FFORUM

Le magazine de la Fédération Française d'Othello

Algorithmes

- *Minimax*
- *Alpha-bêta*
- *Scout*
- *La mobilité*
- *Heuristiques*
- *Transposition*

Les programmes

- *Comp'oth*
- *Iago*
- *Bill*
- *Thor*
- *Othel du Nord*

Stratégie

- *Ouvertures*
- *Tests de finales*

Historique

Bibliographie

7,50 euros

Numéro spécial Informatique

Olympiades 1989

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

Polygon 29-35 Peer Gynt

Divers 1993

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

Polygon 31-33 Cassio

St-Michel sur Orge 1991

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

Thor 32-32 Spock

Paderborn 1994

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

Logistello 36-28 Bugs

Paderborn 1994

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

Keyano 33-31 Spock

Parties Internet 1994

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

Kitty/Rev 41-23 Logistello

Édito...

Éditorial de la première édition

Un éditorial en forme de présentation de ce numéro spécial Informatique de votre magazine préféré, j'ai nommé Fforum *bien sûr.*

Pourquoi l'informatique ? Eh bien, comme les plus anciens d'entre vous s'en souviennent peut-être (silence dans le fond quand grand-papa raconte une histoire), ce noble art (l'informatique, pas raconter des histoires) a joué dès les temps héroïques de la fondation de la F.F.O. (alias la fédé, alias le Machin) un rôle important dans le développement d'Othello (l'art de raconter des histoires aussi d'ailleurs) ; la présence d'informaticiens parmi les dinosaures... pardon les fondateurs évoqués ci-dessus en témoigne, mais aussi et surtout le grand nombre de programmes de jeux d'Othello qui circulaient déjà ainsi que le défunt tournoi de programmes organisé jadis par l'Ordinateur Individuel *au SICOB. Aujourd'hui de nombreux joueurs ont connu Othello par ce biais, qu'ils se soient eux-mêmes essayés à la rédaction d'un programme ou non. Il est peut-être utile de rappeler à ce propos, tout en reconnaissant le niveau de certains programmes (à qui pensez-vous ? À rien, c'était pour faire avancer le schmilblick...), que les annonces fantaisistes d'un « programme-champion-du-monde-qui-bat-tous-les-humains » parues ici où là empruntent plus aux techniques confirmées de l'approximation intéressée et du « coup » publicitaire qu'à un quelconque souci de rigueur ou d'exactitude (bon, on se calme, non mais c'est vrai quoi).*

Passons maintenant au contenu proprement dit de ce numéro spécial que vos éditeurs préférés, sacrifiant leurs vacances à cette tâche d'intérêt public (sanglots étouffés de reconnaissance), vous ont mitonné. Les articles proviennent souvent d'anciens (ou moins anciens) numéros de Fforum *: ainsi rassemblés, ils deviennent plus faciles d'accès (on dit merci) surtout pour les membres de la F.F.O. dont le dernier numéro de* Fforum *n'est pas 99 (NdT : ça veut dire qu'ils sont abonnés à perpétuité et donc sans doute depuis l'âge du chalcidique inférieur). Comme le sommaire vous l'indiquera (c'est juste après), y figurent entre autres : une série d'articles sur les techniques de programmation, la description détaillée des algorithmes de quelques bons programmes, des idées de tests pour évaluer la rapidité de votre créature chérie, une réflexion détaillée sur la machine idéale et enfin un historique détaillé des tournois de programmes. Quant à l'article inédit (merci François) qui ouvre ce numéro, on y décrit un virus étrange et venu d'ailleurs : attention, VOUS êtes peut-être déjà atteint !*

Didier Piau (août 1990)

Éditorial de la deuxième édition

Plus de quatre années depuis que Didier nous concocta ce savoureux mélange de silicium, de réflexion et de bonne humeur. Cette seconde édition du numéro spécial informatique reprend les articles de la première édition ; nous y avons rajouté ceux publiés dans Fforum *depuis, ainsi que plusieurs inédits.*

Nous espérons que ce numéro, remis en forme et d'une meilleure qualité de mise en page, vous aidera à mieux comprendre ces sales bêtes qui ne pensent pas comme nous.

Pour les plus mordus d'entre vous, nous vous signalons qu'il existe un serveur d'Othello sur Internet, basé à l'université de Paderborn, en Allemagne. Pour se connecter, il faut faire un `telnet faust.uni-paderborn.de 5000`. *Signalons également que vous pouvez nous contacter par courrier électronique à l'adresse suivante :* `ffo@fr.st`

Emmanuel Lazard (avril 1995)

Éditorial de la troisième édition

Quelques nouveaux articles (et surtout la perte des originaux sur papier de la précédente édition) m'ont motivé pour refaire une mise en page complète de ce numéro spécial informatique. Depuis dix ans, que de progrès : les meilleurs programmes battent systématiquement les meilleurs mondiaux, la multiplication des serveurs de jeux sur Internet a accéléré la diffusion des connaissances et les machines ont fait des bonds en puissance fulgurants. Point d'orgue, la défaite du champion du monde Takeshi Murakami face à Logistello *sur le score sans appel de 6-0 en août 1997. Écrire un programme d'Othello est toujours aussi facile mais écrire un BON programme d'Othello est toujours aussi compliqué !*

Emmanuel Lazard (février 2002)

Sommaire

Algorithmes

Quelques bons programmes d'Othello

De l'ouverture à la finale...

Ça n'arrive pas qu'aux autres

(ou les confessions d'un programmeur repentant)

par François Aguillon

Si, le voile du palais vibrant à chaque ronflement sonore du citoyen en paix avec son âme, vous n'avez jamais passé vos nuits qu'à dormir sous vos plumes, peut-être n'imaginez-vous pas que, sous la lune, certains que le mal ronge, s'adonnent avec délectation à de turpides activités. Mais que font-ils donc, tous ces mâles (eh oui, rien que des mecs !), les nuits de pleine lune ? Ils tapotent leur clavier d'ordinateur bêtement, essayant de faire comprendre à leur machine comment jouer à Othello, s'arrachant les cheveux sur le « dernier » bug. Jusqu'au jour où ils arrêtent de s'arracher les cheveux : on les appelle alors des chauves.

Car, approchez vos oreilles plus près de la feuille pour que vos voisins n'entendent pas le scoop que je vous livre ici, si vous avez déjà joué contre un programme d'Othello, c'est que quelqu'un l'a écrit ! Et ce quelqu'un, qui est-ce ? Un mâle chauve, bien sûr ! Mais peut-être pensez-vous que ce mâle chauve est un mec pas normal, avec des bits partout et des poils nulle part (oh que c'est fin !) : et bien, vous vous trompez ! Car Messieurs, écoutez-moi bien (Mesdames et Mesdemoiselles aussi, bien que cet article soit surtout un truc viril, comme le service militaire, le port des rouflaquettes, la coupe du monde de football et diverses autres guignolades. Je dois préciser que je n'ai rien contre les fonctions analytiques) : nous, les mecs, on a un truc hormonal qui décline avec l'âge, et qui fait qu'on a tous un avenir de chauve ! Examinez votre peigne demain matin, et vous comprendrez ; ceux qui se coiffent avec une éponge ont déjà compris, les pauvres (et ceux qui ne se coiffent pas sont de gros dégoûtants).

Mais, vous demandez-vous, pourquoi sacrifier ainsi sa chevelure à écrire des programmes d'Othello, puisqu'il en existe de pas mauvais tout faits ? C'est une bonne question que vous pouvez vous remercier de vous être posée. Sans fausse modestie, je pense être très bien placé pour y répondre, étant donné que je programme Othello depuis plus de dix ans, que je vais bientôt m'acheter une moumoute, et que je me la suis posée dès avant-hier soir.

Un premier élément de réponse est que le mal est progressif : un jour, on se retrouve avec un ordinateur à apprendre un langage, et on en a vite marre de programmer la résolution d'équation du deuxième degré ou le tracé de fonctions analytiques (je dois préciser que je n'ai rien contre les femmes non plus). Un programme de jeu « intelligent » présente à ce point de vue bien des avantages : c'est assez original, cela sert à quelque chose (à jouer, c'est important), et surtout c'est très prestigieux : parlez-en à votre coiffeur, vous verrez la flamme de l'admiration s'allumer dans ses pupilles naïves qui ne savent pas qu'elles sont en train de perdre un client. Et en plus, un programme d'Othello, c'est particulièrement simple à écrire : on a vu des programmes dont le listing tiendrait sur une étiquette de lotion capillaire. Pourquoi est-ce simple ? Parce que la règle d'Othello est simple ! Si on veut programmer les échecs, il faut se taper la définition de toutes les pièces, des règles tordues comme la prise en passant, les roques, les promotions de pions, le pat. Les dames, ça n'est pas beaucoup mieux ; enfin le Jeu de dames, parce que les dames, surtout analytiques... Le Go, c'est horriblement compliqué (ne serait-ce que de se rendre compte quand la partie est finie !!), le strip-poker, c'est mieux en vrai, et la planche à voile, c'est hors-sujet. C'est simple aussi parce que le déroulement d'une partie d'Othello est simple : cela commence au coup 1, et ne dépasse le coup 60 que si le séisme atteint l'intensité 5,7 sur l'échelle de Richter, qui en compte 10 (c'est peut-être pour ça que les Japonais sont si forts, même la terre tremble devant eux).

C'est à ce moment qu'il faut faire attention à ne pas tomber gravement atteint : quand on a écrit un programme d'Othello, il ne faut surtout pas chercher à l'améliorer, sous peine de calvitie précoce. Parce que le premier programme d'Othello que vous aurez écrit, ne vous en faites pas, il sera nul ! Et il sera facile à améliorer ! Et le suivant un peu moins nul et un peu moins facile, et ainsi de suite. Et, sans y prendre garde, on se retrouve avec un programme représentant déjà une petite somme de travail, et qui vous bat, vous, son programmeur tonsuré ! Ca fait drôle, je vous l'assure, la première fois ! Après, on s'habitue, vient même le jour où on est surpris de gagner, mais plus tard.

On a alors vite fait de sombrer complètement. Toujours battu par son programme, on en vient vite à le croire quasi-imbattable, et à vouloir le frotter aux autres programmes, qui paraissaient, il y a peu, hors de portée. Et là, il faut une bonne dose d'humilité, car bien souvent, ils sont effectivement hors de portée ! Combien ai-je vu de jeunes chauves (moi le premier !) arriver à un tournoi persuadés de s'offrir le scalp de leurs aînés, en repartir déçus, voire mortifiés, de n'avoir accroché que le scalp de Bip. Et alors on se dit : « S'ils peuvent le faire, pourquoi pas moi ? ». La quantité de travail est alors importante, il faut pinailler le moindre détail et couper les cheveux en quatre, mais qu'importe, on est accro. Et quand on est accro, la monomanie n'est pas loin, le symptôme du tournoi guette : perte de sommeil, manie des voyages vers de nouveaux horizons (Paris, St-Michel-sur-Orge, Pérenchies, Utrecht, Londres), boulimie de Mips...

Mais, ne le répétez à personne, surtout pas à ma femme (elle m'avait épousé pour mes mèches blondes, que le vent de notre amour faisait flotter bien haut, et qui flottent maintenant dans le lavabo conjugal, c'est dégueulasse), ce vice, comme il est doux ! Cela ne peut se comparer qu'à l'alcool sans la cirrhose, la télé sans électricité, le Bois de Boulogne sans le Sida, la vitesse sans les radars de la maréchaussée, ou à Vincent van Gogh.

Alors, pourquoi pas vous ? Vous avez sans doute été séduits à vos débuts par la facilité d'approche du jeu d'Othello, et ensuite par sa richesse insoupçonnable de prime abord. Il en va exactement de même pour sa programmation. Alors laissez-moi vous donner quelques magistraux conseils et recommandations :

• vous n'avez peut-être pas joué votre première partie comme Tamenori. Soyez prudents dans vos objectifs informatiques : essayez au début d'être nul, puis de moins en moins nul. N'essayez pas d'être bon tout de suite, c'est très difficile ;

• essayez de vous faire une interface utilisateur puissante, sinon forcément jolie. Retours en arrière, soumission de problèmes, enregistrement de partie, tout cela aide beaucoup dans les phases de tests ;

• il existe de la littérature traitant le sujet : par exemple, ce numéro de Fforum ! Il faut en profiter, mais surtout ne pas tout lire avant d'écrire une ligne de programme, ne pas taper un listing sans le décortiquer. Je pense qu'il est bon de se poser des questions avant de lire les réponses. Certains papiers sont très techniques, à la limite incompréhensibles. J'en ai d'ailleurs écrit quelques-uns que je n'ai toujours pas compris ;

• dans le midi, on dit que la fiente de pigeon fait repousser les cheveux ;

• il ne faut pas rester isolé. Les tournois de programmes d'Othello sont des endroits privilégiés pour discuter : pendant que les machines jouent entre elles, il faut bien tuer le temps en bavardant ! N'ayez pas peur d'y aller ! A titre d'information, ces dernières années, il y a eu des tournois de programmes réguliers à St-Michel-sur-Orge (91) au mois de mars, à Pérenchies (59) en juin, et aussi à Utrecht (NL) en avril et à Londres (Zimbabwe) en août[1] ;

• les tournois hommes-machines sont aussi très enrichissants, au sens figuré. Tant qu'on en est dans les contingences matérielles, n'espérez pas trop gagner des sous avec un programme d'Othello, quelle que soit sa qualité : il n'y a pas un gros marché pour cela. Le mieux pour gagner des sous, c'est d'écrire un programme qui dessine des billets bien imités sur une imprimante laser, et de faire ses courses avec le résultat produit. Les premiers billets pourront être faits avec une laser noir et blanc et des crayons de couleur ; avec ceux-ci on essaiera d'acheter rapidement une laser couleur. Ensuite, une fois en prison et dégagé des soucis financiers, on pourra consacrer tout son temps à Othello ;

• il y a aussi des programmeurs d'Othello sur 3615 Eliott ou 3614 Ness, ceux que je connais (sauf moi, peut-être), sont des gens charmants qui répondent aux questions qu'on leur pose !! (exemple de question : « Eh, mec, t'as pas cent balles ? », exemple de réponse : « Casse-toi ou je te fais un piège de Stoner. ») ;

• qui pourrait me prêter une douzaine de pigeons pendant une cinquantaine d'années ?

• il n'est pas nécessaire de posséder une machine basée sur un 786 cadencé à 273 MHz et avec tellement de mémoire cache qu'il n'y a plus rien à cacher. Je suis persuadé qu'on peut faire de très bonnes choses avec un matériel « normal » (ceci dit, je suis presque le seul à en être persuadé !). Par contre, il est intéressant de posséder un programme portable, c'est-à-dire écrit dans un langage suffisamment standard et répandu ; de plus, si le langage est souple, c'est bien ; s'il permet une structuration, c'est bien ; s'il est rapide, c'est bien ; s'il est lisible, c'est bien ; s'il est facile à apprendre, c'est bien. S'il existe, je mange mon chien.

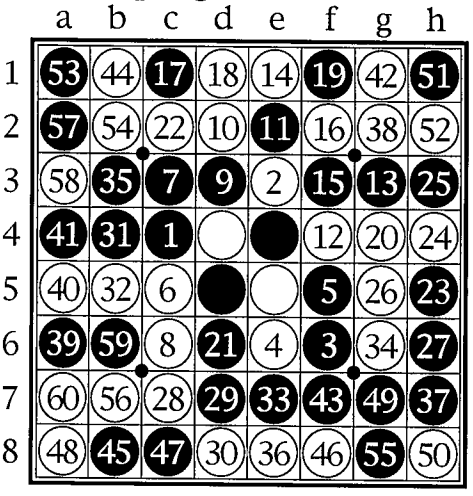
Pour vous fixer un ordre de grandeur, je pense qu'il faut consacrer de l'ordre d'une année de loisirs informatiques avant de pouvoir prétendre jouer dans la cour des grands, à supposer que vous possédiez déjà un niveau othellistique et/ou informatique différent du zéro absolu. C'est long, mais, encore une fois, ça n'est pas un an sans rien voir, c'est un travail très progressif et jamais fastidieux (presque jamais...). Qu'est-ce que cela signifie, jouer dans la cour des grands ? Cela veut dire se situer au même niveau que les tout premiers joueurs français au classement de Jech (et donc, à moins d'être soi-même dans ces tout premiers avoir un programme plus fort que soi), cela signifie pouvoir espérer sinon tout de suite gagner un tournoi du moins tenter d'accrocher les grosses cylindrées du genre, en particulier britanniques ou néerlandaises, cela signifie aussi avoir écrit un programme beaucoup plus fort que tout ce qui se trouve dans le commerce, du moins en ce moment.

Et qu'est-ce que cela apporte ? Si vous lisez cette revue, c'est que vous aimez le jeu. L'aspect ludique est de loin le plus important. Il est à deux étages ici : il y a l'aspect ludique habituel de la programmation que l'on connaît en écrivant le programme (je me fixe un but, et je force la machine à y aller, rapidement si possible), et le jeu du programme lui-même (me suis-je fixé les bons buts ?). De plus, il y a l'aspect technique. Un programme d'Othello peut devenir une chose élaborée, qui initie au développement des programmes importants et des algorithmes sophistiqués. Enfin, l'aspect prestige peut compter : je peux vous dire que je mets toujours dans mon CV que je programme Othello, et je sais que cela fait toujours une bonne impression : au pire, un décideur peut n'y attacher aucune importance.

En bref et en slogans : vous pouvez le faire, l'essayer c'est l'adopter, n'échangez jamais un bon programme contre deux mauvais, voilà ce que j'avais à dire. J'espère qu'après une telle pub, quelques-uns d'entre vous se décideront à tenter l'aventure, et que nous retrouverons de nombreux « bleus » aux prochains tournois de programmes.

Je tiens quand même à préciser avant de terminer que Londres n'est pas au Zimbabwe, c'était pour voir si vous suiviez. Le premier lecteur qui m'envoie sur carte postale le pays où est situé Londres gagne un pion noir à poser sur une case X de son choix pour deux personnes (eh oui, pour que quelque chose soit vraiment X, il faut être deux (ou plus (plus, c'est plus X (tu vois, Olivier, j'arrive aussi à imbriquer les parenthèses)))))(mais pas à les débriquer, zut)).

Championnat de France des programmes 1998



Thor 32-32 Compoth

[1] Ce texte date un peu. Les tournois ont principalement eu lieu à Courchelettes (près de Douai), à Waterloo (Canada) et à Paderborn (Allemagne). Ils se tiennent mainenant sur Internet.

Les petits trucs font (aussi) les bons programmes

par Bernard Daunas

Si l'élément majeur de l'écriture d'un bon programme d'Othello réside principalement dans l'algorithme utilisé, son implémentation joue également un rôle important, parfois déterminant lorsque deux programmes comparables sont en compétition. Nous allons étudier une technique qui permet de résoudre de manière simple et rapide un problème qui se pose couramment lors de la rédaction d'un programme d'Othello : la représentation interne de l'othellier c'est-à-dire la manière dont le programme « voit » à un instant donné la position.

Tableau à deux dimensions

La représentation interne venant le plus naturellement à l'esprit est celle dans laquelle l'othellier est représenté par un tableau à deux dimensions (8 cases par 8 cases). Un tel choix a le mérite de « coller » à la réalité physique : un othellier a effectivement deux dimensions. Mais si cette représentation est simple pour un joueur humain, c'est-à-dire si elle lui décrit l'état du jeu sous une forme qu'il comprend aisément, il n'en va pas de même pour la machine.

Une case se décrit dans ce système par un couple abscisse-ordonnée soit deux nombres distincts. Considérons la génération des coups légaux dans une position donnée : il va être nécessaire pour chaque pion du joueur ayant le trait de parcourir, dans les huit directions partant de ce pion, les cases contiguës aussi longtemps qu'elles appartiennent à l'adversaire. Il faudra arrêter le balayage d'une direction dès que l'on rencontrera :

- une case vide : cette direction était sans intérêt ;
- un bord : même situation ;
- un pion appartenant au joueur ayant le trait : le coup considéré est alors légal et les pions à retourner pour ce qui concerne la direction étudiée sont ceux qui viennent d'être balayés.

Examinons plus précisément comment s'opère le balayage : une direction se définit comme un déplacement, positif ou négatif, à ajouter à la case courante. Ce déplacement est double : un déplacement pour l'abscisse, un autre pour l'ordonnée. Les huit directions sont alors (0,1) (1,1) (1,0) (1,-1) (0,-1) (-1,-1) (-1,0) et (-1,1) en partant du nord et en tournant dans le sens des aiguilles d'une montre.

Ainsi le balayage va être constitué :

a) de l'ajout du double déplacement ;
b) de la comparaison des nouvelles abscisses-ordonnées avec chacun des bords pour déterminer si l'on n'est pas sorti de l'othellier ;
c) enfin de la vérification du contenu de la case (case vide, case amie ou case ennemie).

Ces trois opérations sont assez coûteuses en temps de calcul, particulièrement la seconde, et ce d'autant plus qu'elles se trouvent au cœur de la routine de génération des coups, à l'intérieur de quatre boucles imbriquées : les deux boucles permettant le balayage de l'othellier (une boucle pour les abscisses, une boucle pour les ordonnées), la boucle sur les huit directions dans lesquelles on peut retourner des pions pour une case donnée et enfin la boucle sur les cases successives d'une direction.

C'est ici qu'intervient l'idée que l'on doit au Docteur Samuel, auteur dans les années 50 du premier programme jouant aux dames anglaises (*checkers*). Il est possible tout à la fois :

- de réunir les deux boucles externes en une seule ;
- de transformer le double déplacement (a) en un déplacement simple ;
- de supprimer complètement (b), constituée de plusieurs lignes de code, en la combinant avec (c), sans accroître la complexité de celle-ci.

Tableau à une dimension

Le truc utilisé par Samuel est le suivant : le tableau à deux dimensions représentant l'othellier est devenu un tableau à une dimension, au prix de quelques cases supplémentaires sur les bords. À Othello, la numérotation des cases et leur contenu pour la position de départ serait :

81	82	83	84	85	86	87	88	89	90
72	73	74	75	76	77	78	79	80	
63	64	65	66	67	68	69	70	71	
54	55	56	57	58	59	60	61	62	
45	46	47	48	49	50	51	52	53	
36	37	38	39	40	41	42	43	44	
27	28	29	30	31	32	33	34	35	
18	19	20	21	22	23	24	25	26	
9	10	11	12	13	14	15	16	17	
0	1	2	3	4	5	6	7	8	

-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
-1	0	0	0	0	0	0	0	0	
-1	0	0	0	0	0	0	0	0	
-1	0	0	0	0	0	0	0	0	
-1	0	0	0	1	2	0	0	0	
-1	0	0	0	2	1	0	0	0	
-1	0	0	0	0	0	0	0	0	
-1	0	0	0	0	0	0	0	0	
-1	0	0	0	0	0	0	0	0	
-1	-1	-1	-1	-1	-1	-1	-1	-1	

Quelques remarques

• le fait de trouver un contenu négatif pour une case signifie que l'on vient de sortir de l'othellier. Le test est très rapide, tous les microprocesseurs modernes disposant d'un registre d'état dans lequel un bit se positionne lorsque l'octet en cours de test est négatif ; une simple instruction de saut conditionnel permet le débranchement ;

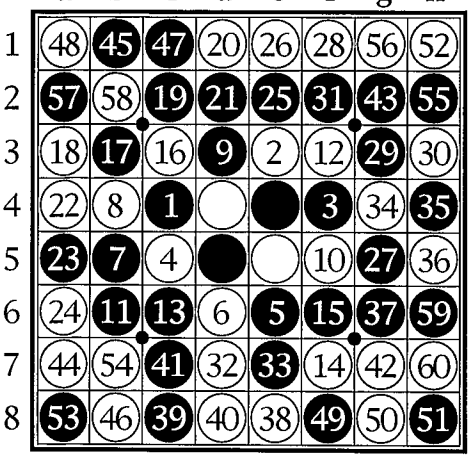
• les déplacements pour aller dans chacune des huit directions sont maintenant : (+9) (+10) (+1) (-8) (-9) (-10) (-1) et (+8) toujours dans le sens des aiguilles d'une montre en partant de la direction nord ;

• il n'est pas nécessaire de doter le côté est de l'othellier de cases supplémentaires, les cases du côté ouest assurant cette fonction pour les deux côtés. On s'en convaincra facilement en tentant de sortir de l'othellier par l'est ;

• un avantage additionnel du procédé est lié au passage d'un tableau de deux à une dimension : le calcul d'indices d'un tableau à deux dimensions exige dans la plupart des langages de réaliser une multiplication entière suivie d'une addition. Sur les machines ne disposant pas de multiplication câblée (c'est le cas de la plupart des 8 bits), la multiplication doit être émulée par un sous-programme spécialisé et très lent (de l'ordre de plusieurs centaines de microsecondes). Ce calcul n'est pas apparent dans un langage évolué mais n'en est pas moins généré automatiquement par le compilateur ou l'interpréteur.

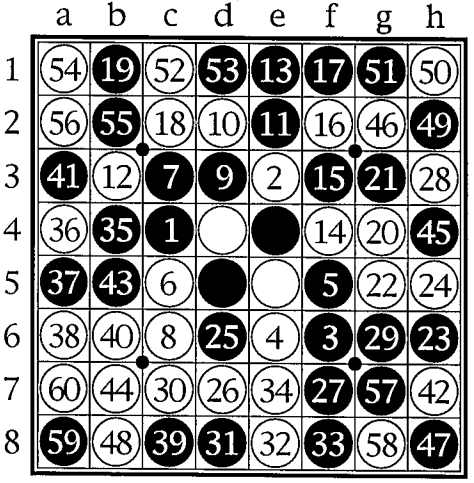
Cette caractéristique originale du programme de Samuel se retrouve plus tard dans la quasi-totalité des programmes d'échecs et d'Othello. Elle nous montre que les échanges sont fréquents et fructueux entre programmes de jeux distincts. Elle nous rappelle également, si nous l'avions oublié, que le soin apporté à la réalisation d'un programme est aussi important que la qualité des algorithmes.

Championnat de France des programmes 2000



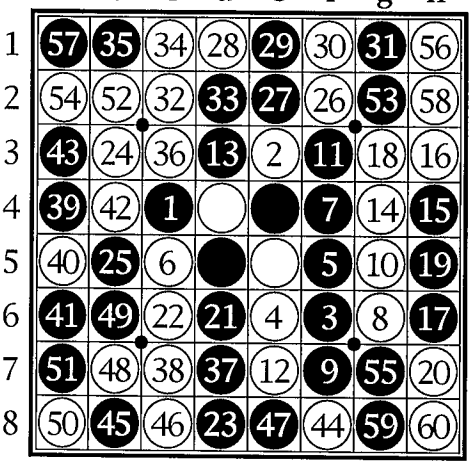
Turtle 23-41 Edax

Championnat de France des programmes 2000



Spock 32-32 Thor

Championnat de France des programmes 2000



Balder 25-39 Mamaju

Le minimax

par Jean-François Puget

Cet algorithme est utilisé dans beaucoup de programmes de jeux (échecs, dames et bien sûr Othello) pour choisir le prochain coup à jouer.

I. Présentation

Prenons un premier exemple. On suppose que deux joueurs que l'on appelle Min et Max jouent à un jeu, que c'est à Max de jouer et qu'il reste 3 coups à jouer pour finir la partie. Max est en fait le programme.

Construisons (figure 1) une sorte d'arbre renversé : plaçons tout d'abord P0 (Position actuelle du jeu). Sur la ligne au-dessous plaçons les positions qui correspondent à l'exécution d'un coup de Max. Par exemple, si Max a trois coups possibles, plaçons P1 qui serait la position du jeu si Max jouait le premier de ses coups, puis P2 et P3. On relie P0 à ces positions, que l'on appelle les successeurs de P0, par des arcs (branches de l'arbre) représentant les coups. P0 est à la profondeur 0 et ses successeurs sont à la profondeur 1.

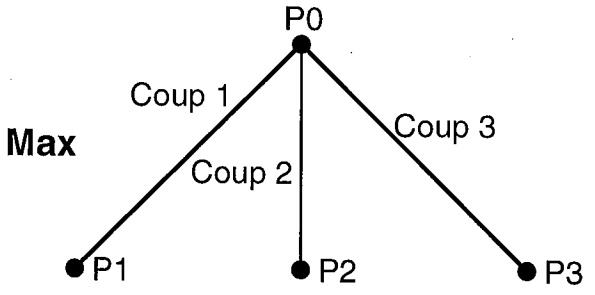


Figure 1

Marquons à gauche Max pour indiquer que l'on passe à l'un de ses successeurs par un coup de Max, puis recommençons : pour chacune des positions P1, P2, ou P3, plaçons toutes les positions qui en découlent par l'application d'un coup de Min (voir figure 2) puis pour chacune de ces nouvelles positions générons les successeurs correspondant à un coup de Max. Nous nous arrêtons alors car nous avons effectué trois coups depuis P0. Sur la figure 2 les positions P10 à P19 correspondent à des parties finies.

Mettons une note à ces positions :

- `+1` si Max a gagné ;
- `0` si la partie est nulle ;
- `-1` si Min a gagné.

Supposons que ces notes soient données à P10,..., P19 comme indiqué sur la figure (c'est le chiffre en gras en-dessous des positions).

Examinons ensuite les positions situées juste au-dessus, à la profondeur 2. Si Max avait le trait en P4, il jouerait le seul coup possible et obtiendrait la position P10 qui est perdante pour lui. P4 est donc perdante pour Max, on la marque à -1.

De même en P5, Max aurait le choix entre trois coups. il choisit le meilleur, c'est-à-dire celui qui conduit à la position ayant la valeur maximum. Ici c'est P13, qui est gagnante, et donc P5 l'est aussi. On remarque donc que la valeur de la position où Max a le trait est le maximum des valeurs de ses successeurs. Nous pouvons ainsi calculer les valeurs des positions P6 à P9.

Vient ensuite le calcul des valeurs des positions situées à la profondeur 1. Si nous étions en P1 et que le trait était à Min, celui-ci aurait le choix entre 2 coups. Il choisirait le meilleur pour lui, c'est-à-dire le moins bon pour Max, donc la position qui conduit à la position ayant la valeur minimum. Ici c'est P4, perdante pour Max; P1 est donc perdante et on la marque à -1. Remarquons que la valeur d'une position où Min a le trait est le minimum des valeurs des ses successeurs. On calcule ainsi les valeurs P2 et P3.

Nous arrivons enfin à P0 et c'est à Max de jouer. Nous avons vu précédemment que si les deux adversaires jouent au mieux, alors P1 et P2 sont perdantes pour Max tandis que P3 est nulle. Le bon coup pour Max est donc le troisième, c'est celui qui conduit à la position ayant la valeur maximum. La raison pour laquelle nous avons appelé les joueurs Min et Max devrait être claire maintenant...

II. La fonction d'évaluation

Dans la pratique un problème insurmontable se pose. Supposons en effet que l'on soit à 10 coups de la fin. L'algorithme fonctionne correctement : il suffit de générer 10 étages de positions au lieu de 3, puis d'évaluer celles situées à la profondeur 10, enfin de remonter les valeurs comme ci-dessus.

Mais si on est à 20 ou 30 coups de la fin, on n'a tout simplement pas le temps, et de très loin, de générer l'arbre jusqu'à la fin du jeu. Fixons- nous donc une profondeur maximale **Pmax**, et générons l'arbre correspondant. La partie n'étant pas terminée en profondeur Pmax, il n'est plus possible d'affirmer qu'une position est gagnante, perdante ou nulle. Nous avons donc besoin d'une fonction d'évaluation statique, qui à toute position P qu'on lui fournit, affecte une valeur `f(P)`, d'autant plus grande que la position est bonne pour Max (c'est-à-dire pour le programme). Si néanmoins il était possible de l'affirmer, alors par convention :

`f(P)` = $+\infty$ si P est gagnante pour Max.
`f(P)` = $-\infty$ si P est perdante pour Max.

Pour résumer il faut donc :

1. générer l'arbre jusqu'à la profondeur Pmax ;

2. marquer chacune des positions à la profondeur Pmax par sa valeur `f(P)` ;

3. remonter ensuite les valeurs à la profondeur Pmax-1, puis Pmax-2,... jusqu'à la profondeur 1 de la manière suivante : calculons la valeur des positions à l'étage de profondeur p :

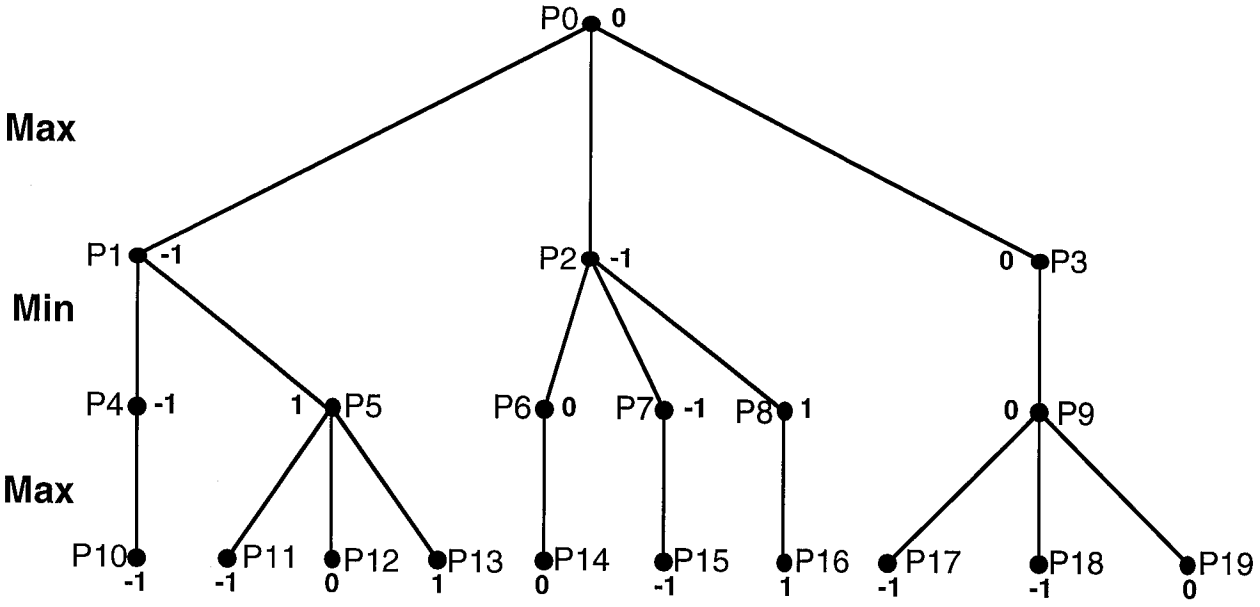


Figure 2

`si` p est pair (trait à Max),
`alors` la valeur d'une position est le maximum des valeurs de ses descendants ;
`sinon` (p est impair, trait à Min),
`alors` la valeur est le minimum de ses descendants ;

4. (après avoir calculé les valeurs des positions à la profondeur 1) le meilleur coup pour Max est celui qui amène à la position ayant la valeur maximale.

Exemple : En figure 2, le meilleur coup est le troisième.
Exercice : reprendre la figure 2 en supposant qu'à la profondeur 3 les valeurs données par la fonction d'évaluation soient les opposées des valeurs marquées. Quel est le meilleur coup ?

Dans le prochain article, je présenterai une procédure Pascal qui implémente cet algorithme, augmenté de quelques améliorations (alpha-bêta).

III. Discussion de l'algorithme

Le principal avantage du minimax est qu'il permet de distinguer entre l'évaluation statique de la position, sans se préoccuper de ce qui se passe ensuite, et sa remise en situation dynamique, dont se charge le minimax.

Son principal inconvénient : l'effet horizon, qui se produit quand on se fixe a priori une profondeur donnée de recherche. Deux effets peuvent se produire :

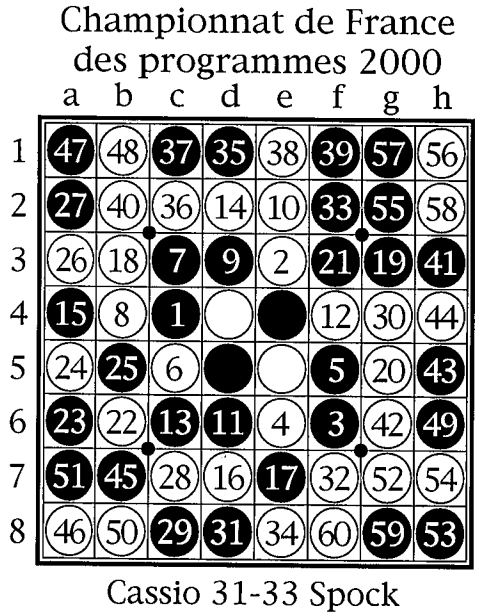
• l'effet horizon négatif : le programme joue des coups d'attente qui repoussent certains effets négatifs (perte de coin par exemple) au-delà de la profondeur limite, alors qu'ils sont inéluctables ;

• l'effet horizon positif : le programme ne joue pas le meilleur coup (il est pour lui apparemment mauvais) car il ne voit pas assez loin.
Il est possible d'améliorer l'algorithme :

• en générant moins de positions : utilisation des coupures alpha-bêta (qui seront traitées la prochaine fois), tentatives de ne conserver comme successeurs que quelques-uns (les meilleurs !) des coups légaux ;

• en essayant de limiter l'effet d'horizon, ceci en continuant la recherche jusqu'à obtenir des positions « stables » (par exemple plus de coins en prise, etc.) au lieu de s'arrêter à une profondeur fixée (le problème se compliquera à Othello car souvent une position n'a absolument pas la même valeur si Max a le trait ou si c'est Min).

Un dernier point laissé en exercice : que se passe-t-il si un des deux joueurs passe ?



Coupures alpha-bêta

par Jean-François Puget

Nous avons déjà examiné un algorithme très général qui permet de choisir le « meilleur » coup dans une position donnée d'un jeu à deux joueurs. Pour cela il suffit de pouvoir générer tous les coups possibles dans chaque position. Ensuite, une valeur est calculée en « minimaxisant » les résultats donnés par une fonction d'évaluation statique. Pour résumer, l'algorithme du minimax permet de réduire le problème du choix du coup au choix d'une fonction d'évaluation. Le principal inconvénient est la très grande puissance de calcul nécessitée.

Plusieurs méthodes ont été imaginées pour accroître l'efficacité de la recherche. La plus connue cherche à réduire la taille de l'arbre en n'examinant pas tous les coups possibles dans une position donnée de l'arbre.

Examinons sur un exemple comment certains coups peuvent être négligés. Supposons que dans la position P0 (voir figure 1), la position P1 ainsi que tous ses descendants aient été examinés par le minimax et que sa valeur soit 1. Supposons de plus que P3 ait été examinée aussi et que sa valeur soit 0. Dans ces conditions, il n'est pas nécessaire d'examiner P4 pour savoir que P1 est meilleure que P2. En effet, la valeur de P2 est le minimum des valeurs de ses descendants car Min a le trait (voir l'article sur le minimax). Donc la valeur de P2 est inférieure ou égale à 0 et donc P1 est meilleure. De plus, la valeur de P0 est le maximum des valeurs de ses descendants donc, ici, celle de P1, c'est-à-dire 1. Ces résultats sont d'ailleurs les mêmes que ceux donnés par le minimax si tout l'arbre est exploré.

Les coups de Max peuvent également être négligés dans une situation comparable (voir figure 2). Si les positions P1 et P3 ont été examinées et ont pour valeurs respectivement -1 et 0, la position P4 n'a pas besoin d'être examinée. Le raisonnement est semblable au cas précédent : ici, P2 est meilleure (pour Max) que P1, donc le meilleur coup de Min est celui qui conduit à P1.

Voici maintenant deux procédures écrites en pseudo-Pascal qui réalisent ces coupures. Elles sont récursives, c'est-à-dire qu'elles s'appellent mutuellement. Les coupures sont réalisées en passant un paramètre, appelé « borne », auquel sont comparées les valeurs des positions examinées. Voyons plus en détail ces procédures présentées figures 3 et 4.

```
FUNCTION Min(P : position,
   borne : integer) : integer ;

VAR m, n, t, i : integer ;
    P1,…,Pn : position ;

BEGIN
   déterminer les descendants
   P1,…,Pn de P ;
   IF n = 0 THEN Min := F(P)
   ELSE BEGIN
      m := infini ;
      FOR i := 1 TO n DO
      BEGIN
         t := Max(Pi, m) ;
         IF t < m THEN m := t ;
         IF m <= borne THEN i := n ;
      END ;
      Min := m ;
   END
END ;
```

figure 3

Commençons par la procédure Min qui calcule les valeurs des positions où Min a le trait. La première action est de déterminer les descendants de la position P en cours d'examen. Les coups légaux sont générés et les conditions d'arrêt sont testées :

- si la profondeur maximale est atteinte, ou si aucun des deux joueurs ne peut jouer, n est fixé à 0 et l'exploration de l'arbre s'arrête ;
- si le joueur Min passe, n est fixé à 1 et le descendant est la position P de départ, ce qui revient à examiner un coup fictif dans l'arbre de jeu.

Ensuite, dans le cas où il n'y a pas de descendant, la fonction d'évaluation statique, désignée par F, est appliquée à la position P. Sinon, (n différent de 0), les valeurs des descendants sont calculées les unes après les autres en faisant appel à la procédure Max : en effet, le joueur Max a le trait dans ces positions. Si l'une des valeurs est plus petite que la borne, il y a coupure. Les descendants suivants ne sont pas examinés et la valeur renvoyée par Min est le minimum des valeurs trouvées.

Exercice : vérifier que cette procédure n'examine pas le coup 4 dans la figure 1 en supposant que P est en fait P2 et que borne vaut 1.

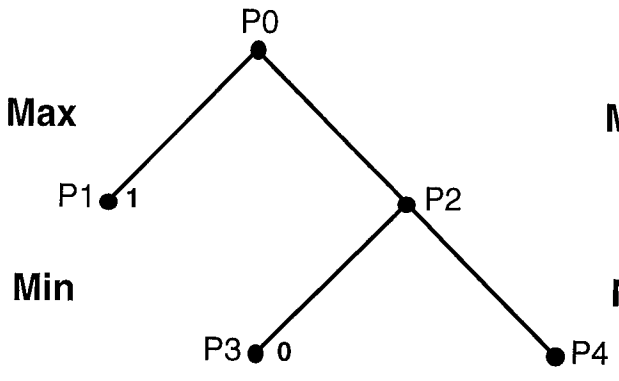


Figure 1

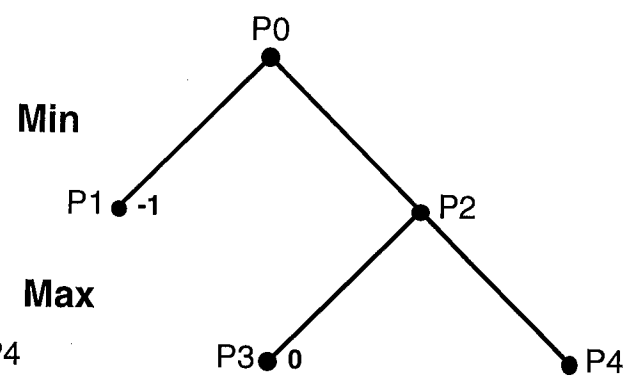


Figure 2

```
FUNCTION Max(P : position,
   borne : integer) : integer ;

VAR m, n, t, i : integer ;
    P1,…,Pn : position ;

BEGIN
   déterminer les descendants
   P1,…,Pn de P ;
   IF n = 0 THEN Max := F(P)
   ELSE BEGIN
      m := -infini ;
      FOR i := 1 TO n DO
      BEGIN
         t := Min(Pi, m) ;
         IF t > m THEN m := t ;
         IF m >= borne THEN i := n ;
      END ;
      Max := m ;
   END
END ;
```

figure 4

La procédure Max fonctionne de manière analogue.
Exercice : le vérifier sur la figure 2.

Il est encore possible d'améliorer cet algorithme pour réaliser encore plus de coupures. Regardons la figure 5 : supposons que les positions P1 et P5 ont été examinées et qu'elles valent 1 et 0 respectivement. Alors, il n'est pas nécessaire d'examiner P6 pour conclure que P2 vaut au plus 0 et que P1 est meilleure. En effet, P4 vaut au plus 0 car Min a le trait et donc la valeur de P4 est le minimum des valeurs de P5 et P6 ; donc P3 et P2 aussi puisqu'il n'y a qu'un seul coup possible à chaque fois. En P0, Max a le trait, donc le meilleur coup est celui qui conduit à P1. Tout ceci peut être déduit sans examiner P6. Or, les procédures précédentes ne permettent pas ce genre de coupure. Il faut introduire deux bornes au lieu d'une seule. L'usage est de les appeler alpha et bêta, d'où le nom de l'algorithme. Les nouvelles procédures sont présentées figure 6.

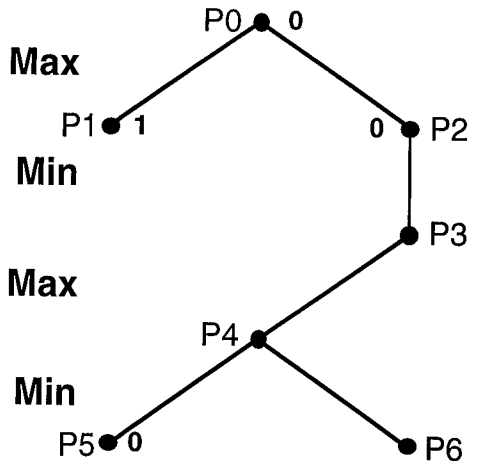


figure 5

Le programme est lancé en appelant Max sur la position de départ, alpha et bêta valant -infini et +infini.
Exercice : vérifier que la coupure décrite figure 5 est bien effectuée par ces nouvelles procédures.

Cet algorithme peut encore être amélioré ; par exemple il n'est pas nécessaire de générer tous les descendants possibles car certains seront ignorés.

```
FUNCTION Min(P : position,
   alpha, beta : integer) : integer;

VAR m, n, t, i : integer ;
    P1,…,Pn : position ;

BEGIN
   déterminer les descendants
   P1,…,Pn de P ;
   IF n = 0 THEN Min := F(P)
   ELSE BEGIN
      m := alpha ;
      FOR i := 1 TO n DO
      BEGIN
         t := Max(Pi, beta, m) ;
         IF t < m THEN m := t ;
         IF m <= beta THEN i := n ;
      END ;
      Min := m ;
   END
END ;
```

```
FUNCTION Max(P : position,
   alpha, beta : integer) : integer;

VAR m, n, t, i : integer ;
    P1,…,Pn : position ;

BEGIN
   déterminer les descendants
   P1,…,Pn de P ;
   IF n = 0 THEN Max := F(P)
   ELSE BEGIN
      m := alpha ;
      FOR i := 1 TO n DO
      BEGIN
         t := Min(Pi, beta, m) ;
         IF t > m THEN m := t ;
         IF m >= beta THEN i := n ;
      END ;
      Max := m ;
   END
END ;
```

figure 6

L'algorithme scout

Les stratégies d'espionnage au service de l'alpha-bêta (et réciproquement)

par Stéphane Nicolet

Résumé des épisodes précédents : après les articles de Jean-François Puget que vous avez lus un peu avant dans ce numéro spécial, vous êtes parfaitement au courant des joies des algorithmes Minimax et Alpha-Bêta. De plus, vous maîtrisez la notation Négamax que je vais utiliser dans cet article[1].

I) Espionner avant d'évaluer

L'algorithme SCOUT repose sur une idée simple : il n'est peut-être pas très utile de passer beaucoup de temps à évaluer exactement un coup si c'est pour s'apercevoir *in fine* que l'on en possède déjà un meilleur. Mais, m'objecterez-vous, comment savoir, sans l'évaluer, si un coup est meilleur ou moins bon que le meilleur que l'on possède ? Chut, c'est un secret, mais c'est entre autres l'un des buts de la section II que de vous fournir le moyen d'effectuer ce test rapidement. Supposons donc pour l'instant que, par un miracle quelconque, une telle possibilité nous soit offerte, et appelons cela « espionner avant d'évaluer ».

```
FUNCTION SCOUT (P: position): integer;
VAR max,n,t,i : integer;
    P1,…, Pn: position;
BEGIN
  déterminer les descendants P1…Pn de P;
  IF n = 0 THEN SCOUT := eval(P)
  ELSE BEGIN
     max := - infini ;
     FOR i := 1 TO n DO
     BEGIN
        IF i = 1
          THEN t := -SCOUT(Pi)
          ELSE
            BEGIN
              IF meilleureNote(Pi,max)
                 (* on espionne *)
                THEN  t := -SCOUT(Pi);
               (* avant d'évaluer *)
            END;
         IF t > max THEN max := t;
     END;
     SCOUT := max ;
  END;
END;
```

Alors un nouvel algorithme vient à l'esprit pour trouver la valeur minimax d'un arbre : on détermine exactement la note de la première branche de l'arbre, puis, pour chacune des autres branches, on jette un coup d'œil (pour vérifier si la recherche sera intéressante) avant de chercher sa vraie valeur.

Bien sûr, l'intérêt de la méthode dépend de la rapidité du coup d'œil, et donc de celle de la procédure *meilleureNote* ci-dessus. S'il faut autant de temps pour savoir si une branche a ou non une valeur minimax V supérieure à une valeur donnée α, si donc il faut autant de temps, dis-je, que pour résoudre effectivement cette branche, alors on aura doublé le temps de recherche lorsque l'inégalité $V > \alpha$ est vraie, et rien gagné dans les autres cas. Par contre, si tester l'inégalité $V > \alpha$ est très rapide, alors on a beaucoup gagné lorsqu'elle est fausse, et seulement un peu perdu lorsqu'elle est vraie.

II) Techniques de réduction de fenêtre

L'idée fondamentale sur laquelle s'appuient les algorithmes de cet article est la suivante : lorsque l'on fait une recherche avec un algorithme de type alpha-bêta, la recherche est d'autant plus efficace que les bornes α et β sont rapprochées, car on fait plus de coupures. Plus précisément :

Si $\alpha_0 \le \alpha_1 < \beta_1 \le \beta_0$, alors la procédure alpha-bêta(P,α_1,β_1) explore un sous-arbre de l'arbre exploré par la procédure alpha-bêta(P,α_0,β_0).

