utilisées, notre programme est donc similaire à MIA [WIN 00], avec la recherche de quiescence et le livre d'ouvertures en moins.

Les plus grandes modifications ont porté sur la représentation du jeu, la gestion des coups et la fonction d'évaluation. La modification des règles du jeu a été facilitée par les similitudes entre les deux jeux, notamment la taille du damier (8x8). GNU Chess utilise une représentation du jeu basée sur les *bitboards*, qui sont en fait des entiers de 64 bits. Plusieurs *bitboards* sont utilisés pour représenter soit l'ensemble des pièces d'une couleur, soit seulement des pièces d'un certain type. La plupart des opérations sont accélérées par des manipulations booléennes sur les bits. Ceci est aidé par le précalcul de divers *bitboards* liés aux divers déplacements possibles. Ceci est aussi aidé par le maintien simultané de *bitboards* « tournés » de 45, 90 ou 315 degrés par rapport aux *bitboards* initiaux. Il est ainsi possible d'extraire rapidement des lignes, mais aussi des colonnes et même des diagonales de la position, et de représenter leur contenu comme des nombres binaires d'au plus 8 bits. Ces nombres peuvent, par exemple, être utilisés comme indices dans des tableaux précalculés, pour obtenir les mouvements possibles sur cette ligne, colonne ou diagonale. Les lignes peuvent être extraites trivialement à partir du bitboard initial, les colonnes à partir d'un bitboard changeant le rôle des lignes et des colonnes, et les diagonales à partir d'un des deux *bitboards* correspondant aux permutations des bits de la figure 6.3.

0	2	5	9	14	20	27	35
1	4	8	13	19	26	34	42
3	7	12	18	25	33	41	48
6	11	17	24	32	40	47	53
10	16	23	31	39	46	52	57
15	22	30	38	45	51	56	60
21	29	37	44	50	55	59	62
28	36	43	49	54	58	61	63

28	36	43	49	54	58	61	63
21	29	37	44	50	55	59	62
15	22	30	38	45	51	56	60
10	16	23	31	39	46	52	57
6	11	17	24	32	40	47	53
3	7	12	18	25	33	41	48
1	4	8	13	19	26	34	42
0	2	5	9	14	20	27	35

Figure 6.3. *permutations des cases pour les* bitboards *tournés de 45 et 315 degrés*

Tout ceci est facile à adapter pour les règles de Lines of action. Nous utilisons dans BING deux *bitboards* pour les pions de chacune des couleurs. Par exemple, la position initiale est représentée par les deux *bitboards* 0x7E0000000000007E pour Blanc et 0x0081818181818100 pour Noir. Nous utilisons aussi trois *bitboards* tournés pour chaque couleur. Dans notre cas, l'intérêt des *bitboards* tournés et d'éviter, quand on cherche les mouvements possibles d'un pion dans une direction, de devoir exécuter une boucle coûteuse sur les cases dans cette direction.

Notre programme recherche environ 100 000 nœuds par seconde sur un Pentium 4 3,2 GHz, contre environ 300 000 pour GNU Chess. Cette différence est due à la lenteur relative de notre fonction d'évaluation. La fonction d'évaluation de GNU Chess est pourtant plus complexe, avec beaucoup plus de cas particuliers traités, mais notre fonction d'évaluation évalue des composantes assez coûteuses (il serait sans doute possible d'accélérer le calcul de ces composantes en le rendant incrémental, mais ce serait nuisible à la simplicité du code). De plus, GNU Chess utilise des coupes d'évaluation paresseuse (*lazy evaluation cuts*), permettant de se contenter d'une évaluation rapide si elle est suffisante.

6.4. Fonction d'évaluation

Nous définissons les composantes de notre fonction d'évaluation, le réseau de neurones qui fournit l'évaluation finale et les méthodes d'apprentissage.

6.4.1. *Les différentes composantes*

La fonction d'évaluation repose sur plusieurs composantes qui sont ensuite données en entrée d'un réseau de neurones. Nous nous sommes inspirés des composantes décrites dans [WIN 03]. Nous commençons par calculer le centre de gravité des pions des ensembles de pions de chaque couleur. Nous définissons l'*excentrisme* d'un pion comme la distance au centre de gravité de la couleur. Quand nous considérons une distance, il s'agit toujours de la distance $d_\infty((x_1, y_1), (x_2, y_2)) = \max(|x_1 - x_2|, |y_1 - y_2|)$. En effet. il s'agit du nombre minimal de déplacement nécessaires en 8-connexité pour se déplacer d'une case à une autre.