Commençons par un raffinement de l'algorithme alpha-bêta dû à M. Fishburn et qui nous sera utile par la suite :

```
FUNCTION Alpha_beta ( P: position ;
      alpha,beta : integer) : integer;
VAR max,n,t,i : integer;
    P1, ... , Pn: position;
BEGIN
  déterminer les descendants P1…Pn de P;
  IF n = 0 THEN Alpha_beta := eval(P)
  ELSE BEGIN
    max := - infini ;
    FOR i := 1 TO n DO
    BEGIN
      t:=-Alpha_beta(Pi,-beta,-alpha);
        IF t > max THEN max := t;
        IF t > alpha THEN alpha := t;
        IF alpha >= beta THEN i := n;
         (* coupure alpha-beta *)
    END;
    Alpha_beta := max ;
  END;
END;
```

L'intérêt de la procédure ci-dessus, par rapport à la procédure alpha-bêta standard, est qu'elle peut retourner une valeur à l'extérieur de l'intervalle $[\alpha,\beta]$: donc on peut réduire la fenêtre et récupérer quand même un peu d'information. Plus précisément, on peut montrer que :

---

[1] Car vous avez remarqué que minimiser une fonction, c'est maximiser son opposé et donc que si on remplace son appel `t := Min(Pi, beta, m)` par `t := -Max(Pi, -beta, -m)`, ça marche aussi bien.

Si l'on définit par V la (vraie) valeur minimax de l'arbre de jeu à explorer, par α et β les bornes passées à cette procédure et par W la valeur retournée par elle, alors de trois choses l'une :

- si $\alpha < W < \beta$, on a exactement $W = V$;
- si $W \leq \alpha$, on sait que $V \leq W \leq \alpha$;
- si $\beta \leq W$, on sait que $\beta \leq W \leq V$.

Sachant cela, il est trivial de fabriquer, comme annoncé, une procédure pour savoir rapidement si la valeur minimax d'un arbre est plus grande ou non qu'une valeur donnée α : il suffit de poser $\beta = \alpha+1$ en appelant la procédure ci-dessus[2]. En prime, la valeur retournée aura les propriétés ci-dessous :

- si $W \leq \alpha$, alors $V \leq W \leq \alpha$;
- si $\alpha < W$, alors $\alpha < W \leq V$.

III) La synthèse

Il serait dommage de s'arrêter en si bon chemin : nous avons maintenant à notre disposition deux algorithmes (scout et alpha-bêta), concurrents et également performants pour faire une recherche dans un arbre de jeu. Cruel dilemme : lequel allons-nous mettre dans notre prochain méchant programme d'Othello ? La réponse est évidente : le troisième ! Nous avons deux idées ? Marions-les ! Pourquoi ne pas introduire des bornes alpha et bêta dans notre procédure SCOUT, ou, ce qui revient au même, une stratégie d'espionnage avant évaluation dans notre procédure alpha-bêta[3]?

Ceci est motivé par les deux jolis dessins ci-dessous, dans lesquels les carrés sont les niveaux maximisant et les ronds les niveaux minimisant :

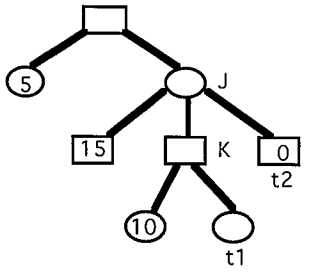


Figure 1

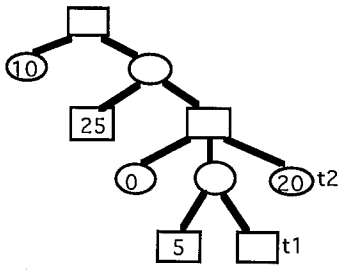


Figure 2

La figure 1 montre un exemple où SCOUT est supérieur à alpha-bêta : le nœud t1 n'est jamais examiné par SCOUT (quand SCOUT examine K, la valeur 10 lui fait sauter t1 ; ensuite, la valeur 0 de t2 suffit à lui faire abandonner à jamais J). L'algorithme alpha-bêta, en revanche, n'a aucune raison d'examiner la valeur de t2 avant celle de t1. La figure 2 nous montre une situation inverse où alpha-bêta est supérieur à SCOUT : vous pourrez vérifier que le nœud t1, visité par SCOUT, est sauté par alpha-bêta. Un algorithme qui emprunterait le meilleur aux deux précédents réaliserait les deux types de coupure.

La procédure finale pourrait alors ressembler à ceci :

```
FUNCTION ABScout (P: position ; alpha,
         beta : integer) : integer;
VAR max,n,t,i : integer;
    P1, ... , Pn: position;
BEGIN
  déterminer les descendants P1...Pn de P;
  IF n = 0 THEN ABScout:= eval(P)
  ELSE BEGIN
    max := - infini ;
    FOR i := 1 TO n DO
    BEGIN
      IF i = 1
        THEN
          t:=-ABScout(Pi,-beta,-alpha)
        ELSE
          BEGIN
            t:=-ABScout(Pi,-(alpha+1),
                                -alpha );
            if (t>alpha) and (t<beta)
              THEN
                t:=-ABScout(Pi,-beta,
                                   -t );
          END;
      IF t > max THEN max := t;
      IF t > alpha THEN alpha := t;
      IF alpha >= beta THEN i := n;
    END;
    ABScout:= max ;
  END;
END;
```

Exercice : vérifier que cette procédure effectue les coupures voulues à la fois dans la figure 1 et dans la figure 2.

Conclusion : Nous avons réalisé un algorithme qui va à la fois plus vite que scout et que l'alpha-bêta. Mais, surtout, surtout, n'oubliez pas : au lieu d'en profiter pour faire bêtement chercher votre programme un peu plus loin, un peu plus vite (et donc propager encore plus les erreurs de sa fonction d'évaluation), utilisez le temps gagné pour améliorer la-dite fonction d'évaluation. Comme dit le sage : rien ne sert de courir, même si on est un ordinateur...

---

[2] La triviale procédure meilleureNote(P, v) se réduit à un test : -Alpha_beta(P, -(v+1), -v) > v

[3] Hein ? Pourquoi ?

Évaluer la mobilité

par Jean-François Puget

Après avoir décrit l'algorithme alpha-bêta, nous nous intéressons à l'écriture d'une fonction d'évaluation simple mais efficace. Il s'agit de pouvoir affecter un nombre à toutes les positions possibles, nombre d'autant plus élevé que les chances de gain du programme seront fortes.

La mobilité

Pour Othello, une composante intéressante à évaluer est la différence de mobilité, notée DM, entre les deux joueurs. Elle se calcule en faisant la différence entre le nombre de coups possibles du programme et celui de l'adversaire. Si cette fonction est calculée bêtement, en générant tous les coups possibles des deux adversaires, le temps de calcul devient prohibitif. Heureusement, il est possible de calculer très rapidement une valeur approchée de DM.

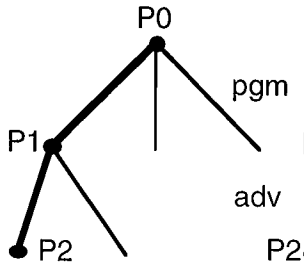


Figure 1

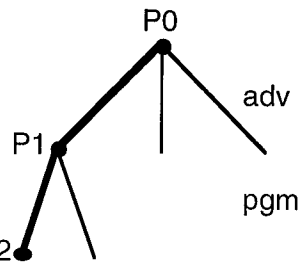


Figure 2

• Premier cas : il s'agit d'une position P2 où le programme a le trait. Supposons que P2 soit à une profondeur supérieure ou égale à 2 dans l'arbre de jeu, et que ses ancêtres soient P0 et P1 (voir la figure 1). Appelons `m(P)` la mobilité du joueur au trait dans la position P. Une bonne approximation de DM dans la position P2 est `m(P0) - m(P1)`.

• Deuxième cas : l'adversaire a le trait (voir la figure 2). Cette fois il faut prendre l'opposé du précédent, soit `m(P1) - m(P0)`.

Dans les deux cas, la fonction croît avec le nombre de coups du programme et décroît avec celui de l'adversaire. Rappelons que si un joueur passe, un coup fictif est introduit dans l'arbre de jeu. Si la mobilité correspondante est prise égale à 0, les choix précédents restent valables.

Cette méthode est très rapide car lorsque le programme arrive à la position P2 durant l'exploration de l'arbre, les coups jouables dans la position P1 sont connus, ainsi que leur nombre qui n'est autre que `m(P1)` ! De même, `m(P0)` est connu, et il ne reste qu'à faire leur différence. Le gain de rapidité du programme est de pratiquement deux coups. Une variante a été utilisée dans le programme BRAND d'Anders Kierulf qui est décrit dans une plaquette disponible auprès de la FFO.

La frontière

La mobilité peut être approchée autrement, par l'analyse de la frontière de la position.

Dans ce qui suit, les voisines d'une case de l'othellier seront les cases qui la touchent par un côté ou un coin. Le nombre des voisines d'une case est donc de 3, 5 ou 8 suivant qu'il s'agisse d'un coin, d'un bord ou d'une case centrale.

Pour l'analyse de la frontière, à chaque case est associée le nombre de ses voisines vides. La somme `F(J)` de ces valeurs est effectuée pour toutes les cases appartenant au joueur J, ceci pour les deux joueurs. La frontière est alors définie par : `F = F(Adv) - F(Prog)`.

Le programme tendra à minimiser sa frontière et à maximiser celle de son adversaire. De plus cette fonction privilégie les positions « compactes » par opposition aux positions « dispersées » qui peuvent être dangereuses.

Donnons un exemple :

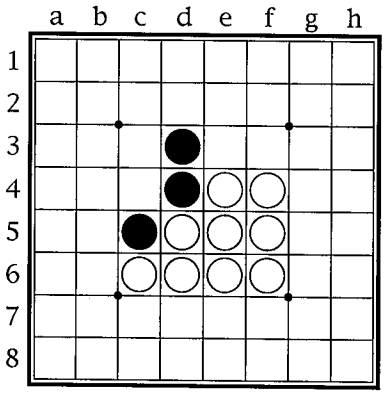


figure 3

	a	b	c	d	e	f	g	h
1	3	5	5	5	5	5	5	3
2	5	8	7	7	7	8	8	5
3	5	8	6	6	4	6	7	5
4	5	7	4	3	2	5	6	5
5	5	6	4	1	0	3	5	5
6	5	6	5	3	3	5	6	5
7	5	7	6	5	5	6	7	5
8	3	5	5	5	5	5	5	3

figure 4

Pour la position de la figure 3, les nombres de voisines vides sont regroupés dans le tableau de la figure 4. La frontière s'analyse comme suit (le programme a les blancs) :

```
F(noir) =  4 (pour c5) + 3 (pour d4)
           + 6 (pour d3) = 13.
F(blanc) = 27.
F = F(Adv) - F(prog) = 13-27 = -14.
```

Une première façon d'accélérer le calcul de cette frontière est basée sur la remarque suivante : le tableau T des nombres de voisines vides ne change pas beaucoup lorsqu'un coup est joué. En fait, seules les valeurs correspondant aux voisines du coup joué changent : elles diminuent de 1 (voir les figures 5 et 6 par exemple). Réciproquement, il est tout aussi simple de « déjouer » un coup sur ce tableau : il suffit d'enlever 1 aux voisines du dernier coup.

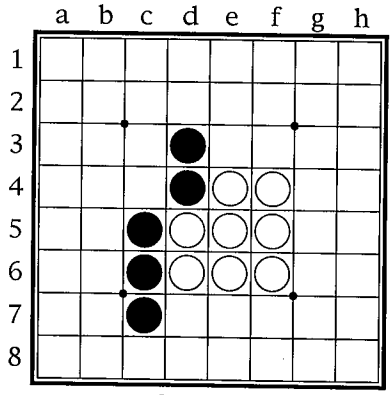


figure 5

	a	b	c	d	e	f	g	h
1	3	5	5	5	5	5	5	3
2	5	8	7	7	7	8	8	5
3	5	8	6	6	4	6	7	5
4	5	7	4	3	2	5	6	5
5	5	6	4	1	0	3	5	5
6	5	5	4	2	3	5	6	5
7	5	6	6	4	5	6	7	5
8	3	4	4	4	5	5	5	3

figure 6

Pour toutes les positions de l'arbre de jeu, un seul tableau sera utilisé et mis à jour lors des changements de position dans l'arbre. Par exemple, en reprenant la figure 1, T est calculé une fois pour toute pour la position P0, puis est mis à jour successivement pour P1 puis P2 (en même temps que l'exploration de l'arbre de jeu). La frontière est alors calculée en P2. Ensuite T est remis à jour pour P1 en « déjouant » le dernier coup. L'examen des autres coups possibles en P1 peut avoir lieu de la même façon. Une fois que tout l'arbre commençant par P1 est exploré, T est mis à jour pour P0 en « déjouant » un coup et les autres coups possibles en P0 sont examinés à leur tour.

En conclusion, l'introduction de T accélère le calcul de la frontière au prix d'un peu plus de mémoire (pour T lui-même et pour mémoriser les coups afin de pouvoir les « déjouer »). De plus, cette méthode, au contraire de celle basée sur la mobilité, ne nécessite pas de générer tous les coups possibles dans une position avant de pouvoir passer à un descendant dans l'arbre de jeu.

Calcul incrémental de la frontière

Il est possible d'accélérer encore ces calculs ! Le principe est de ne calculer que les variations de la fonction lorsqu'un coup est joué, plutôt que la fonction elle-même. La frontière est modifiée de trois façons lorsqu'un coup est joué : une case de plus est occupée, les voisines de cette case ont une valeur différente dans le tableau T et des cases sont retournées.

L'algorithme fonctionne donc en trois étapes que nous allons détailler. Soit F la frontière à calculer dans une position P donnée (par exemple celle de la figure 5). Ce qui nous intéresse est la variation de F par rapport à F0, frontière du père P0 de P dans l'arbre de jeu (la position de la figure 3).

• **Initialisation :**
F:=F0 (:= signifie « reçoit la valeur de », c'est l'affectation dans la notation du langage Pascal). *F := -14.*

• **Étape 1 :** F est modifiée pour tenir compte de la nouvelle case. Le nombre de ses voisines (c'est-à-dire la valeur correspondante de T) est enlevé à F si c'est le programme qui joue, ajouté si c'est l'adversaire. *F := -14 + 6 = -8 (l'adversaire joue en C7).*

• **Étape 2 :** T est mis à jour comme décrit plus haut. Pour chacune des voisines CV de la case jouée, F est modifiée de : +1 si CV appartient au programme, 0 si CV est vide, -1 si elle est à l'adversaire. Ceci a lieu avant que les retournements ne soient effectués. *F:= -8 + 2 = -6 (deux cases au programme parmi les voisines).*

• **Étape 3 :** Les retournements sont effectués. Pour chaque case retournée, le double de la valeur correspondante de T est enlevé si c'est le programme qui joue, ajouté si c'est l'adversaire. *F:= -6 +2*4 = +2. Nous pouvons vérifier par un calcul direct que la frontière vaut 2 dans la position de la figure 5.*

La principale économie de cette deuxième modification vient du fait que l'on ne parcourt plus T pour le calcul de F. En fait, aucun nouveau calcul d'indices n'est nécessaire : dans l'étape 2, les voisines doivent être parcourues pour mettre à jour T, et dans l'étape 3, les cases retournées doivent être parcourues pour les retourner justement ! Le prix à payer pour cet énorme gain de temps est de mémoriser les valeurs successives de la frontière pour pouvoir remonter dans l'arbre de jeu (pour T il était plus rapide de faire la transformation inverse que de mémoriser les états successifs).

Une variante de cette méthode est utilisée dans le programme Comp'Oth de François Aguillon. La principale différence est que le calcul de la frontière reste séparé pour les deux joueurs et qu'une fonction non linéaire est appliquée aux résultats avant d'en faire la différence.

Conclusion

Nous avons examiné différentes approximations de la mobilité pour obtenir des fonctions d'évaluation les plus rapides possibles. Il est bien évident qu'une bonne fonction d'évaluation doit intégrer d'autres caractéristiques d'Othello ne serait-ce que parce que toutes les cases ne jouent pas le même rôle (les coins pour ne citer qu'eux), mais la mobilité est suffisamment importante pour que l'on s'y intéresse.

L'idée générale à retenir est qu'il est toujours possible d'accélérer une fonction d'évaluation au prix d'un plus grand besoin en mémoire. Une bonne façon d'y arriver est de calculer les variations de la fonction plutôt que sa valeur elle même.

Heuristiques de milieu de partie

par Nicolas Becquet et Jean Delteil

I. Introduction

Nous avons tenté d'établir une synthèse (sûrement pas exhaustive) de diverses techniques tendant à améliorer l'algorithme alpha-bêta, ceci en mettant nos forces en commun (docs, sources !, échanges d'idées). Le mieux était de présenter également des résultats chiffrés. Notre étude porte sur le milieu de partie, mais certaines des méthodes sont utilisables en finale. Les finales feront peut-être l'objet d'une publication du même style... Beaucoup de ces techniques doivent leur existence aux recherches sur les logiciels d'échecs et les documents sont nombreux (mais en anglais...). Mais souvent l'implémentation pour Othello est délicate, et plusieurs années de travail ont été nécessaires. Les résultats que nous fournissons sont obtenus pour un alpha-bêta de profondeur fixe, c'est-à-dire ne faisant pas d'extension de recherche sur certains coups comme il est d'usage dans les programmes d'échecs. À notre connaissance l'extension singulière n'est pas ou peu utilisée pour Othello.

II. Heuristiques d'ordonnancement des coups

Nous présentons ici quelques méthodes d'ordonnancement des coups dans la recherche alpha-bêta. Nous en verrons la grande efficacité globale par rapport à un alpha-bêta de base. Ce qui confirme, si besoin en était, la justesse du paradigme « essayer d'abord les meilleurs coups à chaque nœud de l'arbre ».

II.1 Base de comparaisons

Alpha-bêta standard, retournant la cote et la variante principale. Il y aurait aussi à dire sur l'implémentation de l'algorithme, notamment sur l'utilisation d'une version non récursive.

II.2 Recherche courte, fenêtrage, alpha-bêta progressif

Une recherche courte (une constante R_SHORT indiquant la profondeur doit être établie) est effectuée sur chaque coup racine. Les lignes principales et cotes obtenues sont mémorisées. Les coups racines sont triés par cotes décroissantes. Objectif avoué :

- trier rapidement les coups racines ;
- obtenir les lignes de jeux principales ;
- obtenir une cote prévisionnelle du meilleur coup (réglage du fenêtrage).

On effectue alors la recherche réelle. On utilise un alpha-bêta de profondeur progressive, qui démarre à R_SHORT+1 jusqu'au niveau max souhaité. (Nous rappelons au passage que cette méthode offre un avantage considérable dans la gestion du temps. En effet, on peut arrêter le processus quand on le souhaite ; on dispose alors toujours du meilleur coup de la précédente itération, dans le pire des cas celui de la recherche courte.)

Revenons à nos moutons. Dans une boucle de recherche progressive, on doit par souci d'efficacité commencer par examiner le meilleur coup de l'itération précédente. Pour ce coup, la fenêtre alpha-bêta ne sera pas initialisée à [$-\infty$, $+\infty$] mais à [Cote_precedente – valeur, Cote_ precedente + valeur]. Valeur sera implémentée selon votre fonction d'évaluation (ce peut être une constante ou un savant calcul). Bien sûr quelquefois une nouvelle recherche sera nécessaire, lorsque la valeur retournée sera hors fenêtre ($\leq$ alpha ou $\geq$ bêta). Le premier coup résolu, les suivants seront recherchés en *zero-window* (c.-à-d. [α, α+1]). Dans le cas où un coup dépasse alpha, une nouvelle recherche est nécessaire (pas toujours) avec [α+1, $+\infty$] par exemple. Ce coup devra alors être placé en tête de liste pour la prochaine itération.

Pour que tout marche bien, il nous faut un alpha-bêta légèrement modifié (algorithme PVS : Principal Variation Search). L'algorithme reçoit un paramètre supplémentaire par rapport à alpha-bêta, qui est la variante principale. Le coup de la variante principale (s'il existe) est exploré par appel récursif à PVS en premier. Les autres coups sont explorés avec fenêtre zero-window par un appel à l'alpha-bêta standard. Dans le cas où il n'y a plus de coup dans la variante principale, l'alpha-bêta standard est invoqué.

II.3 Tri des coups

À chaque nœud, la liste des coups jouables est triée par mérite (en utilisant une partie de la fonction d'évaluation). Toutefois la priorité (dans PVS) est gardée au coup de la variante principale. Comme nous le verrons les résultats sont très bons. Un bon exemple d'une programmation facile qui rapporte beaucoup. La principale difficulté est de trouver une fonction d'évaluation simplifiée qui doit :

- être très rapide ;
- bien trier les coups.

Beaucoup de tâtonnements sont nécessaires. Si vous tentez l'expérience n'oubliez pas de contrôler bien sûr le nombre de nœuds générés mais surtout le temps utilisé... Pour gagner un peu de temps, on peut arrêter le tri quand on se rapproche de l'horizon de recherche. Nous utilisons une limite de 2 (le tri est effectué quand il reste plus de 2 niveaux à explorer).

II.4 Utilisation d'une table d'élagage

Il s'agit de garder en mémoire des informations sur l'arbre de jeu (par exemple des données issues d'une recherche courte). Il semble en effet dommage dans la recherche courte de ne garder que les lignes principales pour chaque racine. D'où l'idée du stockage mémoire d'informations. Le plus simple semble être pour une position donnée le meilleur coup à jouer. C'est une sorte de table de « coups meutriers » où un coup est associé à une position. Le problème de base d'une telle table est l'indexation. En effet, il est impensable de réserver une place mémoire pour chaque position possible sur l'othellier. Il faut donc se donner :

1) une taille raisonnable de stockage ;

2) un moyen rapide de stocker et de retrouver l'information.

Pour 1), deux tables (une par couleur) de 64Ko peuvent par exemple être utilisées, nous verrons qu'un octet d'information suffit. Une fois cette taille fixée, il nous faut (point 2) un système qui « convertisse » une position en un indice de table qui ici sera compris entre 0 et 65535 ce qui sera très pratique (l'index correspond à un mot de 16 bits). C'est le principe du « haching » ou « adressage dispersé » (aussi appelé « hachage »). Bien entendu les doublons sont inévitables dans le cas général (deux posi-

tions différentes seront converties en un même indice). La fonction d'adressage dispersé doit posséder deux propriétés essentielles :

• rapidité du calcul de l'index ;

• faible probabilité d'obtenir des doublons (dispersion aléatoire et uniforme).

Ici, nous ne pouvons pas utiliser un calcul d'adresse basé sur la méthode classique des nombres premiers (trop de calculs). Une méthode spécifique aux arbres de jeux a été mise au point par Zobrist, qui permet entre autres merveilles un calcul incrémental.

Principe des clefs de Zobrist

Nous nous contenterons d'une description sans justification théorique de la méthode. Pour simplifier, supposons que nous ayons une seule table à gérer THASH dont chaque poste contiendra les informations mémorisées. Il nous faut d'abord une petite table annexe, dont chaque entrée correspond à une case de l'othellier TALEA. Dans cette table nous allons affecter, une fois pour toutes, deux nombres aléatoires à chaque entrée (un nombre pour blanc, un nombre pour noir).

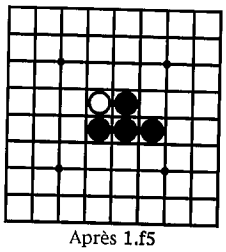
Voici maintenant la méthode pour calculer l'adresse d'une position :

```
clef = 0 ; /* élément neutre du XOR */
Parcours de l'othellier case c
 si la case (c) n'est pas vide
   clef=clef XOR (TALEA[c,couleur(c)])
 Finsi
Fin du Parcours

/* clef contient la clef de la position */
/* l'entrée est retrouvée par THASH[clef] */
```

On le voit, le calcul est fort simple. Pour un calcul incrémental, il suffit de faire l'opération XOR sur la clef courante et la case jouée. En principe, il faudrait également tenir compte des pions retournés, mais cela ne sera pas nécessaire dans notre table d'élagage de base, par contre justifié dans une table de transposition. Dans l'alpha-bêta, il suffit de passer en paramètre la clef courante et de calculer la nouvelle clef avant chaque appel récursif.

Position exemple



Après 1.f5

Soit `1010101111000001` la clef de la position de départ, soit `1000000011110000` la clef de la case f5 pour les Noirs. Après le coup, la clef devient `0010101100110001`.

Implémentation de la table d'élagage

- Éléments de la table

Nous aurons dans une entrée (un octet).

• Le coup à jouer (6 bits).

• Indicateur de profondeur de recherche (0 si < 4, 1 sinon) (1 bit).

• Indicateur de zero-window (1 bit) (1 si le coup a été calculé par zero-window ou est hors-fenêtre alpha-bêta, 0 sinon).

• (Une entrée non utilisée contient 0).

Les indicateurs sont très importants, ils permettent de décider d'une priorité en cas de conflit. Un conflit existe quand on veut stocker des informations dans une entrée déjà utilisée. Nous avons utilisé les règles de priorités suivantes :

1. si (le nouveau coup n'est pas calculé en zero-window), alors stockage ;
2. si (l'ancien coup n'est pas calculé en zero-window), alors pas de stockage ;
3. si (l'indicateur de profondeur(ancien) > (nouveau)), alors pas de stockage ;

Globalement, la préférence est accordée aux coups à l'intérieur de la fenêtre alpha-bêta ou aux coups recherchés au moins à une profondeur donnée.

- Compléments

Il est bon de définir une constante K qui indique un niveau de profondeur maxi à partir duquel il n'est plus nécessaire d'utiliser la table. K doit être défini grosso modo en fonction de la taille de la table, ici nous avons pris K=8. Dans un but de contrôle et de tests, il est pratique de tenir quelques statistiques sur la table sous forme de compteurs :

• nombre d'entrées occupées ;

• nombre d'entrées écrasées (réécrites) ;

• nombre de tentatives d'écritures ;

• nombre de tentatives de lectures ;

• etc.

En effet, attention, le coup trouvé dans une entrée n'est pas forcément légal (doublons de clefs possibles).

Remarques

D'autres types d'heuristiques de tri de coups ont été publiées, en particulier :

• « history » : on tente de trier globalement tous les coups possibles dans une table. Pour Othello cette table pourrait permettre d'améliorer l'ordre dans lequel les coups légaux sont générés.

• « countermove » : on tente de trier globalement une table qui associe à chaque coup possible (60 pour Othello) une liste préférentielle des coups à essayer (voir par exemple la description du programme BILL).

Nous recommandons ces deux méthodes surtout quand on n'utilise pas un tri tel qu'expliqué en II.3, sinon le résultat II.3 + countermove est mauvais (nous ne savons pas pourquoi).

III. Heuristiques d'élagage a priori

Il s'agit ici de techniques beaucoup plus agressives et risquées. Le principe général est d'ignorer ou d'explorer partiellement certaines branches de l'arbre. Nous avons éliminé les méthodes qui consistent à réduire arbitrairement le nombre de coups explorés en fonction de la profondeur. Nous préférons les méthodes basées sur un critère de décision. L'appréciation des résultats obtenus reste subjective, certains estimeront par exemple que x% d'erreur sur le coup ou la cote obtenus à la racine ne sont pas tolérables, même si le temps de recherche est diminué de y%. La vérification ultime est de faire jouer des parties entre deux versions de programme (en double toutes-rondes sur p positions de bases).

III.1 La futilité

Le principe de base de cette heuristique est le suivant : on essaye de ne pas développer une position pour laquelle on peut s'attendre à ce que la cote soit hors de la fenêtre alpha-bêta courante. Ce qui paraît très logique. Pour cela, il nous faudrait deux estimations de la position en cours.

Une estimation optimiste (EO) avec une probabilité faible que `cote réelle > estimation_optimiste`. Une estimation pessimiste (EP) avec une probabilité faible que `cote réelle < estimation_pessimiste`. Alors, par exemple, pour un nœud à maximiser, si on connaît EO et EP on arrête la recherche si `EO =< alpha` ou `EP >= bêta`. Tout l'art est de déterminer le calcul de EO et EP. On peut imaginer beaucoup de méthodes et nous vous encourageons à y réfléchir, ce domaine est très ouvert.

L'implémentation la plus ancienne (connue) remonte au programme CHESS 4.5 (années 70 !). Dans ce programme, la « futilité » était implémentée en gros comme suit : seulement pour des positions terminales jugées tranquilles, l'estimation de la cote est fournie par la cote du matériel et un min ou max de la cote positionnelle qui, elle, n'est pas évaluée exactement. Outre les petits élagages produits, le gain de temps est important lorsque le calcul de la cote positionnelle (long) n'est pas effectué.

L'implémentation originale que nous avons utilisée est plus générale et peut être éventuellement appliquée n'importe où dans l'arbre. Prenons encore l'exemple d'un nœud à maximiser. On estime d'abord sa cote (`Cp`) par la fonction d'évaluation habituelle. On suppose connaître deux valeurs `dim_max` et `aug_max` qui indiquent respectivement la diminution et l'augmentation relatives possibles de la cote. On élague alors si `CP+aug_max<=alpha` ou `CP-dim_max>=bêta`. (Bien sûr, on doit déjà avoir à la base `CP < alpha` ou `CP > bêta`.) Reste à définir comment on obtient `dim_max` et `aug_max`. On pourrait se contenter de constantes, mais il s'avère que c'est peu efficace et très dangereux. Nous allons donc actualiser dynamiquement ces valeurs. Il faut, bien sûr, fixer des valeurs initiales, par tâtonnement. Ces valeurs seront ensuites maximisées (dommage !) lors du parcours de l'arbre. L'opération de maximisation est effectuée, s'il y a lieu, après obtention de la cote réelle (`CR`) de la position.

On a alors :

```
aug_max=Max(aug_max,CR-CP)  (CR > CP)
dim_max=Max(dim_max,CP-CR)  (CR < CP)
```

Comme précaution supplémentaire, on peut décider d'utiliser la futilité seulement à partir d'une certaine profondeur dans l'arbre et/ou à une certaine distance de l'horizon. Il est bon également de gérer une variable logique qui indique à l'alpha-bêta si l'on souhaite utiliser ou non la futilité. Utile entre autres dans la recherche courte, pour bien remplir la hash-table !

III.2 Le « razoring » ou réduction de profondeur

Il s'agit ici de réduire, dans certains cas la profondeur de recherche d'une constante (RAZOR). Cas d'application (pour un nœud) :

- on n'est pas dans une recherche de type zero-window (évite l'utilisation récursive du RAZORING) ;
- ce n'est pas le premier coup testé de la position (pour « assurer » alpha et bêta) ;
- profondeur-RAZOR > 0 (devinez !).

Alors on y va ! : on recherche avec une profondeur p-RAZOR-1 (au lieu de p-1) en passant astucieusement les paramètres alpha et bêta pour le niveau suivant sous la forme (α, α+1). Après retour de la valeur V si celle-ci est dans la fenêtre de base (($V>\alpha$) ET ($V<\beta$)), il vaut mieux alors effectuer le réexamen à profondeur réelle.

Remarques

Encore ici d'autres techniques ont été publiées, on peut citer par exemple le « NULL-MOVE ». C'est encore de la « futilité » ; une cote pessimiste est obtenue par une analyse à une profondeur réduite en passant carrément le trait à l'adversaire (NULL-MOVE). La justification est que si l'on ne joue pas, la valeur retournée sera une cote pessimiste et l'on pourra agir en conséquence. Cela s'applique bien à un jeu comme les échecs où le fait de jouer est généralement avantageux (sauf zugzwang pour les connaisseurs). Mais à Othello, c'est plutôt le contraire qui est la règle. On aurait donc en général une cote optimiste... à essayer ! (Semble difficile à implémenter, beaucoup de cas particuliers.)

Méthode	P	Temps	Feuilles	Nœuds	Degré	Pire	err1	err2
1	8	58	484238	651754	5,13	4367384		
1+2	8	13	98976	152293	4,21	590632		
1+2+3	8	8	45343	73216	3,82	310980		
1+3	8	17	121303	176072	4,32	824152		
1+2+4	8	9	59721	97474	3,95	384225		
1+2+3+4	8	7	46529	77325	3,83	282111		
(*+5)	8	5	27504	48067	3,58	219958	0%	4%
(*+6)	8	7	44569	73892	3,81	259880	0%	25%
(*+5+6)	8	4	25435	45138	3,55	192222	6%	26%

Tableau 1 : comparatif des méthodes

	Occupation	Lectures				Ecritures		
Méthode	Entrées	Tent.	Null	Leg	Best	Tent	Ovr	Null
1+2+4	15620	41212	15620	20411	19179	22137	6339	46
1+2+3+4	14233	34035	14233	15891	14970	19169	4812	30
*+5	10116	23802	11721	9816	9148	13111	2912	19
*+6	14061	32501	14061	14645	13770	18835	4650	37
*+5+6	9942	22884	11629	9086	8466	12792	2767	23

Tableau 2 : utilisation table adressage dispersé

IV. Expérimentation

IV.1 Protocole de tests

Nous sommes partis d'un échantillon de 50 positions aléatoires comprenant de 24 à 34 pions. Un autre test indépendant, sur 20 positions, a donné des résultats très voisins. Pour le comptage des nœuds et feuilles, seule la dernière itération est prise en compte (donc uniquement à profondeur 8). Mais le temps de calcul est pris dans son intégralité (cf. tableau 1).

Bien sûr il faudrait tester aussi avec :
- un échantillon plus volumineux (1000 !) ;
- différentes profondeurs ;
- par catégorie de positions (fortes, faibles, moyennes) ;
- etc.

Mais tout cela prend du temps et aurait surchargé cet exposé que nous voulons assez général.

Légende du tableau 1:
- Méthode : (1) standard, (2) fenêtrage, recherche courte, (3) tri des coups, (4) table d'élagage, (*) 1+2+3+4, (5) futilité, (6) razoring.
- P : profondeur de recherche.
- Degré : degré moyen de branchement.
- Pire : nombre maximum de nœuds générés.
- Feuilles : nombre moyen de positions terminales.
- Nœuds : nombre moyen de nœuds explorés (feuilles comprises).
- err1 : coup retourné faux.
- err2 : cote retournée fausse (mais coup juste).

Utilisation table adressage dispersé (cf. tableau 2)

Légende du tableau 2 :
- Entrées : nombre d'entrées utilisées
- Tent : tentatives de lect/écr
- Null : lect/écr non servie
- Leg : lect et le coup est légal
- Best : lect et le coup est légal et le meilleur
- Ovr : nbr d'écrasements en écriture

- Commentaires :

(1) : le degré de branchement semble plausible. Environ cinq coups sont explorés en moyenne à chaque nœud. À noter le pire des cas, très loin de la moyenne.

(1+2) : bonne efficacité, pour ceux qui douteraient de l'utilité d'une recherche courte et du fenêtrage et de la recherche progressive. Test avec R_SHORT=4.

(1+2+3) : encore mieux ! On peut noter cette fois que la réduction drastique de l'arbre n'est pas convertie entièrement en gain de temps (le tri des coups est relativement coûteux en temps).

(1+3) : à comparer avec (1+2+3). Qui peut encore douter de l'utilité de la recherche courte et de l'alpha-bêta progressif ? On vérifie encore bien l'efficacité redoutable de (3).

(1+2+4) : à comparer surtout avec (1+2), on vérifie bien l'efficacité de la table. Mais très proche de 1+2+3. À noter, chaque fois qu'un coup légal est trouvé dans la table, 19179/20411 fois c'est le meilleur coup ou un coup suffisant pour élaguer !

(1+2+3+4) : un tout petit peu mieux que 1+2+3. Peut-être qu'à des prondeurs supérieures...

(*+5) : la futilité a été désactivée en recherche courte. Le niveau mini d'activation est 6, la distance maxi est 2 d'où en fait utilisée dans le test uniquement à distance de 2 niveaux de l'horizon. (Vous devriez obtenir des résultats meilleurs, nous ne pouvions l'appliquer qu'à des niveaux pairs dans le test.) N'est également pas utilisée, pour un nœud donné, quand un coup légal est trouvé dans la table d'élagage. Relativement peu d'erreurs relevées, uniquement des cotes fausses et du même ordre de grandeur. L'occupation de la table diminue logiquement. En conclusion, une technique assez sûre et efficace.

(*+6) : implémenté avec RAZOR=2, désactivé en recherche courte. Inférieur à (*+5), mais beaucoup plus dangereux ! Et si on restreint encore les conditions d'application, le gain sera minime. (Vous devriez obtenir des résultats meilleurs (avec RAZOR=1), nous ne pouvions l'appliquer qu'à des niveaux pairs dans le test.)

(*+5+6) : cette fois-ci quelques erreurs sur le coup choisi apparaissent. Une combinaison performante mais risquée. (Ici encore sûrement des résultats meilleurs avec RAZOR=1.)

V. Conclusion

Nous avons bien montré l'importance des heuristiques ; certes on est toujours exponentiel mais la différence entre un degré de branchement de 5,00 et 3,60 au niveau 12 représente environ 50 fois moins de nœuds explorés ! Nous espérons que beaucoup d'entre vous tireront profit de cette étude (euh... pas trop tout de même). Si vous faites des tests vous pouvez bien sûr obtenir des résultats sensiblement différents, car, dans tous les cas cette fameuse « fonction d'évaluation » a plus que son mot à dire. Cela tient en particulier (qualité mise à part !), à l'amplitude des cotes possibles et à la probabilité de rencontrer des cotes égales. Nous attendons impatiemment vos critiques, suggestions, publications, questions, cela ne peut que nous encourager à poursuivre nos efforts (courrier à adresser à la FFO qui transmettra).

VI. Références

- Méthode d'adressage dispersé :

- Zobrist A. L (1970), « A New Hashing Method with Applications for Game Playing. » Tech. Rep. 88., Computer Sciences Department, University of Wisconsin, Madison, Reprinted (1990) in ICCA Journal Vol. 13 No. 2. pp 69-73.

- Heuristiques diverses (synthèse et nombreuses références) :

- Chun Ye and T. Antony Marsland, « Experiments In Forward Pruning With Limited Extensions. » Computing department, University of Alberta Edmonton. ICCA journal Vol. 15 No. 2. pp 55-66.

- Futilité :

- Peter W. Frey, « Chess Skill In Man And Machine. », Springer-Verlag New York ISBN 0-387-07957-2 pp 82-118.

Saturday Night Fever

par François Aguillon

Cet article prétend traiter d'informatique, et plus précisément d'un syndrome qui frappe les veilles de tournoi. Comme les tournois ont lieu le dimanche, et que *Saturday* signifie Samedi[1], *night* est une négation et *fever* veut dire dormir, vous avez compris qu'un programmeur d'Othello ne dort jamais la veille d'un tournoi ; voici quelque chose pour eux qui agira comme un somnifère.

Introduction

Mais, se demande le lecteur qui a eu le courage de lire l'article jusqu'ici, pourquoi donc ne pas dormir tranquillement, puisqu'ils n'ont pas de bibliothèque d'ouvertures à regarder à la hâte ?

La réponse tient en deux points : tout d'abord, il faudrait être un peu fou pour tenter la veille d'un tournoi une « amélioration » qui ne soit pas facilement testable et estampillée bugfree[2] ; faute de quoi, la loi de Murphy s'appliquerait, qui dit en l'occurrence que les bugs indétectables dans les conditions usuelles se manifestent très souvent pendant les tournois. Mais d'un autre côté, il faudrait aussi être bien sûr de soi pour affirmer que son programme n'est pas améliorable, et s'en aller dormir tranquille. Alors que faire ? Essayer d'améliorer quelque chose qui ne soit qu'un détail infime sur le plan de la programmation, et qui ait des conséquences certaines sur le niveau de jeu.

Il est tentant alors de jouer sur les coefficients pondérant les diverses composantes de la fonction d'évaluation. En effet, une fonction d'évaluation digne de ce nom prend en compte plusieurs critères : par exemple, à Othello, les critères peuvent être le nombre de pions, la mobilité, la structure des bords, la structure des coins, la parité, etc. En appelant x, y, z... les notes obtenues selon chacun des critères, la manière traditionnelle d'en déduire une cote globale de la position est d'en faire une combinaison linéaire :

$$E = ax + by + cz + \cdots \qquad (1)$$

Il existe évidemment un jeu idéal de coefficients a, b, c,... On peut le chercher par tâtonnement : c'est la fièvre du samedi soir, car c'est une chose que l'on fait très simplement, et il est difficile d'introduire un bug de la sorte, encore que l'overflow[3], ça existe. Le problème de l'optimisation de ces coefficients est toutefois loin d'être trivial :

- le nombre de coefficients peut être élevé. De plus, ces coefficients peuvent varier en cours de jeu : par exemple, au soixantième coup, le seul coefficient non nul est celui qui compte le nombre de pions ;
- bien souvent le programme joue mieux que son concepteur. Celui-ci a donc des difficultés à évaluer l'effet positif ou négatif d'une variation de ces coefficients. Pour compenser sa faiblesse, on peut être tenté d'utiliser un sparring-partner[4] : il existe de bons programmes, après tout ! Mais cela n'est pas très fiable, car on risque alors de ne pas trouver la meilleure fonction d'évaluation, mais celle qui est la plus adaptée à l'adversaire ;
- enfin, gardons à l'esprit qu'Othello est essentiellement un jeu de hasard. Avant de dire qu'un programme est plus fort que sa version antérieure, il convient de faire un nombre significatif de parties avec lui, afin d'avoir une statistique suffisante pour s'affranchir des fluctuations dues au seul hasard.

En résumant, la technique artisanale d'optimisation des coefficients de la fonction d'évaluation arrive à être fastidieuse sans être fiable. Est-il possible de faire autrement ? Je ne m'étais jamais posé la question, jusqu'à la lecture récente d'un article de Kai Fu Lee et Sanjoy Mahajan[5], par ailleurs connus pour être les auteurs du programme Bill. Dommage, parce que la réponse est oui.

Principe

L'idée part d'un constat assez simple : la seule (!) chose que réalise une bonne fonction d'évaluation est de juger correctement la valeur d'une position. Supposons que l'on sache déterminer a priori la valeur de chacune des positions d'une collection donnée ; comparer cette valeur à ce que donne votre fonction d'évaluation permet de « calibrer » celle-ci. Exemple : si, sur toutes les positions gagnantes, votre fonction répond « position gagnante », gardez-la et lisez autre chose. Si au contraire elle répond systématiquement « position perdante », vous n'avez rien compris à Othello mais vous avez de la chance ; mettez un signe « - » au bon endroit, et ne cherchez surtout pas à comprendre. Si elle répond une fois sur deux « gagnante », une fois sur deux « perdante », vous venez de prouver avec un programme qu'Othello est un jeu de pur hasard. Plus sérieusement, on peut alors optimiser les coefficients de la fonction d'évaluation de manière rationnelle : il suffit de chercher l'accord le plus serré possible entre votre fonction et la vraie valeur de la position. On verra même que l'on peut optimiser automatiquement ladite fonction d'évaluation. Magique ?

Que nenni ! Il y a une arnaque ! Il faudrait pour réaliser ce qui est annoncé plus haut disposer d'un truc qui vous permette de dire si une position est gagnante ou perdante. Ce truc, c'est la Fonction d'Évaluation de Dieu (FED dans la suite de l'article). Si l'on connaissait la FED, ce serait bien d'un côté, sauf que, bon, le jeu serait mort. Essayons de voir si l'on peut se passer de cette FED. Une première idée simple consisterait à la remplacer par la fonction d'évaluation de ses saints. C'est en effet l'année ou jamais de profiter du fait que des quatre archanges assis à droite de Dieu (St Hideshi, St

---

1 en haut-allemand.

2 inoxydable, en espagnol.

3 saison des pluies, en bengali.

4 petit frère, en moldave.

5 je ne vais pas vous dire ce que signifie leur nom en araméen, mais donner les références de l'article en question : « A pattern classification approach to evaluation function learning », *Artificial Intelligence*, **36** (1988), pp.1-25.

Didier, St Paul et St Brian)[6], deux ne parlent pas trop mal le français. Hélas, trois fois hélas, les archanges et tous les saints[7] sont très forts pour donner des conseils sur « quoi mettre » dans une fonction d'évaluation — on ne compte plus les articles dans notre bulletin paroissial traitant par exemple de la parité, la mobilité, les bords déséquilibrés, etc. — mais leur discours sur l'équilibre des divers ingrédients est beaucoup moins compréhensible par les cloportes que nous sommes[8]. Peut-être même que ce « discours » (au sens du λογοσ[9]) n'existe pas vraiment et que tout se passe à l'estime, dans la plus sombre irrationalité, caca beurk.

Alors, il n'y a pas d'espoir ? Si je réponds non maintenant après vous avoir abruti de deux pages d'insanités, je vais me faire lyncher[10]. Donc je réponds oui. En effet, il n'est pas nécessaire de disposer de la FED elle-même, mais, et ce n'est pas exactement pareil, d'une collection de positions sur laquelle on connaît la valeur de cette FED. Il suffit donc tout simplement de disposer de transcriptions de parties jouées par Dieu. En effet, Dieu jouant comme un dieu (c'est ce qui fait sa force, d'ailleurs, encore que certains jaloux disent qu'il est surtout fort mentalement, mais, que d'un point de vue purement technique...), il est certain que toutes les positions qu'il atteint au cours d'une partie sont des positions gagnantes. Et donc on peut aligner sa fonction d'évaluation le plus près possible de celle de Dieu.

Comment faire ? Supposons, pour illustrer notre propos, que la fonction d'évaluation ne fasse intervenir que deux composantes :

$$E = ax + by$$

et que nous ayons, pour chacune de ces composantes x et y la même évaluation que Dieu lui-même, le problème qui nous préoccupe n'étant que de déterminer les valeurs de a et b lorsque N coups ont été joués. Il suffit de porter dans un plan de coordonnées x et y un « + » pour chaque évaluation de la position (forcément gagnante) de Dieu, ou, si Dieu n'a pas le trait, un « - » pour évaluer la position (forcément perdante) de son adversaire (cf. figure 1).

Ceci étant fait, comme nos critères x et y sont les critères idéaux, on voit que le plan se découpe en deux demi-plans, l'un gagnant et l'autre perdant. En appelant α l'angle directeur du vecteur « position gagnante moyenne » g, il est clair que la bonne fonction d'évaluation est

$$E = g\cos(\alpha)x + g\sin(\alpha)y \qquad (2)$$

où g est le module de g. À partir de ses composantes individuelles, vous avez trouvé la bonne fonction d'évaluation sans rien connaître à Othello, rien qu'en espionnant. Sa fiabilité est reliée à la précision avec laquelle est défini l'angle α. Il est clair que cette précision est d'autant meilleure que la direction du vecteur g est mieux définie, c'est-à-dire que l'on dispose d'un grand nombre de points pour la calculer, donc d'un grand nombre de transcriptions de parties jouées par Dieu.

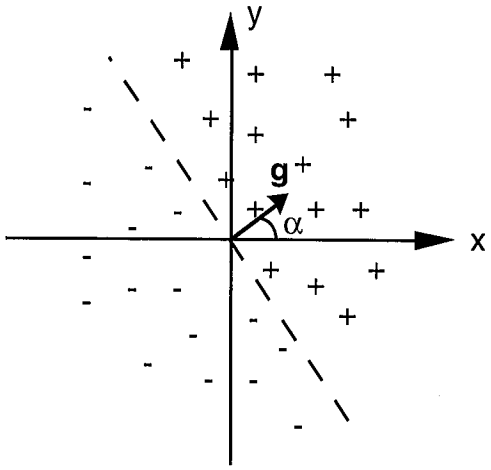


Figure 1

Précaution oratoire

Avez vous remarqué l'anormale proportion d'agrégés de mathématiques parmi les distingués membres de la FFO, surtout dans la catégorie des grosses cylindrées ? C'est seulement à leur intention que je précise que je ne suis qu'un pauvre physicien, que je vais peut-être dire des âneries ou des évidences pour eux, mais tant pis. Que les autres veuillent bien oublier la faiblesse dont je viens de faire preuve avec ce trait de modestie, et continuent à croire que la prochaine médaille Fields[11] est pour moi, merci.

Objections

Il m'étonnerait, me décevrait même, que vous n'en ayez pas. Si vraiment vous séchez, je vais vous en souffler, en vrac :

1) Dieu n'a jamais autorisé que l'on transcrive ses parties ;

2) les critères individuels dont on dispose ne sont pas forcément ceux de Dieu. Le joli (?) petit dessin représenté plus haut risque de ne pas ressembler à celui qu'on obtiendra avec ses propres critères ;

3) et puis d'abord, il y a plus de deux critères à prendre en compte pour bien jouer à Othello.

Les gens rapides voient tout de suite que l'objection (3) est idiote, vue que la généralisation à n dimensions du raisonnement présenté plus haut est triviale. Que ceux qui ne sont pas assez rapides pour avoir vu cela tout de suite

---

6 (NDLR) il s'agit bien sûr des quatre premiers du championnat du Monde 1990, date de parution de cet article.

7 il ne faut vexer personne ! Parmi tous les saints d'aujourd'hui se cache peut-être un futur pape.

8 taxer deux éminents membres de la FFO d'hermétisme et tous les autres lecteurs de cette revue de cloportes est aussi délicat que diplomatique, n'est-il pas ?

9 λογοσ : discours en grec.

10 lynchee : petit fruit très sucré et très juteux poussant en Chine. Par extension, se faire lyncher : expression intraduisible, que l'on peut traduire par « se faire réduire en bouillie ».

11 célèbre mathématicien qui, avec Strawberry et Forever démontra la célèbre conjecture de Crezknÿ-Dugland. La médaille Fields est aux mathématiciens ce que le label rouge est au poulet de Bresse.

se rassurent : l'objection (3) est peut-être la plus lourde de conséquences pratiques et j'ai la faiblesse de faire partie de ceux qui sont assez pragmatiques pour penser qu'en matière d'informatique, la pratique est importante (a-t-on jamais vu une machine de Turing atteindre le mégaflop[12] ?). Comme quoi, à être rapide on ne gagne parfois qu'à se mettre le doigt dans l'œil très vite....

Théologie opérationnelle

Réglons d'abord un problème sous-jacent à tout le début de cet article et qui n'est que confusément exprimé dans la première objection : Dieu existe-t-il ? Beaucoup de philosophes et de métaphysiciens se sont penchés sur la question et, aussi prétentieux que cela puisse paraître, je voudrais apporter à l'édifice de leurs réponses une contribution que je pense assez novatrice : ça m'est égal. De manière raisonnée. Que cherchais-je dans les transcriptions des parties de Dieu ? La certitude qu'une position était gagnante. Que vais-je trouver dans la transcription d'une partie entre par exemple St Hideshi et (vite un nom qui ne vexera personne) F. Aguillon ? La quasi-certitude que, si ce dernier n'a pas gagné dans les 50 demiers coups, la faute n'en incombe pas comme il le prétend avec sa mauvaise foi habituelle à la prétendue lourdeur des saucisses melba qu'il venait d'ingérer, mais plutôt au fait que dès le coup 10, il était en position perdante. En termes plus savants, il est plausible d'admettre qu'il y a une corrélation certaine entre le fait pour un joueur de se retrouver en position favorable et celui de gagner la partie. Si vous n'admettez pas ceci, la suite ne vous concerne plus. On peut donc tout à fait remplacer les transcriptions de Dieu par des transcriptions de joueurs plus plébéiens, à condition toutefois qu'au moins un des deux ne joue pas comme un guignol, faute de quoi la corrélation entre gain de la partie et avantage en cours de partie s'évanouit. Or des transcriptions de parties entre deux joueurs dont un au moins n'est pas un guignol, ça n'est pas cela qui manque. Il en existe même beaucoup maintenant qui sont sur des disquettes ; et une base de données, cela peut aussi servir à améliorer son milieu de partie... Et même, si l'on veut rendre les choses plus sérieuses encore, rien n'empêche de faire jouer divinement par votre programme favori les derniers coups des parties dont vous disposez en lieu et place de leurs auteurs réels.

Pour établir l'équivalent de la figure 1 à partir de transcriptions réelles, la marche à suivre est plus compliquée, parce qu'on ne sait pas a priori lequel des deux joueurs a joué le mieux : on ne le sait qu'à la fin de la partie. On évalue donc la position au coup N, ce qui permet de placer un point dans le plan d'évaluation Oxy. Ensuite, on joue la partie jusqu'à la fin (en faisant éventuellement jouer les derniers coups par le programme). Si celui qui avait le trait au coup N gagne, on transforme le point en « + », s'il perd on transforme le point en « - ». Sur le résultat obtenu, la différence tient au fait que les hommes font des erreurs[13]. Ainsi, ils peuvent perdre une partie gagnante selon les critères de Dieu ; c'est triste pour eux, mais que les optimistes se consolent en disant qu'ils peuvent aussi gagner des parties perdantes. Ceci est illustré sur les figures 2 et 3 sur lesquelles on a reporté les évaluations selon les critères de Dieu de parties jouées par des bons (figure 2) et des guignols (figure 3) : on y constate une certaine interpénétration des « + » et des « - » d'autant plus faible que les joueurs sont plus proches de Dieu.

Pour un lot de joueurs homogènes, on doit s'attendre à ce que les diagrammes obtenus pour des positions de début de partie ressemblent plutôt à la figure 3 et ceux en fin de partie à la figure 2 : en effet, il est plus « facile » de faire une erreur entre le coup 1 et le coup 60 qu'au coup 60.

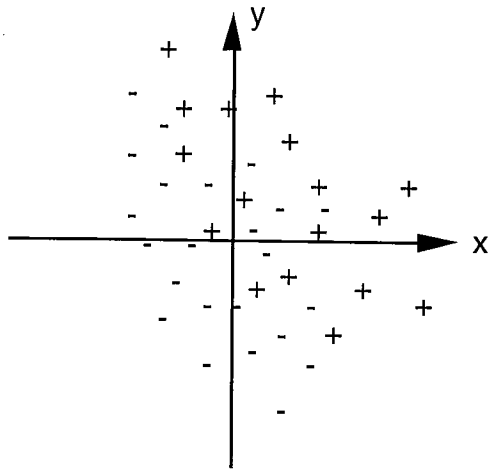


Figure 2

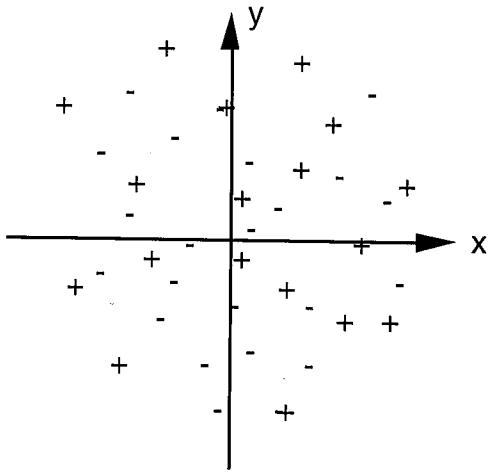


Figure 3

Le secret des dieux

Le paragraphe précédent nous a permis de comprendre ce qui se passait lorsque l'on passait de transcriptions de Dieu à des transcriptions humaines. Toutefois, pour juger de ceci, nous supposions encore que nous connaissions les critères de la FED. Voyons maintenant ce qui résulte du fait que nous ne connaissons pas ces critères (objection numéro deux susdite). Pour cela, nous ferons provisoirement l'hypothèse que nous disposons à nouveau

[12] la machine de Turing est un ordinateur parfaitement ridicule qui a fait un flop commercial retentissant, alors que maintenant les superordinateurs font des gigaflops sans effort apparent.

[13] les femmes aussi. « Les femmes aussi », c'était le titre d'une émission TV féministe dans les années soixante que Maman regardait pendant que Papa apprenait à faire la vaisselle. Pendant ce temps, les enfants rigolaient...

de transcriptions de parties de Dieu. Après, nous mettrons tout à bouillir dans la même marmite, jusqu'à ce qu'elle nous explose à la figure.

Les critères humains présentent évidemment des défauts de comportement par rapport aux critères divins présentés figure 1. Ces défauts sont plus ou moins graves et conduisent à une foule de situations différentes. La situation la plus grave est celle où les critères ne sont pas du tout pertinents : x=l'âge du capitaine, y=l'usure de la moquette[14]. On aura un diagramme du style de celui de la figure 3, où les coups joués par Dieu n'ont pas meilleure cote que ceux joués par son adversaire. Si les critères ne sont pas tout à fait pertinents, la situation se présentera plutôt comme sur la figure 2, séparant une région majoritairement « + » et une région majoritairement « - ». Cela n'est pas pour autant que le diagramme sera forcément identique à celui de la figure 2 ; des différences notables peuvent surgir :

- la droite séparant la zone « + » de la zone « - » ne passe pas forcément par l'origine des coordonnées x et y ;
- cette droite peut même être complètement tordue !

La première dénote une maladie bénigne, qu'il faut néanmoins soigner car ses conséquences peuvent en être désastreuses. Regardons la figure 4 : à cause de ce petit problème, le vecteur « position gagnante moyenne » g et le vecteur « position perdante moyenne » p ont la même direction : combiner les critères pour une fonction d'évaluation normale et pour une fonction d'évaluation « à qui perd gagne » conduira donc aux mêmes coefficients. Gênant...

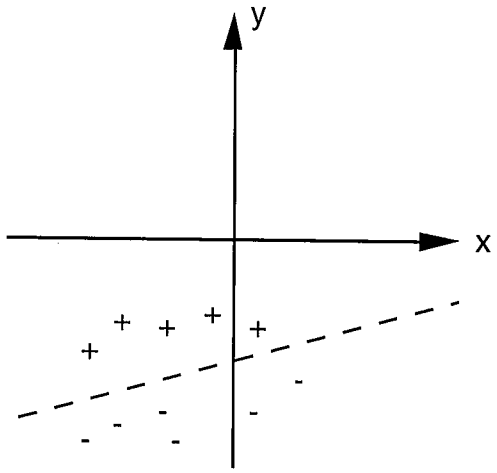


Figure 4

La solution est très simple : il suffit de faire un changement de l'origine des coordonnées tel que g et p soient opposés : on se ramène ainsi automatiquement au cas de la figure 2.

Lorsque la « droite » de séparation des deux régions est tordue, le problème se complique un peu. En fait, on peut distinguer deux cas limites : dans celui illustré sur la figure 5, la fausse droite est une vraie parabole. Ici, on peut encore trouver à l'aide de la formule (2) une droite (une vraie) qui sépare une région clairement à dominante « - » d'une autre région clairement à dominante « + ». Mais il n'y a pas besoin d'être grand clerc pour comprendre qu'il y a mieux à faire : remplacer la forme linéaire habituelle par une forme un peu plus compliquée :

$$E = ax^2 + by \qquad (3)$$

Et voilà que, partant de l'idée d'optimiser une fonction linéaire, on introduit une fonction non-linéaire. Contentons-nous pour l'instant de cette idée et ne cherchons pas à établir ni quelle est la forme la mieux adaptée, ni comment optimiser les coefficients.

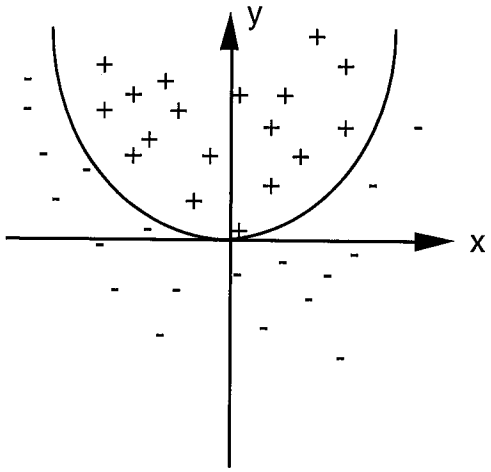


Figure 5

Il y a encore plus spectaculaire : examinons le diagramme de la figure 6. Un seul coup d'œil suffit pour se rendre compte que p=g=0. Si les vecteurs sont rigoureusement égaux, plus question d'un simple changement d'origine pour optimiser une forme linéaire : cela ne peut évidemment pas marcher !

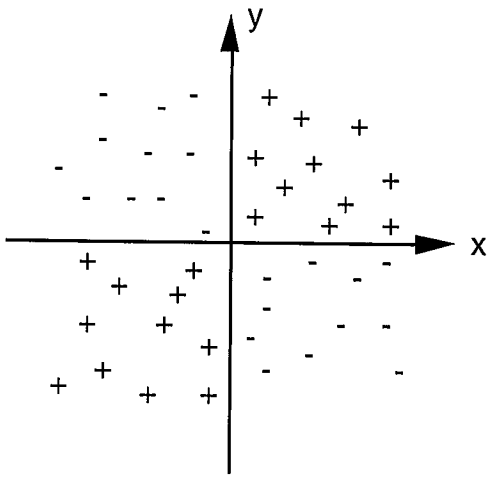


Figure 6

Là aussi, on voit que la bonne fonction d'évaluation n'est pas une fonction linéaire des critères individuels d'évaluation :

$$E = xy \qquad (4)$$

[14] ou vice versa.

Mais, se demande le lecteur critique, les fonctions non-linéaires ci-dessus présentées ne sont-elles pas pure spéculation ? Peu me chaut d'y prêter attention si jamais ne les rencontre dans des situations concrètes ! Et bien, rassure-toi, lecteur méfiant, on les rencontre bel et bien. Pas sous cette forme, évidemment, mais sous la dénomination générale de corrélation : ainsi, dans l'exemple ci-dessus, les caractéristiques x et y ne sont pas pertinentes en elles-mêmes, mais le triplet (x, y, position gagnante) est très fortement corrélé.

Du rêve à la réalité

Il est temps, après cette rapide[15] introduction, de se pencher sur le cas réaliste d'une vraie fonction d'évaluation comptant une touillée[16] de critères d'évaluation, et où l'on ne connaît ni la FED, ni des transcriptions de parties de Dieu. Qu'est-ce qui va changer, que doit-on garder, que doit-on jeter de ce qui a été dit plus haut ?

On doit ici distinguer deux cas : soit on désire, par souci de simplicité, conserver à la fonction d'évaluation une forme linéaire type équation (1) et simplement en optimiser les coefficients, soit on est prêt à introduire de la non-linéarité sur le modèle des équations (3) ou (4), et alors se pose le problème de trouver une méthode pour obtenir une bonne forme non-linéaire.

Envisageons tout d'abord le cas simple de la simple optimisation des coefficients d'une forme linéaire. Comme la fonction d'évaluation ne comporte pas deux mais n critères, les vecteurs « position gagnante moyenne » g et « position perdante moyenne » p (calculés, je le rappelle, pour un nombre N donné de cases jouées) sont maintenant des vecteurs à n dimensions. En appelant v_i les n_g positions gagnantes de la base de parties, l'expression de g est

$$g = \frac{1}{n_g}\sum_{i=1}^{n_g} v_i$$

p recevant une définition analogue. Supposons dans un premier temps que g et p ont le bon goût d'avoir des directions opposées. La fonction d'évaluation d'une position obtenue avec N cases jouées et caractérisée par un vecteur d'évaluation v généralise la relation (2)

$$E(v) = v^t g \quad (5)$$

où g est un vecteur colonne et v^t un vecteur ligne

$$g = \begin{pmatrix} x_g \\ y_g \\ z_g \\ \cdots \end{pmatrix} \qquad v^t = \begin{pmatrix} x & y & z & \cdots \end{pmatrix}$$

et leur produit scalaire est :

$$v^t g = xx_g + yy_g + zz_g + \cdots$$

Pour tenir compte du fait que g et p ne sont pas de directions opposées, il faut décaler l'origine d'un vecteur o tel que

$$g' = g + o = -(p + o) = -p'$$

En remplaçant g par g' dans l'équation (5), on a l'expression générale de l'évaluation linéaire optimisée :

$$E(v) = \tfrac{1}{2}(v^t g - v^t p) \qquad (6)$$

Notons en passant que l'on peut ainsi déterminer un degré de pertinence pour chacune des composantes de l'évaluation : c'est tout simplement le cosinus de l'angle que fait le vecteur g' avec le vecteur directeur de l'axe correspondant à cette composante. Ainsi, par un simple calcul d'angle, on peut voir le degré de pertinence de chacun des critères intervenant sous forme d'une combinaison linéaire dans l'évaluation. Il est clair que les critères correspondant à un angle nul sont strictement inutiles à l'évaluation sous forme linéaire.

Les évaluations non-linéaires ne peuvent être que meilleures, ne serait-ce que parce qu'elles incluent les formes linéaires. La démarche à suivre est dans son principe équivalente à celle exposée plus haut en commentant les figures 5 et 6. Deux problèmes se posent :

- décrire la courbe séparant la zone « + » de la zone « - » ;
- définir quelque chose comme une distance entre un point quelconque de l'espace d'évaluation et ladite courbe.

Définir la courbe ne semble pas compliqué. Toutefois, la dimension de l'espace d'évaluation peut être grand ; en outre, les limites des zones « + » et « - » sont floues, d'une part parce que les parties qui ont été utilisées pour les établir n'ont pas toutes (!) été jouées parfaitement et d'autre part parce que les critères individuels d'évaluation ne sont pas forcément exacts.

Tout ceci complique singulièrement le problème. Déjà que j'éprouve souvent beaucoup de difficulté à me représenter une surface dans un espace à trois dimensions[17], ne me demandez pas à quoi peut ressembler une hypersurface qui, projetée dans un sous-espace 3D donne une ellipsoïde de révolution, et dans un autre sous-espace 3D un cube tout ce qu'il y a de plus conservateur, lorsqu'on la projette dans un plan contenant 1D de chacun des espaces 3D susnommés[18]. Bref, il n'est plus question de dire comme plus haut: « visiblement, c'est une parabole ». Extrapoler une fonction « raisonnable » à partir d'un nombre fini de points est un jeu d'enfant pour le numéricien. Malheureusement, le flou dans la surface empêche de se livrer à ce genre de sport. Si donc l'on veut faire quelque chose de plus que

---

[15] mais ô combien passionnante (remarque signée d'un lecteur qui préfère rester anonyme).

[16] la touillée vaut exactement trois dix-septièmes de floppée.

[17] devrais-je l'avouer ? Parfois même deux !

[18] pour tout dire, je ne suis même pas sûr que ma question ait un sens…

l'évaluation linéaire, il faut transférer sur la puissance de calcul des machines une part importante du travail. Kai Fu Lee et Sanjoy Mahajan utilisent des techniques de reconnaissance de forme avec apprentissage. Deux étapes distinctes existent dans cette technique : une phase d'apprentissage (qui s'effectue une fois pour toutes), au cours de laquelle le programme « apprend » la forme de la surface discriminant les positions gagnantes des positions perdantes à partir d'une base de positions étiquetées « + » ou « - », et une phase d'évaluation (appelée par l'alpha-bêta), où le programme évalue une position en se référant aux informations extraites de la base de connaissances.

Reconnaissance de formes

La phase d'apprentissage consiste à extraire de la base de parties non seulement les vecteurs g et p, qui sont les vecteurs « position gagnante moyenne » et « position perdante moyenne », mais aussi des matrices de covariance G et P, définies par

$$G = \frac{1}{n_g}\sum_{i=1}^{n_g}(v_i - g)(v_i - g)^t$$

où v_i est l'un des n_g vecteurs d'évaluation de la base de parties correspondant à une position gagnante. La définition de P est bien sûr analogue. La matrice vv^t est définie par

$$vv^t = \begin{pmatrix} x^2 & xy & xz & \cdots \\ xy & y^2 & yz & \cdots \\ xz & yz & z^2 & \cdots \\ \cdots & \cdots & \cdots & \cdots \end{pmatrix}$$

en appelant x, y, z,... les composantes individuelles de v. Ces quatre quantités g, p, G et P déterminées, on obtient une fonction d'évaluation par :

$$2E(v) = (v-p)^t P^{-1}(v-p) - (v-g)^t G^{-1}(v-g) + |P| - |G| \qquad (7)$$

où G^{-1} et P^{-1} sont les matrices inverses de G et de P, |P| et |G| leur déterminant (que l'on peut omettre dans une fonction d'évaluation sauf à vouloir assurer un semblant de continuité à la fonction d'onde lorsque l'on change de paramètres g, p, G et P dans l'évolution de la partie).

On peut même déduire de la fonction E(v) ainsi calculée une probabilité de gain par la formule

$$P(v) = \frac{1}{1+\exp(-E(v))} \qquad (8)$$

Bon, voilà pour le gros de la théorie de la chose. L'article déjà cité vous donnera plus de détails et de précisions, des références de saines lectures sur le sujet.

C'est étudié pour !

C'est beau la théorie, surtout quand on y croit. Moi je suis assez genre St Thomas[19], il faut que je regarde dans le détail pour voir comment cela marche. Alors, je me suis donné une maquette stupide : un plateau de jeu compte deux cases, qui peuvent être soit vides, soit occupées par un pion pouvant être noir ou blanc. Une base de parties de ce jeu inconnu jouées par des experts très connus est représentée sur le tableau ci-dessous.

n°	+/-	case 1	case 2
1	+	Noir	Noir
2	+	Blanc	Blanc
3	+	Noir	Vide
4	+	Blanc	Vide
5	-	Noir	Blanc
6	-	Blanc	Noir
7	-	Vide	Noir
8	-	Vide	Blanc

Choisissons deux critères (idiots) pour construire une fonction d'évaluation. Par exemple, le critère x vaut -1 si la case 1 est blanche, 0 si elle est vide et 1 si elle est noire, le critère y est identique pour la case 2. On a alors

$$g = \frac{1}{4}\left(\begin{pmatrix}1\\1\end{pmatrix}+\begin{pmatrix}-1\\-1\end{pmatrix}+\begin{pmatrix}1\\0\end{pmatrix}+\begin{pmatrix}-1\\0\end{pmatrix}\right)$$

$$p = \frac{1}{4}\left(\begin{pmatrix}1\\-1\end{pmatrix}+\begin{pmatrix}-1\\1\end{pmatrix}+\begin{pmatrix}0\\1\end{pmatrix}+\begin{pmatrix}0\\-1\end{pmatrix}\right)$$

et on se trouve dans un cas où p=g, ce qui revient à dire qu'il est impossible de trouver une fonction d'évaluation linéaire (l'exemple a été choisi pour cela). Le calcul des produits vv^t donne les résultats suivants :

positions 1 et 2 : $vv^t = \begin{pmatrix}1&1\\1&1\end{pmatrix}$

positions 3 et 4 : $vv^t = \begin{pmatrix}1&0\\0&0\end{pmatrix}$

positions 5 et 6 : $vv^t = \begin{pmatrix}1&-1\\-1&1\end{pmatrix}$

positions 7 et 8 : $vv^t = \begin{pmatrix}0&0\\0&1\end{pmatrix}$

ce qui conduit à

$$G = \begin{pmatrix}1&\frac{1}{2}\\\frac{1}{2}&\frac{1}{2}\end{pmatrix} \qquad P = \begin{pmatrix}\frac{1}{2}&-\frac{1}{2}\\-\frac{1}{2}&1\end{pmatrix}$$

d'où en inversant

$$G^{-1} = \begin{pmatrix}2&-2\\-2&4\end{pmatrix} \qquad P^{-1} = \begin{pmatrix}4&2\\2&2\end{pmatrix}$$

[19] ne cherchez pas, il y a aussi des saints qui ne sont pas des joueurs d'Othello, après tout !

Reste maintenant à obtenir une fonction d'évaluation. L'équation (7) donne ici :

$$E = x^2 + 4xy - y^2$$

Ouf ! La phase d'apprentissage est terminée. Évidemment, ça n'est pas folichon comme calcul, mais il est clair que, sur un cas réel, cela se fait par ordinateur. Simplement ici, pour des raisons « pédagogiques », j'ai voulu montrer toutes les tripes du calcul.

Cette quantité E peut être prise comme fonction d'évaluation, ou on peut, pour faire plus joli, en déduire la probabilité de gain P(x,y) à l'aide de la relation (8).

Les résultats sont reportés dans le tableau ci-dessous où on a appliqué l'évaluation calculée aux huit positions figurant dans la base de connaissances :

n°	+/-	x	y	E(x,y)	P(x,y)
1	+	1	1	4	98%
2	+	-1	-1	4	98%
3	+	1	0	1	73%
4	+	-1	0	1	73%
5	-	1	-1	-4	2%
6	-	-1	1	-4	2%
7	-	0	1	-1	27%
8	-	0	-1	-1	27%

On constate que notre fonction d'évaluation, bien que construite de bric et de broc, juge très correctement de la qualité d'une position.

Reconnaissance de méforme

Je me suis amusé avec la formule (7) dans plusieurs autres cas, et je trouve qu'elle n'est pas exempte de critiques. Examinons le cas d'un autre jeu dont la fonction d'évaluation est aussi à deux dimensions et dont la base de parties d'experts est :

n°	+/-	x	y	E_l	E_{rf}	E_{pif}
1	+	1	1	3	4,8	5,0
2	+	0	1	2	1,8	0,7
3	+	1	0	1	1,1	2,2
4	+	-1	0	-1	-0,9	0,3
5	-	1	-1	-1	-5,1	-3,0
6	-	-1	1	1	-0,8	-1,0
7	-	-1	-1	-3	-3,5	-1,0
8	-	0	-1	-2	-4,5	-3,3

Les physionomistes auront peut-être reconnu le même jeu que précédemment, à ceci près que les positions 2 et 7 ont été échangées. L'application des formules (6) et (7) donne l'expression de la meilleure fonction d'évaluation linéaire et de la fonction non-linéaire trouvée par reconnaissance de forme :

$$E_l = x + 2y$$

$$E_{rf} = \frac{x^2}{5} + \frac{9xy}{5} - \frac{77y^2}{60} + x + \frac{19y}{6} - \frac{1}{12}$$

La comparaison de ces deux fonctions est présentée dans le tableau ci-dessus. Une fonction parfaite devrait classer en tête les quatre positions étiquetées gagnantes. En attribuant 11 points à la position la meilleure selon l'évaluation, 9 à la seconde, 7 à la troisième, etc jusqu'à -3 à la huitième position (et en partageant les points entre positions ayant la même cote), on devrait donner 32 points pour l'ensemble des quatre positions gagnantes et 0 pour les quatre positions perdantes. À cette échelle, l'évaluation linéaire obtient 28/32 et l'évaluation non-linéaire 30/32 : elle est donc un peu meilleure. Ce qui me chagrine, c'est qu'on peut faire mieux avec une fonction d'évaluation pifométrique :

$$E_{pif} = x + 2y + \frac{4x^2}{3} + 2xy - \frac{4y^2}{3}$$

(qu'on trouve très facilement graphiquement en remarquant que les deux droites d'équation $2y + x = 0$ et $2y - 4x - 3 = 0$ séparent le plan Oxy en deux zones, l'une gagnante et l'autre perdante). On voit que la fonction d'évaluation E_{pif} obtient un score de 32/32 avec le système de mesures ainsi déterminé.

Donc, il y a mieux à faire que la formule (7). J'ai essayé, en vain, de trouver mieux. Quelqu'un peut-il trouver mieux, ou éventuellement, expliquer pourquoi on ne trouve pas mieux ? J'ai une explication, mais elle n'est pas fondée[20]. Cela dépend peut-être du sens que l'on donne au mot « mieux », qui n'est peut-être pas toujours mieux au sens d'Aguillon. Par exemple, on peut noter que la différence entre cotes moyennes des positions « + » et des positions « - » ramenées à la somme des variances des cotes « + » et des cotes « - » est plus grande avec E_{rf} qu'avec E_{pif}. De toute façon, les méthodes graphiques avec mille parties et quatre dimensions, j'ai mieux[21] à faire...

Résultats obtenus

Kai Fu Lee et al. ont mis dans Bill 3.0 une fonction d'évaluation calculée par l'équation (7), sans préciser outre mesure la base de données qu'ils ont exploitée pendant la phase d'apprentissage. Ils clament que cela leur permet de gagner deux niveaux par rapport à la précédente version de Bill 2.0. Ils ont mesuré ceci en faisant jouer 200 parties Bill 3.0 contre Bill 2.0. Le premier en a gagné 139, et il y a eu 6 matches nuls, ce qui est un résultat typique de Bill 2.0 à 8 (ou 7) coups contre Bill 2.0 à 6 (ou 5) coups. Malheureusement, ils n'ont pas fait ou pas publié d'optimisation linéaire, et il serait intéressant de savoir ce qui dans ce gain est attribuable à la non-linéarité, et ce qui serait attribuable à un accord très précis tout au long de la partie des coefficients d'une fonction linéaire.

Bon, moi aussi, j'ai essayé. J'ai d'abord voulu tester l'optimisation linéaire, donnée par la relation (6), qui est censée être la meilleure forme linéaire, donc forcément meilleure que la forme linéaire que possède mon programme. J'ai utilisé la collection de parties dont je dispose, qui doit compter quelques centaines de parties

---

[20] et puis j'ai peur de me faire lapider en la donnant.

[21] dans *tous* les sens du terme.

jouées en compétition. J'ai calculé la fonction d'évaluation E(v) pour les coups 5, 10, 15, 20, ... jusqu'à 55 et interpolé les résultats pour les coups intermédiaires. Tout ceci semble très compliqué, mais c'est l'ordinateur qui fait tout le travail, et le temps d'écrire le programme permettant une telle analyse et le faire tourner ne prend guère qu'une demi-journée.

Les bonnes nouvelles : les fonctions d'évaluation ainsi trouvées ont une bonne tête, c'est-à-dire qu'elles évoluent doucement d'un coup à l'autre et que l'évolution des coefficients semble tout à fait conforme avec ce que j'avais cru comprendre du jeu d'Othello. Par exemple, le poids de la note attribuée à la mobilité, élevé en début et milieu de partie, tombe à zéro en fin de partie. Autre détail peut-être plus impressionnant : la mobilité est évaluée par mon programme en ne considérant que la frontière. Le coefficient qui lui est attribué est très différent en début de partie selon que c'est Noir ou Blanc qui a le trait. Cela se comprend par la très forte influence du trait sur la frontière en début de jeu : après son premier coup, Noir a 4 pions sur la frontière contre 1 pour Blanc ; Noir n'est pourtant pas encore en situation désespérée.

Les mauvaises nouvelles maintenant : le programme ainsi obtenu joue comme un guignol. Pourquoi ? Et bien, parce que mes parties sont jouées par de trop bons joueurs ! Lorsque par exemple on examine la situation au dixième coup, (pratiquement) aucune case X n'a été jouée. Donc le programme n'a aucune opinion sur la valeur des cases X et donc les jouera comme toute autre case. Autrement dit, cette méthode, qui n'est qu'une méthode d'espionnage, fournit une fonction d'évaluation qui est rigoureusement démunie de tout jugement lorsqu'elle fait face à des situations inédites. Je m'y connais encore moins en épidémiologie qu'en mathématiques, mais la situation me fait un peu penser à celle d'un organisme vivant, qui acquiert des anticorps au fur et à mesure qu'il tombe malade. Si donc l'on travaille avec une base de données trop savante, on risque de succomber à la première angine venue (même si par ailleurs on a d'excellents anticorps contre la Tamenorite aiguë et autres complications aussi graves).

Que faire face à cela ? Je n'ai que quelques idées, que je n'ai pas testées :

- sachant que la méthode est d'autant plus fiable qu'on est avancé dans la partie, on peut ne l'appliquer qu'à partir d'un taux de remplissage de l'Othellier suffisant. C'est la méthode de facilité, et c'est donc peut-être la bonne ;
- un projet plus ambitieux serait de compléter la base de données avec des parties débiles. Par exemple, en faisant jouer un progratnme contre lui-même, sauf que le coup 5 (par exemple) est joué aléatoirement ; cela permet de sortir des sentiers battus ;
- enfin, on peut toujours imaginer de mettre à profit l'analogie biologique pour faire un programme évolutif (au sens de Darwin, mais oui), qui joue d'abord totalement au hasard contre lui-même, qui extrait les « bonnes » informations de cette première base, qui joue ensuite un peu moins au hasard, réextrait de meilleures pondérations pour son évaluation, et qui joue de moins en moins au hasard, mais jamais de manière totalement « déterministe ». Cette part de hasard serait le reflet évolutionniste de la variabilité génétique, l'extraction des fonctions d'évaluation constituant l'acte de sélection sous l'effet de la compétition entre programmes qui simule la pression sélective de l'environnement.

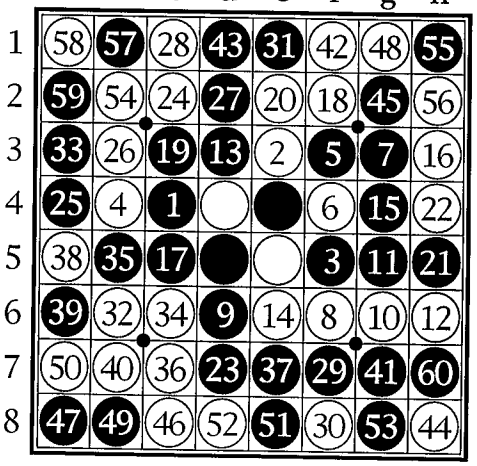
Conclusion

Deux idées ont été présentées dans ce petit texte : comment utiliser une base de données pour optimiser la forme linéaire d'une fonction d'évaluation et comment introduire une forme non-linéaire pour faire encore mieux. Ces idées sont attirantes, mais leur mise en pratique peut être plus délicate qu'il n'y paraît de prime abord.

On pourrait se dire qu'Othello n'est qu'une version à 64 cases des petits jeux que j'ai présentés comme modèle. Vu la qualité des résultats obtenus sur les petits jeux, pourquoi ne pas faire une bête fonction d'évaluation à 64 dimensions contenant chacune l'état d'occupation d'une case et confier à un programme le soin d'en optimiser automatiquement les coefficients ? Après tout, une telle fonction ne compterait que 2080 coefficients, et serait obtenue en ne manipulant que deux matrices 64x64, ce qui est accessible au moindre micro[22]. Oui, mais voilà, une fonction quadratique ne peut prendre en compte que les corrélations entre deux variables. Pour prendre en compte les corrélations entre trois variables, il faut des fonctions de degré 3 et, pour tout prendre en compte sur un othellier, des fonctions du soixante-quatrième degré, qui s'obtiennent en manipulant 64 tenseurs à 64 dimensions de 64 variables, ce qui fait... trop[23]. Autrement dit, ce qui est corrélation d'ordre élevé doit être pris en compte dans une seule dimension de la fonction d'évaluation, ou vous n'avez plus qu'à vous en remettre à l'efficacité de votre alpha-bêta.

Si l'un d'entre vous a le courage de se lancer dans ce genre d'aventures, j'aimerais bien qu'il nous en fasse connaître le fruit, même s'il est amer (ou à vers). Pour ma part, je finis d'élever mes enfants et j'arrive. À dans vingt ans...

Parties Internet 1997



Zebra 40-24 Bill

22 sauf le ZX81, toute notre jeunesse.

23 ma calculette travaille en notation polonaise, et elle m'a répondu « out of range », ce qui signifie en polonais « dette extérieure ».

Les tables de transpositions

par Nicolas Becquet

Introduction

Historiquement, les tables de transpositions ont d'abord été utilisées par les programmes d'échecs, car ce type de jeu conduit à parcourir de nombreuses fois certaines positions du fait des interversions de coups.

À Othello où le nombre de transpositions est négligeable, l'intérêt de ce type de table est qu'il permet de segmenter une même recherche en plusieurs parties distinctes tout en conservant les résultats intermédiaires. Par exemple, l'utilisation d'une zero-window, qui nécessite parfois la ré-exploration d'un sous-arbre est le cas typique de ce type de segmentation. La grille de tests de finales établie par Marc Tastet dans *Fforum 34* a été un réel challenge au cours duquel il a fallu innover, ce qui m'a conduit à déterminer un certain nombre de ces techniques de segmentation.

L'objet de cet article n'est pas de débattre de ces méthodes, mais de décrire le fonctionnement, devenu dans ce contexte capital, de cette mémoire cache jouée par les tables de transpositions dans l'exploration des finales à Othello.

La mise en œuvre de ces tables nécessite :

- de la mémoire ;
- une bonne détermination des informations à conserver pour chacun des nœuds parcourus ;
- une méthode d'adressage permettant de répartir correctement et efficacement ces informations au sein d'un espace mémoire réduit.

Description de l'adressage

L'idée est de construire deux tableaux (un par couleur) de même taille, dont chaque entrée repose sur une structure de données qui sera précisée plus loin. L'adressage de ces tableaux sera effectué au moyen d'une clé de Zobrist de 32 bits de longueur devant être réduite en taille au moment de l'adressage de façon à correspondre à l'espace mémoire alloué. En effet, supposons que nous n'ayons que 128 Ko de place disponible pour chaque table, nous appliquerons la transformation suivante de la clé d'adressage pour la ramener à un index de longueur de 19 bits (dans le cas de notre exemple) :

```
Index = (Clé adressage) MOD (Taille table)
```

Cette méthode présente l'avantage de permettre de gérer des tables dont la taille n'est pas nécessairement une puissance de 2, ce qui autorise une gestion mémoire plus souple.

Nous ne décrirons pas ici la méthode d'adressage dispersé de Zobrist qui a été présentée dans *Fforum 37*, mais j'aimerais toutefois faire deux remarques sur les risques de collisions que nous amène cette méthode.

- Le premier type de collision correspond à une erreur de hachage, i.e., deux positions différentes génèrent la même clé ; ce risque est d'autant plus grand que la longueur de la clé est réduite. Ce phénomène est en probabilité proportionnel au taux de remplissage des tables. Il est donc recommandé d'ajuster de quelque manière la taille de la table (donc de la clé) par rapport au remplissage probable. Ce remplissage en finale peut être estimé à partir du nombre de cases vides de la position, du temps de calcul maximum et de la vitesse de calcul. L'ajustement peut se faire soit en augmentant la taille de la table, soit en limitant son utilisation (par exemple jusqu'à une profondeur donnée). En particulier la probabilité de collision correspond au temps moyen pour parcourir la clé ; par conséquent avec une clé de 32 bits, ce temps est de 2^32 / vitesse de parcours. Autrement dit, avec un couple matériel / logiciel s'exécutant à la vitesse de 500000 nœuds par seconde, il faut 2^32 / 500000 secondes (environ 2 heures 20 minutes) pour parcourir la clé et donc risquer fortement d'avoir une erreur de hachage.

La façon de contourner ce problème, qui survient lorsque l'on cherche à calculer des finales profondes (entre 27 et 30 cases vides), consiste à agrandir la taille de la clé en associant à chaque nœud une signature correspondant à une autre clé de 32 bits, mais calculée à partir d'une autre semence aléatoire de façon à obtenir *in fine* deux clés sans corrélation entre elles. Cette méthode permet d'obtenir une clé de hachage de 64 bits qui, compte tenu des puissances de calcul actuelles rend complètement négligeable le risque lié à ce type de collision.

- Le second type de collision ne génère pas d'anomalie d'identification des positions, mais nuit à la vitesse de calcul ; il s'agit du conflit d'adressage qui se présente lorsque deux positions différentes doivent être stockées dans une même entrée de la table. En pareil cas, à défaut d'agrandir la taille des tables, il est nécessaire de définir un schéma de remplacement de façon à conserver l'entrée présentant le meilleur rapport qualité/prix.

Deux informations sont représentatives de l'effort consommé pour calculer une position :

1. la longueur de la branche principale, c'est-à-dire la profondeur qui sépare la position actuelle de l'horizon de la recherche, autrement dit le nombre de cases vides restant à parcourir pour la finale ; nous appellerons cette information *DEPTH* dans la suite de notre propos ;
2. le nombre de nœuds qui ont été parcourus pour évaluer cette position.

J'ai personnellement retenu la première information (*DEPTH*), d'une part parce qu'elle n'occupe que six bits de mémoire (la profondeur maximum d'un arbre de recherche à Othello est de 64), et d'autre part, parce que cette donnée réduit encore le risque que deux positions ayant la même clé se télescopent, car, en lecture, cette information devra coïncider avec le nombre de cases vides de la position en cours. Ensuite, il suffit, partant de l'index de la position, de parcourir les x positions suivantes, et de stocker l'entrée candidate à l'index présentant le meilleur rapport qualité/prix, c'est-à-dire l'entrée où *DEPTH* est minimum. J'ai enfin déterminé empiriquement, c'est-à-dire avec l'aide d'un chronomètre et en observant le nombre de nœuds parcourus, que le nombre idéal d'entrées à parcourir était 8...

Un défaut lié à cette méthode est qu'elle ne prend pas en compte l'âge des entrées ; en effet, les entrées les plus récentes doivent être conservées, car leur utilisation serait nécessaire en cas de nouveau parcours.

Description des informations devant être stockées

En premier lieu, il me semble important de bien insister sur le fait qu'à la fin de chaque nœud parcouru par l'algorithme de parcours alpha-bêta, nous avons trois cas de figures possibles :

1. le nœud traversé est sur la branche principale et le score retourné est exact ; nous qualifierons ce type de nœud de *TRUESCORE* ;
2. le score trouvé à l'issue de la recherche d'un nœud est inférieur à alpha (*UPPERBOUND*) ;
3. le score est supérieur ou égal à bêta (*LOWERBOUND*).

L'idée est donc en sortie de chaque nœud de stocker ces informations (avec naturellement le coup associé) de façon à pouvoir les réutiliser plus tard.

En effet, à l'entrée de chaque nœud, un parcours de la table est nécessaire et, si une entrée correspondant à la position est retrouvée, alors :

• dans le cas d'un *TRUESCORE*, la recherche du nœud correspondant et de ses descendants peut immédiatement être stoppée ;

• dans le cas d'une *UPPERBOUND* ou d'une *LOWERBOUND*, le coup correspondant doit être évalué en premier et naturellement, la fenêtre alpha-bêta peut être réajustée de la façon suivante :

- dans le cas d'une *UPPERBOUND*, si le score récupéré de la position est inférieur à bêta, alors bêta peut être réduit au score trouvé + 1 ;
- réciproquement, dans le cas d'une *LOWERBOUND*, c'est alpha qui peut être ramené à score - 1 ;

naturellement, la recherche pourra être stoppée si ces ajustements conduisent à ce que la condition alpha >= bêta soit remplie.

Par ailleurs, une fois le coup issu de la table exploré, deux options peuvent être prises :

1. dans la mesure où l'ordre de génération des coups est constant, la recherche peut redémarrer à partir du coup issu de la table de transpositions, ce qui permet d'éliminer une exploration inutile de la partie gauche de l'arbre dans le cas d'une retraversée liée à un parcours zero-window infructueux ;
2. l'autre méthode consiste, une fois les coups triés, à les explorer en ne prenant pas en compte le coup issu de la table.

Résultats et interprétation

Pour convaincre les sceptiques sur la mise en place de tables de transpositions, le tableau suivant fournit le nombre de Knœuds pour obtenir le score parfait des diagrammes 26, 33, 37, 38 (publiés dans le numéro 34 de *Fforum*) en fonction de la taille de la table (en Ko, sachant que le stockage de chaque position nécessite huit octets).

Diag.	0 Ko	512 Ko	1 Mo	16 Mo	32 Mo
26	20 077	10 134	10 077	9 993	9 993
27	5 735	2 921	2 908	2 904	2 904
33	26 632	11 548	11 385	11 281	11 281
37	71 190	34 593	34 061	33 164	33 132
38	273 381	110 231	106 291	96 246	95 913

Premières conclusions

• Ces tables réduisent d'au moins 50% le nombre de nœuds générés.

• Il existe un couple optimal taille de la table / nombre de nœuds à parcourir en terme de rapport qualité / prix.

- On observe que 512 Ko (ce qui correspond à 65 536 positions) d'espace alloué aux tables suffisent pour un nombre de nœuds voisin de 10 000 kN, ce qui correspond à des finales comportant environ 20 cases vides.
- Cette donnée peut, dans une certaine mesure être extrapolée, ce qui permet par exemple, de déterminer que pour le diagramme 38, la quantité de mémoire optimale est d'environ 6 Mo.
- Si on alloue une place supérieure à cet optimum, les gains sont marginaux (moins de 1%).

• Cependant, il faut bien prendre garde au fait que plus ce cache amène d'efficacité, plus il y a eu d'exploitation de résultats intermédiaires.

En pareil cas, la recherche a peut-être été trop segmentée, ce qui peut entraîner un surcoût que l'on peut optimiser...

Prenons un autre point d'observation en se focalisant sur le diagramme 38, évalué avec un Pentium MMX 200. La table est ci-dessous.

Conclusions

• tout d'abord, on vérifie que la taille optimale de la table sur cette position est bien de 6 Mo ;

• les gains en nœuds ne coïncident pas avec les gains en temps, ceci s'explique par le surcoût induit par la gestion de la table (qui est au pire de 10%) ;

• je ne m'étendrais pas sur les autres informations, car je pense qu'elles parlent d'elles-mêmes...

	0 Ko	512 Ko	1 Mo	6 Mo	16 Mo	32 Mo
Nombre de Knœuds	273 381	110 231	106 291	98 464	96 246	95 913
Temps de calcul	9'22"	4'07"	3'59"	3'33"	3'26"	3'23"
Gain Nœuds, Temps	S.O.	59%, 56%	61%, 57%	64%, 62%	65%, 63%	65%, 63%
Lectures réussies	S.O.	7%	8%	13%	18%	19%
Taux de collisions	S.O.	96%	94%	71%	32%	6%
Vitesse (N/s)	486 443	446 291	444 732	461 055	467 213	472 477

Description de l'algorithme

Nous allons examiner les fonctions de lecture et d'écriture dans la table, puis la façon dont la fonction de recherche alpha-bêta les appelle.

La table de transposition est utilisée pour six cases vides ou plus ; en effet, en dessous le rapport qualité/prix est nettement défavorable.

```
#define TrueScore 64
#define LowerBound 128
#define UpperBound 192
#define ReHash 8 // Nbre d'essais à faire pour trouver un emplacement
                    pour le stockage du nœud

typedef struct
{
   long  Sgn;    // La clé secondaire de hachage
   byte  Depth; // Le nombre de cases vides restant à parcourir
   byte  Infos; // Le type de nœud (TRUESCORE, LOWERBOUND,
                                      UPPERBOUND) + le coup joué
   short Score; // Le score de la position
} HashEntry;

void PutHashInfos(long Key, long Sgn, char Color, char Depth,
                  short Score, int Alpha, int Beta, char Move)
{
   register long  i; *Entry, *pBest;

   pBest = Entry = Color > 0 ? &HT1[Key % (NbEntries-ReHash)] :
                               &HT2[Key % (NbEntries-ReHash)] ;
   for (i = ReHash; Entry->Depth && i && Entry->Sgn != Sgn; i--)
   {
      Entry++;
      if (Entry->Depth < pBest->Depth || Entry->Sgn == Sgn)
         pBest = Entry;
   }
   pBest->Sgn   = Sgn;
   pBest->Depth = Depth;
   pBest->Infos = packmove[Move]; // Transformation d'un coup de
                                     l'intervalle 11..88 à 0..63
   pBest->Score = Score;
   if (Score < Alpha)
      pBest->Infos |= UpperBound;
   else if (Score >= Beta)
      pBest->Infos |= LowerBound;
   else
      pBest->Infos |= TrueScore;
} // PutHashInfos

int GetHashInfos(long Key, long Sgn, char Color, char Depth, int *Alpha,
                 int *Beta, char *Move)
{
   register long  i, *Entry;

   Entry= Color > 0 ? &HT1[NbEntries-ReHash] : &HT2[NbEntries-ReHash];
   for (i = ReHash; Entry->Infos && i; Entry++, i--)
      if (Entry->Depth == Depth && Entry->Sgn == Sgn)
      {
         *Move = unpackmove[Entry->Infos & 63];
            // Transformation d'un coup de l'intervalle 0..63 à 11..88
         switch (Entry->Infos & 192)
         {
            case TrueScore:
               *Alpha = Entry->Score;
               *Beta  = -infini;
               break;
            case UpperBound:
```

```
                if (Entry->Score < *Beta) *Beta = Entry->Score + 1;
                break;
             case LowerBound:
                if (Entry->Score > *Alpha) *Alpha = Entry->Score - 1;
                break;
          }
          return true;
       }
    return false;
 } // GetHashInfos

 int Search(POSITION P, char Color, int depth, int Alpha, int Beta,
            long NodeKey, long NodeSgn)
 {
    int      t, i, j, Cj, MaxMv, Nc;
    MoveList LM;
    RetList  LR;
    long     PosKey, PosSgn;

    Cj = 0;
    MaxMv = 0;
    t = -infini;
    if ((DEPTH >= 6) && GetHashInfos(NodeKey, NodeSgn, Color, depth,
                                     &Alpha, &Beta, (byte *)&Cj))
    {
       if (Alpha >= Beta) goto label_100;
       if (P.Board[Cj] == 0 && DoMove(&P, Cj, LR))
       {
          UpdateHashCode(NodeKey, NodeSgn, &PosKey, &PosSgn, Color LR);
                         // actualisation de la clé de hachage
          t= -SearchF(P, -Color, depth-1, -Beta, -Alpha, PosKey, PosSgn);
          UndoMove(&P, Cj, LR);
       }
       if (t > Alpha)
       {
          Alpha = t;
          MaxMv = Cj;
          if (Alpha >= Beta) goto label_99;
          UpdateBestLine(Cj);
       }
    }
    if ((Nc = SortMoves(LM)) == 0)
    {
       // Traitement joueur passe
       goto label_100;
    }
    if (Cj > 0)
       for (j = 0; j < Nc && LM[j] != (byte)Cj; j++);
       // À ordre des coups constant, on repart du coup de la table
    else
       j = -1; // Sinon, on part du premier
    for (i = j+1; i < Nc; i++)
    {
       // Boucle de parcours des coups d'un nœud...
    }
 label_99:
    if ((DEPTH >= 6) && (MaxMv != 0))
       PutHashInfos(NodeKey, NodeSgn, Color, depth, (short)Alpha,
                    Alpha, Beta, (byte)MaxMv);
 label_100:
    return Alpha;
 } // Search
```

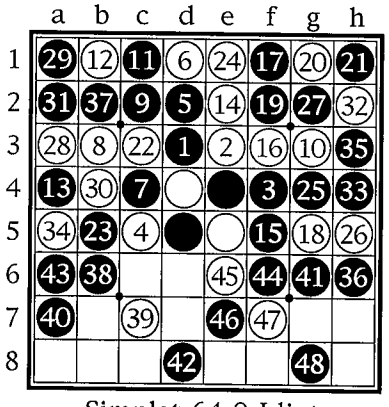
Évaluation aléatoire

par Stéphane Nicolet

Cet article explique scientifiquement pourquoi Othello est, par essence même, un jeu de hasard.

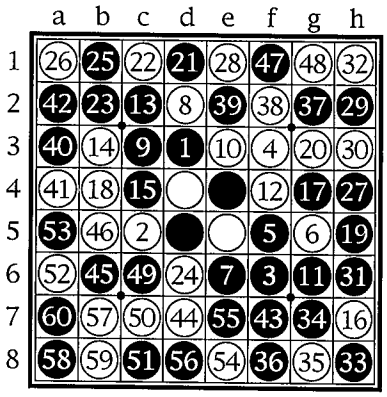
Imaginez deux programmes jouant à Othello qui, pour choisir leurs coups, utilisent deux stratégies idiotes : l'un (appelons-le Idiot) dresse la liste de tous ses coups jouables et élit un des coups au hasard ; l'autre (que nous nommerons Simplet) utilise tout l'attirail normal d'un programme classique, c'est-à-dire qu'il essaie, pour chacun de ses coups, d'anticiper toutes les réponses possibles de l'adversaire, puis toutes ses possibilités après ces réponses, et ainsi de suite jusqu'à une grande profondeur (il « voit » par exemple 4 ou 5 coups à l'avance), puisqu'il remonte les notes renvoyées par sa fonction d'évaluation suivant le principe du minimax. La seule subtilité est que sa fonction d'évaluation renvoie un nombre purement aléatoire, indépendant de la position, ne tenant donc aucun compte des critères reconnus comme importants à Othello : coins, bords, centre, frontière, parité, etc.

On pourrait s'attendre, intuitivement, à ce que les deux programmes aient des performances équivalentes, *e.g.* qu'ils fassent match nul sur un grand nombre de parties, ou tout au moins qu'ils fassent des parties aussi nulles l'un que l'autre. J'ai, bien sûr, essayé, et voici ce qui arriva !



Simplet 64-0 Idiot

Impressionant, n'est-ce pas ? Non seulement Simplet écrase Idiot 2-0, mais en plus il le fait avec style, lui retournant tous ses pions dans la première partie après un béton presque parfait, et « inventant » des séquences de grande classe dans la seconde (la séquence 27-31, qui échange deux coins pour éviter d'ouvrir, est particulièrement rusée).



Simplet 39-25 Idiot

Il semble paradoxal, à première vue, que des évaluations « aléatoires » de positions d'Othello puissent aboutir à autre chose qu'à un jeu aléatoire quand on les utilise dans une recherche minimax : le réflexe naturel consiste à dire que la recherche sur des nombres purement aléatoires ne peut entraîner qu'un choix aléatoire à la racine.

Comment expliquer le phénomène ?

Il faut bien distinguer, dans un algorithme de recherche dans un arbre de jeu, la partie « fonction d'évaluation » de la partie « processus de rétro-propagation des notes des feuilles ». Les propriétés d'une fonction d'évaluation ne se retrouvent pas forcément dans la note renvoyée par le processus de filtrage qu'est le minimax : la recherche « sélectionne » les évaluations des feuilles de l'arbre de jeu, en maximisant aux niveaux pairs et en minimisant aux niveaux impairs. Or le facteur de branchement d'un nœud influence la valeur aléatoire choisie par le minimax : plus un nœud a de fils, plus il y a de chances qu'on trouve une grande (resp. petite) valeur aléatoire parmi eux, donc plus, en moyenne, la note renvoyée à un niveau maximisant (resp. minimisant) sera grande (resp. faible)[1]. Donc le minimax aléatoire de Simplet sélectionne naturellement les coups qui donnent peu de coups à l'adversaire, et lui assurent beaucoup de coups jouables. Autrement dit, sans rien savoir d'Othello, il joue la mobilité !