Pour chacun des joueurs, les composantes sont :

1) Le nombre de pions de la couleur ;

2) La distance moyenne au centre de gravité des pions de la couleur ;

3) La distance entre le centre de gravité de la couleur et le centre du damier ;

4) Une évaluation du nombre de composantes 8-connexes de la couleur et de leurs excentrismes (l'excentrisme d'une composante connexe est défini comme l'excentrisme minimal des pions qui la composent). Il est évidemment bon d'avoir peu de composantes connexes ; cependant, une composante connexe excentrée, composée de n pions, est presque aussi mauvaise que n pions séparés, puisque ces pions devront probablement être déplacés séparément pour les connecter au reste des pions. Ces idées amènent à la formule suivante : une composante connexe de n pions compte pour 1 si elle est peu excentrée (distance inférieure à 1,5), pour n si elle est très excentrée (distance supérieure à 2,5), avec une transition affine pour les distances intermédiaires ;

5) Un évaluation des connexions entre pions adjacents de la couleur, une connexion comptant plus si elle est proche du centre de gravité de la couleur. Plus précisément, nous donnons à chaque connexion la valeur $\max(2,5 - d, 0)$, où d est la distance au centre de gravité de la couleur, et nous calculons la somme sur toutes les paires de pions adjacents de la couleur. Cette composante donne l'avantage à des formes ramassées, qui sont connectées de nombreuses façons différentes et donc plus « solides » ;

6) Une évaluation du nombre de pions de la couleur « bloqués » par des pions adverses. Nous considérons qu'un pion est bloqué si il est à distance supérieure à 2 du centre de gravité, et si les directions qui lui permettraient de se rapprocher du centre de gravité sont bloquées par des pions adverses adjacents. L'évaluation précise est un peu compliquée ; le blocage d'un pion dépend de la distance exacte au centre de gravité et de la meilleure direction libre qu'il possède, dans le sens où elle le rapproche ou au moins ne l'éloigne pas trop du centre de gravité. La formule précise est :

$$\max(d-2,0) \times \max(0.5+\min(\min_{\text{dir. libre}}(d_2-d),1),0),$$

où d est la distance originale au centre de gravité, d_2 la distance au centre de gravité d'une case adjacente libre, et où on considère que le minimum d'un ensemble vide est infini. Ainsi, la partie droite de l'expression varie entre 1,5 pour des pions ne pouvant même pas reculer et 0 pour des pions dont une des cases adjacentes libres lui permet d'avancer d'au moins 0,5.

Les composantes 4, 5 et 6 ressemblent à des composantes utilisées dans d'autres programmes tels que MIA [WIN 03], mais les détails sont un peu originaux. Nous avons fait en sorte, pour chacune des composantes, de ne pas introduire de variations brusques mais d'utiliser des fonctions continues. Il y a des paramètres un peu arbitraires qu'il aurait été fastidieux de tester rigoureusement. Une composante qui aurait pu être rajoutée, et qui est présente dans YL et MIA, est le nombre de coups possibles pour chaque joueur.

Il n'est pas évident de décider si la composante sur le nombre de pions doit intervenir positivement ou négativement (il est plus facile de se connecter avec peu de pions, mais plus facile de gêner l'adversaire avec beaucoup de pions). De plus, les composantes 4 et 5 évaluent des choses un peu similaires, et sont fortement corrélées au nombre de pions. Nous avions d'abord utilisé une combinaison linéaire de ces composantes avec des poids qui nous ont paru raisonnables, mais nous l'avons ensuite remplacée par un réseau de neurones, avec apprentissage des poids par différences temporelles. L'amélioration du niveau a été très claire.

6.4.2. *Calcul du nombre d'Euler avec les quads*

Le nombre d'Euler d'une partie finie d'un plan discrétisé est la différence entre le nombre de composantes connexes (dans notre cas, 8-connexes) et le nombre de trous. Ce nombre peut être calculé efficacement, et surtout incrémentalement, avec les *quads* [GRA 71].

En pratique, ce nombre est une bonne estimation du nombre de composantes connexes, puisque les trous dans les composantes sont rares. Le nombre de composantes brut n'est pas utilisé dans notre fonction d'évaluation (puisque la composante 4

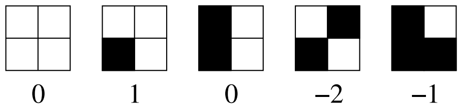

Figure 6.4. *Poids des quads pour le calcul du nombre de composantes connexes*

fait intervenir l'excentrisme des composantes), mais le nombre d'Euler reste utile pour les tests de victoire. En effet, si le nombre d'Euler est strictement supérieur à 1, il y a certainement plus d'une composante connexe, et le calcul exact n'est pas nécessaire.