On peut penser que les nombres aléatoires constituent un moyen inefficace et peu précis d'estimer la mobilité, par rapport à la méthode consistant à compter les coups légaux ou la frontière dans la fonction d'évaluation. C'est vrai, d'une certaine manière, mais il faut bien voir que ce minimax appliqué à des évaluations aléatoires fait des choses subtiles dont les effets sont difficilement quantifiables. Par exemple, il permet d'estimer la mobilité à toutes les profondeurs — le nombre de branches des positions près de la racine influe autant que le nombre de branches des positions plus profondes dans l'arbre ; au contraire, les programmes classiques ne prennent en compte que la mobilité à l'horizon de recherche.

Ceci nous amène d'ailleurs à envisager une façon amusante d'améliorer une fonction d'évaluation existante. Supposons que l'on dispose d'une fonction d'évaluation classique **E**, alors, d'après ce qui précède, on peut envisager que la fonction d'évaluation **E+u** (où **u** est un bruit blanc, c'est-à-dire une composante purement aléatoire suffisamment petite pour que **E** soit toujours dominante), utilisée à des profondeurs suffisantes, améliore le jeu du programme !

De l'art de créer de l'ordre à partir du chaos...

Référence : D.Deal et M. Smith, *Random Evaluation in Chess*, ICCA Journal **17-1**, p. 3-9, 1994.

[1] Pour les matheux de l'assistance : si $X_1,\ldots, X_n$ sont des variables indépendantes suivant une loi uniforme, alors $E(\max(X_1,\ldots,X_n))=E(X_1)*(n-1)/n$

Résoudre le jeu : des chiffres

par Emmanuel Lazard

L'article de Jean Delteil paru dans les échos d'Othello de *Fforum 47* m'a donné envie de faire une petite simulation. Souvenez-vous, il faisait état de 10^{30} nœuds à parcourir pour pouvoir résoudre complètement l'arbre de jeu. Je me suis amusé à faire jouer aléatoirement par un programme vingt millions de parties et à compter à chaque coup la mobilité dont dispose le joueur. Ceci en vu d'estimer le facteur de branchement moyen de l'arbre. Les résultats ont été les suivants.

• Une croissance régulière de la mobilité moyenne, avec un maximum de 12,02 aux coups 28 et 30, puis une décroissance régulière. La valeur maximum de la mobilité atteinte lors d'une partie a été de 27 coups légaux (aux coups 24, 27, 28, 31 et 32, bien sûr pas dans la même partie).

• La somme des produits partiels de cette mobilité moyenne me permet d'estimer le nombre total de nœuds dans l'arbre de jeu (c'est-à-dire le nombre total de position possibles à Othello, à partir de la position de départ) à environ 10^{53}. C'est ce chiffre qu'il faut comparer à celui des dames anglaises (estimé à 10^{40}) et à celui des échecs (de l'ordre de 10^{120}), même si le nombre total de positions possibles n'est pas un bon indicateur de la complexité d'un jeu. L'analyse théorique de l'algorithme alpha-bêta utilisé par la plupart des programmes montre qu'un ordonnancement optimal des coups à chaque niveau permet de ne parcourir qu'une fraction de l'arbre total, estimée à deux fois la racine carrée du nombre de nœuds. Ceci nous donnerait ici environ 10^{27} positions à parcourir. De plus, un certain nombre d'interversions peuvent encore abaisser ce nombre. Les chiffres donnés par Jean dans son article ne sont donc pas irréalistes. Bien sûr, il ne s'agit ici que de simulation et de calculs « théoriques », mais je pense que les ordres de grandeur sont respectés. Le même travail a été effectué sur les 43000 parties de la base : la mobilité moyenne est plus élevée qu'avec les parties aléatoires au début mais repasse en dessous vers le coup 25. Tous ces résultats sont illustrés dans les courbes ci-dessous.

Pour votre plaisir, j'ai inclus ici les parties aléatoires qui ont la mobilité cumulée (c'est-à-dire la somme, sur toute la partie, de la mobilité à chaque coup) la plus faible et la plus élevée.

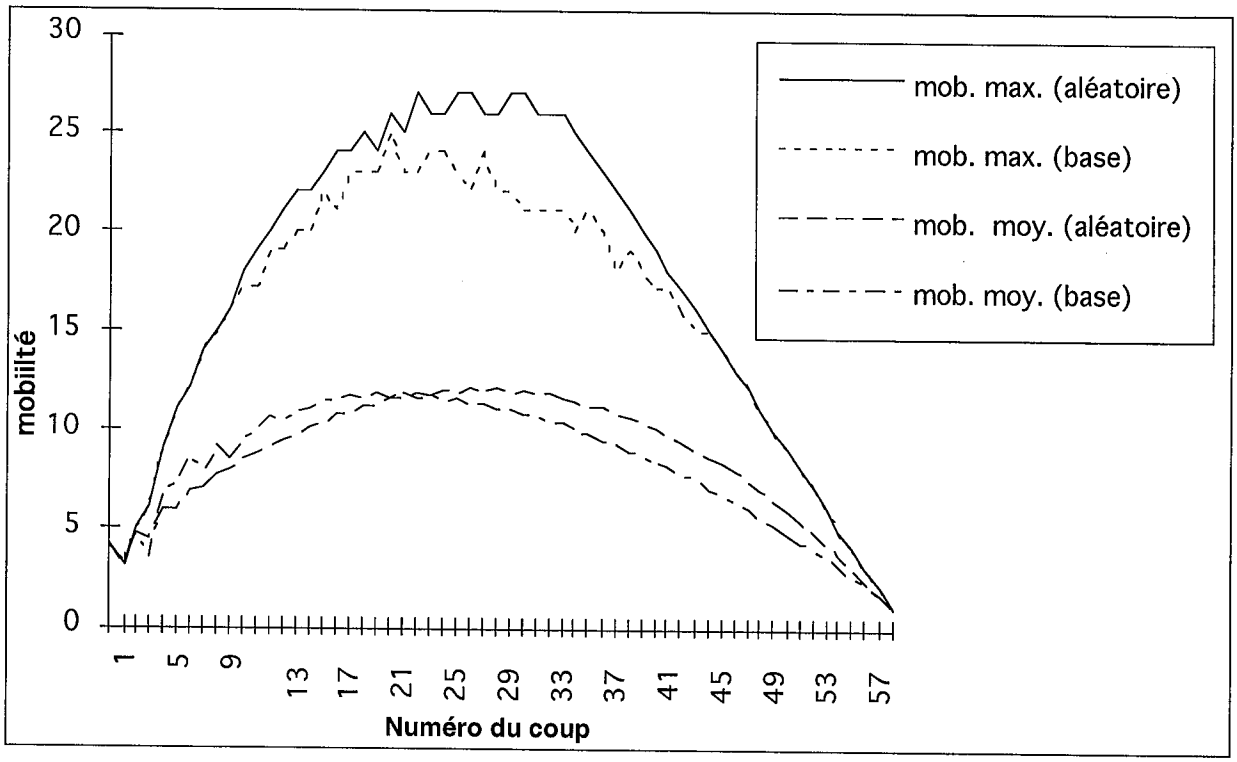
NDLR : David Haigh m'a depuis fait remarquer que chacune des 64 cases peut soit être vide, soit contenir un pion noir, soit un pion blanc. Cela fait environ 10^{30} positions possibles (et encore certaines sont clairement illégales).

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

mobilité cumulée : 255

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

mobilité cumulée : 685



Le programme Comp'oth

par François Aguillon

I. Vue générale

Comp'oth est articulé en cinq parties principales, qui se répartissent le travail ainsi :

• le « gestionnaire » accomplit les fonctions subalternes : représentation de l'othellier, saisie des réponses, gestion des interruptions qui sont nécessaires car le programme réfléchit sur le temps de son adversaire et se chronomètre afin de pouvoir gérer son temps de calcul (Puisque je ne parlerai pas plus de cette partie du programme, il me semble important de dire que le fait de disposer de toute facilité pour revenir en arrière dans la partie, poser un problème et afficher l'état d'avancement des différents calculs constitue le premier outil du programmeur pour déterminer et tester les parties plus nobles de son œuvre.) ;

• trois modules différents calculent la réponse du programme en début, milieu et fin de partie. Ce sont évidemment les plus intéressants ;

• le « superviseur », enfin, est le chef d'orchestre de la réflexion de l'ordinateur. C'est lui qui choisit un des trois modules précédents en fonction de l'othellier et de l'horloge. De plus, en milieu de partie, le superviseur détermine la profondeur de l'analyse à entreprendre.

Ces cinq modules ne sont évidemment pas de même importance. Celui qui détermine la physionomie du programme entier est le milieu de partie que nous allons détailler maintenant. Ensuite, nous parlerons des annexes que sont le début et la fin de partie.

II. Le milieu de partie

1) Généralités

La colonne vertébrale de cette partie du programme est bien évidemment un algorithme « minimax » avec des coupures « alpha-bêta », tel que l'a décrit J.-F. Puget dans un article précédent.

Le dilemme de l'adepte de ce genre d'algorithme est bien connu :

• soit on utilise une fonction d'évaluation très élaborée qui permet de bien juger une situation, au prix d'une lenteur interdisant une recherche très approfondie par l'algorithme « minimax » ; on obtient ainsi des programmes capables d'une bonne évaluation stratégique de la position, mais dont la faible profondeur limite la force tactique ; la limite absolue de cette philosophie aboutit à l'intelligence artificielle, mettant souvent en jeu une somme considérable de connaissances ;

• soit on choisit une fonction d'évaluation très simple, voire simpliste, mais rapide, ce qui permet au minimax de développer une recherche assez profonde ; cette philosophie de la « force brute » conduit à des programmes d'une grande force tactique mais d'une faiblesse stratégique dangereuse.

La philosophie de Comp'oth est celle de la force brute. Pourquoi ? La raison en est très simple : pour écrire une fonction d'évaluation sophistiquée et fiable, il faut être un bon joueur d'Othello, ce que je ne suis pas ! Par contre, la deuxième option fait beaucoup appel à des qualités d'informaticien que je pense posséder (il faut bien que j'aie quelque chose de bon !). Le fait que Comp'oth ne soit pas mauvais doit être dû au fait qu'Othello est un jeu où la tactique est très importante, comme aux échecs, à l'inverse du Go où la stratégie prend le pas.

2) La fonction d'évaluation

La fonction d'évaluation de Comp'oth se doit d'être sommaire. En fait, pour coter une position, le programme ne dispose que de deux informations : une note de « mobilité » et une note de « bords ». Elles reflètent les deux principaux points que surveille le joueur d'Othello lambda : conserver des libertés et ne pas donner de coin trop facilement sans compensation.

2.1) La mobilité

Je pense que beaucoup de programmes d'Othello évaluent la mobilité en comptant le nombre de coups jouables par chacun des deux joueurs à partir d'une position donnée. Cette méthode est très lente pour un résultat pas formidable. Pour évaluer la mobilité, Comp'oth se contente de faire une analyse de la frontière du jeu (moins il en a, mieux il se porte), ce qui est autrement plus rapide et plus proche de ce que font les joueurs humains.

Le décompte des mobilités, ou plutôt de la frontière, est pondéré par une fonction non linéaire, ce qui rend compte du fait suivant : il y a une énorme différence entre avoir peu et très peu de mobilité, alors qu'il est presque indifférent d'en avoir beaucoup ou énormément.

2.2) Les bords

La notation des bords est autrement plus délicate. Pour ne prendre que deux exemples classiques, cette notation est censée apporter une réponse par oui ou non à des questions aussi simples que « dois-je prendre le bord de cinq maintenant, ou plus tard ? » ou « y a-t-il une possibilité immédiate ou future de déclenchement d'un piège de Stoner ? »

En fait, il y a même une question triviale qui n'a pas de réponse simple : qu'est-ce qu'un bord, en tant qu'entité que l'on peut isoler du reste de l'othellier ? Au minimum, il y a les huit cases qui constituent le bord au sens propre. Mais une foule de cases influencent le jugement que l'on peut avoir d'un bord : la ligne qui lui est parallèle, les lignes perpendiculaires issues des cases C (pour les bords déséquilibrés en particulier), les grandes diagonales, pour ne citer que les plus évidentes. Or cela représente déjà 38 des 64 cases de l'othellier. Une fonction d'évaluation simple ne peut pas se permettre un tel luxe de... « détails ».

Je n'ai donc retenu comme définition d'un bord que dix cases : les huit cases du bord proprement dit et les deux cases X attenantes. Toutes les formes possibles de bord sont évaluées dans un tableau, et non calculées au cours de l'exploration, ce qui permet de gagner du temps. C'est également plus précis, car le calcul du tableau, fait une fois pour toutes, n'a pas besoin d'être rapide.

Dix cases avec trois possibilités par cases (blanc, noir, vide), cela fait $3^{10} = 59049$ bords différents à coter. Ce chiffre est assez important et explique deux choses : sur les 256 Koctets nécessaires à Comp'oth, un peu plus de 115 sont occupés par ce tableau ; d'autre part, il est impossible de tous les coter à la main sans faire d'erreur.

C'est pourquoi ces cotes sont calculées par le programme en début de partie pendant la phase d'initialisation. Ce calcul prend environ 45 secondes. Il est intéressant de constater qu'en introduisant deux cases de plus dans la définition d'un bord, il faudrait disposer de 519 Koctets pour le tableau et de sept minutes pour le calculer.

La cotation des bords est établie par une récurrence descendante. Les bords sans cases vides sont évalués en comptant les pions de chaque couleur.

Ensuite un algorithme permet de coter tout bord à (n-1) cases occupées à partir des bords à n cases occupées de la façon suivante : à partir d'un bord de (n-l) cases, les (11-n) bords à n cases obtenus en jouant un pion sont générés. Les probabilités de pouvoir jouer ces pions ne sont pas égales. La probabilité d'accès vaut 1 lorsque le pion est jouable uniquement en considérant le bord, et varie jusqu'à une valeur très faible lorsqu'il s'agit d'un coin sur un bord complètement vide. La note du bord à (n-1) cases est alors la moyenne des notes des bords à n cases pondérée par ces probabilités d'accès, ceci pour chacun des deux joueurs (celui qui a le trait et l'autre). Si la note pour le joueur au trait est meilleure que celle de son adversaire (c'est-à-dire s'il est intéressant d'y jouer), la note définitive du bord est la moyenne des deux notes avec une pondération favorable au joueur qui a le trait. Malheureusement, beaucoup de situations fréquemment rencontrées sur l'othellier ne sont pas dans ce cas (aucun des deux joueurs n'a intérêt à jouer sur le bord). Par exemple pour un bord équilibré de quatre pions d'une seule couleur, seuls les coins présentent de l'intérêt — et encore — mais leur probabilité d'accès est faible. Il s'ensuit que la situation est bloquée. Dans une partie réelle, les deux joueurs ne se précipiteront pas pour jouer sur le bord, la situation aux alentours va donc changer et les joueurs auront accès à toutes les cases. On simule ceci en recalculant les notes avec des probabilités d'accès supérieures en espérant que tout ira mieux. Certaines situations sont inextricables, par exemple celle décrite figure 1 n'évoluera jamais dans le bon sens : le premier qui jouera perdra le bord sauf dans le cas improbable où Blanc peut prendre le coin sans que Noir puisse s'insérer. Dans ces cas-là, le jugement de Salomon s'impose, la note définitive est la moyenne des deux notes partielles.

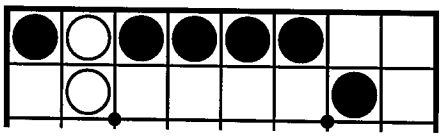


figure 1

Il est paradoxal de constater que le résultat ultime de ce calcul consiste à coter le bord vide. Pas de révélations sensationnelles, le programme trouve qu'un bord totalement vide n'avantage ni Blanc, ni Noir.

2.3) Équilibre mobilité/bord

S'il suffisait de prendre les bords pour gagner, Othello serait un jeu stupide. Le point central d'Othello est peut-être la contradiction entre l'intérêt tactique de prendre les bords, c'est-à-dire bétonner (pour avoir une bonne mobilité) et l'intérêt stratégique de ne pas les prendre ! Comp'oth retrouve cette contradiction de manière simple en constatant que jouer un bord améliore souvent sa note de mobilité au détriment de sa note de bord.

Attacher trop d'importance à la mobilité conduit Comp'oth à bétonner outrancièrement, l'inverse conduit à se faire battre par le premier « bétonneur » venu. L'équilibre entre les deux donne un programme qui opte assez tôt pour l'une ou l'autre des options, en fonction des réponses de l'adversaire, sans hésitation entre les deux (ce qui peut être catastrophique).

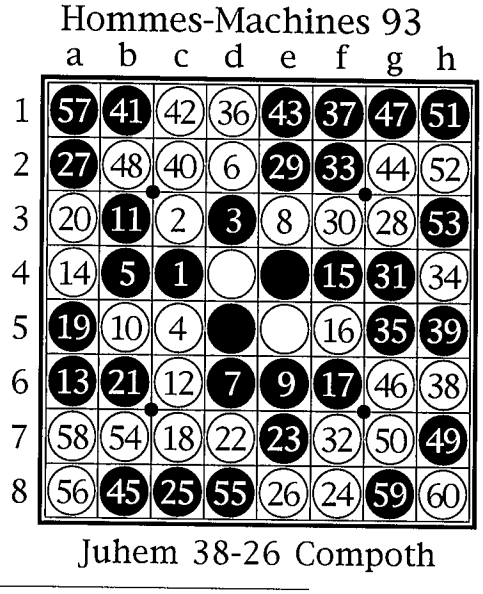
III. Le début de partie

La bibliothèque d'ouverture comprend actuellement[1] 2500 coups. Elle a été établie de manière très simple en faisant tourner le programme de milieu de partie à une profondeur de douze coups. Je dispose à peu près de 150 lignes allant au moins jusqu'au vingtième coup. Cela représente une centaine de jours de calcul pour la machine (deux ventilateurs en assurent en permanence le refroidissement).

IV. La fin de partie

Comp'oth lance sa recherche finale jusqu'à 17 coups avant la fin. L'algorithme utilisé est l'alpha-bêta, je ne m'étendrai pas sur la fonction d'évaluation...

Cette partie est vraiment placée sous la haute surveillance du superviseur et souvent la recherche finale est interrompue faute de temps. C'est pourquoi, grâce à une fenêtre (c'est-à-dire les bornes alpha et bêta initiales) réduite à sa plus faible largeur, Comp'oth recherche très rapidement un coup gagnant, même s'il est d'une débilité à faire pleurer ! Ensuite le superviseur regarde si le meilleur coup a des chances d'être trouvé en un temps raisonnable. Si c'est le cas, la recherche finale continue jusqu'à l'obtention du meilleur coup, du moins dans un temps limité. Cette gestion du temps, assez compliquée, n'a jamais été prise en défaut. Le record à battre est de 4mn 59s de réflexion pour Comp'oth, avec un temps alloué de 5mn. Enfin, si Comp'oth ne trouve pas de coup gagnant (horreur !), il choisira (sauf à l'extrême fin de partie) de faire calculer sa réponse par l'algorithme de milieu de partie, car il arrive souvent que le moins mauvais coup dans l'absolu donne à son adversaire une suite gagnante beaucoup plus facile à trouver qu'un coup joué par le milieu de partie.



[1] À la date où l'auteur écrit ces lignes bien sûr (1987).

Iago

par François Aguillon

Je voudrais, dans cet article, me livrer à la présentation critique d'un programme d'Othello de haut niveau nommé Iago. Les quelques remarques que je serai amené à développer ne sont pas à prendre pour parole d'évangile mais comme le simple avis de quelqu'un qui s'est frotté aux mêmes problèmes — et qui a d'ailleurs souvent adopté les mêmes solutions.

Iago a été écrit par Paul Rosenbloom, du *department of computer science* de l'université Carnegie-Mellon de Pittsburgh (USA), dans le cadre d'une coopération avec la défense américaine, tout ce qui porte le label « intelligence artificielle » intéressant de très près les militaires. Le côté professionnel du programmeur apparaît d'ailleurs clairement à travers la présentation du programme[1], l'auteur essayant d'apporter à chaque fois qu'il le peut des preuves à ses assertions, et n'hésitant pas à passer énormément de temps à se donner des outils de diagnostics et d'essais sur son programme.

Comme l'écrasante majorité des programmes d'Othello existants, Iago utilise un algorithme de type alpha-bêta couplé à une fonction d'évaluation et géré par un « superviseur » qui détermine selon l'état de l'othellier la valeur des paramètres intervenant dans la fonction d'évaluation et le temps que le programme peut s'accorder pour chaque coup. La description du programme se scinde donc naturellement en description de la fonction d'évaluation, de l'algorithme alpha-bêta et du superviseur.

I. Fonction d'évaluation

La fonction d'évaluation est calculée à partir de quatre facteurs :

- B : rend compte de la structure des bords ;
- D : rend compte des pions définitifs ;
- M : rend compte de la mobilité ;
- F : rend compte de la frontière.

Ces facteurs sont pondérés par quatre coefficients b, d, m et f, de telle sorte que l'évaluation d'une position s'écrit :

$$Eval = bB + dD + mM + fF.$$

La valeur des coefficients dépend, pour certains d'entre eux, du numéro n du coup auquel Iago est en train d'effectuer ces calculs. Ainsi :

- $b = 312 + 6{,}24n$;
- $d = 36$;
- $m = 50 + 2n$ si $n \leq 25$,
  $= 75 + n$ si $n > 25$;
- $f = 99$.

Restent à décrire la manière de calculer les quatre facteurs B, D, M et F.

1) Structure des bords

La structure des bords est analysée d'une manière analogue à celle déjà décrite dans l'article sur le programme Comp'oth (cf. article précédent). Iago calcule une fois pour toutes un tableau dans lequel sont évaluées les positions de tous les bords possibles. Comme un bord comprend huit cases, que chaque case peut se trouver dans trois états différents, ce tableau comporte $3^8 = 6561$ entrées.

Pour calculer ce tableau, Iago commence par attribuer une note statique pour toutes les structures de bords à huit pions : comme ces pions sont des pions définitifs si le bord entier est occupé, il suffit, pour obtenir la note statique d'un bord, de compter les pions de chaque couleur ! Les bords à 7 pions sont ensuite envisagés tout simplement en regardant vers quels bords de 8 pions ils évoluent. Ensuite, on évalue les bords à 6 pions en regardant vers quels bords à 7 pions ils évoluent, puis les bords à 5, etc.

Quand on regarde les évolutions possibles d'un bord à n pions, on obtient 8-n bords différents à n+1 pions. Chacun de ces 8-n bords n'a pas la même probabilité, au cours d'une partie, d'être obtenu légalement. Par exemple, si un bord est vide, la probabilité que Noir puisse s'installer dans un coin est très faible. La note attribuée à un bord est donc la moyenne des notes de ses « fils » pondérée par leur probabilité d'accès. Cette probabilité vaut 1 si le pion posé retourne des pions à l'intérieur du bord lui-même ; dans le cas contraire, elle est nulle pour une case de coin et dépend de l'état des voisins sur le bord pour les six autres cases. Enfin, on laisse toujours au joueur la possibilité de ne pas jouer sur le bord, et ce avec une probabilité de 1.

P. Rosenbloom critique lui-même son tableau de stabilité des bords, en lui reprochant de donner à son programme un penchant certain pour prendre les bords. Je serais assez tenté d'attribuer cela à une gestion par trop simpliste des bords quasi-stables, c'est-à-dire des bords où aucun des deux joueurs n'a intérêt à jouer le premier (exemple typique, un bord équilibré de quatre pions de la même couleur). En effet, la probabilité de ne pas jouer sur ce bord ne restera pas égale à 1 durant toute la partie : un des deux joueurs sera bien obligé d'y jouer le premier ! Le problème est d'autant plus important que de tels bords quasi-stables sont des points de passage pratiquement inévitables dans une partie d'Othello.

La faiblesse habituelle de cette façon de coter les bords réside dans le fait qu'elle considère le bord comme une entité en soi et néglige l'interaction du bord avec l'othellier en général et les autres bords en particulier. Or ces interactions tendent à devenir plus importantes au fur et à mesure que les bords se remplissent, pour devenir capitales en fin de partie. Iago prend en compte ce couplage en focalisant son attention sur les cases X.

---

[1] Paul Rosenbloom, « A World-Championship-Level Othello Program », *Artificial Intelligence* 19 (1982) 279-320.

Ainsi, avant d'envisager de jouer une case X, Iago évalue d'une part la probabilité que cette case X donne l'accès au coin et d'autre part le coût du sacrifice du coin. Ce coût est évalué à l'aide d'une sorte de petit alpha-bêta « local » à deux coups de profondeur.

Ce traitement particulier des couplages entre coins semble donner de bons résultats si l'on en croit P. Rosenbloom. Toutefois, il oublie tous les sacrifices de coin qui ne se font pas à l'aide d'une case X, mais d'une case C d'un bord adjacent. De plus, je doute qu'il puisse amener à de bonnes évaluations dans le cas de pièges de Stoner ; mais après tout, c'est aussi le rôle de l'algorithme alpha-bêta de détecter de telles combinaisons !

2) Pions définitifs

Cette partie de la fonction d'évaluation compte les pions définitifs, c'est-à-dire les pions qui ne peuvent plus être retournés. Malheureusement, les pions définitifs ne sont pas faciles à reconnaître : seuls sont reconnus par Iago comme tels les coins et les pions comptant un voisin définitif de leur couleur dans leur ligne, leur colonne et leurs deux diagonales.

Bien que ne reconnaissant pas tous les pions définitifs, l'algorithme de Iago n'en consomme pas moins beaucoup de temps. C'est pourquoi ils ne sont pas évalués dans les nœuds terminaux de la recherche, mais dans les nœuds situés juste au-dessus. Ceci permet d'économiser un facteur 4 sur le temps consacré à cette partie de l'évaluation, au prix d'une perte de précision.

De toute façon, de l'aveu même de P. Rosenbloom, cette stabilité interne n'influe pas beaucoup le comportement de Iago. En effet, la partie est généralement gagnée (ou perdue !) depuis longtemps lorsque la balance des pions définitifs commence seulement à pencher d'un côté ou de l'autre.

3) Mobilité

La mobilité est évaluée en tenant compte de l'accessibilité de chaque case libre de l'othellier pour chacun des deux joueurs. On obtient pour le joueur (respectivement pour son adversaire) une note j (respectivement a) en additionnant le double du nombre de cases auquel il a accès seul, et le nombre de cases auquel les deux joueurs ont accès. L'évaluation M de la mobilité se fait alors de manière non-linéaire :

$$M = 1000(j - a)(j + a + 2)$$

Je ne vais pas résister à la tentation de critiquer vertement ce calcul de mobilité ! En effet, il présente deux graves inconvénients : il est très pénalisant en temps (Iago passe 85% de son temps consacré à l'évaluation pour ce seul calcul), et il attribue la même valeur aux mobilités acceptables et à celles qui ne le sont pas.

Jean-François Puget a décrit la méthode employée par A. Kierulf dans son programme Brand pour contourner le premier inconvénient : il suffit, comme le fait d'ailleurs Iago, d'évaluer la mobilité à un coup de moins que le reste de la fonction d'évaluation.

Mais je m'interroge encore sur la validité d'une telle méthode : de même que l'évaluation d'une position résulte d'un savant équilibre entre avantages et inconvénients, le choix d'un bon coup est le fruit d'un compromis entre forces et faiblesses de chaque position. Or, si en un coup on retourne tout un pan de la frontière — sans s'en rendre compte par l'analyse de la mobilité qui ne sera mise à jour qu'au niveau suivant, à condition qu'il y en ait un — et que l'on ne compte pour ce coup que les avantages qu'il apporte par ailleurs, on commet une erreur grossière qui risque de n'être que peu « amortie » par la profondeur de la recherche. La méthode qui consiste à évaluer à un coup de moins ne devrait s'appliquer, à mon sens, qu'aux paramètres qui ne sont pas susceptibles de grosses variations en un coup, ce qui n'est pas le cas de la mobilité ! Allez, on se calme.

4) Frontière

La note de frontière est attribuée à partir :

- du nombre de pions adverses adjacents à une case vide ;
- du nombre de cases vides adjacentes à un pion adverse ;
- de la somme des nombres de cases vides adjacentes à chacun des pions de l'adversaire.

Bien que semblant redondantes à première vue, ces quantités sont très pertinentes pour rendre compte simplement de la mobilité potentielle d'une position. Iago estime (mais on ne sait pas comment exactement), pour le joueur et pour son adversaire, deux notes j' et a' à partir desquelles il établit la note de frontière par la relation

$$F = 1000(j' - a')(j' + a' + 2)$$

Cette note de frontière est beaucoup plus vite calculée que la mobilité, car elle peut être calculée de manière incrémentale à chaque profondeur et donc ne nécessite pas un examen complet de l'othellier à chaque coup.

5) Conclusion

Voilà, j'espère bien détaillée, la fonction d'évaluation de Iago. L'expert à Othello sera sans doute surpris d'y trouver si peu de choses, mais ça marche (par exemple, Iago s'est permis de battre en tournoi un prototype du Reversi Challenger sur le score de 60 à 3)[2] ! Il est évident que cette fonction d'évaluation ne concerne que le milieu de partie, la fonction d'évaluation en fin de jeu consistant en un simple décompte de pions.

Il faut maintenant regarder les parties plus typiquement informatiques de Iago, à savoir l'exploration de l'arbre alpha-bêta et la gestion du jeu lui-même.

II. L'algorithme alpha-bêta

Le but de ce paragraphe n'est pas d'expliquer une fois de plus ce qu'est l'algorithme alpha-bêta. Le lecteur novice est renvoyé aux articles précédents de ce numéro

[2] (NDLR) N'oublions pas que cet article date de juin 88 et que l'article de Rosenbloom est encore plus vieux.

spécial. Par contre, il s'agira de décrire les raffinements que P. Rosenbloom a apportés à cet algorithme et, de plus, à en mesurer l'efficacité.

Un petit rafraîchissement de mémoire sur les méfaits de la croissance exponentielle ne peut faire de mal à personne. Iago effectue sa recherche à une profondeur typique de six coups. La divergence moyenne du jeu a été évalué par P. Rosenbloom à $9,9$. Une exploration brutale de l'arbre (type minimax) conduirait donc à évaluer $9,9^6 = 941480$ nœuds terminaux. L'alpha-bêta le plus « bête » fait tomber la divergence effective à $5,6$. À 6 coups, on n'évalue alors plus que $5,6^6 = 30841$ nœuds terminaux, soit trente fois moins qu'avec le minimax. Avec l'alpha-bêta un peu plus évolué utilisé par Iago, la divergence effective tombe à $4,1$; le nombre de nœuds terminaux est alors ramené à $4,1^6 = 4750$, soit $6,5$ fois moins que l'alpha-bêta brut. Enfin, un alpha-bêta idéal ferait tomber la divergence à $3,7$ pour gagner encore presque un facteur 2 sur la vitesse. Programmeurs, investissez dans la matière grise avant d'investir dans la dernière bécane super-rapide qui sera démodée avant l'automne (je ne suis pas vraiment bien placé pour donner de tels conseils !).

Mais j'en vois qui, par l'odeur alléchés, se demandent quels sont ces trucs qui améliorent tant l'efficacité des algorithmes. La réponse tient en deux points : mise en mémoire d'une partie de l'arbre et du tableau des coups meurtriers.

Iago explore de manière incrémentale l'arbre de recherche : il cherche d'abord à un coup de profondeur, puis deux, puis trois et ainsi de suite jusqu'à six ou sept. Mais au lieu d'oublier à chaque fois le résultat de la recherche précédente, il conserve les informations obtenues. Malheureusement, du fait de problèmes de taille de mémoire, il ne peut garder toutes ces informations. Il conserve donc celles qui offrent le meilleur « rapport qualité/prix », c'est-à-dire les trois premiers niveaux de l'arbre. Au prix d'un tout petit peu de mémoire (3 niveaux représentent 1000 nœuds terminaux seulement), il améliore ainsi sa vitesse d'un facteur plus grand que deux.

Ensuite, pour les niveaux de l'arbre supérieurs à 3, Iago établit dynamiquement une table de coups meurtriers pour ordonner sa recherche. Cette table contient des informations du type : « quand Noir vient de jouer a3, alors Blanc peut jouer les cases b2, c7, d5, etc. Parmi celles-ci, c7 est la meilleure, mais d5 n'est pas mauvais, b2 est nul... »

Cette table est à la fois mise à jour au fur et à mesure de la recherche et utilisée pour l'ordonner. Évidemment, les informations contenues dans cette table ne sont pas complètement fiables mais statistiquement, elles doivent se révéler valables puisque l'utilisation de la table permet un gain de temps d'un facteur plus grand que 3.

III. Le superviseur

C'est la partie du programme qui joue le rôle du chef d'orchestre. Tout d'abord, il décide a priori du temps qu'il peut consacrer à sa recherche. Comme beaucoup de programmes. Iago joue vite en début de partie pour pouvoir consacrer plus de temps vers la fin de partie. Je ne suis pas très sûr que ce soit là une bonne façon de faire, tant il est vrai qu'un handicap même minime en début de jeu ne va souvent qu'en s'aggravant.

Une fois que son temps limite est fixé, Iago fait une recherche de type milieu de partie à un coup, en chronométrant le temps qu'il met pour ce faire. Ainsi, sachant qu'il mettra 4 fois plus de temps pour aller à un coup supplémentaire, il sait s'il peut lancer une recherche à deux coups ou non. En procédant de la même façon plusieurs fois de suite, il va ainsi typiquement à une profondeur de 6 coups, mais est capable seul de pousser à 7 ou de limiter à 5 sa recherche.

Le problème se complique légèrement en fin de partie, car Iago doit alors choisir entre trois fonctions d'évaluations :

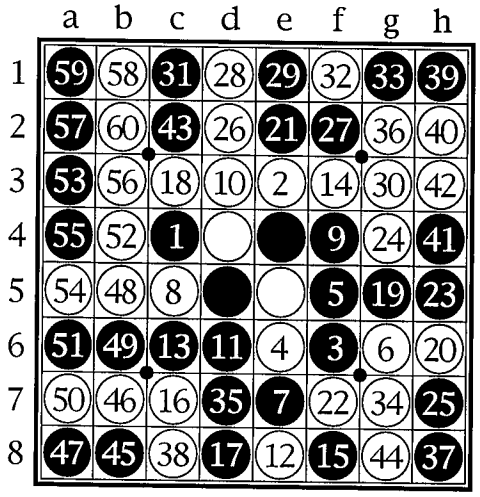
- celle de milieu de partie, décrite à la section I ;
- celle de fin de partie, qui trouve le meilleur coup ;
- celle de fin de partie rapide, qui trouve le premier coup gagnant s'il existe.

P. Rosenbloom a établi que la deuxième fonction d'évaluation est 250 fois plus rapide que la première et 16 fois plus lente que la troisième. Iago est donc capable de prédire à quel moment il peut lancer sa recherche finale.

Ces techniques de gestion du temps, simples à mettre en œuvre, sont trop rarement appliquées par les programmeurs. Pourtant, en condition de tournoi, c'est-à-dire en jouant à l'horloge, les fluctuations des temps de calcul d'une partie à l'autre à profondeur égale sont telles que les programmes non gérés en temps sont obligés de jouer beaucoup trop vite dans la majorité des parties s'ils veulent être sûrs de ne pas perdre au temps.

En conclusion, je ne saurais que trop recommander à ceux qui programment Othello, et aussi les autres jeux de ce type, de lire le papier duquel j'ai tiré cet article, qui constitue non seulement une mine d'informations, mais peut aussi être lu comme un guide méthodologique tant est grande sa clarté.

Et même si je me suis permis quelques remarques sévères sur sa fonction d'évaluation, je n'en demeure pas moins persuadé que Iago doit constituer un adversaire largement hors de ma portée !!!



Iago 40-24 Aldaron

Bill, la terreur de l'ouest

par François Aguillon

I. Introduction

Un précédent article sur le programme Iago ayant soulevé l'enthousiasme général d'au moins un lecteur de *Fforum* (moi-même), j'ai décidé à la demande générale de ce lecteur de tenter la même démarche pour présenter le programme Bill : cette démarche consiste tout simplement à paraphraser un rapport de recherche[1], en inventant un peu lorsque je ne comprends pas l'anglais.

Bill est un programme écrit par Kai-Fu Lee et Sanjoy Mahajan : comme le nom de ses auteurs le laisse deviner, c'est (encore) un programme U.S. conçu à l'université Carnegie-Mellon, les célèbres duettistes de Pittsburgh, dans le cadre d'un travail universitaire. Ce programme est véritablement la terreur de l'ouest, car il semble gagner tous les tournois qu'il veut[2]. Tremblez, programmeurs, rien qu'à l'idée de découvrir la bête !

Foin de billevesées, entrons dans le vif du sujet. Comme tous les programmes connus jouant à l'Othello potablement, Bill utilise un algorithme d'exploration d'arbre contenant la science programmatique de ses auteurs, couplé à une fonction d'évaluation contenant leur savoir othellistique. Je commencerai par parler du moteur du programme, l'exploration de l'arbre, puis de son carburant, la fonction d'évaluation.

II. Alpha-bêta

Que ceux qui ne connaissent pas cet algorithme sautent directement à la section III, parce que je n'ai pas l'intention d'en tenter ici une explication détaillée, si tant est que j'en aie les moyens intellectuels. Pour les vilains petits curieux qui voudraient savoir quand même, qu'ils lisent donc les articles de J.F. Puget dans les pages précédentes ou qu'ils se procurent les pages 255 à 297 de l'excellent livre de J.L. Laurière, « Intelligence Artificielle », chez Eyrolles (il faudra bien que je le regarde un jour, à force de dire à tout le monde qu'il est très bien, ça me donne envie de le lire).

Bon, donc si je ne décris pas l'algorithme standard, je vais au moins parler des « trucs » que possède Bill pour en améliorer les possibilités : fenêtre nulle, plan d'élagage, table de coups meurtriers.

1) Fenêtre nulle (zero-window)

L'alpha-bêta recherche le meilleur coup en supposant a priori que sa cote est comprise à l'intérieur d'une fenêtre définie par ses valeurs minimale (alpha) et maximale (bêta). Le temps mis par l'alpha-bêta pour donner sa réponse est évidemment d'autant plus court que la fenêtre en est plus étroite. Il faut donc la réduire le plus possible, mais en gardant présent à l'esprit que plus cette fenêtre est réduite, plus grand est le risque que l'alpha-bêta ne donne pas la solution, que le coup soit meilleur qu'attendu (auquel cas on se cogne sur le haut de la fenêtre) ou plus mauvais (et alors on se cogne en bas de la fenêtre).

Bill réduit sa fenêtre de recherche dynamiquement. Utilisant une profondeur incrémentale (un coup, puis 2, 3, etc.), il sait qu'il commence par examiner un coup qui, s'il n'est pas le meilleur, n'est pas totalement idiot. Partant de ce principe, une fois qu'il a déterminé l'évaluation E de ce premier coup, il fixe pour la suite de la recherche alpha à E et bêta à $E+1$. Dans ces conditions, tous les autres coups doivent se cogner soit en haut, soit en bas de la fenêtre (d'où le nom de fenêtre nulle). Trois cas peuvent alors se produire dans la suite de la recherche :

- tous les autres coups se cognent dans le bas de la fenêtre ; ceci arrive lorsque le premier coup était effectivement le meilleur ;
- tous les autres coups se cognent dans le bas de la fenêtre, sauf un qui se cogne dans le haut ; c'est alors celui qui s'est cogné dans le haut qui est le meilleur coup ;
- au moins deux coups se sont cognés dans le haut de la fenêtre ; ces coups sont meilleurs que le premier coup envisagé, mais on ne peut pas les comparer entre eux et il faut recommencer la recherche.

Il est clair que les deux premiers cas conduisent à un gain de temps sur l'algorithme classique, alors que dans le troisième cas, il y aura au contraire perte de temps. Le bilan global dépend très fortement de l'ordre de la recherche. Dans le cas de Bill, une statistique sur 22 coups conduit à un gain global de temps de 40% à huit coups de profondeur.

2) Plan d'élagage (hash table)

Je ne connais pas la traduction française consacrée de l'idiome anglais « hash table ». Mon dictionnaire unique et favori me propose « table pour hacher la viande », si vous préférez...[3]

Ces considérations sémantico-philologiques écartées, le principe du plan d'élagage est simple : sachant qu'une situation donnée est forcément rencontrée plusieurs fois au cours de la recherche incrémentale, il semble efficace de mémoriser la meilleure réponse à lui apporter (afin d'élaguer l'arbre de recherche au plus vite). Pour cela, on se réserve en mémoire une table de T éléments et on extrait de chaque position une clef C, qui n'est qu'un nombre compris entre 1 et T calculé à partir d'informations extraites de la position. Après examen d'une position, on prend soin de ranger le résultat, c'est-à-dire la meilleure réponse, à l'entrée numérotée C de la table. Avant examen d'une position, on regarde dans la table à l'entrée C quelle est la réponse apportée par les précédents examens de la position et on commence par celle-là.

Comme à Othello les interversions de coups sont rares, l'usage d'une telle table n'est donc justifiée que parce que

---

[1] K.-F. Lee and S. Mahajan, *« BILL: A table-based knowledge-intensive Othello program »*, Tech. Rept. CMU-CS-86-141, Carnegie-Mellon University, Pittsburgh, PA (86).
NDLR : Signalons qu'un article plus récent est paru : K.-F. Lee and S. Mahajan, « The development of a World class Othello program », *Artificial Intelligence* 43 (1990) 21-36.

[2] (NDLR) Au moins dans les années 86-89.

[3] On parle plutôt maintenant de « table de hachage ».

la recherche est incrémentale. La taille de la table doit être aussi grande que possible pour pouvoir contenir le plus d'informations possibles. Comme Bill n'accorde que 32768 entrées à cette table et qu'il examine couramment 100000 positions avant de choisir son coup, chaque entrée pointe en fait sur une liste décrivant de manière plus précise les positions ayant une même clef, précision apportée par une « serrure » codée sur 16 bits, le nombre de coups joués et la couleur du joueur qui a le trait. Malgré tout, l'ensemble clef + serrure + nombre de pions + trait ne donne encore que 38 bits, ce qui est insuffisant pour discriminer toutes les positions de l'othellier ; il se peut donc que deux positions différentes téléscopent leurs données. C'est pourquoi Bill vérifie systématiquement la légalité de la réponse proposée dans la table d'élagage.

Le gain de temps apporté par cette technique par rapport à une technique où les coups sont examinés dans un ordre fixe (les coins, puis les cases centrales... en dernier les cases X) est de l'ordre de 2,8 à une profondeur de huit coups. Mais un problème important subsiste : que faire lorsque le coup conseillé dans le plan d'élagage est illégal (ce qui doit être rare), ou bien inexistant (ce qui est fréquent, car systématique pour les nœuds terminaux de l'arbre, qui sont majoritaires) ? Les parents de Bill ont décidé de ne pas laisser leur rejeton chercher à l'aveuglette et de fournir à leur charmant bambin une table de coups meurtriers.

3) Table de coups meurtriers (killer table)

Si les deux précédentes sections étaient de la belle ouvrage (enfin décrivaient de la belle ouvrage, plutôt), ils n'en étaient pas moins assez standards. Par contre, la gestion de la table des coups meurtriers par Bill est très jolie et je pense originale.

Cette table comprend 60 entrées pour chaque couleur. L'entrée x pour la couleur c est une liste triée de toutes les cases vides, le critère heuristique de tri étant la valeur supposée du coup. Cette table est mise à jour de la manière suivante :

- en début de partie, elle est dressée de manière heuristique ; par exemple, pour un coup noir en b2, l'ordre est a1, a2, b1, a8, h1, h8, ..., g2, b7, g7 ;
- en cours de partie, elle est actualisée de la manière suivante : quand un coup est trouvé bon (ou du moins conduit à un élagage), il monte d'un cran dans la liste et son prédécesseur baisse d'un cran ; évidemment, cela rappelle l'affreux tri à bulles réputé pour sa lenteur, mais dans ce cas, la lenteur même du processus doit être un avantage ;
- enfin, quand un coup est joué, il est évidemment supprimé dans toutes les entrées de la table.

L'efficacité de cette table a elle aussi été mesurée à un facteur 2,2. Si l'on va un peu vite, on en conclut donc que plan d'élagage + coup meurtrier apportent ensemble un gain de l'ordre de 6,2. Les choses ne sont pas aussi simples car la table des coups meurtriers n'entre en jeu que lorsque le plan d'élagage échoue : souvent les deux font double emploi, ce qui fait que leur cumul n'apporte un gain de temps « que » de 3,8.

Reste donc à voir ce que Bill sait faire avec cette implémentation de l'algorithme alpha-bêta. Pour cela, examinons donc la fonction d'évaluation.

III. Fonction d'évaluation

La fonction d'évaluation est la somme de trois termes représentant les trois critères utilisés par Bill pour juger d'une position : les bords, la mobilité et l'occupation de l'othellier.

$$Eval = b * bords + m * mobilité + o * occupation$$

où b, m et o sont des coefficients dépendant du nombre p de pions joués, suivant l'algorithme suivant :

$b = 500, \quad m = 350 - 2p$

Si $p < 10, o = 200 - p$

Si $9 < p < 20, o = 210 - 2p$

Si $19 < p < 40, o = 270 - 5p$

Si $39 < p < 50, o = 350 - 7p$

Si $p > 50, o = 0$

Ces coefficients ont été sélectionnés parmi un jeu de dix coefficients en faisant jouer Bill contre lui-même dans un tournoi toutes-rondes.

L'idée directrice de Bill est d'utiliser autant que faire se peut une fonction d'évaluation rapide. Pour cela, les cotes de bord, mobilité et occupation pour toutes les situations rencontrées sont calculées dans des tableaux avant le début de la partie. Le cœur de Bill bat donc complètement à l'intérieur de ces tableaux (je trouve par moments ma prose aussi imagée que le chocolat Suchard).

1) Tableau des bords

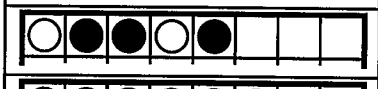
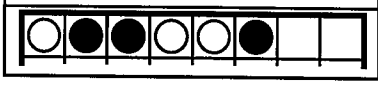
Un bord pour Bill est constitué des huit cases de bord proprement dites, auxquelles sont adjointes les deux cases X attenantes. Je trouve ce choix absolument génial, parce que c'est celui que j'avais fait pour Comp'oth, qui a aussi un tableau des cotes de bords : peut-être ne suis-je pas impartial dans ce jugement. J'ai décrit dans le passé la cote des bords effectuée par Comp'oth, voici maintenant celle de Bill. Bien qu'elles aient été conçues indépendamment l'une de l'autre, les ressemblances entre elles sont énormes, mais pas au point d'être totales.

Bill calcule son tableau des bords en deux passes : une passe statique et une passe dynamique.

1.1) Passe statique

Cette passe attribue une note à chaque bord en considérant sa stabilité. Dans ce but, sept classes de stabilité sont définies :

- les pions définitifs, qui ne pourront jamais être retournés ;
- à l'opposé, les pions instables, qui peuvent être retournés immédiatement ;
- entre ces deux extrêmes se trouvent les pions très stables, qui ne peuvent être retournés qu'après un coup donné pour chacun des deux joueurs ;
- les pions stables, qui ne peuvent être retournés qu'après un coup joué par leur propre couleur ;
- les pions flottants, enserrés entre deux pions instables de l'adversaire ;
- les pions isolés, qui sont seuls (!) ;
- enfin tous les autres pions, dits semi-stables.

	Pions instables
	Pion isolé
	Pions semi-stables
	Les noirs sont des pions flottants
	Les noirs sont des pions stables
	Ces mêmes pions sont ici très stables
	Et là, ils sont définitifs

Ensuite, chaque type de cases est cotée suivant sa stabilité, selon la loi suivante :

	coin	case C	case A	case B
instable		-50	20	15
isolé		-75	-25	-50
semi-stable		-125	100	100
flottant			300	200
stable		800	800	800
très stable		1000	1000	1000
définitif	800	1200	1000	1000

La cote statique d'un bord pour Noir est obtenue en additionnant les valeurs des cases noires et en soustrayant la valeur des cases blanches. Ensuite, les cases X sont évaluées, mais leur valeur est petite du point de vue de la stabilité.

Cette valeur statique est souvent correcte, parfois mauvaise. Par exemple, un bord du type

est estimé à +50 (sur une échelle allant de -8000 à +8000), alors qu'il est évidemment mauvais. C'est pourquoi la valeur statique est corrigée par un traitement dynamique du tableau des bords.

1.2) Passe dynamique

Le principe de cette passe est d'apporter des modifications à la cote statique des bords en analysant vers quelles positions ces bords sont susceptibles d'évoluer. Les bords pleins étant définitifs, leur cote définitive est égale à leur cote statique. Tous les autres bords voient leur cote statique être modifiée par celle de leurs enfants, en faisant appel à la fonction récursive eval_def(bord vide), définie par :

```
FUNCTION eval_def(père)
  IF cote(père) non définitive
    REPEAT
      génère un nouveau bord fils
      IF fils <> père
           (si on passe, fils = père !)
        cote(fils) = eval_def(fils)
    UNTIL tous les fils ont été
          envisagés
    cote(père)=cote_dynamique(père)
    cote(père) définitive
  END
  RETURN cote(père)
END
```

Le secret réside évidemment dans l'appel de la fonction cote_dynamique(père). Cette fonction ressemble beaucoup à une fonction minimax, à une différence importante près : on ne sait pas de manière certaine si les coups joués sont légaux. S'ils retournent des pions dans la ligne, ils sont forcément légaux, mais s'ils n'en retournent pas, il ne sont pas forcément illégaux : cela dépend du reste de l'othellier. D'autre part, parmi les coups légaux, il y a la possibilité de passer : on n'a pas le droit de passer sur l'othellier, mais on a sinon toujours, du moins la plupart du temps, le droit de jouer ailleurs que sur le bord envisagé.

Bill effectue donc un minimax « probabiliste » : chaque coup fils est caractérisé non seulement par sa cote, mais aussi par sa probabilité de légalité, déterminée ainsi :

- un coup légal a une probabilité de 1 ;
- l'accès à un coin est improbable, sauf si :
- il est démontré légal (accès par une case C) ;
- la case X est occupée par l'adversaire ;
- l'accès à une autre case est :
- plus probable s'il y a des pions adverses à côté ;
- moins probable si ce sont des pions amis ;
- plus probable si la ligne est plus remplie.

Les probabilités de légalité des fils étant ainsi déterminées dans un tableau proba(fils), Bill estime à partir de leur cote celle de leur père, en suivant l'algorithme suivant :

```
FUNCTION cote_dynamique(père)
  valeur = 0
  proba_rest = 1
  max_légal = cote du meilleur coup
              légal

  TRIE les coups possibles
       (et non légaux !)
  FOR coups-fils possibles
      du meilleur au pire
    IF cote(fils) < max_légal
      QUITTE LA BOUCLE
    ELSE
      valeur = valeur + proba_rest*
               proba(fils)*cote(fils)
      proba_rest = proba_rest -
                   proba(fils)
    END
  END
  valeur = valeur+proba_rest*max_légal
  RETURN valeur
END
```

(Ceci est l'algorithme publié et il a je pense deux défauts : il n'envisage pas le cas où il n'y a pas de coup légal, ce qui n'est pas très compliqué à corriger ; d'autre part, l'ajustement à l'intérieur de la boucle de la variable proba_rest doit, je pense, se faire plutôt ainsi

```
proba_rest=proba_rest*(1-proba(fils))
```

afin d'éviter d'avoir à faire à des probabilités négatives conduisant à des résultats incongrus.)

1.3) Commentaires personnels

À quoi sert le tableau statique, puisqu'il semble à première vue que seuls les bords complets se voient attribuer une valeur statique ? La réponse est que parmi les coups légaux, il y a toujours la possibilité de passer. Il faut donc connaître la cote d'une position pour pouvoir la calculer, puisqu'une position se génère elle-même après le coup « passe ». C'est là qu'intervient la cote statique, attribuée à une position en temps que « fils » pour déterminer la cote de cette même position en temps que « père ».

Mais alors me diront les fins matheux, pourquoi s'arrêter en si bon chemin et pourquoi ne pas réinjecter la cote ainsi déterminée dans la position « fils » pour recalculer celle de la position « père » ? Pour tout vous dire, les auteurs de Bill n'ont pas abordé le problème dans leur papier, mais il ne m'étonnerait pas du tout que ce soit ce qu'ils fassent faire à leur rejeton. Trois raisons de penser cela :

1) ce sont visiblement des gens subtils ;

2) le terme *converged* — assez facile à traduire de l'anglais en français — est employé pour caractériser une position dont la cote dynamique a été établie. Cela laisse entendre que le « développement perturbatif » a été poussé à plusieurs ordres jusqu'à ce qu'il converge.

3) Bill met cinq minutes pour calculer ce tableau. Comp'oth fait le même genre de fantaisie en 45 secondes, sur une machine plus modeste, mais sans itération de ce type — et là, j'en suis sûr !

Question subsidiaire pour matheux enragés : un tel développement converge-t-il toujours ? Sous la torture et bien qu'incapable de me livrer à de tels batifolages mathématiques, je dirais que oui. Mais la limite de convergence dépend de la cote statique ; ceci se voit facilement lorsque le meilleur coup consiste à passer : la cote est alors égale à la cote statique.

Enfin, et bien que cela risque de n'intéresser que moi, on peut comparer les techniques de Bill et de Comp'oth pour calculer tous les deux par un minimax probabiliste le même tableau. Comp'oth a une passe statique très réduite (seulement les bords complets, où elle est totalement valable) et évite le problème du mouvement qui autorise à passer en disant que passer revient à laisser l'adversaire jouer sur le bord. Ainsi, un coup ne peut pas être son propre fils et la cote est entièrement dynamique à partir des positions complètes.

2) Évaluation de la mobilité et de l'occupation

Bill ne tient évidemment pas compte que de la structure des bords pour juger une position. Il tient compte aussi d'autres facteurs, regroupés sous le terme générique de facteurs internes (par opposition aux bords, qui sont externes à l'othellier) :

- de combien de coups légaux dispose chaque joueur ?
- quelle est la position de ces coups sur l'othellier ?
- combien de pions seront retournés par ces coups ?
- combien y a-t-il de cases vides adjacentes aux pions de chaque joueur ?
- combien y a-t-il de pions de chaque couleur adjacents à une case vide ?
- quelle est la longueur et la position de chaque séquence de pions ?
- les pions de chaque joueur sont-ils centraux ou périphériques ?

Bill utilise le fait que l'évaluation de toutes ces questions peut être réalisée ou approximée en examinant une à une toutes les séquences de pions alignés figurant sur l'othellier. Toute l'information sur les facteurs internes a donc été concentrée dans neuf tableaux.

- Trois tableaux à 6561 entrées pour les lignes et les colonnes. Je pense deviner qu'il y a un tableau pour les bords (non pas en tant que tels, mais pour leur influence sur la mobilité), les prébords et les rangées centrales. Pour ceux qui n'ont pas une machine à calculer fonctionnant en coprocesseur avec leurs circonvolutions méningées, 6561 n'est autre que 3 à la puissance 8, c'est-à-dire le nombre total de configurations possibles d'une rangée de huit cases.
- Un tableau à 6561 entrées pour les grandes diagonales.
- Un tableau à 2187 entrées pour les diagonales à sept cases.
- Un tableau à 729 entrées pour les diagonales à six cases.
- Un tableau à 243 entrées pour les diagonales à cinq cases.
- Un tableau à 81 entrées pour les diagonales à quatre cases.
- Un tableau à 9 entrées pour les diagonales à trois cases (enfin, le chiffre 9 publié me semble devoir être lu 27, mais c'est un détail).

Les diagonales à deux cases ou à une case n'ont aucune influence sur la mobilité, vu que l'on ne peut retourner de pions le long d'une telle diagonale.

L'évaluation d'une position se fait alors en lisant les informations contenues dans ces tableaux pour tous les alignements de plus de deux cases existant sur l'othellier. L'évaluation de la mobilité d'une position se ramène donc à un certain nombre de lectures de tableaux (4 diagonales à 3, 4, 5, 6 ou 7 cases, 2 diagonales à 8 cases, 4 bords, 4 prébords et 8 rangées centrales).

Les tableaux ainsi structurés contiennent des informations sur la mobilité vraie pondérée, la mobilité potentielle pondérée et la structure des pions.

2.1) Mobilité vraie

Bill n'évalue pas la mobilité en comptant bêtement le nombre de coups légaux : il les pondère suivant leur qualité. Le problème est encore une fois que leur qualité n'est pas connue, puisque c'est justement le rôle de la fonction d'évaluation de la calculer. Cette qualité est donc déterminée, d'après ce que j'ai cru deviner :

- en attribuant une valeur statique aux cases jouables (par exemple un accès à une case X est moins bon qu'un accès à une case centrale) ;
- en attribuant une valeur statique aux cases retournées, en faisant une évaluation sommaire de leur importance (par exemple, l'accès à une case A sera beaucoup moins estimé s'il retourne une case X, car la probabilité est importante que le coin voisin ne soit pas occupé et que la case X donne accès à ce coin) ;
- enfin, en minimisant le nombre de pions retournés.

Bien que la plus importante pour Bill, la mobilité vraie n'est pas la seule composante de l'évaluation de la mobilité. Est aussi prise en compte la mobilité potentielle, qui présente sur la mobilité vraie l'avantage de varier moins vite et donc de minimiser l'effet horizon du minimax.

2.2) Mobilité potentielle

Elle est évaluée de la manière suivante :

• pour chaque case vide, soustraire des points à chaque couleur adjacente à cette case vide ;

• pour chaque case noire (resp. blanche), soustraire des points à Noir (resp. à Blanc) pour chaque case vide adjacente.

Le nombre de points soustraits à chaque fois dépend de la qualité supposée de la mobilité potentielle considérée, de façon similaire à la méthode employée pour la mobilité vraie. La note de mobilité est évaluée en combinant les mobilités vraie et potentielle, avec une pondération en faveur de la première.

2.3) Évaluation de l'occupation

Le but de la note d'occupation est double : éviter les longs alignements et pondérer la valeur des cases occupées.

Eviter les longs alignements est, à ma connaissance, une disposition particulière à Bill. L'argument avancé par ses auteurs pour la justifier est qu'un long alignement peut constituer un mur, s'il est à l'extérieur de la position, et donc est potentiellement mauvais (Bill n'a pas moyen de savoir si un alignement est un mur ou non ; il sait très bien si un pion est à la frontière ou pas, mais ne sait rien sur un alignement desdits pions de frontière). De plus, un long alignement compte nécessairement beaucoup de pions et est donc mauvais dans la perspective de réduire le nombre de pions.

La valeur des cases occupées est pondérée, mais de manière autrement plus fine que ce que l'on voit d'habitude :

• il est très mauvais de laisser un vide entre deux pions de sa couleur ;

• il est mauvais d'avoir un pion adverse entre deux de ses pions ;

• les pions qui ne peuvent ni retourner ni être retournés sont mauvais (attention, on ne parle toujours que dans une direction, il ne s'agit pas encore de pions définitifs !) ;

• les pions centraux sont meilleurs que les pions périphériques.

Comme il a déjà été expliqué, la cote d'occupation est de moins en moins importante au fur et à mesure du déroulement de la partie. Enfin, on doit noter que Bill compte les pions de chaque couleur, mais seulement pour le cas où un joueur n'a plus de pion.

2.4) Commentaires

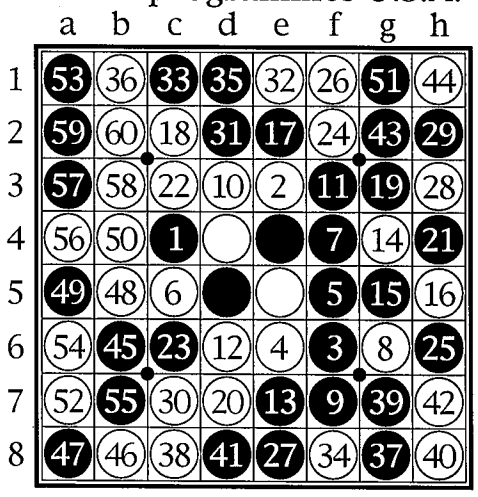
Bon, tout d'abord une information capitale : Bill-la-terreur ne sait rien de la parité et toute étude de parité me semble impossible à mener à l'aide des tableaux décrits. Donc, pour battre Bill, tout sur la parité (mais ça m'étonnerait que Bill n'ait pas évolué sur ce plan-là aussi, mes informations datent de 1986...).

Ensuite le point original de Bill pour les facteurs internes me semble être la pondération de l'occupation. En particulier, le fait de ne pas aimer les alignements est surprenant. Je dois dire que les raisons invoquées par les auteurs de Bill pour justifier cette attitude ne me convainquent pas : en effet, pour minimiser le nombre de pions, il me semble plus simple et plus efficace de les compter. D'autre part, le nombre d'alignements susceptibles de constituer un mur est petit devant le nombre d'alignements internes : je ne pense pas là non plus que la fonction proposée soit efficace pour détecter la présence de murs. Le fait d'éviter les longs alignements est peut-être un gage de gain de mobilité à long terme : un long alignement interne conduira tôt ou tard à jouer des coups retournant des pions externes dans plusieurs directions à la fois. C'est probablement une combinaison de tous ces facteurs qui justifie cette attitude.

IV.Conclusion

Voilà pour l'essentiel de Bill. Je passe sous silence les choses plus standard : réflexion sur le temps de l'adversaire, bibliothèque d'ouvertures, fin de partie, gestion du temps sont autant de « détails » intéressants exposés dans le papier cité dans l'introduction. Je livre en pâture à votre sagacité quelques parties jouées par Bill : pour remonter le moral des programmeurs affolés par la quantité de travail à fournir pour rejoindre Bill et espérer le battre, j'y inclus la partie perdue par Bill en tournoi face à Peer Gynt[4], le programme d'Anders Kierulf (bon, mais pour écrire un programme comme Peer Gynt, il faut aussi faire « quelques » efforts !).

Tournoi de programmes U.S.A. 86

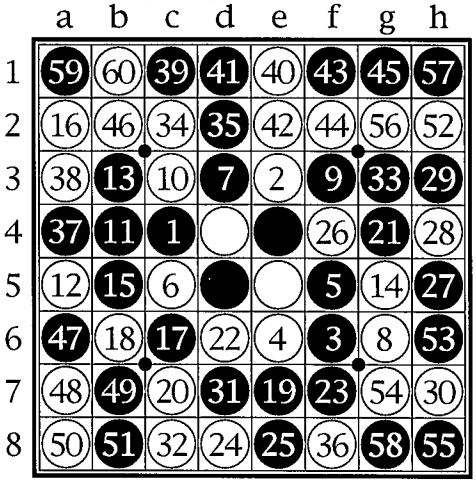


Bill 44-20 Aldaron

---

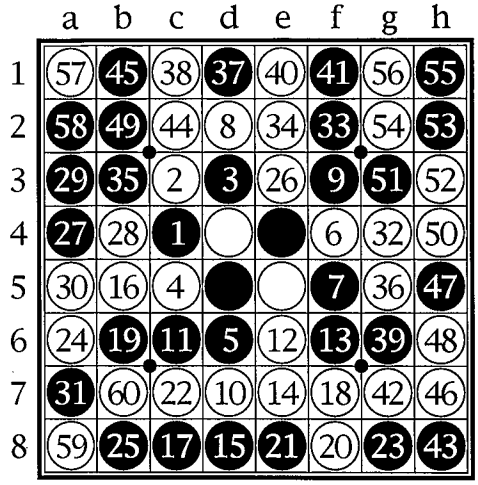
4 (NDLR) Bill en a bien sûr perdu d'autres depuis.

Tournoi de programmes U.S.A. 86



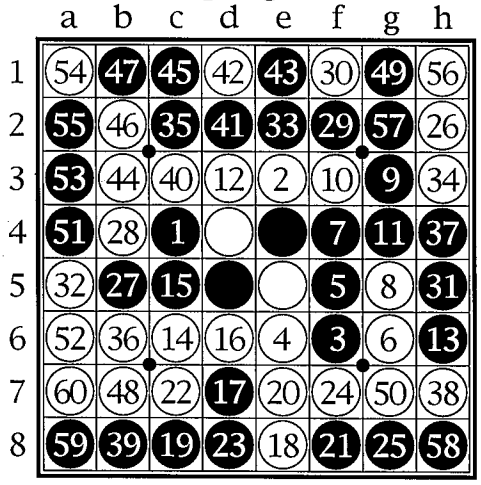
Aldaron 45-19 Bill

Courchelettes 93



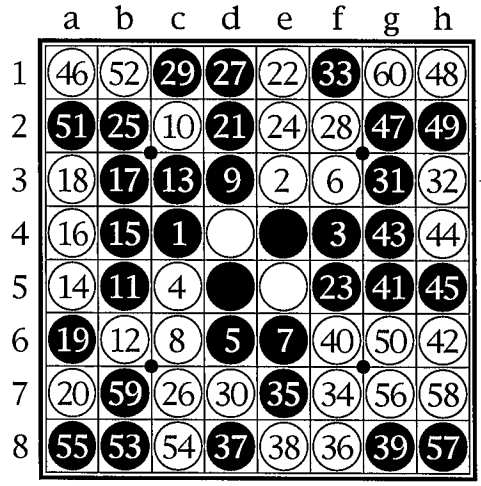
Bill 24-40 Rev

Tournoi de programmes 89



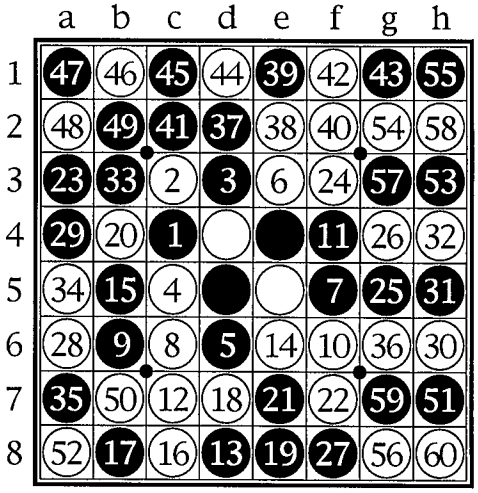
Peer Gynt 35-29 Bill

Courchelettes 93



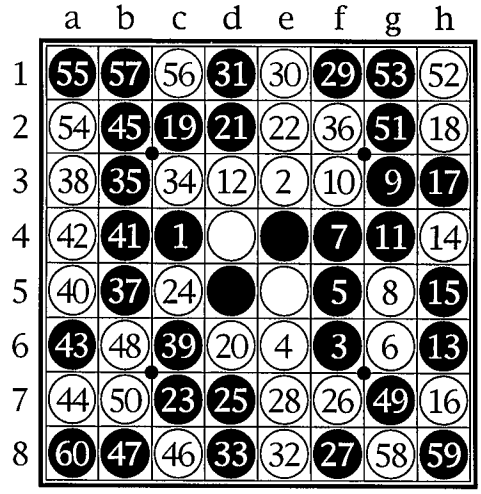
Gros Thello 26-38 Bill

Courchelettes 93



Bill 29-35 Jacpoth

Courchelettes 93



Fugazi 28-36 Bill

Test Othello Killer

par Bruno de la Boisserie

Othello est un jeu qui a toujours fasciné les informaticiens. Beaucoup de joueurs font d'ailleurs ce métier. Il est donc très logique que beaucoup de programmes aient été écrits... On en trouve un grand nombre dans le domaine public : une centaine rien que sur IBM PC et compatibles. D'autres sont diffusés par leurs auteurs, comme Comp'Oth ou Thor. Quelques-uns, enfin ont été commercialisés.

Parmi eux, Reversal, Odin et Master Reversi ont connu leur heure de gloire sur Apple 2 et Trs-80. Plus récemment, l'Oric et l'Amstrad se sont vus dotés de Reversi Champion, écrit par Vincent Baillet, un des meilleurs joueurs français il y a quelques années. Mais... beaucoup de mauvais programmes ont aussi été vendus, comme le nullissime Prise sur Thomson.

Othello Killer, sur Commodore Amiga, fait-il partie du premier groupe ou du second ? Angoissante question pour Ubi Soft, l'éditeur...

Pour le savoir, j'ai fait jouer Othello Killer contre Comp'Oth (limité à 5 minutes de réflexion). Le début s'est avéré relativement équilibré, mais la situation s'est vite dégradée en milieu de partie. Comp'Oth est parti beaucoup plus tôt en finale, et s'est imposé largement.

Mis en confiance par ce résultat, j'ai joué une partie contre ce programme. J'ai pu constater qu'il savait surveiller ses libertés, jouait les coups tranquilles quand il y en avait, pouvait éventuellement damiériser son adversaire pour réduire ses libertés. Il n'est pas atteint d'Hasegawite (ancienne maladie d'origine japonaise consistant en un goût immodéré pour les cases A). Mais, comme beaucoup de joueurs informatiques et humains, il a du mal à juger les chances de succès d'un bétonnage. J'ai appliqué cette stratégie, il s'est trouvé assez vite à court de libertés et a perdu la partie. Déception chez Ubi Soft, qui me prenait pour un super-champion, quand je leur ai montré mon classement (là, en bas...)

Les meilleurs programmes d'amateur restent donc nettement meilleurs que ceux des « professionnels », sur le plan du niveau de jeu. Il faut dire qu'Othello Killer ne possède ni bibliothèque d'ouverture, ni gestion globale du temps (le niveau est déterminé par la profondeur), ni réflexion sur le temps de son adversaire. Enfin, il faut savoir que le meilleur niveau travaille à seulement six coups de profondeur (en moyenne) et que la recherche finale ne porte que sur les neuf derniers coups.

Arrêtons-nous sur un détail qui explique beaucoup de choses : Ubi Soft ne connaissait ni l'existence de la fédération et de ses tournois réguliers, ni le livre « Othello par l'exemple ». On comprend mieux ainsi leur difficulté à juger du niveau de leur programme...

Si nous ne sommes pas en présence d'un tueur, nous avons par contre un bon professeur sous la main : en effet, Othello Killer dispose d'options qui faciliteront les progrès des débutants et des joueurs moyens :

- les quatre niveaux leur permettront de toujours trouver un partenaire adapté et de mesurer leurs progrès ;
- la sauvegarde et le chargement de parties autoriseront l'étude à tête reposée de toute partie ; il est possible d'avancer et de reculer librement du début à la fin, pour apprécier des évolutions à long terme ;
- Othello Killer pourra leur indiquer ce qu'il aurait joué à leur place, que ce soit dans une partie contre l'ordinateur ou sur un problème précis, qu'une option spécifique permet de préparer.

On peut tout de même déplorer l'impossibilité d'afficher des coordonnées, bien utiles pour suivre les articles de Fforum, et les quelques entorses faites à la règle : on peut commencer avec les blancs, et les pions sont inversés quand le joueur commence avec les noirs (les deux pions noirs sont en d4 et e5).

Un mot enfin sur la qualité de réalisation d'Othello Killer : le graphisme est sympathique, bien qu'un peu décevant pour un Amiga. Le retournement des pions est un petit spectacle à lui tout seul. Une musique et quelques commentaires parlés (numérisés) sont censés agrémenter le jeu, mais on s'en lasse vite. Il est heureusement possible de les mettre hors fonction. La notice est correctement écrite, assez complète, en 10 pages environ. Une page sur deux est imprimée en noir sur fond sombre, pour décourager les photocopieurs. La protection se fait en effet à chaque lancement, en indiquant un mot présent à un endroit précis de la notice.

Au total, Othello Killer est un produit conçu sérieusement, pour un public le plus large possible. C'est un excellent cadeau de Noël pour votre petit neveu « qui voudrait bien jouer mais qui n'a jamais de partenaire ». Grâce à ce professeur patient, il pourra arriver rapidement au niveau d'un joueur de club, pour peu qu'il s'accroche, et viendra tout naturellement grossir les rangs de la F.F.O.

Pour vous qui n'en êtes plus à votre premier piège de Stoner, il vaut mieux contacter les auteurs des bons programmes d'amateur (voir à ce sujet les différents comptes rendus de tournois de programmes dans *Fforum*), et vous frotter à ces adversaires d'un niveau international !

Parties Internet 1997

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

Zebra 29-35 Bill

La force brute ou les limites de la machine

par Yannick Hervé

I. Préliminaires

Enfin autre chose que de la stratégie pour champions !

Les pages de notre bon *Fforum* étant (aussi) fréquentées par quelques auteurs de (bons) programmes, ils peuvent y puiser toutes les astuces, techniques, stratégies et tactiques des meilleurs humains pour mettre dans leur joujou binaire.

Avec cet article, j'aimerais apporter quelques réflexions sur l'informatique et Othello. Après les articles de B. DAUNAS et J.F. PUGET dans des vieux *Fforum* puis ceux de J.C DELBARRE dans les derniers, cet article et les suivants[1] font et feront le point de deux ans de réflexion sur ce qu'est une machine qui joue à Othello, mais en se focalisant plutôt sur la machine que sur l'algorithmique.

1) Les premiers essais infructueux

Avec l'accord et la participation de Sylvain QUIN, nous avons présenté deux versions de Thor au championnat du monde des ordinateurs de jeux à Londres en août 1989. Une version tournait sur un PC 386 à 20 MHz et une autre sur une machine parallèle à 32 Transputer T800 à 20 MHz et RAM statique(C&Bek).

La version PC a eu quelques problèmes pendant les deux premiers des cinq jours de tournois ; nous avions une version du programme complètement farfelue due aux modifications de dernière minute. Après changement de version, ce programme a fini 5ème.

La version machine rapide devait théoriquement faire un malheur, mais un portage et une parallélisation en 5 jours et nuits, une incompréhension entre Sylvain (à Paris) et moi (à Strasbourg) et un GROS bug dans l'analyse de milieu de partie ont fait que celui-ci est arrivé avant-dernier. Ce programme était pourtant le seul à donner le score 20 à 22 coups avant la fin (la fin de partie fonctionnait heureusement).

La puissance de calcul disponible laissait rêveur bon nombre de programmeurs, mais celle-ci était mal utilisée. Cet échec m'a dans un premier temps abattu puis titillé, aussi avec l'auteur de Thor nous avons décidé de renouveler l'expérience en s'y prenant plus tôt et en cherchant la meilleure machine possible.

2) Le travail en cours

Pour les championnats 1990, je dois donc porter Thor sur toutes les machines me tombant sous la main pour déterminer la plus efficace[2]. Pendant ce travail j'ai été amené à me demander comment optimiser Thor en particulier et un programme d'Othello en général ; j'ai donc ressorti de mes cartons une étude que j'avais menée il y a deux ans pour la reprendre et la compléter ; ce papier rapporte quelques réflexions issues de ce travail.

[1] (NDLR) qui malheureusement n'ont jamais été écrits...

[2] (NDLR) Probablement à cause du coût élevé de l'inscription, seulement six programmes (dont un français, Microb) participèrent à ces championnats.

II. Introduction

L'écriture d'un programme d'Othello efficace demande un langage, une machine, une bonne dose de patience et des compétences informatiques et othellistiques. On le voit de plus en plus dans les tournois, quel que soit le jeu d'ailleurs, les programmes en langages rapides (C ou assembleur) sur des machines rapides gagnent plus facilement que les programmes en Basic interprété (ou LISP... beurk) sur des processeurs 8 bits.

Comme je ne programme pas au top niveau (quand j'ai un gros boulot... je fais sous-traiter par un collègue informaticien) et comme ma spécialité (professionnelle) est l'architecture des ordinateurs à usage général ou spécialisé, je me pose certaines questions sur l'utilisation rationnelle de « machines » pour jouer à Othello.

J'entends par machine, un dispositif automatique et efficace pouvant être amené sur un lieu de tournoi, ou utilisé pour l'entraînement ; donc loin de moi l'idée de faire un réseau de CRAY-3 à usage uniquement réservé à mon jeu favori.

Par formation, j'ai plutôt tendance à penser « force brute » plutôt que finesse d'analyse et je pense sincèrement que les progrès des machines à jouer passent par l'association d'une machine très rapide et d'une analyse très poussée. (C'est une lapalissade mais il fallait le dire.)

III. Les questions

Je me pose donc les questions suivantes.

- Quelle est l'efficacité d'un processeur classique pour résoudre des problèmes d'Othello ?
- Quelle est la limite de performance pouvant être atteinte avec une machine conventionnelle ou non-conventionnelle ?
- Avec la technologie actuelle, combien de coups avant la fin peut-on connaître le score ?
- Pour évaluer une position à une profondeur donnée, en un temps donné, de quelle puissance de calcul doit-on disposer ?
- Quelle est la machine dont la puissance serait la mieux utilisée par un programme d'Othello ?

IV. Les outils d'un bon programme

Pour faire fonctionner un programme d'Othello, il faut un certain nombre de modules logiciels, ou procédures, se répartissant en deux familles.

Les routines locales :

- Cj - À partir d'une position et du trait, déterminer la liste des coups valides.
- Jc - À partir d'une position, du trait et de la case choisie, déterminer la nouvelle position.
- Ev - Pour une position, calculer une note permettant l'évaluation. Cette routine est « l'intelligence » du programme.

• Np - Pour une position, calculer le nombre de pions de chaque couleur et qui gagne si c'est le dernier pion posé.

Les routines globales :

• Ar - À partir d'une position et du trait, il faut construire l'arbre des séquences de coups possibles pour « aller voir » le plus loin possible les conséquences d'un choix.

• Mm - Sur l'arbre des coups, pour connaître les conséquences d'un choix il faut « remonter » l'évaluation des positions « feuilles » par une technique de minimax.

• AB - Pour éviter de scruter des directions dans l'arbre qu'on peut savoir inutiles (ceci de façon algorithmique et non heuristique), il faut pratiquer une technique d'alpha-bêta ou algorithme similaire. Cette routine donne sa « force » au programme en diminuant la divergence moyenne (voir les articles précédents de J.F. PUGET).

Pour toutes ces routines, plusieurs algorithmes existent et c'est la compétence ou l'ingéniosité du programmeur qui fait la différence. On peut choisir d'écrire un algorithme récursif ou non, profondeur itérée, stockage d'une partie de l'arbre de recherche, technique du *zero-window*... La littérature spécialisée est bourrée de ces algorithmes tout aussi compliqués les uns que les autres. Les structures de données présentent aussi une grande importance : codage de la position, codage des coups, stockage de l'arbre...

Enfin le séquencement d'un programme est constitué de plusieurs phases qui peuvent être réduites à

• P1 : Séquence d'ouverture (débuts de parties en bibliothèque).

• P2 : Milieu de partie (souvent découpé en plusieurs phases).

• P3 : Recherche d'un coup gagnant.

• P4 : Recherche du meilleur coup gagnant ou du moins mauvais coup perdant.

Le programme peut aussi avoir la capacité de calculer sur le temps de l'adversaire en faisant l'hypothèse que l'adversaire va jouer un certain coup. Si cette hypothèse se révèle exacte, tant mieux c'est du temps en plus, si elle est fausse, on arrête l'évaluation et on reprend avec la nouvelle configuration.

On ne parlera évidemment pas de l'interface avec l'utilisateur qui est à la portée du premier programmeur venu et qui n'intervient en rien dans la force de jeu d'un programme.

V. Quelques réflexions

Dans une première approche, les routines Cj, Jc et Np sont purement mécaniques et il est dommage de perdre du temps processeur (processeur est pris ici au sens le plus large du terme) pour les effectuer. Pour Thor, le temps d'analyse est constitué de 30% pour Cj et 37% pour Jc.

Ce que l'on peut remarquer d'abord : le calcul d'un coup valide (Cj) est indépendant du calcul d'un autre coup jouable ; malheureusement un processeur classique ne peut les rechercher que de facon séquentielle (les uns après les autres).

Les phases P3 et P4 ne font intervenir aucune intelligence puisqu'il suffit de savoir compter les pions.

À part l'évaluation (Ev), tous les modules font intervenir des opérations très simples comme des comparaisons, des incréments...

VI. Quelques calculs

Les machines actuelles[3] (celles que je connais) évaluent entre 600 et 10.000 coups par seconde en milieu de partie, la moyenne étant aux alentours de 1200. Pour les fins de parties je n'ai que quelques chiffres qui s'échelonnent entre 8.000 et 30.000 coups par seconde.

Pour une profondeur d'analyse P et une divergence D on a :

$$\text{Noeuds} \approx D^{P+1}/(D-1)$$

$$\text{Feuilles} \approx D^{P}$$

$$\text{Noeuds\_non\_feuilles} \approx D^{P}/(D-1)$$

Un coup de milieu de partie se jouant en environ 50 secondes en cadence de tournoi, regardons en fonction des profondeurs et des divergences, les cadences d'analyse dont il faudrait disposer pour effectuer le boulot en 50 s.

P\D	3.35	3.4	3.45	3.5	3.55
10	5075	5849	6728	7723	8851
11	17000	19886	23211	27030	31421
12	56950	67614	80080	94606	111544
13	190782	229889	276276	331123	395983
14	639121	781622	953152	1958932	1405742
15	2141000	2657517	3288375	4046264	4990385

CM(P,D) en noeuds par seconde

On peut faire la même analyse pour une recherche de coup gagnant.

On peut remarquer que l'efficacité de l'alpha-bêta, qui diminue la divergence moyenne, est extrêmement importante. La force de jeu d'une machine sera liée au couple puissance intrinsèque/efficacité de recherche. On peut donc déjà dire que rien ne sert de fabriquer une machine qui joue extrêmement vite si elle ne possède pas un algorithme de type alpha-bêta efficace. Cette dernière remarque est vraie quelle que soit la phase de jeu : milieu de partie, recherche de coup gagnant ou recherche du meilleur coup gagnant.

On va chercher la machine la plus puissante réalisable avec la technologie actuelle, accessible à un bidouilleur averti (possédant quand même quelques économies).

VII. Eléments de réponses et architectures

1) Le processeur super-efficace

Les nouveaux processeurs arrivant sur le marché sont très prometteurs, c'est une bagarre de MIPS, que peut-on en faire pour Othello ?

Sur une machine purement séquentielle de puissance de calcul Pc, le temps d'évaluation de la profondeur Pr est de T. Si on a une divergence D de l'arbre de recherche, il faut D fois plus de puissance pour scruter jusqu'à la profondeur Pr+1 dans le même temps.

---

[3] (NDLR) N'oublions pas que cet article a été écrit en 1990 !

Faisons le point sur quelques machines actuelles :
(MIPS=Million d'Instructions par Seconde ; les MIPS indiqués ici sont des MIPS mesurés à l'aide de Thor et sont donc les MIPS effectivement utilisables, rien à voir avec les MIPS VAX, RISC ou CISC des documentations flatteuses.)

80386/33MHz = 2,4 MIPS
80960/20MHz = 4,2 MIPS
SPARC/20MHz = 4,4 MIPS (SUN 4/110)
SPARC/25MHz = 8 MIPS (SUN 4/330)
80486/33MHZ = 11 MIPS (estimé avec Cache 64Ko)

On voit que le nec plus ultra des processeurs actuels, le 80486, ne permet de « voir » qu'à peine deux profondeurs supplémentaires qu'un 80386 à 16MHz, et il faudrait une divergence inférieure à 3,3.
Pour les algorithmes de fin de partie où la divergence est plus faible, la différence se fait nettement plus sentir (ce sera vrai aussi pour les paragraphes suivants).

2) Le multi-processeur

Des études théoriques et pratiques ont montré que l'algorithme alpha-bêta n'était pas facilement parallélisable. Avec N processeurs, et pour les versions actuelles de l'alpha-bêta parallèle, on ne peut espérer faire mieux qu'un *speed-up* de $\sqrt{N}$. Donc, pour gagner un niveau de profondeur, si D est la divergence moyenne, il faut D^2 fois plus de processeurs. (speed-up : temps avec un processeur/temps avec N processeurs.)

Hypothèse : D=3,4 et profondeur P effectuée avec 1 processeur en T secondes.
Il faut donc un speed-up de 3,4 pour espérer analyser une profondeur de plus dans le même temps.

Prof.	speed − up	Nb. Proc.	si $D = 3,4$
P	1	1	1
$P+1$	D	D^2	12
$P+2$	D^2	D^4	134
$P+3$	D^3	D^6	1545 !!!
...	...	...	...

Ceci est une performance théorique et ne tient pas compte des échanges entre processeurs, des processeurs en attente de résultats, du temps de configuration du réseau, de l'*over-head*... (over-head : partie de l'algorithme non distribuable sur le multi-processeur et effectuée par un seul.)
On voit que cette solution, avec les versions parallèles de l'alpha-bêta existantes, n'est pas très économique en processeurs, sans parler de la façon de les interconnecter.
Il existe néanmoins des variantes de l'alpha-bêta donnant de meilleurs résultats mais au prix d'une complexité algorithmique effrayante.

3) Les architectures parallèles encore plus bizarres

Aucune architecture parallèle existant actuellement ne peut répondre à toutes les fonctions à implémenter (tout l'algorithme n'étant pas parallélisable), il faudrait donc avoir un processeur séquentiel gérant et distribuant des bouts d'application à une ou plusieurs machines parallèles, et alors la loi d'Amdahl s'applique.

Soit un algorithme se déroulant en un temps T sur un processeur classique, r% de cet algorithme est parallélisable et se déroule A fois plus vite avec l'extension parallèle, on a :

avant parallélisation $T = rT + (1-r)T$
et après parallélisation $T' = rT/A + (1-r)T$
Speed-up total
$$S = (rt + (1-r)T)/(rT/A + (1-r)T)$$
Si A tend vers l'infini $S = 1/(1-r)$

Avec une extension parallèle infiniment efficace pour r% de l'algorithme, on a donc les résultats suivants :

r%	Speed-up
5	1,05
50	2
71	3,5
90	10

Il faut donc, si la divergence est de 3,5, avoir 71% de l'algorithme parallélisable sur une machine infiniment efficace pour gagner une profondeur d'analyse !!! Je vous laisse faire les calculs si A n'est pas infini.

4) Les coprocesseurs spécialisés

Pour les coprocesseurs spécialisés, le phénomène vu au-dessus s'applique encore mais ces processeurs peuvent être beaucoup plus simples et optimisés. Si on veut utiliser cette méthode il vaut mieux avoir le maximum de performances, donc des processeurs câblés quand c'est possible.

5) Les modules câblés

Dans la mesure où certaines fonctions sont simples à réaliser, on va chercher à en optimiser le fonctionnement au niveau le plus bas en les câblant ; travail délicat demandant de bien sérier les problèmes rencontrés et l'enchaînement des opérations. L'étude des paramètres complexité, rapidité, coût, performances, encombrement pour les différentes fonctions énumérées plus haut peut permettre de faire un choix de réalisation.
Juste pour l'anecdote, j'ai étudié pour les fonctions Cj et Jc des solutions aussi farfelues que (ces solutions fonctionnent en simulation pour la fonction Cj) :

- réseau SIMD (GAPP de NCR et ELSA produit maison) ;
- réseau de neurones ;
- un processeur classique par case ;
- une EPLD par case, archi. à message ;
- une EPLD par case, archi. à feedback ;
- un réseau analogique (carrément) ;
- câblage total des équations (qq. milliers de portes).

Toutes ces solutions sont trop coûteuses, trop encombrantes, trop lentes ou carrément trop tordues.
Mais je vais vous livrer, au fil de plusieurs articles[4], quelques études plus sérieuses et que je compte essayer en grandeur réelle si je trouve le temps, l'argent et la main-d'œuvre.

---

4 (NDLR) Malheureusement...

6) L'association de modules câblés

On a vu avec la loi d'Amdahl que c'est la partie de l'algorithme traitée séquentiellement par la machine lente qui ralentit énormément les performances ; aussi il est préférable de faire traiter quasiment 100% de l'algorithme par les modules câblés en les reliant entre eux si possible, plutôt que de repasser par le maillon faible de la chaîne. Le chaînage et la gestion des modules feront l'objet d'un autre papier.

VIII. La solution retenue

La solution que je préconise n'est pas très originale :

- associer des modules câblés spécialisés à l'unité centrale la plus rapide possible ;
- l'unité centrale doit avoir le moins de boulot possible (Loi d'Amdahl) ;
- les modules câblés seront connectés entre eux de la façon la plus efficace possible.

IX. Un peu de topologie othellistique

Le jeu d'Othello est composé de 64 cases. Blanc et Noir vont jouer sur 60 d'entre elles au maximum. Chaque case peut influencer n directions avec le schéma suivant :

	a	b	c	d	e	f	g	h
1	3	3	5	5	5	5	3	3
2	3	3	5	5	5	5	3	3
3	5	5	8	8	8	8	5	5
4	5	5	8	0	0	8	5	5
5	5	5	8	0	0	8	5	5
6	5	5	8	8	8	8	5	5
7	3	3	5	5	5	5	3	3
8	3	3	5	5	5	5	3	3

Remarque : les cases centrales ne sont jamais jouables.

Pour chaque case, il faudra chercher dans les n directions pour voir si elle est jouable (Cj), et si elle l'est, chercher dans ces n directions pour voir s'il y a des pions à retourner (Jc).

Pour un calcul parallèle de tous les coups jouables, on doit considérer toutes les « lignes » à prendre en compte. Il y a dans un jeu d'Othello 38 petits Othello linéaires :

- 8 colonnes de 8 cases ;
- 8 lignes de 8 cases ;
- 22 diagonales dont 2 à 8 cases, 4 à 7 cases, 4 à 6 cases, 4 à 5 cases, 4 à 4 cases, 4 à 3 cases.

X. Les premiers modules rapides

1) Préliminaire : le codage de la position

La case d'un jeu d'Othello peut avoir trois états de configuration et un état supplémentaire :

- Vide (pas de pion sur cette case) ;
- Noir (un pion noir sur cette case) ;
- Blanc (un pion blanc sur cette case) ;
- Coup (celui qui a le trait joue sur cette case).

Une configuration de jeu peut donc être vue comme un mot de 64 nombres ternaires (base 3) et il y a donc au plus 3^{64} configurations possibles. Trouver les coups jouables c'est associer chacune de ces configurations à la liste correspondante des coups jouables, ceci est technologiquement impossible par une méthode directe.

On considère que le codage d'une case est effectué sur deux chiffres binaires (bit) :

Q = occupation et C = couleur :

Q	C	état correspondant au code
0	0	case vide
0	1	état supplémentaire en réserve
1	0	pion blanc
1	1	pion noir

Une configuration sera donc un mot de 128 bits. Ceci paraît énorme mais on verra que l'unité centrale n'aura jamais à manipuler ces mots.

2) Le module Cj : liste des cases jouables

On préfère l'approche « direction » à l'approche « case », c'est-à-dire qu'on ne va pas s'intéresser au fait qu'une case est jouable mais on va découper le jeu d'Othello 2D en 38 Othello 1D. On recherche les coups possibles dans nos Othello unidimensionnels. Une case intervenant dans plusieurs directions, il faudra combiner les résultats pour trouver la solution. On va scinder en deux le module Cj :

- Cj1 : savoir si une case est jouable ou pas (60 résultats) ;
- Cj2 : en fonction des 60 résultats, fabriquer une liste des coups valides.

Pour le module Cj1, on a besoin de 38 sous-modules correspondant à chaque direction. Une direction comporte N cases, donc 2N bits de codage ; avec 2N bits on peut avoir 2^{2N} configurations différentes qui doivent engendrer N résultats, c'est-à-dire lesquelles des N cases sont jouables.

Les plus gros othelliers unidimensionnels comportent :

- 8 cases à 6561 configurations différentes ;
- avec le code manipulable par la machine : 16 bits ;
- 65536 configurations possibles en tenant compte du codage ;
- 1 résultat par case, jouable ou non, codable chacun sur un bit.

Pour obtenir ces résultats, il suffit de construire une table de transfert ou LUT (Look-Up Table) à 16 adresses, donc 64 Kmots de 8 bits. Cette capacité est tout à fait réaliste avec les circuits classiques existants. On choisira de tabuler par exemple les coups de Noir (voir plus loin pourquoi).

Exemple : à l'adresse (suite de QC) :

11 10 10 00 00 00 10 11

on aura programmé la réponse :

0 0 0 1 0 1 0 0

Pour la table de transfert, on pourra choisir de l'EPROM ou de la RAM statique. En cas de choix de la RAM, il faudra prévoir un système initialisateur au démarrage de la machine.

Si on fait le bilan sur les 38 directions, il faut :

Avant association	Après association
12*64 Kmots de 8 bits	18*64Ko
6*16 Kmots de 6 bits	6*16Ko
4*8 Kmots de 6 bits	4*8Ko

4*4 Kmots de 6 bits	4*4Ko
4*1 Kmots de 5 bits	(ex : assoc. diag 5 et 3)
4*256 mots de 4 bits	(boîtiers standards)
4*64 mots de 2 bits	(Q=1 implicite case centre)

Tous ces boîtiers n'existant pas ou étant trop exotiques, les plus courants peuvent très bien être employés à la place (en association ou en sous-utilisation : cf. colonne de droite).

Une case est jouable si elle est jouable sur l'une des directions qu'elle influence, donc une fonction OU logique entre les résultats permet d'obtenir les 60 résultats recherchés par l'intermédiaire de 12 OU à 3 entrées et 44 OU à 4 entrées.

On a tabulé les coups de Noir. Comment connaître les coups de Blanc ? Si on remarque que les coups de Noir sont les coups de Blanc quand TOUS les pions sont inversés, il suffit de faire une inversion conditionnelle du bit représentant la couleur en fonction du résultat recherché. Si on appelle Coul le bit donnant la couleur du trait, il suffit de faire un ou exclusif de tous les bits C avec Coul avant d'attaquer les mémoires (8 boitiers de XOR).

Si le jeu d'Othello ne consistait qu'à extraire les coups valides, on aurait une machine capable d'examiner 22,2 millions de configurations par seconde pour la solution la plus rapide.

Le module Cj2 devant être optimisé pour être relié au module Jc, on l'exposera plus tard.

3) Le module Jc : jouer un coup

La philosophie générale est la même : on dispose de la configuration, du coup à jouer (toujours valide) et de la couleur qui joue le coup. Pour ne pas encombrer le codage et comme il nous reste un code inutilisé, on fait l'hypothèse que la case sur laquelle on joue est codée par Q=0 et C=1.

On divise encore le jeu en 38 sous-othelliers unidimensionnels et on va tabuler dans les mémoires les pions qui sont retournés par chaque coup.

Exemple : à l'adresse (suite de QC) : 11 10 10 10 01 10 11 11, on a le mot : 0 1 1 1 0 1 0 0

qui correspond aux pions à retourner si le coup est joué par Noir.

Pour prendre en compte les retournements de Blanc, il suffira de faire encore une inversion conditionnelle des bits C par Coul à l'entrée des mémoires.

Un pion est retourné s'il est retourné dans au moins une direction, donc le bit qui dit si un pion est retournable sera fabriqué par une porte OU sur les résultats de chaque direction l'incriminant (36 OU à 4 entrées).

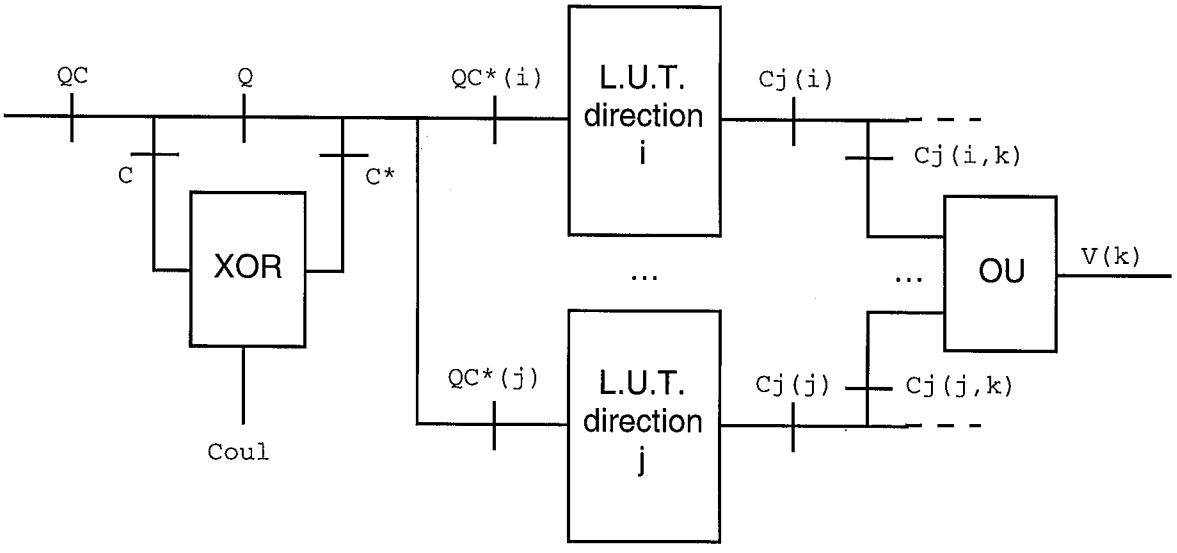
Pour fabriquer la nouvelle configuration, il suffit alors de faire un XOR entre le bit C et le bit indicateur de retournement pour chacune des cases (8 boitiers de XOR). Pour fabriquer le nouvel ensemble de bits Q, il suffit de faire sur les bits d'entrées un OU entre Q et C puisque seules les cases vides ont le code Q=0 et C=0.

XI. Et les prochaines fois ?

Comment fabriquer le module Cj2, comment brancher de façon efficace Cj et Jc. Comment générer l'arbre de jeu en hard, comment compter les pions de façon parallèle, le minimax, l'alpha-bêta, la gestion de la bête...

Schéma de principe du module Cj1

QC	bus configuration
C*	bus C modifié par Coul
QC*(i)	bus QC (C modifié) pour la direction i
Cj(i)	résultat pour la direction i
Cj(i,k)	résultat pour la case k dans la direction i
V(k)	indicateur de case k jouable pour Coul

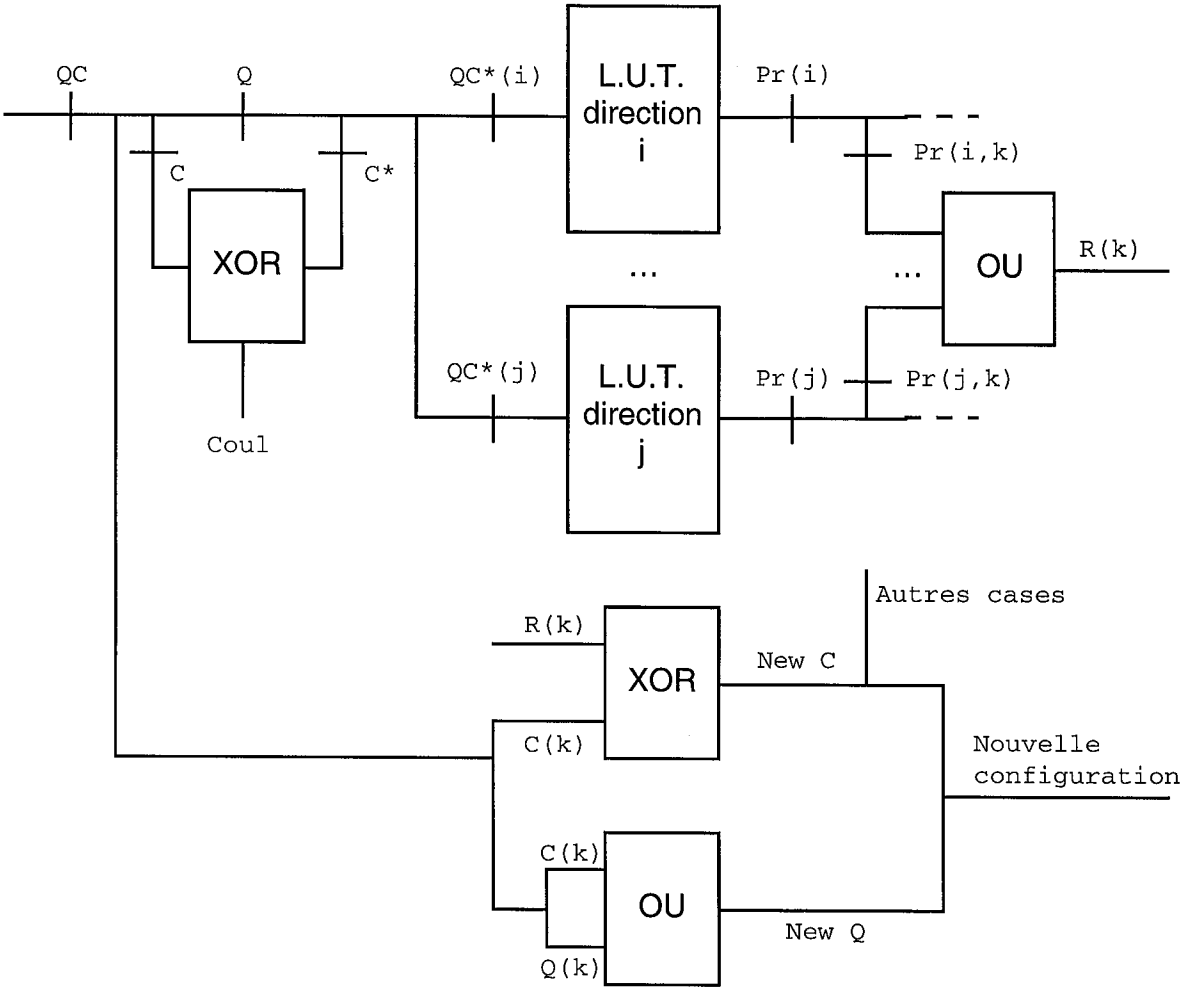


Encombrement : 46 boîtiers (variable en fonction de la technologie).
Prix : aux alentours de 2000 F chez un très bon fournisseur (pour des EPROMs).
Performances : en fonction de la technologie utilisée, mais aux limites :

- TTL-AS pour les portes et RAM 35 ns d'où 45 ns maximum pour extraire les coups valides ;
- TTL-AS pour les portes et EPROM 250 ns (la moins chère), d'où 260 ns maximum.