Le nombre d'Euler se calcule par une somme sur chacun des carrés 2x2 du plateau (y compris sur les bords en dépassant d'une case), avec les poids de la figure 6.4 (les autres carrés qui se déduisent par rotation ont les mêmes poids). Ces poids correspondent au nombre de quarts de tours que font les frontières des composantes connexes. Diviser la somme par quatre donne donc le nombre d'Euler. L'intérêt de ce calcul est la possibilité de calcul incrémental quand un coup est joué.

6.4.3. *Apprentissage*

Nous présentons notre méthode d'apprentissage par différences temporelles, pour l'apprentissage des poids d'un petit réseau de neurones. Nous comparons avec des programmes similaires dans d'autres jeux. Un succès notable de l'algorithme des différences temporelles est le programme de Backgammon TD-Gammon [TES 02] ; cependant le domaine est stochastique et les résultats doivent donc être comparés avec précaution. Une tentative d'appliquer la méthode dans des jeux reposant plus sur la recherche, comme les échecs, est le programme KnightCap [BAX 00]. Le programme de Lines of action MIA a aussi utilisé des différences temporelles pour l'apprentissage des poids d'une combinaison linéaire [WIN 02]. Un autre exemple d'application est Neurogo [ENZ 03].

6.4.3.1. *Le réseau de neurones*

Nous utilisons un petit réseau de neurones, prenant en entrées les composantes ci-dessus, pour calculer l'évaluation d'une position. La taille de ce réseau est limitée par des contraintes de temps de calcul ; et de toute façon l'intérêt d'utiliser un gros réseau de neurones n'est pas clair, puisqu'il est plutôt habituel d'utiliser de simples combinaisons linéaires dans ce genre de programmes en *alpha-beta* (comme le font par exemple MIA ou KnightCap). Nous avons choisi un réseau avec une couche cachée, mais seulement 2 neurones dans cette couche. Nous aurions aussi obtenu de bons résultats avec une simple combinaison linéaire ; en fait, des essais avec un seul neurone dans la couche cachée (donc essentiellement une combinaison linéaire) ont donné des résultats très proches.

Les entrées de ce réseau de neurones sont au nombre de 14 : les 6 composantes pour chacun des joueurs, une entrée indiquant le trait (± 1) et une entrée de biais fixée à 1.

Les poids du réseau de neurones sont modifiés par l'algorithme classique de rétro-propagation. Nous avons également implémenté l'algorithme RPROP [RIE 93] qui est une amélioration, mais ce n'était pas forcément nécessaire puisque l'apprentissage est suffisamment rapide avec l'algorithme classique (quelques minutes).

6.4.3.2. *Les différences temporelles*

L'apprentissage s'est fait par l'algorithme des différences temporelles [SUT 88] TD(λ). Étant donnée une partie, cet algorithme fait apprendre au réseau de neurones, dans chacune des positions de la partie, l'évaluation telle qu'elle est fournie par le programme lui-même mais dans les positions ultérieures. La méthode dépend d'un paramètre λ. La valeur $\lambda = 0$ signifie que le programme apprend l'évaluation de la position immédiatement suivante, alors que la valeur $\lambda = 1$ signifie que c'est la valeur finale de la partie qui est apprise, et toutes les valeurs intermédiaires sont possibles. Nous avons choisi de fixer $\lambda = 0,9$, ce qui est une valeur plutôt élevée. Le programme TD-Gammon a utilisé $\lambda = 0,7$ et $\lambda = 0$; Neurogo a utilisé $\lambda = 0$; MIA a utilisé $\lambda = 0,8$. G. Tesauro déconseille des valeurs de λ trop proches de 1 [TES 02], mais le jeu Lines of action étant moins stochastique que le Backgammon, nous pensons qu'une valeur de λ plus élevée est justifiée.

L'apprentissage de notre programme s'est fait à partir d'une base statique de quelques centaines de parties. Les différentes bases que nous avons utilisées contenaient entre 600 et 4 000 parties. Les parties sont jouées en boucle jusqu'à stabilisation des poids du réseau de neurones. En cela, notre approche diffère de celle, plus habituelle, consistant à faire apprendre au programme des parties jouées dynamiquement par la version courante du programme contre lui-même (comme TD-Gammon, Neurogo ou MIA) ou bien entre le programme et des joueurs humains (comme KnightCap, 308 parties jouées sur le serveur Internet FICS). Un avantage de notre méthode est la plus grande rapidité d'apprentissage, puisque moins de parties doivent être construites. Les poids du réseau de neurones de BING se stabilisent après environ 500 lectures de la base de parties.