Schéma de principe du module Jc

Pr(i)	pions retournables pour la direction i
Pr(i,k)	case k retournable pour la direction i
R(k)	indicateur de case k retournable pour coul



Encombrement : 16 boîtiers XOR, 8 boîtiers OU, 32 mémoires 4 PALs pour les OU sur les directions.
Prix : comme Cj1 à quelques francs près.
Performances : Cj1 plus quelques nanosecondes.

Olympiades 1989

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

C&bek 34-30 Jotel

Olympiades 1989

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

Thor 25-39 C&bek

Thor, naissance d'une passion

par Thor

Au commencement, il y eut une revue, l'Ordinateur Individuel. Dans son numéro 1, d'octobre 1978, un programme d'Othello en BASIC.

Puis, il y eut les tournois que cette revue organisa à partir de 1979. Sylvain, bien que provincial à l'époque (tous ces choses-là se passant majoritairement à Paris) fut un lecteur intéressé, puis un spectateur passionné en 1980.

Enfin naquit mon grand-père, Thor version 1, écrit en assembleur Z80 (un processeur 8 bits, ce qui ne nous rajeunit pas) sur une machine de ce temps-là (un TRS-80 modèle 1 pour les connaisseurs). Il fut quatrième au tournoi de Septembre 1981, puis deuxième en Septembre 1982. Début 1983, la nuit descendit sur le monde (moui, j'en fais peut-être un peu trop là). En 1988, la rencontre avec un joueur coriace (tsoin-tsoin) relance la machine humaine. Sylvain participe à l'arbitrage du championnat du Monde de Paris, questionne habilement François Aguillon (qui lui répond encore plus habilement le traître !) et, à la fin de l'année, papa Thor version 2 sort des circuits intégrés d'un compatible PC (on progresse question puissance). Mars 1989 arrive, le premier tournoi, la première victoire (ex aequo) grâce (soyons honnête) à un cadeau de Paul Ralle pourtant largement gagnant. Puis les tournois s'enchaînent au danger du rythme cardiaque de l'auteur (la légende veut qu'il soit plus stressé quand je joue que mes adversaires humains (même Dominique (mais là, ça doit se valoir))). Puis, je sors, petite version 3, dotée à la naissance de tas de belles choses (évaluation paramétrable, base de données et autres colifichets dont je parle ici).

Pour commencer, une maniaquerie de l'auteur de mes jours, qui tient à dire régulièrement que son but est de faire un outil utilisable pour les joueurs humains et non une bête à tournois et à points ELO (quoique ce ne soit pas forcément incompatible). Maintenant, passons à la description de la chose (mais ça va faire des déçus).

Je suis écrit en langage C (un truc né au début des années 70) et je mesure (à ce jour) 6300 lignes de code (avec les colifichets auxquels j'ai fait allusion plus haut). J'utilise normalement un compatible PC. J'ai appris à utiliser les écrans graphiques de type CGA, puis EGA (et donc VGA) et donc à dessiner des pions ronds (avec un bel othellier vert en EGA et VGA). Je possède plusieurs niveaux de jeu, avec un temps fixé par coup qui va de 1-2 secondes à 99 heures, plus un niveau avec temps global pour l'ensemble de la partie (habituel en tournoi). J'utilise le classique algorithme alpha-bêta, déjà largement décrit ici, avec une petite amélioration : à chaque niveau de l'arbre, une fois que le premier coup est calculé (avec sa suite naturellement) j'utilise sa valeur comme bêta pour les coups suivants (et ceci pour tout l'arbre qui se trouve en dessous).

Sylvain a ajouté des fonctions que l'on trouve rarement dans des programmes écrits uniquement pour jouer en tournoi. Par exemple, les coefficients appliqués dans l'évaluation aux paramètres de mobilité, nombre de pions et position (cotes des cases de bords, X et C) sont partiellement modifiables par l'utilisateur (seulement en partie, on peut changer leur importance (de 50 % à 200 % de la valeur initiale) mais pas les inverser). Cela permet de changer le style de jeu (par exemple, mettre le nombre de pions à 200 % et les deux autres coefficients à 50 % donne un style de joueur qui minimise de préférence à toute autre stratégie (mais bon, pas de façon folle quand même), ou bien la mobilité à 200 % et le nombre de pions à 50 % donne un joueur qui joue surtout la mobilité et peut accepter de faire une grosse masse plus facilement que d'autres). Cela permet au joueur de s'entraîner contre différents styles de jeu et de lever (un peu) le reproche qu'on fait généralement aux programmes, à savoir qu'ils permettent de s'entraîner seulement à jouer contre eux et pas contre d'autres joueurs humains.

Autre fonction, celle d'évaluation de la position (cette appellation n'est d'ailleurs pas tout à fait adéquate). Généralement un programme vous dit (enfin, s'il parlait) : ce coup-là est le meilleur et il vaut tant de points. Maintenant, vous auriez plutôt joué tel autre coup et vous aimeriez savoir si c'est valable ou pas. Avec d'autres programmes, il vous faut alors le jouer puis voir ce qu'ils font après. Avec cette fonction, je calcule non seulement le (seul) meilleur coup pour moi, mais aussi les deux suivants. En examinant les cotes relatives, on peut facilement voir si le meilleur coup est vraiment loin des autres ou si, au contraire, il y a deux ou trois coups très proches (dont le vôtre justement non ?).

La dernière fonction (celle qui a pris plus de temps que les autres) est la base de parties. Mais je finirai par là.

La fonction d'évaluation est le cœur d'un programme d'Othello. Mais c'est probablement ce que j'ai de moins au point. En gros, on minimise (enfin un peu) les pions, on maximise la mobilité et on essaie de ne pas trop jouer les cases X et C (enfin, juste un peu moins que les coins quoi !). Sylvain n'a (snobisme ?) jamais été d'accord avec la technique de calcul des tables de bords « à la Comp'oth », mais n'a jamais vraiment été satisfait par aucune autre méthode. Donc, je me contente d'avoir quelques règles comme :

• les cases C, beurk ! sauf si

1) le coin est pris (soit c'est par moi, et c'est tout bon, soit c'est par l'adversaire et les possibilités d'insertions deviennent fortes) ;

2) l'autre case C sur le même bord est déjà prise et il y a plusieurs pions sur le bord (en gros, les bords de 6 et les bords où les deux joueurs sont en opposition et où il faut gagner un temps).

• les cases X, double-beurk ! sauf si

1) même chose que le point (1) ci-dessus.

• les coins, miam miam mais

1) on regarde quand même si on a des pions définitifs avec. Sinon, c'est déjà moins bien vu.

Dernier petit détail qui a surpris Dominique, les pions sont minimisés (enfin avant le coup 40), mais avec une petite modification qui fait qu'on minimise moins quand les coins commencent à être pris. En effet, je donne rarement des coins très tôt sans contrepartie équivalente, et généralement, à ce moment-là, il est souvent bon de faire des pions définitifs plus tôt dans la partie.

Passons au plus intéressant de l'avis du concepteur : la base de parties. À ce jour[1] plus de 24000 parties de tout niveau (mais bon, il y a quand même surtout des championnats du Monde, de France, des Internationaux, des parties japonaises, etc.).

Le point principal est que toutes ces parties ont été vérifiées après le coup 44 afin de savoir qui était réellement gagnant à cet instant précis. Cela permet quand on voit les statistiques en mode « théorique » de mieux juger d'une position en ne tenant pas compte des pertes au temps, des pertes sur gaffe finale (justement à cause du temps souvent (n'est-ce pas Dominique ?))...

Pour le reste, c'est assez simple. La base est prévue pour contenir 400 tournois (on en est à 320 à ce jour), 2000 joueurs (1044 pour l'instant) et 30000 parties. Bientôt va donc se poser le problème d'un changement de format. Justement, le format : suite à quelques demandes, et sachant le nombre d'informaticiens chez les joueurs, tous prêts à faire des extractions d'informations pour construire leur propre bibliothèque, ou simplement leur base, voici le format complet du fichier THOR.DBA.

1) Header : 68 octets comme suit :
   2 octets pour le nombre de parties de la base ;
   2 octets pour le nombre de joueurs dans la base ;
   64 non utilisés à ce jour.

2) Tournois : 400 enregistrements de 32 octets comme suit :
   30 octets pour le nom du tournoi (le dernier doit toujours être un zéro binaire (fin de chaîne de caractères en C)) ;
   2 octets pour le nombre de parties de ce tournoi.

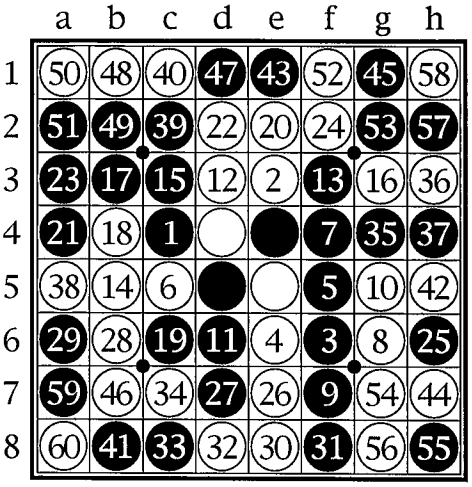
3) Joueurs : 2000 enregistrements de 20 octets comme suit :
   20 octets pour le nom du joueur (le dernier doit toujours être un zéro binaire (fin de chaîne de caractères en C)).

4) Parties : N enregistrements de 68 octets comme suit :
   2 octets pour le numéro de tournoi (de 0 à 399) ;
   1 octet pour le nombre de pions noirs du score final ;
   1 octet pour savoir qui aurait dû gagner après le coup 44 ('N' pour Noir, 'E' pour nul-Égal, 'B' pour Blanc) ;
   2 octets pour le numéro du joueur Noir (de 0 à 1999) ;
   2 octets pour le numéro du joueur Blanc (de 0 à 1999) ;
   60 octets pour les coups de la partie (toutes débutent par f5). Chaque octet a une valeur N, elle-même associée à une case par la méthode suivante :
      les cases a1 à h1 sont numérotées de 11 à 18 ;
      les cases a2 à h2 sont numérotées de 21 à 28 ;
      les cases a3 à h3 sont numérotées de 31 à 38 ;
      les cases a4 à h4 sont numérotées de 41 à 48 ;
      les cases a5 à h5 sont numérotées de 51 à 58 ;
      les cases a6 à h6 sont numérotées de 61 à 68 ;
      les cases a7 à h7 sont numérotées de 71 à 78 ;
      les cases a8 à h8 sont numérotées de 81 à 88.

   En fait, les cases d4 (44), e4 (45), d5 (54) et e5 (55) sont inutiles car elles ne sont jamais jouées puisque occupées dès le départ !

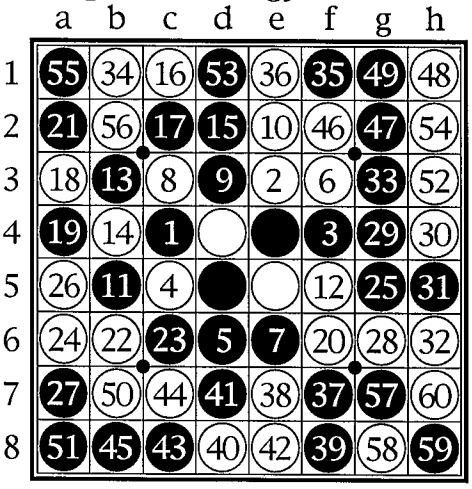
Voilà une rapide description de ce que je contiens. Nous n'aurons probablement pas la chance de nous rencontrer en tournoi, ayant décidé avec mon auteur que je prenais une retraite méritée (enfin, comme toutes les vedettes, je ferai un petit « come-back » à l'open de Bernissart dont je suis le tenant du titre[2]). Mais je m'aperçois que j'avais oublié une information importante : j'ai un petit frère en version Atari ST, adapté de mes gènes par un certain coriace faisable (je ne sais ce qui est le prénom et le nom)[3].

Pérenchies 1991



Thor 31-33 Othel du Nord

Open de Cergy 1991



Nicolet 41-23 Thor

[1] (NDLR) Au 1er janvier 1995

[2] (NDLR) Au 1er janvier 1992.

[3] (NDLR) Et également une version pour Apple Macintosh et une sous Unix.

Othel du Nord

par Jean-Claude Delbarre

Je vais essayer, dans cet article, de mettre à nu Othel du Nord, espérant que cela aidera certains à écrire leur propre programme. ODN est né de la fascination exercée sur moi par le programme Comp'oth et par les articles de son génial auteur, François AGUILLON. ODN, comme l'immense majorité des programmes, utilise un algorithme de type alpha-bêta couplé à une fonction d'évaluation.

I. L'alpha-bêta

ODN explore l'arbre de recherche sur les niveaux pairs à partir de 4. Il stocke les 3 premiers niveaux, plus le « coup meurtrier » du niveau 4, soit en fait les 4 premiers niveaux. Il va de 2 en 2 car une position étudiée est totalement différente selon qui a le trait, et il m'a paru trop difficile de trouver une fonction équivalente pour chacun des 2 cas. J'ai choisi trait à ODN, mais on aurait aussi bien pu choisir trait à l'autre. Cela tient au fait que mon « objectif » était la profondeur 8, il y a deux ans.

Au premier niveau de l'arbre, ODN « élargit » sa fourchette selon la profondeur étudiée : ainsi, si l'appel à profondeur d'étude 4 au niveau 1 est `min(bêta, m-200)`, à profondeur 6 elle est de `min(bêta, m-50)` et à profondeur 8 de `min(bêta, m-20)`, ce qui présente l'avantage de n'étudier que les coups « envisageables » et de les trier. Dans le cas où un coup est seul « nettement meilleur », il est joué instantanément (à profondeur 6 seulement parfois). Il évite ainsi de « perdre du temps ». Dans ce même but, si une recherche est lancée à profondeur 10, il étudiera au moins les deux meilleurs coups du niveau 8 (bien sûr, une telle recherche ne sera lancée qu'après estimation du temps basée sur une divergence de 3,5).

Des essais sur la largeur de la fenêtre d'étude m'ont montré qu'il était plus « rentable » de travailler avec une telle fenêtre, plutôt que d'utiliser la *zero-window*.

Curieusement, j'obtiens de meilleurs résultats en prenant comme coup meurtrier pour chaque niveau supérieur à 4 le coup meurtrier du même niveau mémorisé lors du dernier appel, plutôt qu'en établissant une table de coups meurtriers « réponses » au dernier coup joué !

II. La fonction d'évaluation

Elle privilégie tout ce qui peut s'évaluer de manière incrémentale dans l'arbre.

La frontière bien sûr, évaluée de la manière décrite par J.F. PUGET dans un article précédent, dont l'importance décroît en cours de partie.

La mobilité également, qui présente l'originalité (?) de ne pas tenir compte du nombre de coups jouables, mais du nombre de zones où l'on peut jouer (l'othellier est découpé en huit zones, plus les diagonales). Moins précise que l'évaluation de la mobilité adverse (dernier niveau de l'arbre), celle de l'ordinateur est constituée à l'avant-dernier niveau (chronomètre oblige !).

Une parité primaire (othellier divisé en quatre carrés), comptant le nombre de cases vides, associée aux coups jouables dans chaque zone et à l'occupation des bords, utilise aussi ces données de l'arbre.

Non incrémental, l'inévitable (n'est-ce pas Sylvain ?) tableau de bords. À noter que je me contente d'un tableau à huit cases seulement, les cases X étant étudiées séparément par une espèce « d'alpha-bêta local ».

Un zeste de connexité (tableau de lignes et diagonales, comme pour les bords), contrôle des grandes diagonales et c'est tout.

Manque encore (mais ça ne saurait tarder) une estimation des pions définitifs (qui a déjà coûté pas mal de parties au dernier coup de type « milieu de partie »).

Mais je crois que la force d'Othel du Nord vient de son « évaluation au niveau zéro ». Avant de lancer l'alpha-bêta, chaque coup jouable est affecté d'une note de « niveau zéro » qui sera ajoutée à sa valeur évaluée par l'alpha-bêta. Inspirée par l'article de Paul Ralle dans *Fforum 4*, elle tient compte des conseils de ce champion : par exemple, un coup ou l'on peut jouer seul sera pénalisé par rapport à un coup ou l'on peut jouer à deux ; la prise d'un bord en premier sera pénalisée, de manière différente d'ailleurs selon sa position sur ce bord. La parité sérieuse peut aussi être étudiée à cet endroit, puisque l'on peut à ce niveau se permettre d'utiliser un temps extraordinairement grand, voire un dixième de seconde !

Toujours avant de lancer l'alpha-bêta, cette étude permet également de modifier les coefficients de la fonction d'évaluation : par exemple, si l'adversaire a une très grande frontière, les coefficients de mobilité seront augmentés, ceux de frontière diminués (système « anti-béton »). Bien sûr, tout ceci est fait pour être satisfaisant dans la majorité des cas rencontrés et je modifie tous ces paramètres après chaque défaite (oui, ça fait du boulot !).

Une des choses que je demande aussi à ma fonction d'évaluation, c'est d'aider à élaguer plus vite ! En effet, si j'enlève la note de frontière par exemple, les parties sont sensiblement pareilles, mais avec cinq fois plus de temps pour la même profondeur !... étonnant, non ?

À noter que lors de la recherche du meilleur coup (31-33), l'arbre est préévalué avec une fonction très différente : plus de frontière, mais des bords, des zones jouables, les diagonales et la parité.

III. L'ouverture

Extrêmement importante, elle fait la différence à partir d'un certain niveau. Le premier avantage, immédiat, est évident : plus on va loin dans l'ouverture, plus il nous reste de temps (ah ! cette obsession !) pour le reste. Le second, beaucoup plus difficile à mettre en œuvre, est qu'elle permet de sortir de l'ouverture avec un type de partie adapté au style de jeu du programme. Depuis l'année dernière[1], ODN essaie d'utiliser pour ses ouvertures la base de données de Sylvain, mais cela nécessite un tri sur la qualité des joueurs concernés et sur leur style de jeu. Il utilise donc pour ses ouvertures un mélange de cette base et de sa précédente bibliothèque

1 (NDLR) c'est-à-dire depuis fin 1991.

(mais c'est encore loin d'être au point), avec ajout des parties gagnées au fur et à mesure, et retrait des variantes perdues. Mais c'est vraisemblablement là que se trouve l'amélioration la plus urgente à apporter à ODN, car ses ouvertures actuelles sont très loin d'être satisfaisantes !

IV. La gestion du temps

ODN possède pour chaque coup un tableau « limite » à ne pas dépasser en fonction du numéro du coup, ce qui fait qu'il peut en général consacrer beaucoup de temps au sortir de l'ouverture. Lorsqu'un coup est « évident », seul envisageable pour lui, il le joue immédiatement. Cette gestion fait que le temps consacré à un coup peut varier d'une ou deux secondes à une dizaine de minutes (dans le cas de plusieurs coups « proches » à profondeur 8, mais c'est souvent là que se joue une partie).

Bien évidemment, la réflexion sur temps adverse est un atout dont on ne peut plus se passer.

En conclusion, je constaterai l'énorme travail qu'il y a encore à effectuer pour qu'ODN joue comme je le souhaite. Je dirai qu'une multitude de directions s'ouvre encore devant chaque programmeur et qu'il me paraît impossible qu'un seul les explore toutes. Alors, qui trouvera l'arme absolue ?

Hommes-Machines 92

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

Penloup 36-28 Othel du Nord

St-Michel sur Orge 1990

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

Othel du Nord 57-7 Moore

Paderborn 1993

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

Othel du Nord 31-33 Keyano

Pérenchies 1991

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

Jacpoth 32-32 Othel du Nord

Courchelettes 1994

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

Othel du Nord 33-31 Théole

À propos des bibliothèques d'ouvertures

par Jean Delteil (auteur de Spock)

Bon nombre de programmeurs éprouvent des difficultés diverses à gérer une bibliothèque d'ouvertures. Le terrible dilemme se pose souvent en ces termes :

• Utiliser les ouvertures standard ? Elles sont fausses et incomplètes, car mises au point par des humains.

• Faire calculer les coups par le programme ? « Oui, mais ma fonction d'évaluation n'est pas... »

Je vous propose de combiner astucieusement ces deux possibilités et donc les deux défauts !

Tout d'abord, il faut se munir d'une petite base de données, qui servira à enregistrer des POSITIONS de jeu. Oui, j'ai bien dit POSITION, car cela permet entre autres de gérer plus facilement les interversions de coups (plus nombreuses qu'il n'y paraît). Pour chaque position on aura par exemple au moins les champs suivants :

• N° de version (voir ci-après).
• Coup conseillé (le coup à jouer dans la position).
• Cote de ce coup (cf. votre fonction d'évaluation).
• Profondeur de recherche du coup.
• Description de la position.
• Index de la position fille (issue du coup conseillé).
• Index de la position parente (1).
• Index de la position parente (2).

Ces deux derniers champs permettent de retrouver la ou les position(s) parente(s). S'il existe plus de deux positions parentes, il faut faire pointer un des index vers un enregistrement fictif qui contiendra deux autres index, et ainsi de suite...

Je vous laisse le soin d'optimiser jusqu'à plus soif le stockage et l'indexation.

Calcul des coups par le programme

Le principe est simple (si !).

Supposons que nous ayons déjà dans la base les positions suivantes :

Position	Conseil	Cote
Départ	f5	10
(f5)	d6	-10
(f5d6)	c3	10

Nous voulons ajouter alors la position issue de (f5f6). Le programme calcule le coup (alpha-bêta standard) pour la position (f5f6) qui donne une cote de 5. On ajoute donc ce résultat. Ce qui donne :

Position	Conseil	Cote
Départ	f5	10
(f5)	d6	-10
(f5d6)	c3	10
(f5f6)	e6	5

Mais, ce n'est pas tout ! Propageons donc ce résultat sur les positions parentes. On utilise alors encore un alpha-bêta, mais cette fois-ci, au premier niveau de recherche on « regarde » si la position n'est pas dans la base et, si c'est le cas, on récupère la cote (apprentissage !).

Poursuivons notre exemple : on réévalue la position parente, c'est-à-dire celle issue du coup (f5) :

Soit trois coups à tester (f6, d6, f4).

• Test de f6 :
La position est trouvée dans la base pour cote 5, qui donne -5 à f6.
• Test de d6 :
La position est trouvée dans la base pour cote 10, qui donne -10 à d6.
• Test de f4 :
La position n'est pas trouvée, alpha-bêta normal cote de -30.

La position se voit attribuer une valeur de max(-5, -10, -30), soit -5. Le conseil est remplacé par le meilleur coup (f6). La base de données devient alors :

Position	Conseil	Cote
Départ	f5	10
(f5)	f6	-5
(f5d6)	c3	10
(f5f6)	e6	5

La position parente doit encore être évaluée (position de départ), qui lui donne une cote de 5 (le conseil f5 ne change pas). La base de données devient alors :

Position	Conseil	Cote
Départ	f5	5
(f5)	f6	-5
(f5d6)	c3	10
(f5f6)	e6	5

Il s'agit en fait d'utiliser l'alpha-bêta (ou autre) en tenant compte de positions pouvant être présentes dans la base de données. Chaque position trouvée fait gagner au moins un niveau de profondeur.

Règle pour la propagation : la propagation vers les positions parentes continue si — et seulement si — la cote de la position en cours est modifiée.

Règle pour l'évaluation d'une position parente :

Soit CF = -la cote de la position fille, MF le coup joué pour l'obtenir, CP la cote de la position en cours, MP le conseil de la position en cours.

```
Si MF= MP alors
     Si CF < CP alors
            recalculer alpha-bêta avec
            alpha = CF et bêta = CP.
     Sinon
```

```
            On prend directement le
            résultat : CP = CF.
Finsi
sinon
     Si CF <= CP alors
            La position ne change pas
            (cote et conseil).
     Sinon
            On prend directement le
            résultat : CP = CF, MP = MF.
     Finsi
Finsi.
```

Cette règle permet d'éviter de nombreux appels inutiles à une procédure alpha-bêta. On utilise la propriété que la cote du conseil était un maximum.

Apport des ouvertures standards

Pour faire fonctionner tout ça on pourrait se contenter de faire tourner le programme, par exemple, sur la ligne de jeu principale de la base de données (suite des conseils). La position de chaque coup conseillé est analysée puis stockée dans la base. La base grossissant au fur et à mesure des changements de conseil. À cela un inconvénient : par exemple, si le programme considère que le meilleur coup après (f5d6) est c5, la base de données n'aura pas enregistré la position issue de (f5d6c3). Pour y remédier, j'utilise un fichier qui indique les variantes à « travailler ». Ce fichier est constitué par documentation (*Fforum*, parties perdues, etc).

Remarques

• Un problème majeur se pose quand on modifie la fonction d'évaluation. C'est là qu'intervient le numéro de version. Il permet des moulinettes en tout genre.

• Les calculs sont relativement longs (propagations) un disque RAM n'est pas superflu.

• Un autre phénomène, le « rejet », est également ennuyeux. Supposons (encore !) qu'après le coup f5 et force calculs les réponses f6 et d6 soient mal cotées et que la base de données conseille provisoirement f4 jusqu'à plus ample informé, juste la veille du tournoi... L'exemple est pris ici au deuxième coup pour plus de clarté mais le mal est partout... J'ai même vu des cas où le coup 20 propage sa cote jusqu'à la racine.

Des choses rassurantes apparaissent tout de même : beaucoup de lignes classiques sont retrouvées.

• Une condition de fond est nécessaire concernant la fonction d'évaluation (c'est trivial, mais...) : la cotation doit appartenir à une même échelle de valeurs, quel que soit le nombre de pions présents sur l'othellier. En effet, dans cette méthode les cotes comparées ne sont pas forcément obtenues pour une même profondeur d'analyse.

Conclusions

La méthode me paraît intéressante car elle possède des possibilités d'apprentissage automatique ou dirigé qui permettent de remettre perpétuellement et progressivement en question les lignes d'ouvertures de la bibliothèque, tout en enregistrant les essais infructueux sans toutefois envisager les coups idiots.

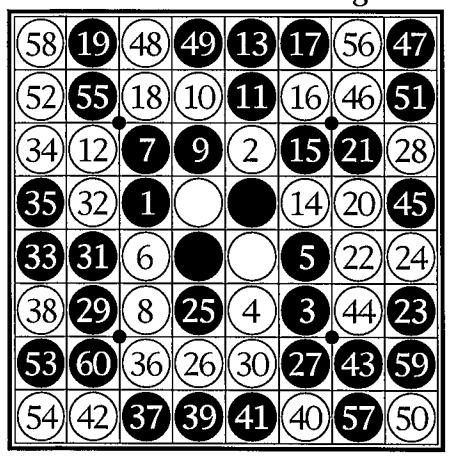
Je ne veux plus entendre dans les couloirs : « j'étais mal après l'ouverture ».

Bibliographie

« Some studies in machine learning using the game of Checkers », Samuel A. L., IBM Journal of research and devel. (Vol 3, 1959, p. 211-229).

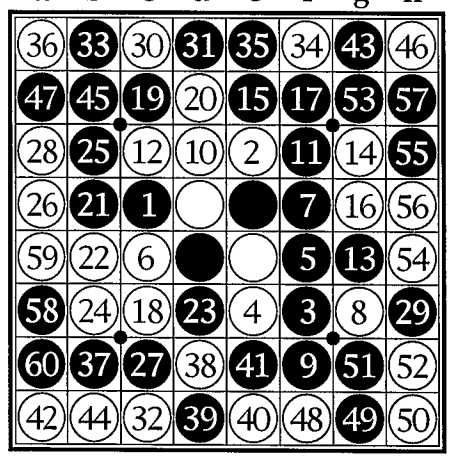
« Some studies in machine learning using the game of Checkers recent progress », Samuel A. L., IBM Journal of research and devel. (Vol 11, nov 1967, p. 601-617).

Championnat de France 1998



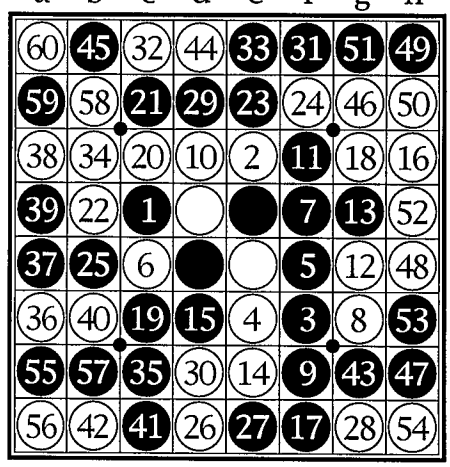
Spock 30-34 Turtle

Thématique semi-rapide 1999



Tastet 32-32 Spock

Thématique semi-rapide 2001



Spock 30-34 Turtle

L'apprentissage des ouvertures chez Logistello

Par Michael Buro

La force des programmes jouant à des jeux de réflexion a considérablement augmenté ces dernières années. Cela a été rendu possible, d'une part, par l'augmentation de la profondeur de recherche due à l'accéleration du matériel et aux raffinements des algorithmes de recherche dans les arbres et, d'autre part, par la mise au point de meilleures fonctions d'évaluation qui sont capables d'estimer les chances de gains à la fois précisement et rapidement. Dans les jeux tels que les Échecs, le Go ou Othello, la phase d'ouverture est très importante. Or c'est là que repose une des faiblesses connues des algoritmes de jeu, sans doute due à la difficulté de concevoir des plans stratégiques. D'où l'utilisation des bibliothèques d'ouvertures pour circonvenir à ce problème. Jusqu'à récemment on n'attachait pas beaucoup d'intérêt à la génération automatique de telles bibliothèques, dans la mesure où on pouvait trouver les « bonnes » suites de coups dans la littérature, les rentrer manuellement dans sa machine en vérifiant éventuellement que son programme était capable de dominer les complications tactiques qui en résultaient, et en changer si nécessaire après chaque partie perdue. Cependant, la mise en œuvre de serveurs de jeux (en particulier sur Internet) sur lesquels les programmes connectés peuvent jouer contre d'autres programmes ou des humains toute la journée, et la multiplication du nombre des parties qui en a résulté, ont rendu nécessaire que les programmes puissent modifier dynamiquement leur répertoire d'ouvertures.

Cet article décrit un algorithme qui trouve des alternatives raisonnables dans l'ouverture. Il est basé sur une idée simple mais efficace : refuser de perdre deux fois la même partie. De plus, il est même capable, si on le laisse tourner assez longtemps, de battre un adversaire plus fort qui n'aurait pas ses capacités d'apprentissage, voire de corriger automatiquement une bibliothèque d'ouvertures existante en en trouvant les faiblesses.

Stratégies pour une suite de parties.

Si un joueur électronique veut non seulement pouvoir être efficace dans une partie précise contre un adversaire dont il ne sait rien, mais aussi dans une séquence de parties contre cet adversaire, il devra sans doute résoudre les problèmes particuliers que posent les stratégies de match, simples mais néanmoins efficaces, que ne peuvent pas contrer les techniques connues de recherche dans les arbres de jeu (minimax, alpha-bêta, etc.). La plus évidente et la plus simple de ces stratégies est la suivante :

I. Essayer de rejouer les parties gagnées.

Un programme dépourvu de mécanisme d'apprentissage et déterministe suit cette stratégie, mais, d'un autre côté, en est en même temps victime, puisqu'il ne sait pas dévier des parties qu'il a perdues. La première idée pour remédier à ce problème est l'utilisation d'une biblothèque d'ouverture avec une composante aléatoire pour choisir les coups. Mais cette approche n'est pas très flexible, ni d'un grand intérêt sur le plan théorique. De plus, ce n'est pas vraiment une solution puisque la probabilité de perdre augmente après chaque défaite (si l'adversaire suit la stratégie I). Pour contrer la stratégie I, il est nécessaire de savoir trouver de bonnes alternatives. Une manière de le faire, passive mais qui marche néanmoins, est de suivre le principe suivant :

II. Essayer de replacer à l'adversaire les variantes avec lesquelles il vous a battu.

L'idée derrière cette méthode élégante est, en quelque sorte, de laisser l'adversaire vous montrer vos propres fautes afin de jouer vous-même la prochaine fois les coups gagnants qu'il aura trouvés. Même un adversaire largement plus fort, mais dépourvu de mécanisme d'apprentissage, peut être mis en difficulté de cette manière puisque, à terme, il jouera contre lui-même. Comment peut-on battre la stratégie II ? Pour se montrer supérieur, il est nécessaire d'être inventif, pour surprendre l'adversaire dans les parties suivantes. Ce qui mène finalement au troisième principe :

III. Si une position a déjà été rencontrée mais qu'aucun coup gagnant n'est connu, alors chercher des coups de rechange prometteurs.

Un programme qui suit ces stratégies est un adversaire très déplaisant à jouer, dans la mesure où il essaye de replacer ses victoires et celles que vous lui avez infligées, où il ne perd jamais deux fois de la même manière, où il apprend les grands schémas d'ouverture qu'il n'aurait peut-être jamais trouvés par lui-même, où, enfin, il corrige ses erreurs dans les parties successives. De plus, un tel programme peut vérifier la qualité de ses innovations avant les tournois en jouant contre lui-même.

L'algorithme de génération de la bibliothèque d'ouvertures

Les stratégies I et II peuvent être implémentées facilement : si le programme garde trace de toutes ses parties perdues jusqu'à un instant donné, il peut, à partir d'elles, reconstruire un arbre de jeu dans lequel les résultats des parties sont propagées des feuilles vers la racine suivant le principe du Minimax. Quand il joue une partie, il lui suffit alors de rechercher dans l'arbre ainsi construit la position courante : si elle a déjà été rencontrée et qu'elle est étiquetée comme gagnante, on connait un coup gagnant dans l'arbre. Sinon, c'est soit qu'elle est nouvelle (auquel cas le programme effectue une recherche alpha-bêta normale pour trouver un coup), soit qu'elle est connue mais étiquetée comme perdante ou nulle. Dans ce cas, la stratégie III indique qu'il faut regarder s'il n'y a pas des alternatives plus prometteuses. Bien sûr, ce procédé ne doit pas concerner que la position en cours

puisque la faute peut avoir été commise plus tard. Il est par conséquent nécessaire de connaître les évaluations heuristiques des coups qui ne sont pas encore joués pour effectuer le meilleur choix global.

Pour faire marcher concrètement l'algorithme, on définit (récursivement) ci-dessous la *valeur déviante* V_D d'une position : c'est la valeur de la variante que suivrait, à partir de cette position, deux joueurs appliquant les stratégies I, II et III.

Étant donné un arbre de jeu T construit à partir de toutes les parties déja jouées, notons $V_T(v) \in \{Perte, Nulle, Gain\}$ *la valeur minimax de la position v (pour la couleur qui a le trait) découlant des résultats des parties de T, S(v) l'ensemble des successeurs de v dans T et S'(v) l'ensemble des successeurs de v qui ne sont pas dans T.*

Alors la valeur déviante $V_D(v)$ *d'une position est définie comme suit :*

• *si v n'est pas dans T alors* $V_D(v)$ *est l'évaluation heuristique de v.*

• *pour une position terminale v dans T,*

si $V_T(v) = Gain$, $V_D(v) = +\infty$
si $V_T(v) = Nulle$, $V_D(v) = 0$
si $V_T(v) = Perte$, $V_D(v) = -\infty$

on modélise ainsi le résultat exact de la partie.

• *si v n'est pas une position terminale dans T, alors*

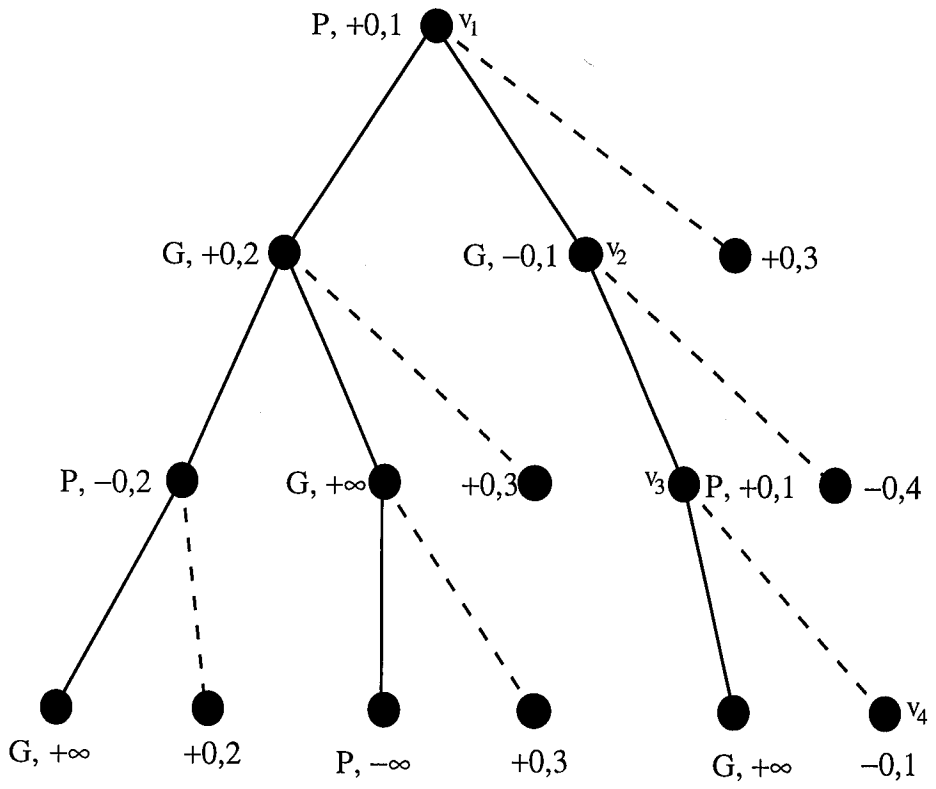
si $V_T(v) = Gain$, $V_D(v) = \max\{-V_D(w) \mid w \in S(v)$ *et* $V_T(w) = Perte\}$
si $V_T(v) = Nulle$, $V_D(v) = \max\{-V_D(w) \mid w \in S(v)$ *et* $V_T(w) = Nulle$ *ou* $w \in S'(v)\}$
si $V_T(v) = Perte$, $V_D(v) = \max\{-V_D(w) \mid w \in S(v)$ *et* $V_T(w) = Gain$ *ou* $w \in S'(v)\}$

Dans chacun des trois cas, la valeur déviante est calculée récursivement à partir des successeurs de v qui ont mené aux meilleurs résultats dans les parties réelles. Cela assure de suivre la stratégie II. Si la position considérée n'a mené qu'à des nulles ou à des pertes, alors on regarde en plus dans toutes les alternatives pour choisir heuristiquement la meilleure suite.

La figure montre un exemple d'arbre de jeu T construit à partir de trois parties. Les positions sont étiquetées, d'une part, par les valeurs minimax $V_T(v)$ (G pour Gain, N pour Nulle et P pour Perte) issue de T et, d'autre part, par les valeurs déviantes $V_D(v)$, les alternatives heuristiques étant dessinées avec des pointillés. À la position racine v_1 l'algorithme doit choisir entre les deux coups « perdants » et l'alternative heuristique. Si l'on suppose que les deux joueurs suivent les stratégies décrites plus haut, alors la séquence optimale est $v_1,\ldots, v_4$. À cette position v_4 le programme considère qu'il y a de l'espoir puisque l'évaluation heuristique donne –0,1 pour l'adversaire. Par conséquent, en v_1 il va choisir le second coup « perdant » et dévier des lignes connues en v_3.

Avant d'utiliser cet algorithme, il est nécessaire de répondre à deux questions : quelles parties utiliser pour construire l'arbre T, et comment évaluer les alternatives ?

La réponse à la seconde question est claire. Il faut bien évidemment avoir une évaluation qui permettent de comparer les notes de différentes phases de la partie : le mieux semble être d'avoir une fonction d'évaluation dont l'interprétation soit globale, par exemple une estimation de la probabilité de gain ou de l'espérance de la différence de pions finale. Ceci posé, on peut, pour



chaque position de la partie en cours d'analyse, chercher les notes (heuristiques) des alternatives éventuelles en faisant une recherche normale de milieu de partie, de préference à une profondeur au moins égale à celle que l'on atteint en tournoi pour éviter les mauvaises surprises.

La première question pose des problèmes plus subtils. Pour les parties perdues ou nulles, pas de problème : elles seront analysées et incluses dans l'arbre sans aucune restriction, puisque les premières seront répétées avec couleurs inversées, tandis que les secondes ne devront pas être rejouées contre un adversaire plus faible. Mais que faire des parties gagnées ? Afin d'éviter l'évaluation de nombreuses positions claires, il faut tester, avant d'importer une partie gagnée, si l'adversaire n'a pas eu, à un moment de la partie, une possibilité prometteuse (*i.e.* avec une forte note), auquel cas la partie doit être examinée puisque c'est une défaite potentielle. De telles parties peuvent être incluses dans l'arbre d'ouverture et corrigées en faisant jouer le programme contre lui-même. Cette pratique tend cependant à biaiser la bibliothèque d'ouvertures, dans la mesure où seules les parties les plus récentes sont rejouées. On peut, pour réduire cet effet néfaste, rechercher de temps en temps dans toute la bibliothèque d'ouvertures les positions « suspectes » qui ne sont pas gagnantes mais auxquelles existent de bonnes alternatives : les parties correspondantes seront finies pour les corriger.

Discussion

Nous avons présenté dans cet article un algorithme qui permet d'apprendre à partir de ses parties, de telle sorte que des variantes raisonnables peuvent être trouvées pour éviter de perdre deux fois la même partie, et que même des adversaires plus forts peuvent être mis en danger en utilisant leurs coups gagnants. Le temps nécessaire pour évaluer les alternatives possibles dans une partie est du même ordre de grandeur que celui pris pour la partie elle-même si la même profondeur est utilisé, et est, par conséquent, acceptable.

En pratique, cette nouvelle technique est implementée dans Logistello, l'un des plus forts programmes d'Othello actuels, sous une forme cependant légérement modifiée : Logistello joue pour gagner, et donc compte les nulles comme des défaites pour éviter leur répétition. Connecté au serveur Internet d'Othello (IOS), il a démontré ces derniers mois sa force au cours de centaines de parties contre d'autres programmes qui étaient aussi équipés de procédés d'apprentissage, en leur resservant leurs propres coups ou innovations.

Remerciements

Je tiens à remercier toutes les personnes qui ont pris part, sur IOS, aux discussions sur les mécanismes d'apprentissage. Les défaites de Logistello contre les forts programmes de l'IOS ont motivé ce travail.

Traduction : Stéphane Nicolet

Références

M. Buro, « *Techniken für die Bewertung von Spielsituationen anhand von Beispielen* », thèse de Ph.D., 1994, Université de Paderborn, Allemagne.

A. L. Samuel, « *Some studies in marchine learning using the game of checkers* », *IBM Journal of Research and Development* **3** (1959), 210-229.

T. Scherzer, L. Scherzer, D. Tjaden, « *Learning in Bebe* », *in* T. A. Marsland et J. Schaeffer (eds.), *Computer, Chess and Cognition*, Springer-Verlag (1990).

Partie Internet 1993

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

Logistello 38-26 Kitty/Rev

Princeton 1997

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

Hannibal 31-33 Logistello

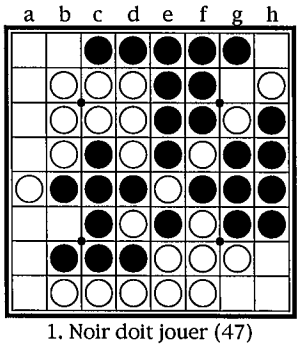
Princeton 1998

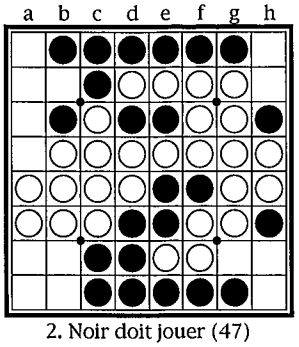
	a	b	c	d	e	f	g	h
1	57	44	43	42	46	52	60	58
2	59	54	47	41	35	33	53	31
3	39	45	22	24	2	12	16	29
4	40	26	1			7	15	27
5	51	17	6			5	13	14
6	28	30	25	10	4	3	8	19
7	50	48	37	18	11	9	32	56
8	49	38	23	20	21	36	34	55

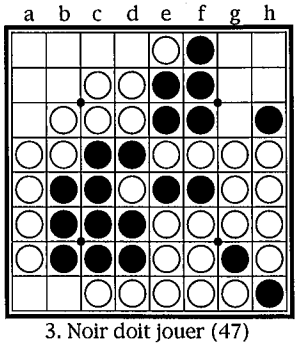
Logistello 31-33 Hannibal

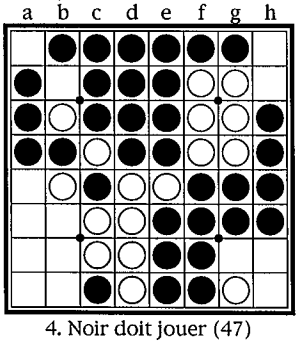
Tests de finales

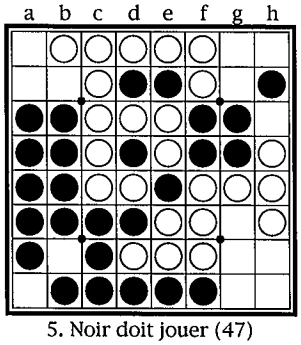
Voici trois grilles de tests de finales pour programmes d'Othello. Les positions 1 à 19 ont été publiées en 1989, comportent de 14 à 16 cases vides et sont donc plus faciles à résoudre que les positions 20 à 39, destinées à tester les programmes les plus performants de 1995. La dernière grille (diagrammes 40 à 59) date de 1999.

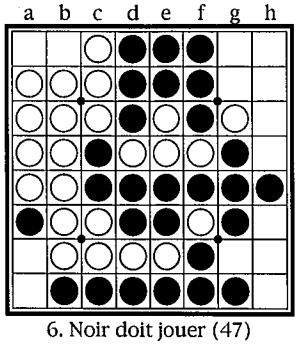

1. Noir doit jouer (47)

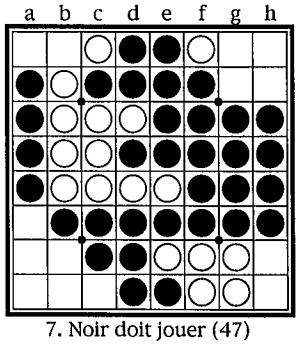

2. Noir doit jouer (47)

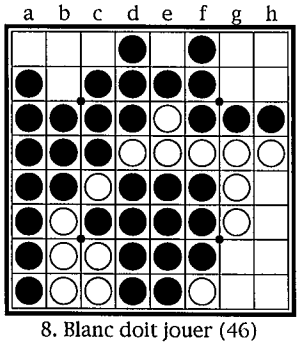

3. Noir doit jouer (47)

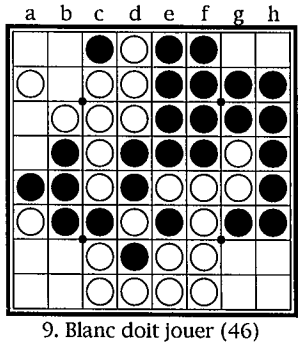

4. Noir doit jouer (47)

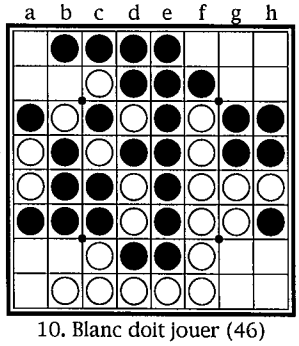

5. Noir doit jouer (47)

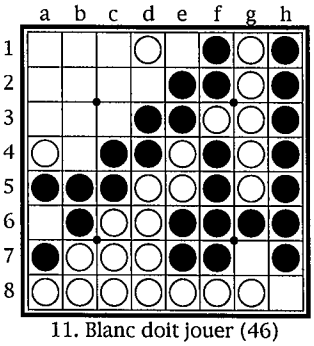

6. Noir doit jouer (47)

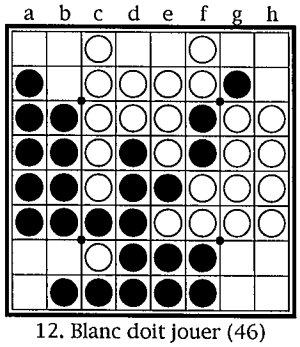

7. Noir doit jouer (47)

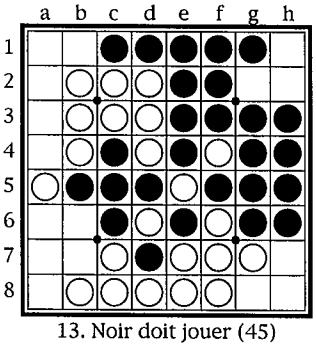

8. Blanc doit jouer (46)

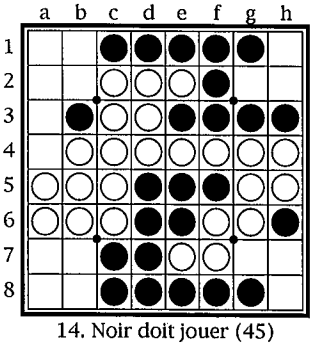

9. Blanc doit jouer (46)

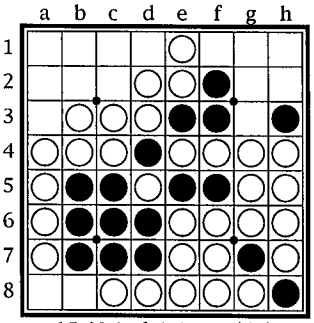

10. Blanc doit jouer (46)

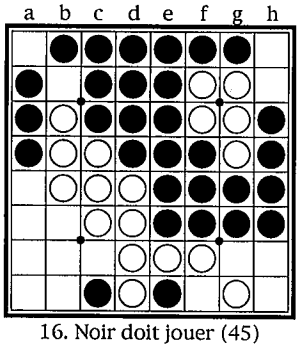

11. Blanc doit jouer (46)

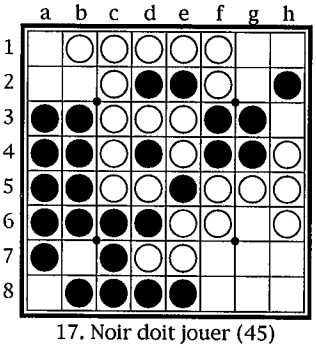

12. Blanc doit jouer (46)

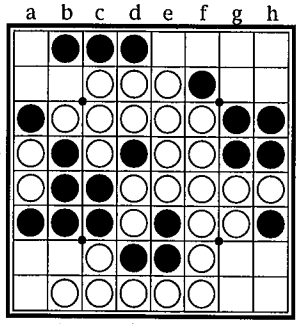

13. Noir doit jouer (45)

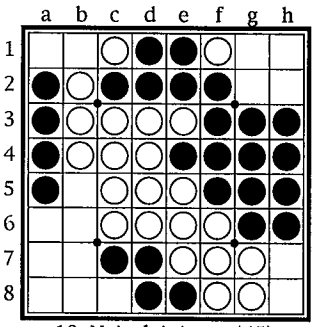

14. Noir doit jouer (45)


15. Noir doit jouer (45)


16. Noir doit jouer (45)

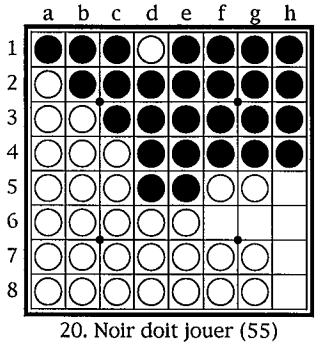

17. Noir doit jouer (45)

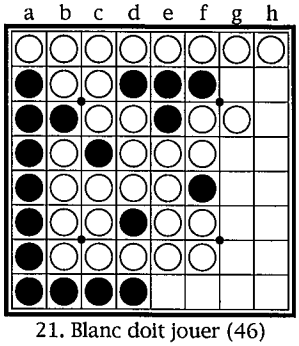

18. Noir doit jouer (45)

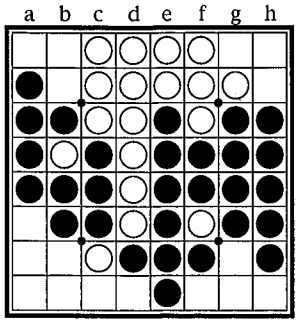

19. Noir doit jouer (45)

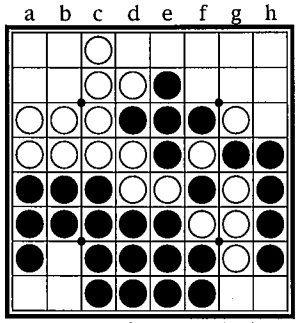
Mondial 1984

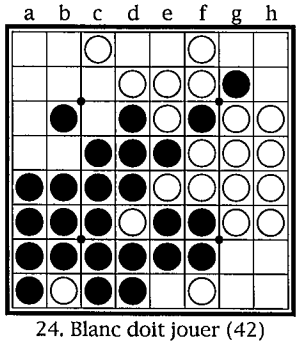
1. Ralle - Taniguchi
2. Brusca - Ralle
3. Taniguchi - Ralle
4. Sharman - Ralle
5. Taniguchi - Landau
6. Landau - Taniguchi
7. Ralle - Taniguchi
8. Brusca - Taniguchi
9. Landau - Sharman
10. Landau - Taniguchi
11. Taniguchi - Ralle
12. Sharman - Landau
13. Ralle - Taniguchi
14. Brusca - Ralle
15. Taniguchi - Ralle
16. Sharman - Ralle
17. Taniguchi - Landau
18. Landau - Taniguchi
19. Ralle - Taniguchi

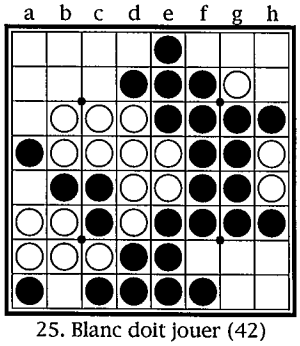

20. Noir doit jouer (55)

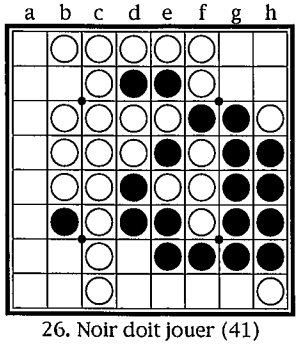

21. Blanc doit jouer (46)

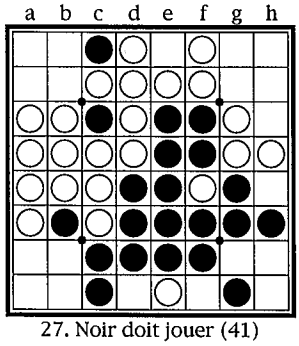

22. Blanc doit jouer (44)

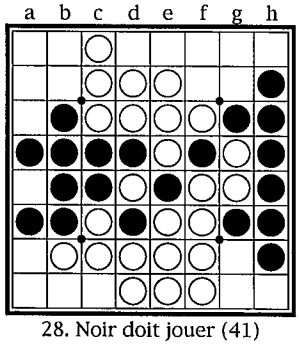

23. Noir doit jouer (43)

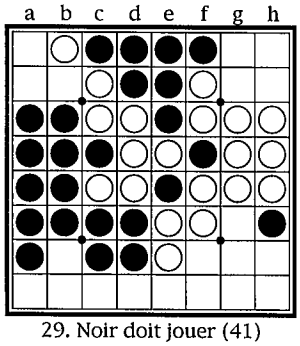

24. Blanc doit jouer (42)

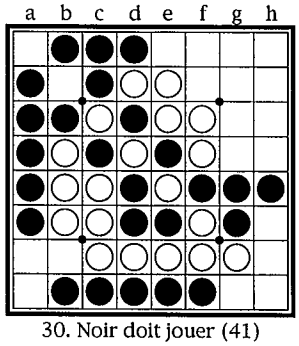

25. Blanc doit jouer (42)

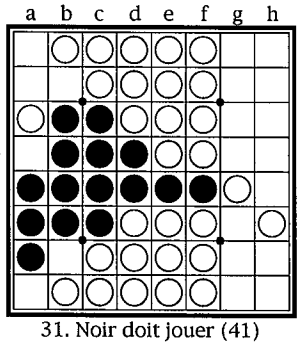

26. Noir doit jouer (41)

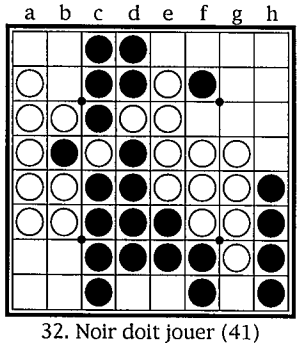

27. Noir doit jouer (41)

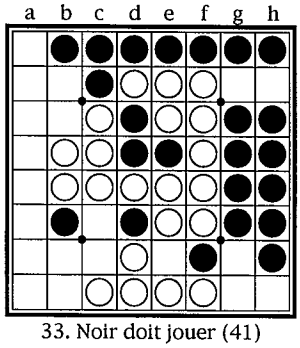

28. Noir doit jouer (41)

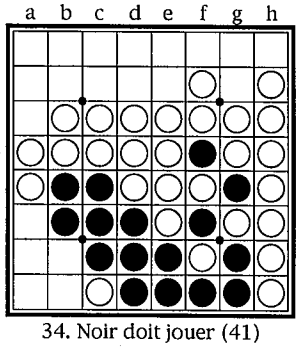

29. Noir doit jouer (41)

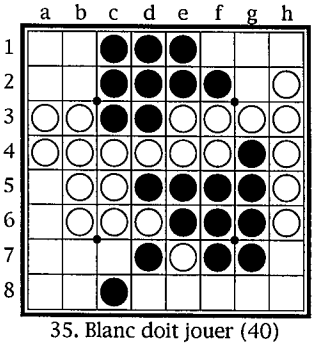

30. Noir doit jouer (41)

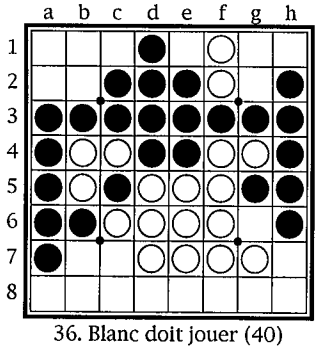

31. Noir doit jouer (41)

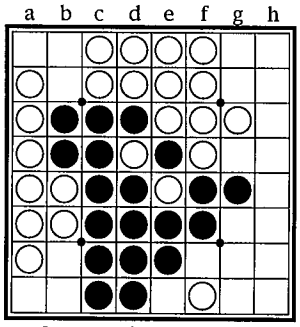

32. Noir doit jouer (41)

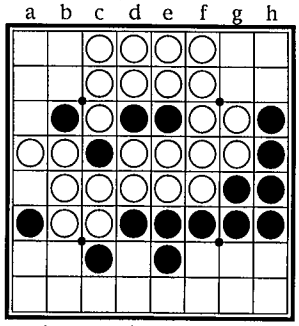

33. Noir doit jouer (41)

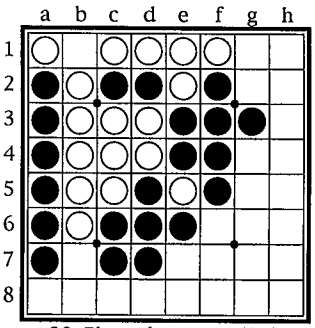

34. Noir doit jouer (41)


35. Blanc doit jouer (40)


36. Blanc doit jouer (40)


37. Noir doit jouer (39)


38. Noir doit jouer (37)


39. Blanc doit jouer (35)

22. Spock - Tastet, Match Hommes - Machines 94
23. Pee-Wee - Tastet, US Open 94
24. Nicolet - Juhem, IDF3 92
25. Rev - Logistello, Internet 94
26. Nicolet - Handel, Finale Cambridge 94
27. Suekuni - Nakajima, Finale Paris 94
28. Logistello - Keyano, Waterloo 93
30. Logistello - Rev, Internet 94
31. Theofanopoulos - Fuller, Monde 84
32. Forest - Tastet, Match Hommes - Machines 94
33. Logistello - Rev, Internet 94
35. Melnikov - Stepanov, Grand Prix Russie (1) 93
36. Logistello - Cassio, Courchelettes 94
37. Shaman - Tastet, Finale Monde 92
38. Tastet - Juhem, Grand Prix de France 93
39. Takahashi - Penloup, Monde 93

Solutions des tests

Première grille :

Diag. 1 : g8 h7 a8 a6 a4 a7 b6 a2 h8 a3 a1 b1 h1 g2 41-23

Diag. 2 : a4 b7 a3 a2 b8 a7 g7 h8 a8 h7 a1 b2 h2 h1 37-27

Diag. 3 : d1 g1 h1 h2 c1 b2 b8 a8 a2 a3 a1 b1 g2 g3 33-31

Diag. 4 : h8 b6 a7 h7 g7 a5 b7 a6 h1 h2 b2 a1 a8 b8 32-32

Diag. 5 : g8 g7 h8 g2 b2 a2 a1 g6 h7 b7 a8 h3 g1 47-16

Diag. 6 : h3 a7 a8 h4 h6 h7 h8 g7 a1 b1 h2 g2 g1 h1 39-25

Diag. 7 : a6 c8 b7 a7 a8 b8 h8 g2 h1 h7 g1 h2 b1 a1 36-28

Diag. 8 : e1 h7 h6 g7 h8 g8 h2 h5 g2 h1 g1 b2 b1 c1 a1 28-36

Diag. 9 : a4 a3 b2 a1 b1 b7 a7 g7 h1 g1 g8 a8 b8 h7 h8 36-28

Diag. 10 : b2 b7 g1 g7 a8 a7 g8 a1 a2 f1 h8 h7 g2 h1 h2 27-37

Diag. 11 : b3 a3 a6 c3 b4 a2 c2 b2 d2 e1 a1 b1 c1 g7 17-46

Diag. 12 : b7 h2 a7 a8 h1 g1 b2 b1 a1 e1 d1 h7 h8 g7 g8 36-28

Diag. 13 : b7 h7 h8 a8 g8 g2 a7 b6 b1 a6 a1 a2 a3 a4 h2 h1 39-25

Diag. 14 : a3 b7 a4 b2 b1 g2 a1 a2 b8 a8 a7 h8 g7 h7 h2 h1 41-23

Diag. 15 : g3 f1 b8 a8 c1 d1 g1 g2 c2 b1 a1 b2 a3 a2 h2 h1 34-30

Diag. 16 : f8 b6 a7 c7 h1 g7 h7 b2 h8 h2 a6 a5 a1 a8 b7 b8 44-20

Diag. 17 : f8 f7 g8 h3 h7 b7 b2 a1 g6 g7 a2 a8 h8 g2 g1 h1 36-28

Diag. 18 : g2 b7 a8 a7 g8 h1 f1 h7 h8 g7 e1 h2 b2 a1 g1 a2 31-33

Diag. 19 : b6 b5 a6 b8 c8 a7 a8 b7 a1 b1 g1 h1 h7 h8 h2 g2 36-28

Deuxième grille :

Diag. 20 : h5 30-29 ou 35-29 (en comptant les cases vides)
• L'intérêt de cette position est de vérifier que les programmes comptent bien la différence de pions et non le total de pions. En effet, Noir peut faire 31 pions (mais perdre) avec la suite 55.g6 h5 f6 h8 h6 h7 alors que 30 pions (et des cases vides) lui suffisent pour gagner.

Diag. 21 : g5 e8 g2 g4 g6 f8 h4 h7 h6 g7 h8 h2 g8 h3 h5 32-32
• Cette position demande plus de temps que la plupart des positions avec 15 cases vides. Cela est dû au fait que toutes les cases vides sont regroupées au lieu d'être séparées dans les quatre coins de l'Othellier.

Diag. 22 : g8 d8 b8 b2 h2 h1 g1 g7 a7 a6 a1 b1 h8 f8 c8 b7 a8 31-33
• Le meilleur coup n'est pas très naturel ici. g8 est le seul coup gagnant et je pense qu'aucun programme n'a considéré ce coup en premier.

Diag. 23 : a2 b7 e1 d1 f2 f1 g2 h1 g1 h2 h3 h8 a8 b8 g8 b2 a1 b1 34-30
• Ici, le meilleur coup est évident, c'est pour voir si les programmes le trouvent vite ou pas.

Diag. 24 : c3 e1 d1 h2 g7 c2 h1 h8 a3 b4 a4 h7 e8 g8 g1 b2 b1 a2 a1 32-32

Diag. 25 : g1 c2 a5 a3 c1 d1 f1 b2 b1 a1 a2 h1 h2 b8 g7 g8 f7 h7 h8 32-32 ou a5 h1 h2 c2 d1 c1 b2 a1 a3 b8 a2 f1 g1 b1 f7 g8 g7 h8 h7 32-32
• Il y a deux coups qui font nulle : a5 et g1. Tous les autres perdent.

Diag. 26 : d8 a6 a4 d7 a5 e8 a3 g2 b2 a1 a2 b7 f8 g8 g1 h1 h2 a8 a7 b8 32-32

Diag. 27 : b7 a7 e1 b1 h2 h3 h5 d8 f8 a8 b2 g2 b8 a1 a2 h8 h1 g1 g7 h7 31-33
• Le meilleur coup n'est pas très évident, car Blanc peut répondre a7 sans retourner la case X b7.

Diag. 28 : b2 a1 b1 a2 a3 a5 a8 b8 f2 f1 e1 g7 d1 g2 c8 a7 h1 g1 g8 h8 32-32 ou e1 f1 a5 f2 g7 a8 b8 h8 g8 a7 c8 b2 d1 g2 a3 a1 h1 a2 b1 32-32 ou f1 a3 f2 d1 a2 a5 b1 a1 b2 a7 e1 g1 g7 h8 c8 b8 a8 h1 g2 g8 32-32
• Il y a trois coups qui font nulle : b2, f1 et e1. Tous les autres perdent.

Diag. 29 : g2 g1 h1 h2 f8 d8 g6 f7 g8 c8 a1 e8 b8 a2 g7 h8 h7 a8 b2 b7 37-27
• Ici, a1 est plus naturel mais ne gagne que 34-30.

Diag. 30 : g3 h4 g4 f2 h8 g8 h7 h6 b7 a8 a7 a1 b2 e1 h3 g2 h1 g1 f1 h2 32-32

Diag. 31 : g6 g4 g2 g3 h4 h1 h2 h7 g1 h3 a1 h5 b2 b7 g8 h8 g7 a2 a4 31-32
• La meilleure suite perd 31-32 seulement. Permet de voir les programmes qui comptent les cases vides. À noter que Brutus détermine le meilleur coup dès la recherche de gain/nulle/perte car il utilise une fenêtre [-1,1] et que le coup se trouve dans cette fenêtre. Sur cette position, c'est donc plus rapide, mais en général, compter les cases vides pour le vainqueur permet d'obtenir davantage de coupures α–β.

Diag. 32 : g3 g8 f3 h4 h3 b2 e8 d8 a1 e1 f1 h2 a7 b7 a8 b8 h1 b1 g2 g1 30-34
• Très peu de programmes envisagent g3 en premier.

Diag. 33 : e7 c7 g2 g8 a4 b3 a6 a5 a2 a3 c6 a1 b7 a7 a8 h2 b2 g7 b8 h8 28-36

Diag. 34 : c2 d2 a3 a2 e1 d1 e2 f1 b1 a6 b7 a8 b2 a1 c1 a7 b8 g2 h1 g1 31-33
• Cette position a permis de trouver un bug dans un programme car il y a des suites qui donnent deux cases vides à la fin.

Diag. 35 : c7 b8 h8 h7 d8 e8 g8 h1 f8 b7 g1 f1 g2 a6 a8 b2 a7 a5 a1 a2 b1 32-32

Diag. 36 : b7 e1 c1 a8 c7 b8 g6 c8 d8 f8 e8 h7 h8 g8 h1 b1 a1 b2 g2 g1 a2 32-32

Diag. 37 : g2 g4 h4 h1 b7 h6 h5 h3 h2 g6 f7 a8 b8 e8 h7 h8 g1 b2 a1 b1 g7 g8 22-42
• Il n'y a qu'un meilleur coup, g2, mais il y a trois autres coups qui ne font qu'un pion de moins : g4, h3 et b7.

Diag. 38 : b2 c8 h2 g2 a5 a7 h1 g1 d7 d8 e8 b7 b1 a1 a8 b8 a3 h7 h8 a2 f7 g8 g7 f8 34-30
• Une jolie case X au coup 37. C'est le seul coup gagnant.

Diag. 39 : le style de partie où un joueur humain sait tout de suite qu'il est gagnant. Combien faut-il de temps à un programme ? Cela dépend. Évidemment, les programmes qui lancent très vite la finale (sans faire beaucoup d'analyse de milieu de partie) sont très avantagés ici.
Blanc peut-il faire 64-0 à coup sûr ? Oui. C'est d'ailleurs ce qu'avait réussi à faire Dominique avec g5 g6 b1 h5 a8 g4 c8 b8 g1 b7 g2 d8 e7 f6 f8 e8 f7 h2 h4 h3 h6 0-64.

Résultats des tests

Merci aux onze programmeurs de cinq pays différents qui nous ont fait parvenir les résultats des tests de finale 20 à 39. Voici une description des programmes puis les résultats des tests. Évidemment, comme les tests ont été faits sur des machines différentes et même de types différents, il est très difficile de comparer les temps. Pour faciliter cela, nous avons classé les machines par ordre de rapidité croissante. Si donc un temps est meilleur que tous ceux qui sont au-dessous de lui, le programme est plus rapide. On peut aussi comparer le nombre de nœuds parcourus (exprimé en milliers de nœuds : Knœuds dans la grille, sauf pour la position 20).

Toutefois Jean Delteil fait la remarque suivante : « Le nombre de nœuds parcourus n'est pas toujours un gage d'efficacité absolue ; par exemple, je peux modifier mon algorithme pour gagner jusqu'à 20 % de nœuds mais il tourne 50 % moins vite ! Alors comment faire ? Je propose l'emploi d'un "étalon" qui permet de mesurer (approximativement) la puissance du matériel appliquée aux finales. J'utilise actuellement un programme dédié à ce calcul (il donne un indice de performance). Il ne reste plus qu'à faire une règle de trois pour ajuster les temps. Bien sûr, seulement valable sur PC ! » Nous avons essayé de calculer un « coefficient machine » qui indique le rapport de vitesse estimé par rapport à un 486DX2-66. Toutefois, vu l'incertitude qui entoure ces coefficients, nous avons préféré ne pas modifier les temps donnés. Vous pourrez faire vous-même certaines multiplications pour voir. Nous avons mis en gras le meilleur temps (brut) et le plus faible nombre de nœuds.

Gros-Thello
par Simon Pinta (F)

486 33Mhz, 4Mo RAM, 256Ko cache.
Langage Quick C pour Windows (Avec débug car sinon problème)
• Pour chaque position, il y a une recherche de coup gagnant, puis s'il n'y a pas de coup gagnant, une recherche de partie nulle est lancée. Enfin, une recherche du meilleur coup si la partie n'est pas nulle. Pour la grille gain/nulle/perte, nous avons pris les résultats qui correspondaient au moment où le programme n'avait pas trouvé de gain (pdcg : pas de coup gagnant) si le résultat final est la nulle, et la somme des résultats pdcg et pdpn (pas de partie nulle) si le résultat final est une position perdante.

Jam'oth
par Guy Rump (F)

Version du 30/11/94,
486 33Mhz, 16Mo de RAM.
Langage C, sous UNIX SCO.
• A donné seulement les temps pour la recherche de la valeur de la position : gain/nulle/perte.
Ce programme joue sur Minitel : 36.14 ou 36.15 JAM*JEU (version limitée en 36.14).

Thor
par Sylvain Quin (F)

Version 3.37ß du 25/10/94,
486 DX2-50Mhz, mode VGA 640x480 en 16 couleurs,
Langage C pur.
• Le nombre de nœuds n'a pas été fourni mais a été estimé à partir du temps mis et du nombre de nœuds par seconde en finale. Comme le programme commence par faire une recherche de milieu de partie, au cours de laquelle il parcourt moins de nœuds par seconde, le nombre de nœuds est surévalué pour les temps courts (moins d'une minute). Par ailleurs, dans le cas où la position est nulle, les résultats donnés dans la grille gain/nulle/perte correspondent à la preuve qu'il y a au moins la nulle. La différence avec la deuxième grille est donc le temps qui correspond à montrer qu'il n'y a pas mieux que la nulle.

Cassio
par Stéphane Nicolet

Version de novembre 1994.
Macintosh LC630. (68040-33Mhz)
Écrit en Turbo-Pascal.
• Cassio est le seul programme à donner directement toute la suite en mode finale parfaite. Ce sont ses suites que nous avons reprises pour les solutions des tests (cf. page précédente).
• Cassio est le programme qui examine en moyenne le moins de nœuds.

Brutus I
Louis Geoffroy, Martin Piotte (CND)

Version du 22/10/94.
Clone 486DX2-66Mhz, 256K cache 4Mo RAM (mais utilise 900Ko seulement). Fonctionne seulement sur PC 386 ou plus avec écran VGA ou plus.
Écrit en C (7000 lignes) et assembleur (13000 lignes).
• Le nombre de nœuds a été fourni pour le meilleur coup mais pas pour la valeur de la position (sauf dans les cas de nulle). Il a alors été estimé assez précisément d'après le temps.
• Brutus semble être le programme le plus rapide actuellement pour déterminer le score exact.

PeeweeRv
par Alan Whinery (USA)

Version 2.xx
Hewlett Packard 486DX2-66Mhz, (indice Norton Sysinfo 141)
Écrit en C (Microsoft C/C++ 7.0) et assembleur (MASM 5.1)
• PeeweeRv ne donne malheureusement pas le nombre de nœuds.

Isaac
par Luigi Lamberti (I)

Version 4.3
486DX2-66Mhz, 256Ko de cache, 4Mo RAM (tests effectués par Donato Barnaba).
Écrit en assembleur. Tourne en mode mémoire « small ».
• Isaac ne peut trouver le meilleur score avec plus de 22 cases vides. Isaac ne donne pas la suite. À titre indicatif, voici les temps nécessaires pour toute la suite dans les tests 35 : 4'06" ; 36 : 25'42" ; 37 : 33'01".
• Les temps d'Isaac sont rarement les meilleurs, mais ils sont quand même toujours très bons.

Spock
par Jean Delteil (F)

Version V8.0 (DOS)
Pentium 66Mhz, 256Ko de cache, 8Mo RAM
Langage C + assembleur.
• Spock fait environ 142 milliers de nœuds par seconde en finale sur ce matériel. Le programme donne les onze premiers coups de la suite parfaite mais la calcule toute entière.

Forest
par Olivier Casile (F)

Version 2.1 (disposant de nouveaux algorithmes de fin de partie).
Pentium 90Mhz, 256Ko de cache, 8Mo RAM.
Compilateur Watcom C++ très optimisé (cf. article dans ce numéro).
• Forest ne donne pas la suite optimale, pour cause de chasse au gaspi. Ce programme a effectué des progrès considérables ces derniers mois et est désormais souvent le plus rapide en mode gain/nulle/perte.

Logistello
par Michael Buro (D)

Version de 11/94
Station Sparc 10M30, 32Mo RAM.
Écrit en ANSI-C.
• Dans une position avec n cases vides, une recherche de milieu de partie jusqu'à une profondeur n-7 (environ mais cela peut varier en fonction des résultats obtenus) est effectuée. Ensuite, une recherche « vite fait mal fait » de fin de partie est effectuée avec une fenêtre [-1,0] pour trouver au moins la nulle. Si cette recherche est fructueuse, une recherche encore « vite fait mal fait » est effectuée avec une fenêtre [-1,1] pour trouver un gain éventuel, sinon on passe directement à l'étape suivante. On effectue alors une recherche exacte de gain/nulle/perte. Si la partie n'est pas nulle, le score optimal est déterminé en utilisant une fenêtre d'amplitude 9 qui se déplace si nécessaire.

Keyano
par Mark Brockington (CND)

Version 3.4 du 31/10/94
Station Sparc 20/50 (50? Mhz), 96Mo RAM (Keyano en utilise environ 3,2).
5000 lignes illisibles de C.
• Du code écrit par Warren Smith et Jean-Christophe Weill a été utilisé pour évaluer les positions avec moins de 9 cases vides.
• Dans une position avec n cases vides, une recherche de milieu de partie à profondeur n-9 est effectuée pour ordonner les coups (sauf pour le diagramme 39 où la recherche a été limitée à profondeur 12 au lieu de 17 qui aurait pris trop de temps). Puis vient la recherche de gain/ nulle/perte et enfin le score optimal.

Pour motiver les programmeurs débutants, nous publions dans les deux premiers tableaux les résultats obtenus en 1990 par leurs illustres prédécesseurs : Comp'oth par François Aguillon sur un 68000 à 10 MHz et Othel du Nord par Jean-Claude Delbarre sur un 80286 à 10 MHz. Sachez cependant que les meilleurs programmes actuels résolvent chacun de ces problèmes (du numéro 1 au numéro 19) en une poignée de secondes.

Test	1		2		3		4		5		6		7		8		9		10	
	Ggnt	Pft	Ggnt	Pft	Ggnt	Pft	Ggnt	Pft	Ggnt	Pft	Ggnt	Pft	Ggnt	Pft	Ggnt	Pft	Ggnt	Pft	Ggnt	Pft
Comp'oth	4"	1'14"	9"	0'54"	12"	1'02"	29"	1'02"	6"	0'15"	16"	2'23"	13"	0'31"	6"	>3'	30"	2'14"	11"	2'09"
Othel du Nord	11"	1'43"	3"	1'09"	4"	2'13"	26"	1'39"	1"	0'24"	11"	2'13"	14"	1'36"	47"	>5'	38"	4'04"	40"	5'15"
Coup/Score	G8	41	A4	37	D1	33	H8	32	G8	48	H3	39	A6	36	E1	36	A4	28	B2	37

Test	11		12		13		14		15		16		17		18		19	
	Ggnt	Pft	Ggnt	Pft	Ggnt	Pft	Ggnt	Pft	Ggnt	Pft	Ggnt	Pft	Ggnt	Pft	Ggnt	Pft	Ggnt	Pft
Comp'oth	3"	2'51"	1'38"	>3'	2'50"		1'39"		2'52"		1"		30"	1'41"	2'20"	5'00"	46"	
Othel du Nord	0"	1'30"	47"	5'42"	3'25"		0'57"		3'51"		50"		11"	3'20"	3'34"		1'02"	
Coup/Score	B3	47	B7	28	B7	39	A3	41	G3	34	F8	44	F8	36	G2	31	B6	36

Recherche d'un coup gagnant

	Coef	Diagramme 20			Diagramme 21			Diagramme 22			Diagramme 23			Diagramme 24		
	Mach.	Tps	Nds	Suite	Tps	Knds	Suite	Temps	Knds	Suite	Temps	Knds	Suite	Temps	Knœuds	Suite
Gros-Thello	0,57	0"	15	H5+	43"	271	pdcg	6'40"	2 112	G8+	2'19"	747	A2+	45'59"	15 170	pdcg
Jam'oth	0,57	0"	52	H5 +6	35"	639	G5 E8 G2=	1'41"	1 500	G8 C8 H2+				8'45"	11 060	C3 E1 D1=
Thor	0,75	0"		H5+	21"	1 050	G5=	1'04"	3 200	G8+	0'22"	1 100	A2+	5'15"	15 750	C3=
Cassio	0,80	0"	10	H5+	16"	311	G5=	1'00"	847	G8+	0'31"	427	A2+	2'27"	2 180	C3=
Brutus	1,00	0"	109	H5 PS PS +1	02"	574	G5 G6 H5 0	0'05"	1 156	G8 F8 D8 +2	0'07"	1 100	A2 E1 F2 +2	0'33"	7 249	C3 E1 D1 0
PeeweeRv	1,00	0"		H5 +1	01"		G5 E8 G2 0	0'08"		G8 F8 D8 +1	0'05"		A2 E1 F2 +2	0'33"		C3 E1 D1 0
Isaac	1,00	0"		gagnant	03"		nulle	0'09"		gagnant	0'05"		gagnant	0'29"		nulle
Spock	1,40	1"	3	H5+	03"	304	G5 E8 G2=	0'13"	1 011	G8 C8 H2+	0'06"	362	A2 F2 F1+	0'19"	1 942	C3 E1 D1=
Forest	1,90	0"	227	H5 +6	01"	577	G5 E8=	0'04"	1 379	G8 F8+	0'03"	1 156	A2 F2+	0'28"	10 090	C3 G7=
Logistello	2,00	0"	195	H5 +6	04"	557	G5 G6 H5 0	0'13"	1 167	G8 F8 D8 >=2	0'09"	916	A2 F2 F1 >=2	1'28"	8 661	C3 E1 D1 0
Keyano	3,28	0"	3	H5 PS PS +6	03"	527	G5 E8 G2 0	0'06"	806	G8 D8 B8 +2	0'04"	457	A2 B2 E1 +2	0'49"	5 605	C3 E1 D1 0

	Coef	Diagramme 25			Diagramme 26			Diagramme 27			Diagramme 28		
	Mach.	Temps	Knœuds	Suite	Temps	Knœuds	Suite	Temps	Knœuds	Suite	Temps	Knœuds	Suite
Gros-Thello	0,57	1h03'08"	22 868	pdcg	10h43'24"	199 752	pdcg	6h37'47"	121 192	perdant	6h13'01"	121 447	pdcg
Jam'oth	0,57	15'15"	19 012	A5 C2 G1=	23'37"	28 313	D8 A6 A4=	13'12"	13 107	E1 H5 B7-	57'47"	68 944	E1 F1 G1=
Thor	0,75	8'22"	25 100	A5=	16'25"	49 250	D8=	41'39"	124 950	B7-	15'19"	45 950	E1=
Cassio	0,80	5'21"	5 024	G1=	29'20"	26 377	D8=	4'10"	3 510	perdant	20'50"	18 619	B2=
Brutus	1,00	1'23"	25 936	A5 C2 G1 0	4'32"	98 272	D8 A6 A4 0	1'24"	25 099	H5 H3 H2 -2	3'40"	79 156	F1 A3 F2 0
PeeweeRv	1,00	1'09"		A5 H1 H2 0	4'08"		D8 A6 A4 0	1'14"		B7 A7 B1 -1	3'13"		B2 A1 B1 0
Isaac	1,00	1'03"		nulle	3'17"		nulle	0'24"		perdant	2'55"		nulle
Spock	1,40	1'00"	6 224	A5 H1 H2=	3'23"	23 619	D8 A6 A4=	0'19"	1 404	H2 A7 B1-	2'38"	19 384	B2 A1 B1=
Forest	1,90	0'43"	15 202	A5 B8=	2'18"	44 659	D8 A6=	1'19"	25 765	B7 H5-	3'00"	65 291	E1 F1=
Logistello	2,00	2'18"	14 289	A5 C2 G1 0	9'14"	52 931	D8 A6 A4 0	1'20"	6 391	B7 A7 B1<=-1	5'10"	29 871	F1 A3 F2 0
Keyano	3,28	1'23"	10 531	A5 C2 G1 0	5'33"	32 774	D8 A6 A4 0	1'24"	7 346	B7 H5 E1 -2	4'36"	28 744	B2 A1 B1 0

	Coef	Diagramme 29			Diagramme 30			Diagramme 31			Diagramme 32		
	Mach.	Temps	Knœuds	Suite	Temps	Knœuds	Suite	Temps	Knœuds	Suite	Temps	Knœuds	Suite
Gros-Thello	0,57	25'43"	8 357	A1+	1h31'51"	26 548	pdcg	3h03'55"	58 266	perdant	3h22'15"	66 073	perdant
Jam'oth	0,57	1'18"	1 397	A1 A2 G1+	9'31"	10 157	=	10'46"	11 573	G4 H5 G2-	7'14"	7 609	G3 G8 E8-
Thor	0,75	2'48"	8 400	G2+	3'13"	9 650	G3=	19'27"	58 350	G6-	3'45"	11 250	G3-
Cassio	0,80	0'53"	758	A1+	6'50"	5 308	G3=	6'24"	4 983	perdant	2'30"	2 168	perdant
Brutus	1,00	0'22"	5 250	G2 D8 H2 +2	1'12"	21 042	G3 H4 G4 0	1'22"	25 654	G6 G4 G2 -1	0'45"	13 761	H4 G8 F3 -2
PeeweeRv	1,00	0'09"		A1 H7 G1 +2	0'43"		G3 H4 G4 0	1'07"		G6 G4 H5 -1	0'42"		B7 A8 A7 -1
Isaac	1,00	0'11"		gagnant	0'46"		nulle	1'04"		perdant	0'38"		perdant
Spock	1,40	0'05"	354	A1 H7 G1+	0'44"	3 954	G3 H4 G4=	0'59"	6 020	G6 G4 H5-	0'29"	2 987	H3 H4 G3
Forest	1,90	0'24"	7 303	A1 A2+	0'45"	13 886	G3 G2=	0'27"	8 108	G6 G4-	0'17"	5 471	B7 E1-
Logistello	2,00	0'15"	1 008	G2 G1 H1 >=2	1'32"	7 944	G3 H4 G4 0	1'13"	5 793	G6 G4 G2<=-1	0'58"	4 316	B7 E1 F1<=-1
Keyano	3,28	0'21"	1 259	G2 D8 H2 +2	1'25"	7 310	G3 H4 G4 0	1'17"	6 163	G3 G4 G7 -2	0'59"	5 625	G3 F3 E1 -2

	Coef	Diagramme 33			Diagramme 34			Diagramme 35			Diagramme 36		
	Mach.	Temps	Knœuds	Suite	Temps	Knœuds	Suite	Temps	Knœuds	Suite	Temps	Knœuds	Suite
Gros-Thello	0,57	7h43'02"	159 908	perdant	5h43'37"	108 794	perdant						
Jam'oth	0,57	18'25"	22 924	E7 G8 C7-	24'27"	30 902	C2 D2 E2-	32'58"	31 186	C7 B8 D8=	2h05'30"	156 814	B7 E1 C1=
Thor	0,75	17'03"	51 150	E7-	9'55"	29 750	C2-	16'01"	48 050	C7=	3h06'32"	559 600	B7=
Cassio	0,80	5'48"	**5 262**	perdant	11'33"	**10 079**	perdant	12'36"	**9 354**	C7=	1h07'33"	57 492	B7=
Brutus	1,00	2'42"	59 454	E7 C7 A4 -2	2'28"	51 129	C2 D2 E2 -2	2'44"	49 596	C7 B8 D8 0	**5'28"**	109 154	B7 E1 C1 0
PeeweeRv	1,00	1'44"		E7 H2 G2 -1	1'52"		A2 C2 D2 -1	1'54"		C7 A7 A5 0	5'38"		B7 E1 C1 0
Isaac	1,00	1'23"		perdant	1'54"		perdant	**1'36"**		nulle	5'48"		nulle
Spock	1,40	1'08"	8 210	E7 H2 B2-	1'46"	12 581	E2 D1 C2-	2'16"	10 888	C7 A7 A6=	8'40"	61 313	B7 E1 C1=
Forest	1,90	1'52"	39 231	E7 H2-	**1'38"**	34 980	F1 E2-	1'59"	34 596	C7 D8=	8'07"	162 478	B7 A8=
Logistello	2,00	2'35"	10 564	E7 C3 B2<=-1	4'12"	24 558	D2 E1 C2<=-1	3'08"	15 380	C7 B8 D8 0	9'47"	55 322	B7 E1 C1 0
Keyano	3,28	**1'06"**	7 046	E7 H2 B2 -2	3'58"	27 280	D2 E2 C2 -2	4'45"	14 254	C7 B8 D8 0	8'38"	**41 039**	B7 E1 C1 0

	Coef	Diagramme 37			Diagramme 38			Diagramme 39		
	Mach.	Temps	Knœuds	Suite	Temps	Knœuds	Suite	Temps	Knœuds	Suite
Gros-Thello	0,57									
Jam'oth	0,57	9'43"	9 130	H3 E8 G4-	>4h (stop)	260 000	A5 A3 A2=?	3'10"	1 037	A8 D8 B8+
Thor	0,75	4'19"	12 950	G2-						
Cassio	0,80	2'42"	2 125	perdant	1h56'56"	**94 561**	B2+	0'04"	**0,7**	F7+
Brutus	1,00	0'44"	13 974	G4 H4 B7 -2	44'59"	871 051	B2 D8 A5 +2	1h51'00"	2 088 871	A8 PS B8 +2
PeeweeRv	1,00	1'24"		B7 A8 B8 -1	1h04'03"		B2 C8 H2 +2	8'39"		F7 G8 B1 +1
Isaac	1,00	0'18"		perdant	**20'27"**		gagnant	1'04"		gagnant
Spock	1,40	0'41"	**1 471**	B7 H6 H4-	50'40"	425 714	B2 D8 A5+	0'22"	1	A8 PS B1+
Forest	1,90	**0'10"**	2 467	G2 G4-	37'44"	670 784	B2 C8+	**0'00"**	46	F7 F6+
Logistello	2,00	2'31"	7 484	H3 G4 G2<=-1	22'39"	110 444	B2 D7 >= 2	2'02"	3 755	A8 >=14
Keyano	3,28	3'19"	5 929	H3 G4 G2 -2	1h15'06"	156 670	B2 C8 H2 +2	22'32"	22 681	A8 PS >=+20

Recherche du meilleur coup

	Coef	Diagramme 20			Diagramme 21			Diagramme 22			Diagramme 23			Diagramme 24		
	Mach.	Tps	Nds	Suite	Tps	Knds	Suite	Temps	Knds	Suite	Temps	Knds	Suite	Temps	Knœuds	Suite
Gros-Thello	0,57	0"	99	H5 35 à 29	2'03"	750	G5 32-32	44'50"	14 906	G8 33 à 31	16'42"	5 147	A2 34 à 30	1h25'57"	28 344	C3 32-32
Thor	0,75	0"		H5 +6	22"	1 100	G5 =	1'32"	4 600	G8 +2	1'14"	3 700	A2 +4	12'08"	36 400	C3 =
Cassio	0,80	0"	59	H5 30-29	27"	495	G5 E8 G2 32	1'59"	1 689	G8 D8 B8 33	1'11"	975	A2 B7 E1 34	3'25"	**3 099**	C3 E1 D1 32
Brutus	1,00	0"	109	H5 PS PS +1	02"	574	G5 G6 H5 0	**0'09"**	2 082	G8 D8 B8 +2	0'12"	1 886	A2 B7 E1 +4	**0'33"**	7 249	C3 E1 D1 0
PeeweeRv	1,00	0"		H5 +1	03"		G5 E8 G2 0	0'20"		G8 D8 B8 +2	0'20"		A2 B7 E1 +4	1'02"		C3 E1 D1 0
Isaac	1,00	0"		+1	07"		0	0'23"		+2	0'16"		+4	1'03"		0
Spock	1,40	1"	62	H5 PS PS 35	03"	304	G5 E8 G2 32	0'18"	1 424	G8 D8 B8 33	0'13"	986	A2 B7 E1 34	0'19"	1 942	C3 E1 D1 32
Forest	1,90	0"	227	H5 +6	04"	1 579	G5 E8 =	0'17"	5 439	G8 D8 +2	**0'10"**	2 722	A2 B7 +4	1'04"	23 368	C3 E1 =
Logistello	2,00	0"	263	H5 +6	04"	557	G5 G6 H5 0	0'19"	1 820	G8 D8 B8 +2	0'21"	1 957	A2 B7 E1 4	1'28"	8 661	C3 E1 D1 0
Keyano	3,28	0"	51	H5 PS PS +6	03"	527	G5 E8 G2 0	0'09"	**1 218**	G8 D8 B8 +2	0'19"	2 089	A2 B2 E1 +4	0'49"	5 605	C3 E1 D1 0

	Coef	Diagramme 25			Diagramme 26			Diagramme 27			Diagramme 28		
	Mach.	Temps	Knœuds	Suite	Temps	Knœuds	Suite	Temps	Knœuds	Suite	Temps	Knœuds	Suite
Gros-Thello	0,57	1h43'30"	37 252	A5 32-32	12h34'16"	235 529	D8 32-32	17h54'50"	337 381	B7 31 à 33	7h42'13"	150 507	E1 32-32
Thor	0,75	20'14"	60 700	A5 =	1h34'05"	282 250	D8 =	1h06'48"	200 400	B7 -2	59'37"	178 850	E1 =
Cassio	0,80	10'08"	**9 595**	G1 C2 A5 32	36'57"	**32 336**	D8 A6 A4 32	8'28"	**7 038**	B7 A7 E1 31	24'17"	**21 898**	B2 A1 B1 32
Brutus	1,00	**1'23"**	25 936	A5 C2 G1 0	4'32"	98 272	D8 A6 A4 0	2'01"	36 155	B7 A7 E1 -2	**3'40"**	79 156	F1 A3 F2 0
PeeweeRv	1,00	2'55"		A5 H1 H2 0	10'58"		D8 A6 A4 0	3'32"		B7 A7 E1 -2	5'29"		F1 A3 F2 0
Isaac	1,00	1'57"		0	4'57"		0	2'23"		-2	5'18"		0
Spock	1,40	1'00"	6 224	A5 H1 H2 32	3'23"	23 619	D8 A6 A4 32	2'14"	13 901	B7 A7 E1 31	2'38"	19 384	B2 A1 B1 32
Forest	1,90	1'43"	36 290	A5 H1 =	**3'16"**	63 840	D8 A6 =	2'11"	40 132	B7 H5 -2	3'48"	81 214	E1 F1 =
Logistello	2,00	2'18"	14 289	A5 C2 G1 0	9'14"	52 931	D8 A6 A4 0	2'01"	10 660	B7 A7 E1 -2	5'10"	29 871	F1 A3 F2 0
Keyano	3,28	**1'23"**	10 531	A5 C2 G1 0	5'33"	32 774	D8 A6 A4 0	**1'59"**	10 787	B7 H5 H3 -2	4'36"	28 744	B2 A1 B1 0

	Coef	Diagramme 29			Diagramme 30			Diagramme 31			Diagramme 32		
	Mach.	Temps	Knœuds	Suite	Temps	Knœuds	Suite	Temps	Knœuds	Suite	Temps	Knœuds	Suite
Gros-Thello	0,57	5h13'50"	99 273	G2 37 à 27	2h22'30"	40 926	G3 32-32	8h51'55"	171 194	G6 31 à 33	13h00'21"	262 529	G3 30 à 34
Thor	0,75	21'14"	63 700	G2 +10	6'07"	18 350	G3 =	24'06"	72 300	G6 -2	20'18"	60 900	G3 -4
Cassio	0,80	7'31"	6 246	G2 G1 H1 37	6'24"	**4 977**	G3 H4 G4 32	13'29"	10 900	G6 G4 G2 31	21'00"	**18 382**	G3 G8 F3 30
Brutus	1,00	0'48"	11 455	G2 G1 H1 +10	**1'12"**	21 042	G3 H4 G4 0	1'22"	25 654	G6 G4 G2 -1	**2'01"**	37 003	G3 G8 F3 -4
PeeweeRv	1,00	1'18"		G2 G1 H1 +10	1'38"		G3 H4 G4 0	2'25"		G6 G4 G2 -1	6'03"		G3 G8 F3 -4
Isaac	1,00	1'31"		+10	2'29"		0	1'38"		-1	5'05"		-4
Spock	1,40	**0'37"**	**3 776**	G2 G1 H1 37	0'44"	3 954	G3 H4 G4 32	1'13"	7 436	G6 G4 G2 31	4'01"	27 888	G3 G8 F3 30
Forest	1,90	2'13"	39 771	G2 G1 +10	1'38"	30 460	G3 H4 =	**0'45"**	12 755	G6 G4 -2	3'57"	82 635	G3 G8 -4
Logistello	2,00	1'13"	6 426	G2 G1 H1 +10	1'32"	7 944	G3 H4 G4 0	1'28"	**7 147**	G6 G4 G2 -2	5'50"	36 126	G3 G8 F3 -4
Keyano	3,28	1'15"	6 814	G2 G1 H1 +10	1'25"	7 310	G3 H4 G4 0	2'02"	10 884	G6 G4 G2 -2	3'20"	23 549	G3 G8 F3 -4

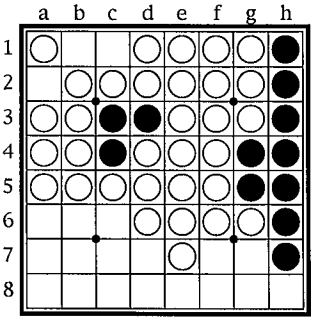
	Coef	Diagramme 33			Diagramme 34			Diagramme 35			Diagramme 36		
	Mach.	Temps	Knœuds	Suite	Temps	Knœuds	Suite	Temps	Knœuds	Suite	Temps	Knœuds	Suite
Gros-Thello	0,57	17h51'47"	896 147	E7 28 à 36	11h03'50"	213 299	C2 31 à 33						
Thor	0,75	2h08'10"	384 500	E7 -8	17'27"	52 350	C2 -2	28'28"	85 400	C7 =	3h06'32"	559 600	B7 =
Cassio	0,80	23'33"	**22 522**	E7 C7 G2 28	18'34"	**16 122**	C2 D2 A3 31	15'18"	11 744	C7 B8 H8 32	1h34'53"	81 551	B7 E1 C1 32
Brutus	1,00	5'54"	129 919	E7 C7 A4 -8	**2'46"**	58 470	C2 D2 A3 -2	2'44"	49 596	C7 B8 D8 0	**5'28"**	109 154	B7 E1 C1 0
PeeweeRv	1,00	8'20"		E7 C7 G2 -8	6'50"		C2 D2 A3 -2	4'22"		C7 A7 A5 0	23'32"		B7 E1 C1 0
Isaac	1,00	8'19"		-8	3'12"		-2	2'04"		0	18'22"		0
Spock	1,40	4'19"	33 544	E7 C7 G2 28	3'05"	22 503	C2 D2 A3 31	2'16"	**10 888**	C7 A7 A6 32	8'40"	61 313	B7 E1 C1 32
Forest	1,90	5'49"	122 076	E7 C7 -8	4'14"	87 900	C2 D2 -2	2'18"	39 264	C7 B8 =	24'45"	457 314	B7 E1 =
Logistello	2,00	9'46"	54 867	E7 C3 C7 -8	6'20"	37 311	C2 D2 A3 -2	3'08"	15 380	C7 B8 D8 0	9'47"	55 322	B7 E1 C1 0
Keyano	3,28	**4'12"**	30 618	E7 C7 A4 -8	6'17"	45 513	C2 D2 A3 -2	4'45"	14 254	C7 B8 D8 0	8'38"	**41 039**	B7 E1 C1 0

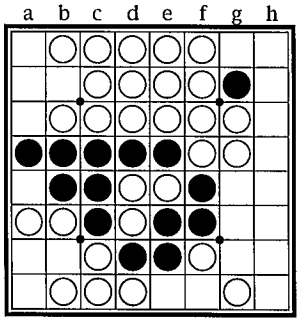
	Coef	Diagramme 37			Diagramme 38			Diagramme 39		
	Mach.	Temps	Knœuds	Suite	Temps	Knœuds	Suite	Temps	Knœuds	Suite
Gros-Thello	0,57									
Thor	0,75	2h01'49"	365 450	G2 -20						
Cassio	0,80	1h39'59"	**78 783**	G2 G4 H4 22	3h28'56"	**169 645**	B2 C8 H2 34			
Brutus	1,00	**13'35"**	258 844	G2 G4 H4 -20	1h34'34"	1 845 209	B2 C8 H2 +4	2h01'50"	2 292 740	A8 PS B8 +64
PeeweeRv	1,00	23'42"		G2 G4 H4 -20	2h13'54"		B2 C8 H2 +4	2h02'06"		A8 B8 C8 +64
Isaac	1,00	17'37"		-20						
Spock	1,40	14'06"	87 930	G2 G4 H4 22	2h01'57"	957 840	B2 C8 H2 34	29'43"	247 619	A8 PS B1 64
Forest	1,90	22'34"	439 476	G2 G4 -20	1h13'42"	1 282 070	B2 C8 +4	14h43'17"	16 126 651	G6 +64
Logistello	2,00	24'31"	135 349	G2 G4 H4 -20	**1h02'21"**	317 323	B2 C8 +4	**13'52"**	**93 961**	A8 >=64
Keyano	3,28	24'27"	123 697	G2 G4 H4 -20	1h39'09"	229 041	B2 C8 H2 +4			

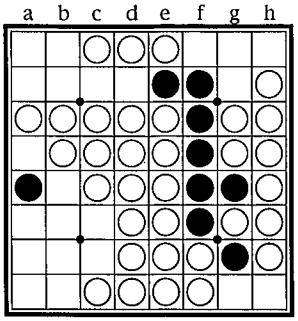
Nouveaux tests de finales

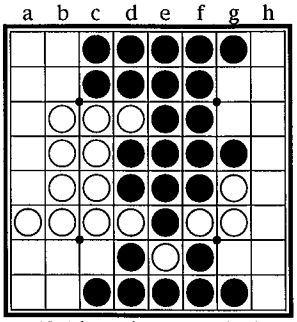
Voici une grille de problèmes destinés en priorité à tester les algorithmes de fin de partie des programmes. Évidemment, les humains peuvent aussi essayer de trouver le meilleur coup et la suite (au moins le début).