Notre application de l'algorithme des différences temporelles a une différence avec l'algorithme original : les coups qui sont appris sont produits par une recherche en *alpha-beta*, et non par une simple recherche à profondeur 1. Avec la terminologie de [BAX 00], l'algorithme que nous utilisons s'appelle en fait TD-DIRECTED(λ). Les résultats sur KnightCap indiquent que cet algorithme, dans le domaine des échecs, a une efficacité supérieure à TD(λ) et inférieure à l'algorithme appelé TDLEAF(λ).

6.4.3.3. *Construction des bases de parties*

La principale difficulté provient de la construction de la base de parties. Nous avons construit des bases qui donnent un bon niveau au programme, et d'autres beaucoup moins bonnes. Il n'est pas encore clair pour nous quelles sont les bonnes façons de construire une base de parties pour l'apprentissage. Toutes les parties de nos bases ont été jouées par BING contre lui-même, en fixant le temps total d'une partie à seulement quelques secondes.

Nous pensons qu'une bonne base de parties doit offrir une certaine diversité, c'est pourquoi nous avons mélangé des parties jouées à différents stades du développement de BING, en commençant par la version avec une combinaison linéaire réglée à la main, et en passant par des versions avec des composantes différentes en entrée du réseau de neurones. Différentes bases de parties ont produit, après apprentissage, des programmes avec des styles de jeu différents. Par exemple, certaines versions de BING accordent plus ou moins d'importance au nombre de pions. Ceci n'empêche pas des niveaux de jeu similaires, comme il a été vérifié expérimentalement.

Les expériences faites avec une base de parties tirées uniquement de la dernière version du programme ont abouti à des résultats souvent nettement inférieurs. Ceci est en accord avec les résultats sur KnightCap [BAX 00], indiquant que des parties d'un seul programme contre lui-même introduisent trop peu de diversité, et que le niveau après apprentissage s'en ressent.

6.5. Conclusion

Le succès de notre programme, BING (2^e place aux 8^e et 9^e *Computer Olympiads*) est dû à la réutilisation du code de recherche de GNU Chess, à un choix de composantes inspiré des travaux existants avec quelques modifications, et à l'utilisation de méthodes d'apprentissages pour régler les poids de la fonction d'évaluation. Ces choix ont permis un développement rapide, compte tenu du niveau obtenu. Des améliorations restent possibles en recherche (extensions ou recherche de quiescence par exemple), pour le choix des composantes de la fonction d'évaluation ou pour la construction de la base de parties utilisée pour l'apprentissage.

6.6. Bibliographie

[BAX 00] BAXTER J., TRIDGELL A., WEAVER L., « Learning to Play Chess Using Temporal-Differences », *Machine Learning*, vol. 40, n°3, p. 243-263, 2000.

[ENZ 03] ENZENBERGER M., « Evaluation in Go by a Neural Network Using Soft Segmentation », KLUWER, Ed., *Advances in Computer Games (ACG10)*, p. 97-108, 2003.

[GRA 71] GRAY S. B., « Local properties of binary images in two dimensions », *IEEE Transactions on Computers*, vol. C-20, p. 551-561, 1971.

[MAR 86] MARSLAND T. A., « A Review of Game-Tree Pruning », *ICGA journal*, vol. 9, n° 1, p. 3-19, 1986.

[RIE 93] RIEDMILLER M., BRAUN H., « A direct adaptive method for faster backpropagation learning : the RPROP algorithm », *IEEE International Conference on Neural Networks*, vol. 1, p. 586-591, 1993.

[SUT 88] SUTTON R. S., « Learning to predict by the methods of temporal differences », *Machine Learning*, vol. 3, n° 1, p. 9-44, August 1988.

[TES 02] TESAURO G., « Programming backgammon using self-teaching neural nets », *Artificial Intelligence*, vol. 134, n° 1-2, p. 181-199, 2002.

[WIN 00] WINANDS M., Analysis and Implementation of Lines of Action, Master's thesis, Faculty of General Sciences of the Universiteit Maastricht, 2000.

[WIN 02] WINANDS M. H. M., KOCSIS L., UITERWIJK J. W. H. M., VAN DEN HERIK H. J., « Learning in Lines of Action », BLOCKEEL H., DENECKER M., Eds., *Proceedings of the Fourteenth Belgium-Netherlands Conference on Artificial Intelligence (BNAIC)*, p. 371-378, 2002.

[WIN 03] WINANDS M., VAN DEN HERIK H., UITERWIJK J., « An Evaluation Function for Lines of Action », H. VAN DEN HERIK H. I., HEINZ E., Eds., *Advances in Computer Games 10*, Kluwer Academic Publishers/Boston, p. 249-260, 2003.