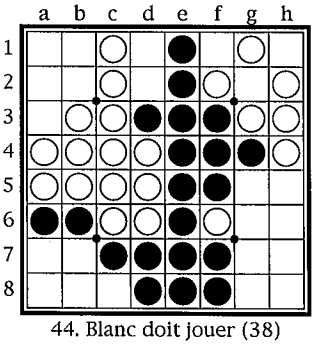
Deux grilles du même type sont déjà parues dans *Fforum 12* puis dans *Fforum 34*, mais les progrès des ordinateurs d'une part et surtout ceux des algorithmes d'autre part ont rendu ces deux grilles complètement obsolètes, les programmes les plus rapides résolvant chaque problème en une poignée de secondes.

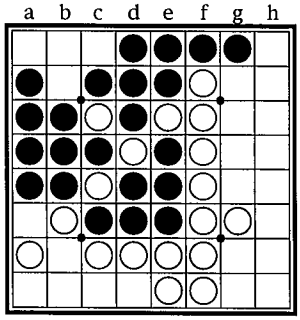

40. Noir doit jouer (41)

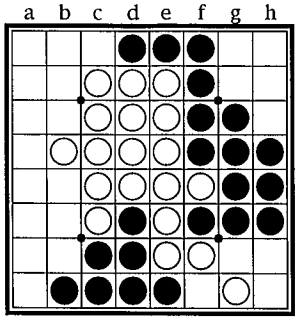

41. Noir doit jouer (39)

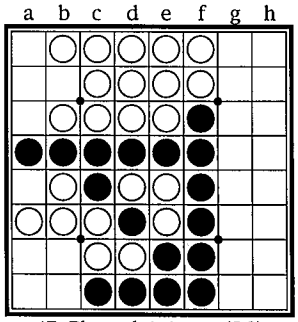

42. Noir doit jouer (39)

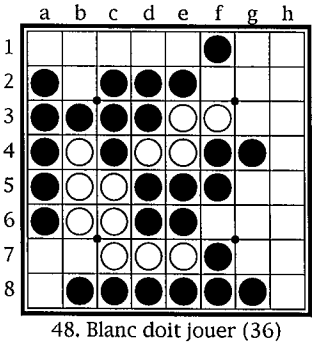

43. Blanc doit jouer (38)

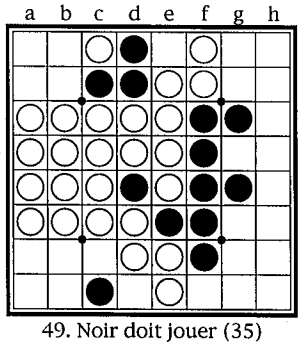

44. Blanc doit jouer (38)

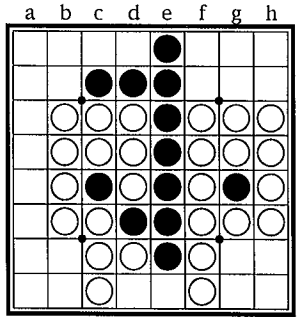

45. Noir doit jouer (37)

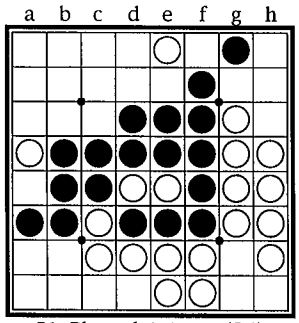

46. Noir doit jouer (37)

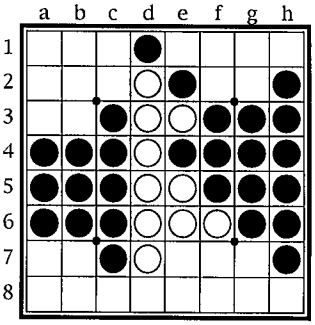

47. Blanc doit jouer (36)

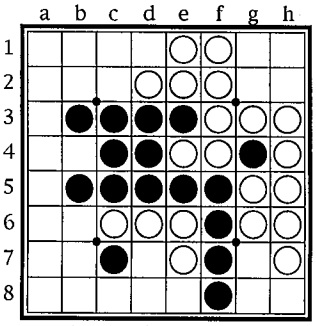

48. Blanc doit jouer (36)

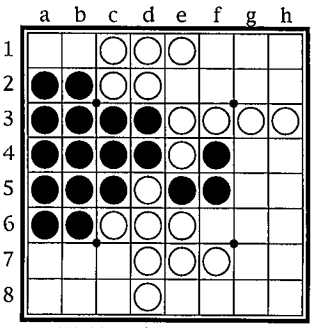

49. Noir doit jouer (35)

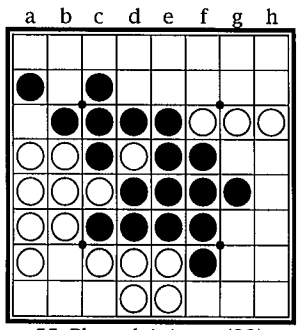

50. Noir doit jouer (35)

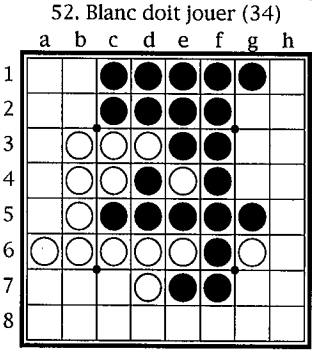

51. Blanc doit jouer (34)

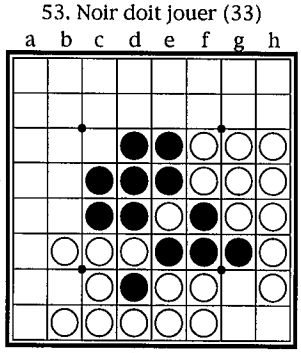

52. Blanc doit jouer (34)

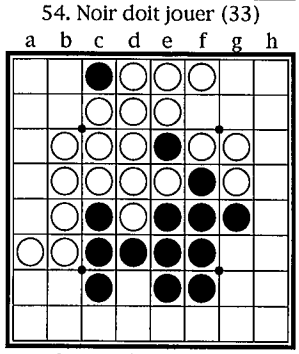

53. Noir doit jouer (33)

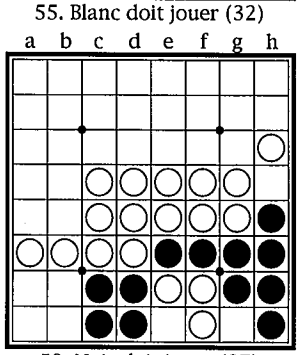

54. Noir doit jouer (33)


55. Blanc doit jouer (32)


56. Blanc doit jouer (32)


57. Noir doit jouer (31)


58. Noir doit jouer (31)


59. Noir doit jouer (27)

40. Turner 34-30 Monnom, Bruxelles 97
41. Eclipse 31-33 Logistello, Internet (7-12) 96
42. Penloup 35-29 Shaman, Cambridge 98
43. Brightwell 29-35 Suekuni, Monde 97 (finale)
44. Shaman 31-33 Tastet, Monde 95
45. Tamenori 38-26 Shaman, Monde 95 (finale)
46. Caspard 24-40 Juhem, Championnat de France 94
47. Brightwell 34-30 Tastet, Monde 97
48. Brightwell 21-43 Tastet, Paris 97
49. Lazard 40-24 Cali, Championnat de France 91
50. Nicolet 26-38 Feinstein, Monde 96
51. Tastet 40-24 Caspard, Monde 97
52. Edmead 19-45 Lazard, Cambridge 96
53. Nicolet 29-35 Murakami, Monde 96 (finale)
54. Logistello 35-29 Brutus, Tournois IOS 97
55. Logistello 32-32 Hannibal, Internet (1-6) 97
56. Logistello 30-34 Hannibal, Internet (1-6) 96
57. Taniguchi 15-49 Ralle, Monde 84 (finale)
58. Logistello 37-27 Murakami, Match 97
59. Tastet 64-0 Parsons, Mind Sports Olympiads 97

Résultats des tests

Logistello
Michael Buro (D)

Version du 23/9/98.
Pentium II 333 MHz.
8 Mo de Hash Table.

Keyano
Mark Brockington (CND)

Version du 26/6/98.
Ultra Sparc E2 170 24Mo RAM

NewThor
Nicolas Becquet,Sylvain Quin (F)

Version 6.4. Pentium II 233MHz
512K cache 32Mo RAM
Langage C sous Windows NT 4

Isaac
Luigi Lamberti (I)

Version 5.2
Pentium 266 MHz MMX.
512 Ko de cache, 64 Mo RAM.

Brutus I
Louis Geoffroy, Martin Piotte (CND)

Version du 10/9/98.
AMD K6-2 333 MHz MMX.
1024 Ko de cache, 64 Mo RAM
(dont 62 Mo utilisés).
Langage C + assembleur.

Zebra
Gunnar Andersson (S)

Version du 8/9/99.
Pentium Celeron 400 Mhz, poussé à 450MHz.
128 Mo RAM.
Langage C sous Windows 98.

Spock
Jean Delteil (F)

Version 13.2w de juin 1999.
Pentium 233 MHz MMX.
512 Ko de cache, 64 Mo RAM.
Langage C.

Hannibal
Martin Piotte, Louis Geoffroy (CND)

Version 0.5.
AMD K6-2 333 MHz MMX.
1024 Ko de cache, 64 Mo RAM
(dont 40 Mo utilisés).
Langage C + assembleur.

Forest
Olivier Casile (F)

Version 3.5 (Windows 95)
Pentium 233MHz, MMX
512Ko de cache, 32Mo RAM.
Compilateur Watcom C++

Saio
Benedetto Romano (I)

Version 5.0 du 15/12/98.
Pentium 266 MHz MMX.
512 Ko de cache, 64 Mo RAM.
Langage C + assembleur.

Voici une solution optimale pour chacune des positions (ligne fournie par NewThor). Le nombre entre parenthèses est le score du joueur qui a le trait.

Diag. 40 : (51) a2 b1 c1 ps b6 c7 a7 b7 b8 d7 f8 c6 a8 e8 g7 f7 a6 g8 c8 h8 d8
Diag. 41 : (32) h4 a3 a2 g6 g5 h5 h6 g7 e8 f8 h2 a5 a7 h3 h1 g1 a1 b2 h8 h7 a8 b7
Diag. 42 : (35) g2 h1 c2 g1 f1 d2 b2 a2 b6 c6 b5 a4 a6 g8 a1 b1 h8 a7 b8 ps b7 c7 a8
Diag. 43 : (26) c7 h4 h5 b8 b7 a8 a7 a5 h3 a4 g3 h6 a3 a2 g7 h2 b1 a1 b2 g2 h1 h7 h8
Diag. 44 : (25) d2 g5 b8 a3 b7 a8 a7 c8 g7 g8 h6 h5 g6 f1 d1 h1 g2 h7 h8 b2 b1 a1 a2
Diag. 45 : (35) b2 c1 g5 h6 g4 h3 b1 g3 h5 h4 g2 a6 a8 a1 h2 h1 b7 b8 d8 c8 g8 g7 h7 h8
Diag. 46 : (28) b3 c1 b1 a3 b2 h3 a5 a4 a2 h7 g7 g2 g1 h1 h2 a1 h8 f8 b6 a8 b5 a7 b7 a6
Diag. 47 : (34) g2 b8 b7 a2 a5 b2 g3 h3 a1 a3 h2 h1 g1 g4 h5 h4 g5 h6 g6 h7 g8 h8 g7 a8 a7
Diag. 48 : (46) f6 g5 g3 f2 h4 h5 h3 g6 g2 a7 h6 g7 d1 c1 b1 e1 g1 b7 a8 h7 h8 b2 a1 h2 h1
Diag. 49 : (40) e1 h4 g6 g4 h5 h6 b2 f8 d8 b8 c7 h3 g2 a1 b1 h2 h1 g1 h7 g7 a2 b7 a8
Diag. 50 : (37) d8 e8 g8 h8 f2 g7 b7 f1 g1 d1 c1 a8 a4 g2 a5 a6 a7 b1 a1 a3 b8 ps h1 h2 h7 ps b2 a2
Diag. 51 : (35) e2 h2 f1 d1 g7 d2 c1 c3 h1 h3 g2 d8 c8 h8 g8 b8 a3 a5 a7 b7 b3 a2 a1 c2 b2 b1 a8
Diag. 52 : (32) a3 f2 e1 f1 c2 b1 b3 c1 g2 a2 g7 e8 d8 h8 f7 g8 b7 c8 b8 h1 g1 e7 a1 b2 a7 a8 f8
Diag. 53 : (31) d8 b6 d7 c2 b4 g7 a5 a3 d1 c1 h8 g8 h2 e8 g2 a7 a6 a4 b7 c8 b8 a8 b1 a1 b2 a2 h1 g1
Diag. 54 : (31) c7 g6 g5 f6 g4 c8 b7 e2 b8 h6 h5 h4 b1 a8 a7 a1 g2 h1 f2 f1 g1 ps e8 f8 g8 h8 h2 g7 h7
Diag. 55 : (32) g6 h7 e2 f2 d2 c1 e1 d1 f1 f8 h6 h5 g8 b2 g4 g1 a1 a3 b1 c8 h4 h8 g7 h2 h1 g2 b7 a8 ps b8
Diag. 56 : (33) h5 h4 h3 d8 c8 c7 e8 g8 g4 g3 f8 b8 b2 a5 a4 h7 h6 h2 g2 a1 a2 b1 a3 a7 b7 g7 h1 ps a8 ps h8
Diag. 57 : (27) a6 b5 b4 e2 f2 e1 g1 d2 c3 f1 d1 a3 a5 a4 a2 b3 c2 g7 g2 b1 b2 a1 c1 b7 h8 g8 a8 a7 h2 h1
Diag. 58 : (34) g1 f2 h3 h5 h6 h7 h2 c8 a3 f8 g6 g7 a5 e8 d7 d8 h8 h4 g8 b7 a8 b8 b1 h1 g2 a1 a7 a2 b2 a4
Diag. 59 : (64) g8 e8 h4 b7 h2 b8 a8 ps a7 ps a5 ps g3 f3 b5 ps f2 g2 h1 ps c3

La première grille donne le temps nécessaire au programme pour choisir le coup qui s'avérera a posteriori le meilleur sans plus changer d'avis ensuite. L'idée est la suivante : si le programme, jouant en temps limité, devait être interrompu avant d'avoir terminé sa recherche, ce temps indique à partir de quand il jouerait le bon coup (sans toutefois être sûr que c'est le bon). Plus classiquement, la deuxième grille donne le temps nécessaire pour déterminer la valeur de la position (gain/nulle/perte) et la troisième, pour donner le score exact.

Résultats des nouveaux tests

Choix du bon coup

	Coef	Diag. 40	Diag. 41	Diag. 42	Diag. 43	Diag. 44	Diag. 45	Diag. 46	Diag. 47	Diag. 48	Diag. 49
	Mach.	Temps	Temps	Temps	Temps	Temps	Temps	Temps	Temps	Temps	Temps
Spock	4,5	0"	3"	8"	2"	2"	11"	5'24"	2"	4"	12"
NewThor	5,2	21"	1"	28"	2"	0"	3'17"	3'25"	0"	0"	0"
Logistello	5,2	0"	3"	18"	2"	38"	0"	0"	0"	0"	0"
Hannibal	5,33	0"	0"	8"	0"	2"	0"	0"	0"	0"	0"
Brutus	5,33	0"	6"	20"	0"	0"	3'58"	2'45"	0"	0"	0"
Zebra	7	0"	0"	7"	0"	25"	0"	2'39"	0"	0"	0"

	Coef	Diag. 50	Diag. 51	Diag. 52	Diag. 53	Diag. 54	Diag. 55	Diag. 56	Diag. 57	Diag. 58	Diag. 59
	Mach.	Temps	Temps	Temps	Temps	Temps	Temps	Temps	Temps	Temps	Temps
Spock	4,5	33"	13"	11"	36"	28'08"	18"	18"	14"	24"	11"
NewThor	5,2	0"	0"	14'45"		7h59'36"	29"	0"	0"	0"	1"
Logistello	5,2	6"	0"	2'23"	1"	21'16"	3"	0"	0"	0"	10"
Hannibal	5,33	0"	0"	1'52"	3"	12h57'27"	0"	0"	0"	0"	1"
Brutus	5,33	14"	5"	6'07"	1h45'10"	2h26'45"	5h30'24"	25"	3h02'52"	0"	5"
Zebra	7	0"	3"	8'50"	10"	1h50'51"	23"	1"	22'20"	0"	0"

Recherche d'un coup gagnant

	Coef	Diagramme 40			Diagramme 41			Diagramme 42			Diagramme 43		
	Mach.	Tps	Nœuds	Suite	Tps	Knœuds	Suite	Temps	Knœuds	Suite	Temps	Knœuds	Suite
Spock	4,5	**0"**	**2 000**	+ A2 C1 D7	1'04"	28 484	= H4 A3 A2	14"	**5 560**	+ G2 H1 C2	11"	4 650	- C7 B8 G2
Forest	4,5	**0"**	17 816	+ B1 C1	2'07"	134 106	= H4 A3	1'13"	84 409	+ G2 H1	16"	12 390	- C7 H4
Isaac	5	**0"**			1'06"			54"		+ G2	11"		- C7
Saio	5	**0"**		+ A2 B1 C1	34"		= H4 A3 A2			+ G2 H1 C2	**3"**		- C7
NewThor	5,2	**0"**	8 178	+ C1 B1 B6 C6	**21"**	31 319	= H4 A3 A2 G6	28"	18 169	+ G2 H1 C2 H8	9"	3 675	- C7 B8 B7 A4
Logistello	5,2	**0"**	13 653	+ A2 C1 G7	1'44"	75 979	= H4 A3	18"	13 415	+ G2 H1 C2	14"	9 384	- G3 C7 B2
Brutus	5,33	**1"**		+ A2 B1 C1 PS	43"		= H4 A3 A2 G6	20"		+ G2 H1 C2 B1	13"		- C7 B8 B1 A1
Hannibal	5,33	**11"**		+ A2 B1 C1	1'09"		= H4 A3 A2 G6	10"		+ G2 H1 C2 D2	9"		- C7 B8 G3 H3
Zebra	7	**0"**	224 000	+ A2	28"	**27 500**	= H4 A3 A2 G6	**7"**	7 150	+ G2 H1 C2	**3"**	**2 860**	- C7 H4 H5
Keyano	11,5	**0"**	56167	+ C1 B1 B6	4'38"	63 155	= H4 A3 A2	3'13"	51 081	+ G2 H1 C2	1'38"	26 293	- C7 B8 G3

	Coef	Diagramme 44			Diagramme 45			Diagramme 46			Diagramme 47		
	Mach.	Temps	Knœuds	Suite	Temps	Knœuds	Suite	Temps	Knœuds	Suite	Temps	Knœuds	Suite
Spock	4,5	30"	8 726	- B8 G5 D2	33"	15 389	+ B2 C1 A6	45"	**13 579**	- B7 B6 A6	19"	**9 091**	+ G2 B8 B7
Forest	4,5	1'11"	57 243	- G2 F1	6'04"	444 185	+ B2 B7	1'29"	71 357	- B7 B6	1'43"	119 268	+ G2 B8
Isaac	5	35"		- D2	1'38"		+ B2	51"		- B3	33"		+ G2
Saio	5	**4"**		- B8 G5 D2 A3	1'25"		+ B2 C1 G5 H4	**24"**		- B3 C1 B1 A3	**17"**		+ G2 B8 B7 A2
NewThor	5,2	18"	**7 946**	- D2 G5 B8 A3	3'19"	104 865	+ B2 C1 B1 B7	42"	19 321	- B5 B3 C1 B2	41"	25 826	+ G2 B8 B7 A2
Logistello	5,2	56"	33 308	- G2 F1 D1	15"	5 090	+ B2 C1 A6	1'16"	39 163	- B3 C1 B7	38"	25 361	+ G2 B8 B7
Brutus	5,33	27"		- D2 G5 G6 B7	2'43"		+ G5 H4 B2 G3	46"		- B7 G7 A4 A3	33"		+ G2 B8 B7 G1
Hannibal	5,33	35"		- D2 G5 B8 A3	18"		+ B2 C1 G5 H6	59"		- B3 C1 B1 G2	1'35"		+ G2 B8 B7 A2
Zebra	7	16"	12 700	- G2 F1 D1	**4"**	**3 900**	+ B2 C1 G5	55"	53 900	- B7 G7 B6	26"	29 500	+ G2 B8 B7
Keyano	11,5	4'03"	49 429	- D2 G5 B8	4'20"	72 572	+ B2 C1 G5	8'25"	91 292	- B3 H3 H2	3'50"	57 595	+ G2 B8 B7

	Coef	Diagramme 48			Diagramme 49			Diagramme 50			Diagramme 51		
	Mach.	Temps	Knœuds	Suite	Temps	Knœuds	Suite	Temps	Knœuds	Suite	Temps	Knœuds	Suite
Spock	4,5	9"	**1 403**	+ F6 G5 G3	4'02"	91 716	+ E1 H4 G6	31"	14 248	+ D8 E2 H7	1'38"	33 227	+ E2 H2 F1
Forest	4,5	15"	8 787	+ F6 F2	13'04"	762 801	+ E1 G4	7'43"	577 959	+ H7 F2	13'58"	851 771	+ E2 F1
Isaac	5	1'02"		+ F6	1'55"		+ E1	5'34"		+ D8	1'42"		+ A3
Saio	5	**3"**		+ F6 G5 F2 G3	**41"**		+ E1 H4 G6 G4	2'02"		+ D8 E8 G8 H8	5'53"		+ E2 H2 G7 F1
NewThor	5,2	11"	4 430	+ F6 G5 F2 G3	2'17"	69 031	+ E1 H4 H6 H5	6'37"	208 166	+ D8 E8 A4 F2	1'25"	39 416	+ E2 H2 G7 F1
Logistello	5,2	12"	4 722	+ F6 G5	2'18"	66 404	+ E1 H4	57"	21 276	+ D8 E2 H7	2'10"	67 289	+ E2 H2
Brutus	5,33	19"		+ F6 G5 F2 G3	1'04"		+ E1 H4 G6 H6	38"		+ D8 E8 G8 H8	44"		+ E2 H2 G7 H8
Hannibal	5,33	10"		+ F6 G5 F2 G3	2'54"		+ E1 H5 G4 G6	17"		+ D8 F2 B8 F1	2'50"		+ E2 H2 F1 D2
Zebra	7	4"	3 000	+ F6 G5 F2	54"	**36 000**	+ E1 G6 B1	**8"**	**8 050**	+ D8 E8 H7	**31"**	**2 510**	+ E2 H2 G7
Keyano	11,5	1'59"	16 650	+ F6 G5 F2	13'43"	106 677	+ E1 H4 H5	14'57"	182 773	+ D8 E8 H7	14'37"	125 254	+ E2 F1 H1

	Coef	Diagramme 52			Diagramme 53			Diagramme 54			Diagramme 55		
	Mach.	Temps	Knœuds	Suite	Temps	Mnœuds	Suite	Temps	Mnœuds	Suite	Temps	Mnœuds	Suite
Spock	4,5	20'10"	492 852	= A3 F2 E1	1h49'11"	3 190	- D8 B6	4h05'50"	7 189	- C8 C7 E2	19h52'08"	29 273	= G6 H7 E2
Forest	4,5	1h08'15"	3 982 734	= A3 F2	8h30'42"	15 588	- D8 B6	14h58'07"	27 039	- F2 G4			
Isaac	5												
Saio	5												
NewThor	5,2	20'24"	610 951	= A3 F2 E1 F1				5h25'54"	10 798	- F2 G6 G5 H5	19h39'54"	33 607	= G6 H7 E2 F2
Logistello	5,2	48'09"	1 840 167	= A3 F2									
Brutus	5,33	12'06"		= A3 F2 E1 F1	1h33'44"		- E8 D8 C8 D7	1h56'47"		- F8 G6 G5 C7	13h32'09"		= G6 H7 C1 F2
Hannibal	5,33	22'12"		= A3 F2 E1 F1	3h20'59"		- D8 B6 D7 A5	10h34'01"		- F8 G4 G5 H6	27h38'47"		= G6 H7 C1 F2
Zebra	7	**8'50"**	**462 000**	= A3 F2 E1	**45'11"**	**2 570**	- D8 B6 D7	**1h44'09"**	**5 730**	- F8 C7 E8	**7h39'44"**	**24 000**	= G6 H7 C1
Keyano	11,5	2h18'50"	1 480 788	= A3 F2 E1									

	Coef	Diagramme 56			Diagramme 57			Diagramme 58			Diagramme 59		
	Mach.	Temps	Mnœuds	Suite	Temps	Mnœuds	Suite	Temps	Mnœuds	Suite	Temps	KNœuds	Suite
Spock	4,5	1h39'18"	1 828	+ H5 H4 H3	15'25"	**356**	- B5 A4 F2	2h13'55"	3 662	+ G1 F2 H3	3"	**16**	+ G8 B7 E8
Forest	4,5										1"	216	+ G8 E8
Isaac	5	1h43'52"		+ H5				2h17'14"		+ G1			
Saio	5			+ H5							2"		+ H4
NewThor	5,2	1h43'33"	2 584	+ H5 D8 C7 G4	2h02'00"	3 696	- A6 B5 B4 E2	1h35'28"	2 718	+ G1 F2 H3 H5	39"	3 467	+ G8 E8 H4 B8
Logistello	5,2	3h19'57"		+ H5 H4	1h52'13"		- A6 B5 A5	3h51'22"	3 689	+ G1 F2	10"	2765	+ G8 E8 H4
Brutus	5,33	48'42"		+ H5 D8 C7 C8	1h50'44"		- B5 A4 G8 H8	50'58"		+ G1 F2 H3 H5	8"		+ G8 E8 H4 B8
Hannibal	5,33	2h06'13"		+ H5 H4 H3 D8	3h06'53"		- A6 B5 B4 E2	1h57'22"		+ G1 F2 H3 D8	10'31"		+ G8 E8 H4 B8
Zebra	7	**29'41"**	**1 200**	+ H5 H4 H3	**8'19"**	406	- G7 B5 A5	**26'00"**	**1300**	+ G1 F2 H3	3"	1080	+ G8 E8 H4
Keyano	11,5												

Recherche du meilleur coup

	Coef	Diagramme 40			Diagramme 41			Diagramme 42			Diagramme 43		
	Mach.	Tps	Knœuds	Suite	Tps	Knœuds	Suite	Temps	Knœuds	Suite	Temps	Knœuds	Suite
Spock	4,5	40"	24 879	A2 B1 C1	1'04"	28 484	H4 A3 A2	1'15"	37 449	G2 H1 C2	5'06"	117 611	G3 H6 H5
Forest	4,5	1'05"	79 341	A2 B1	4'55"	316 711	H4 A3	2'48"	198 238	G2 H1	10'09"	552 241	C7 H4
Saio	5	**11"**	**7 354**	A2 B1 C1 PS	51"	30 600	H4 A3 A2 G6	2'28"	111 749	G2 H1 C2 G1	1'10"	43 072	C7 H4 H5 B8
NewThor	5,2	30"	20 065	A2 B1 C1 PS	58"	31 319	H4 A3 A2 G6	1'17"	51 346	G2 H1 C2 G1	3'03"	85 831	C7 H4 H5 B8
Logistello	5,2	57"	47 976	A2 B1 C1	1'44"	75 979	H4 A3	1'16"	56 387	G2 H1 C2	7'16"	292 415	G3 H6
Brutus	5,33	21"		A2 B1 C1 PS	43"		H4 A3 A2 G6	43"		G2 H1 C2 G1	2'48"		C7 H4 H5 B8
Hannibal	5,33	37"		A2 B1 C1	1'09"		H4 A3 A2 G6	55"		G2 H1 C2 G1	1'23"		C7 H4 H5 B8
Zebra	7	27"	21 700	A2 B1 C1	**28"**	**27 500**	H4 A3 A2	**28"**	**28 200**	G2 H1 C2	**37"**	**31 500**	C7 H4 H5
Keyano	11,5	2'42"	50 139	A2 B1 C1	4'38"	63 155	H4 A3 A2	4'45"	76 764	G2 H1 C2	21'52"	326 660	C7 H4 H5

	Coef	Diagramme 44			Diagramme 45			Diagramme 46			Diagramme 47		
	Mach.	Temps	Knœuds	Suite	Temps	Knœuds	Suite	Temps	Knœuds	Suite	Temps	Knœuds	Suite
Spock	4,5	**1'15"**	**23 886**	B8 G5 D2	18'28"	493 603	B2 C1 G5	4'32"	**139 523**	B3 C1 B1	1'04"	**26 777**	G2 B8 B7
Forest	4,5	4'08"	256 094	D2 G5	1h13'25"	529 012	B2 C1	14'28"	969 866	B3 C1	3'32"	227 093	G2 B8
Saio	5	3'29"	90 053	B8 G5 D2 A3	24'10"	1 196 140	B2 C1 G5 H4	7'44"	261 485	B3 C1 B1 A3	**57"**	30 989	G2 B8 B7 A2
NewThor	5,2	54"	28 827	D2 G5 B8 A3	13'25"	**407 878**	B2 C1 G5 H6	6'21"	222 025	B3 C1 B1 A3	1'21"	47 171	G2 B8 B7 A2
Logistello	5,2	3'19"	126 972	D2 G5 B8 A3	32'27"	1 366 795	B2 C1 G5	6'16"	255 850	B3 C1	2'19"	89 445	G2 B8 B7
Brutus	5,33	2'29"		D2 G5 B8 A3	11'59"		B2 C1 G5 H6	2'45"		B3 C1 B1 A3	1'06"		G2 B8 B7 B2
Hannibal	5,33	1'22"		D2 G5 B8 A3	14'51"		B2 C1 G5 H6	4'35"		B3 C1 B1 A3	2'42"		G2 B8 B7 A2
Zebra	7	**51"**	45 500	D2 G5 B8	**8'14"**	488 000	B2 C1 G5	**2'39"**	168 000	B3 C1 B1	1'20"	82 200	G2 B8 B7
Keyano	11,5	9'18"	126 803	D2 G5 B8	1h59'33"	1 786 990	B2 C1 G5	32'27"	451 307	B3 H3 H2	12'29"	173 623	G2 B8 B7

	Coef	Diagramme 48			Diagramme 49			Diagramme 50			Diagramme 51		
	Mach.	Temps	Knœuds	Suite	Temps	Knœuds	Suite	Temps	Knœuds	Suite	Temps	Knœuds	Suite
Spock	4,5	20'33"	663 558	F6 G5 G3	19'31"	**597 849**	E1 H4 G6	1h51'06"	2 751 847	D8 E8 G8	26'49"	673 340	E2 H2 G2
Forest	4,5	1h53'46"	3 620 231	F6 F2	2h23'40"	5 114 589	E1 H4						
Saio	5	**2'46"**	**91 522**	F6 G5 G3 F2				**31'52"**	1 648 852	D8 E8 G8 H8	42'46"	1 327 680	E2 H2 G7 F1
NewThor	5,2	21'16"	652 173	F6 G5 G3 F2	29'44"	988 921	E1 H4 G6 G4	42'54"	**1 211 000**	D8 E8 G8 H8	24'18"	724 762	E2 H2 F1 D1
Logistello	5,2	33'34"	1 414 798	F6 G5	46'51"	1 950 435	E1 H4	6h10'09"		D8 E8	37'14"	1 418 896	E2 H2
Brutus	5,33	11'53"		F6 G5 G3 F2	**11'01"**		E1 H4 G6 H6	41'22"		D8 E8 G8 H8	**10'39"**		E2 H2 G7 F1
Hannibal	5,33	13'02"		F6 G5 G3 F2	28'33"		E1 H4 G6 G4	2h21'14"		D8 E8 G8 H8	27'48"		E2 H2 F1 D2
Zebra	7	8'45"	500 000	F6 G5 G3	19'55"	1 200 000	E1 H4 G6	42'55"	2 240 000	D8 E8 G8	10'44"	**577 000**	E2 H2 G7
Keyano	11,5	2h40'43"	2 641 415	F6 G5 G3	3h24'51"	2 658 018	E1 H4 G6	18h32'06"	14 854 039	D8 E8 G8	3h49'06"	2 617 529	E2 H2 G7

	Coef	Diagramme 52			Diagramme 53			Diagramme 54			Diagramme 55		
	Mach.	Temps	Knœuds	Suite	Temps	Mnœuds	Suite	Temps	Mnœuds	Suite	Temps	Mnœuds	Suite
Spock	4,5	20'10"	492 852	A3 F2 E1	3h19'42"	6 002	D8 B6 D7	5h38'18"	9 927	C7 G6 G5	19h52'08"	29 273	G6 H7 E2
Forest	4,5												
Saio	5	45'13"	1 131 941	A3 F2 E1 F1									
NewThor	5,2	20'24"	610 951	A3 F2 E1 F1				7h59'36"	15 823	C7 G6 G5 F6	19h39'54"	33 607	G6 H7 E2 F2
Logistello	5,2	48'09"	1 840 167	A3 F2									
Brutus	5,33	12'06"		A3 F2 E1 F1	1h45'10"		D8 B6 D7 C2	2h26'45"		C7 G6 G5 F6	13h32'09"		G6 H7 C1 F2
Hannibal	5,33	22'12"		A3 F2 E1 F1	4h20'55"		D8 B6 D7 A5	12h57'27"		C7 G6 G5 F6	27h38'47"		G6 H7 C1 F2
Zebra	7	**8'50"**	**462 000**	A3 F2 E1	**1h10'18"**	**3 950**	D8 B6 D7	**1h53'14"**	**6 330**	C7 G6 G5	**7h39'44"**	**24 000**	G6 H7 C1
Keyano	11,5	2h19'24"	1 480 788	A3 F2 E1									

	Coef	Diagramme 56			Diagramme 57			Diagramme 58			Diagramme 59		
	Mach.	Temps	Mnœuds	Suite	Temps	Mnœuds	Suite	Temps	Mnœuds	Suite	Temps	Knœuds	Suite
Spock	4,5	2h05'29"	**2 384**	H5 H4 H3	3h00'41"	4 731	A6 B5 B4	3h05'37"	4 606	G1 F2 H3	12"	**27**	G8 B7
Forest	4,5										5"	4 202	G8 E8
Saio	5	3h28'06"		H5 H4 H3 D8				2h48'49"		G1 F2 H3 H5			
NewThor	5,2	4h03'25"	6 548	H5 H4 H3 D8	8h15'41"	14 758	A6 B5 B4 E2	4h50'55"	8 308	G1 F2 H3 H5	40"	3 467	G8 E8 H4 B7
Logistello	5,2										10"	2765	G8 E8 H4
Brutus	5,33	1h06'59"		H5 H4 H3 D8	3h02'52"		A6 B5 B4 E2	2h08'35"		G1 F2 H3 H5	8"		G8 E8 H4 B8
Hannibal	5,33	2h28'55"		H5 H4 H3 D8	16h43'38"		A6 B5 B4 E2	5h02'34"		G1 F2 H3 D8	31"		G8 E8 H4 B7
Zebra	7	**58'44"**	2 420	H5 H4 H3	**1h23'45"**	**4 630**	A6 B5 B4	**1h05'36"**	**3 210**	G1 F2 H3	**3"**	1 080	G8 E8 H4
Keyano	11,5												

Tests de finales par le programme Forest

par Olivier Casile

Mettre au point un programme d'Othello est un plaisir sans fin pour le passionné de programmation. D'abord parce que les règles relativement simples du jeu se prêtent particulièrement bien à l'utilisation de toute la palette des algorithmes classiques (alpha-bêta, minimax, dichotomie, tris, hashing), ensuite parce que sa stratégie est suffisamment complexe pour inciter à la création de nouveaux algorithmes toujours plus efficaces. En dehors de cet aspect purement algorithmique, il en est un autre qui me passionne, c'est l'optimisation du code, dont le but est de le rendre aussi efficace que possible. À chaque fois que je me penche sur mon programme Forest pour en optimiser un module, une fonction, ou même une simple boucle, je me dis après des heures de labeur pour ne gagner parfois qu'une portion de % : « cette fois j'ai été jusqu'au bout, cette routine est parfaite, je ne ferai jamais mieux ». Eh bien l'expérience m'a prouvé que c'est faux, et que l'on finit toujours par trouver encore mieux.

L'article qui suit fait donc le point sur les performances que l'on peut obtenir d'un programme d'Othello, en jouant sur les algorithmes, les processeurs, les systèmes d'exploitation, et bien sûr les compilateurs.

Une remarque de dernière minute avant de se lancer dans les chiffres : au vu des résultats comparatifs publiés ici, Forest se situe au meilleur niveau pour la recherche de l'issue d'une partie (gain/perte/nulle), et un peu en retrait pour la recherche du score exact. J'observe toutefois que les programmes obtenant les meilleurs résultats sont ceux qui utilisent de grandes quantités de mémoire pour stocker et réutiliser l'arbre d'analyse lors de passes successives, ce que Forest ne fait pas afin de pouvoir tourner sur le PC de monsieur tout le monde. Mais la concurrence créant l'émulation, je viens de faire quelques essais dans cette direction, montrant un gain allant jusqu'à 40%, mais au prix d'une consommation de mémoire pouvant se chiffrer en mégaoctets.

Je m'interroge donc sur l'utilité de livrer ce genre d'algorithme dans une prochaine version de Forest. Si vous voulez donner votre avis...

Simplification de l'analyse

Afin de réduire le temps de calcul lors de la recherche de l'issue de la partie à des profondeurs importantes (>18), Forest effectue, au niveau « tournoi » seulement, un léger élagage de l'arbre des coups à examiner. Si comme le montre le tableau 1 le gain est en moyenne de 50%, cet élagage a par contre pour effet de rendre le résultat incertain. Le taux de confiance que j'accorde aux résultats obtenus de cette manière est d'environ 90%, mais même lorsqu'il y a erreur dans le diagnostic, le coup sélectionné est généralement l'un des meilleurs. Il est bien sûr possible d'augmenter le gain en élaguant plus, mais le risque d'erreur augmente d'autant...

• Deux erreurs (*) ont donc été commises par Forest : dans la position 25, le coup A5 donnant la nulle n'a pas été trouvé. Dans ce cas Forest choisit le meilleur coup résultant de la première passe alpha-bêta, soit H2 qui est en fait le moins mauvais coup perdant, par -12. Dans la position 32, Forest trouve le coup G3 avantageux (nul ou gagnant) alors qu'il ne l'est pas, mais constitue néanmoins le meilleur coup perdant, par -4. L'erreur est donc sans conséquences dans ce cas. À noter que dans la position 28, c'est le coup F1, et non E1, qui a été trouvé avantageux, ce qui est aussi exact.

On peut donc considérer qu'il y a eu une erreur et demie sur 16 cas, ce qui est conforme aux 90% de réussite que j'ai estimés, d'autant que les positions proposées sont pratiquement toutes des cas limites, et que l'échantillon n'est donc pas représentatif des parties généralement évaluées par Forest.

Comparaisons entre processeurs

Afin de comparer les performances de divers processeurs, le problème 27 (position perdante) a été analysé avec les systèmes suivants (voir tableau 2). On remarquera dans ce tableau que les deux systèmes équipés de 486DX2-66 ne sont pas d'égale performance : mon clone perso, avant que je ne change sa carte mère était environ 30% plus rapide qu'un PS/2 Server 9585.

Pb.	sans élagage			avec élagage		
	résultat	temps	nœuds	résultat	temps	nœuds
24	nulle (C3)	28,1	10090861	avantage (C3)	22.5	7974911
25	nulle (A5)	43,3	15202978	perte (*)	23.4	8102487
26	nulle (D8)	138.4	44659181	avantage (D8)	51.8	16621301
27	perte	79.8	25765043	perte	22.5	7014801
28	nulle (E1)	180.0	65291616	avantage (F1)	44.2	15371247
29	gain (A1)	24.6	7303212	avantage (A1)	14.5	4120411
30	nulle (G3)	45.8	13886417	avantage (G3)	26.9	7968871
31	perte	27.6	8108595	perte	13.5	3873613
32	perte	17.3	5471795	avantage (G3) (*)	25.8	8435210
33	perte	112.1	39231076	perte	57.3	19642423
34	perte	98.9	34980131	perte	27.3	9408876
35	nulle (C7)	119.8	34596434	avantage (C7)	52.9	14962993
36	nulle (B7)	487.1	162478920	avantage (B7)	339.8	111123275
37	perte	10.2	2467078	perte	5.6	1257688
38	gain (B2)	2264.2	670784185	avantage (B2)	791.9	237912828
39	gain (F7)	0.4	46912	avantage (F7)	0.4	46912

Tableau 1 : résultat de la recherche avec ou sans élagage

J'attribue cela au fait qu'il disposait de RAM 70 ns et que j'avais réduit au maximum les wait-states lors du setup, alors que les PS/2 ont des RAMs 80 ns, et que leurs wait-states sont auto-déterminés au démarrage. Autre surprise, et ce malgré les publicités tapageuses d'Intel, un Pentium 90 n'est même pas deux fois plus rapide qu'un 486DX2 au mieux de sa forme ! Il est toutefois généralement admis que le Pentium ne révèle toute sa puissance que dans les calculs en virgule flottante, quand il ne se trompe pas dans les résultats... Forest n'utilisant que des entiers, je n'ai pu vérifier.

Comparaisons entre compilateurs

Afin de comparer les performances du code généré par les deux compilateurs C dont je dispose sur mon PC, et d'étudier l'influence des diverses options de compilation possibles, j'ai effectué les mesures suivantes pour le problème 27 (voir tableau 3).

Commentaires

Influence de l'alignement des données

Le Pentium accédant aux données par bloc de 64 bits, c'est logiquement l'alignement des données sur frontière de 64 bits qui donne le meilleur résultat, mais de très peu, sans doute car Forest utilise beaucoup plus de tableaux que de structures, ce qui réduit le gain possible. Les alignements sur frontière de 2 et 4 octets donnent le même résultat, alors que l'absence d'alignement est nettement défavorable, car dans ce cas même les tableaux d'entiers courts ou longs peuvent être mal alignés. À noter que les résultats seraient un peu différents sur un 486 : le meilleur résultat serait obtenu avec l'alignement sur 32 bits, alors que l'alignement sur 64 bits n'apporterait absolument rien, les 486 (sauf SLC d'IBM et de Cyrix) étant des processeurs 32 bits. L'absence d'alignement serait elle aussi pénalisante sur un 486.

Influence du mode de passage des paramètres

Le compilateur Watcom gagne au moins 5% en passant les paramètres aux routines via les registres du processeur. Dans certains cas, j'ai même pu observer des gains allant jusqu'à 30%, mais le passage par registre « consomme » les registres EAX, EBX, ECX et EDX, ce qui pénalise l'optimiseur car ces registres sont alors moins disponibles pour conserver des variables fréquemment utilisées. Il y a néanmoins toujours un gain net à passer les paramètres par registre. Le compilateur Borland dispose aussi d'une option permettant de passer les paramètres par registre (fastcall), mais je n'ai jamais pu obtenir de gain de performance en l'utilisant.

Influence de l'optimisation Pentium

Le Pentium dispose de deux unités de calcul séparées, ce qui lui permet d'effectuer deux opérations en même temps, pour peu qu'elles ne dépendent pas l'une de l'autre. Le Pentium dispose aussi d'un sous-ensemble d'instruction RISC s'exécutant en 1 cycle d'horloge. Le compilateur Watcom est capable de jouer sur les deux tableaux :

- il « RISCifie » (terme Watcom) le code, c'est-à-dire décompose les instructions complexes en instructions courtes,
- il réarrange les instructions de façon à regrouper celles qui peuvent s'exécuter en même temps. Par exemple la ligne de code C :

```
if (X==2) Y++; else Y--;
```

produira

```
    mov   edx, Y
    cmp   X, 2
    jne   L1
    inc   edx
    jmp   L2
L1  dec   edx
L2  mov   Y, edx
```

ce qui est apparement plus long, mais en fait plus efficace que

```
    cmp   X, 2
    jne   L1
    inc   Y
    jmp   L2
L1  dec   Y
L2
```

Dans le cas d'Othello, le gain est toutefois faible, car les portions de code critiques, comme par exemple la boucle qui retourne les pions adverses compris entre deux pions du joueur, sont composées d'instructions interdépendantes, qui ne peuvent être réarrangées. Il est important de noter que l'optimisation Pentium n'apporte de gain que sur un Pentium. Elle provoque par contre une dégradation équivalente des performances sur un 486. Il suffit de regarder les séquences d'instructions de l'exemple précédent pour le comprendre. La riscification ajoute des instructions, qu'un 486 ne pourra paralléliser, et qu'il perdra donc des cycles à exécuter. Donc si l'on veut un code efficace sur la majorité des processeurs Intel, mieux vaut éviter les optimisations Pentium. Le compilateur Borland, dans sa version 32 bits, dispose

Système	processeur	cache	indice Norton Sysinfo	Temps de calcul issue
PC/AT	Pentium 90	16k interne 256k externe	286	79.9
PC/AT	486DX2-66	8k interne 256k externe	144	150.8
PS/2 9585-0NG	486DX2-66	8k interne 256k externe	123	220.6
PS/1 2133-174	486DX-33	8k interne 128k externe	69	310.6
PS/VP 425SX/S	486SX-25	8k interne	54	410.3

Tableau 2 : analyse du problème 27 sur différents systèmes.

aussi d'une option d'optimisation Pentium, qui ne fait que du réarrangement sans riscification.

Influence de la stratégie d'optimisation (taille contre vitesse)

La contradiction entre la recherche d'un code de taille minimale et la recherche de la performance maximale apparaît clairement ici : l'optimisation de la taille de code réduit la taille de l'exécutable de 5%, et la taille du code hors runtime de 10%, mais les performances s'en trouvent dégradées de 12%. En ce qui concerne le code de Forest, qui pour pouvoir se limiter au modèle mémoire « small » lorsqu'il est compilé en mode 16 bits, doit être aussi compact que possible sans dégrader les performances, j'utilise une stratégie mixte : tous les modules de calcul sont compilés avec les options donnant le maximum de performances, alors que tous les modules gérant l'affichage, qui représentent environ la moitié du code, sont compilés avec des options réduisant sa taille. Le code de Forest, en modèle small, fait ainsi environ 60 Ko, et il reste donc encore un peu de place pour quelques nouvelles fonctions... jusqu'à la disparition définitive du mode 16 bits.

Utilisation de l'assembleur

Forest dispose de trois fonctions critiques réécrites en assembleur. J'ai choisi ces fonctions car elles représentent à elles seules environ 25% du temps de calcul, sont aisément isolables et faciles à écrire en assembleur. Le module équivalent en C est toutefois maintenu, pour des raisons de portabilité. Pour la position 27, le temps de calcul avec le compilateur Watcom et avec les fonctions critiques en assembleur était de 79,8 s. En utilisant du C « pur », le temps de calcul est de 80,0 s. Il est donc clair qu'un bon compilateur C fait aussi bien qu'un programmeur en assembleur, et qu'il ne faut utiliser le langage machine que si l'on tient absolument à grappiller les derniers pouillèmes de secondes de calcul. À noter toutefois que la différence est plus sensible en mode 16 bits ou avec un compilateur faible en optimisation, comme le Borland.

Influence du mode de fonctionnement du processeur

Le mode protégé (32 bits) des 386/486/Pentium est dans l'absolu plus performant que le mode réel (16 bits), même dans son modèle mémoire le plus favorable (small). La raison principale est que le jeu d'instructions 32 bits est à peu près orthogonal : tous les registres généraux (EAX, EBX, ECX, EDX, ESI, EDI, EBP) sont équivalents et peuvent être utilisés comme registres d'index dans la plupart des instructions, alors que dans le jeu d'instructions 16 bits seuls BX, SI, DI sont utilisables pour cela.

Un compilateur 32 bits dispose donc de beaucoup plus de latitude pour affecter les registres aux variables, et donc pour optimiser le code. Pour être efficace en mode 32 bits, il faut toutefois utiliser autant que faire ce peut des entiers de type « int », et bannir absolument les « short », qui obligent le compilateur à insérer des instructions du mode 16 bits (un octet de préfixe permet au processeur de s'y retrouver) qui ralentissent considérablement le processeur.

Influence du modèle mémoire

Il existe pour les x86 six modèles mémoire : tiny, small, compact, medium, large et flat. J'exclus ici le modèle huge qui est en fait un modèle large avec ajout par les compilateurs d'une arithmétique sur les pointeurs permettant de s'affranchir de la limite de 64Ko (un segment) par variable.

• Le modèle tiny est un archaïsme permettant essentiellement de produire des fichiers exécutables au format .COM pour DOS. Le code, les variables et la pile se trouvant dans un seul et unique segment, le modèle tiny n'est utilisable que pour des programmes très simples, et n'est pas supporté par Windows.

• Le modèle small autorise un segment (64Ko) de code et un segment de données. C'est le modèle mémoire permettant de générer le code le plus rapide car il permet d'éviter toute manipulation des registres de segment du x86. Tous les adressages, code et données, sont alors « near ». Forest utilise ce modèle mémoire lorsqu'il est compilé en mode 16 bits, et les temps d'exécution montrent que c'est effectivement le plus rapide des modèles 16 bits. Dans le cas d'un programme d'Othello, qui peut nécessiter beaucoup de données, comme des tables de bords ou des bibliothèques d'ouvertures, l'unique segment de données peut s'avérer insuffisant. Heureusement la plupart des compilateurs 16 bits permettent, même en modèle small, de déclarer des variables « far » qui seront mises dans des segments séparés.

• le modèle medium autorise plusieurs segments de code et un segment de données. Il n'est pas trop pénalisant (13% tout de même) si l'on ne peut vraiment plus tenir dans un segment de code.

• le modèle compact autorise plusieurs segments de données et un segment de code. Il est extrêmement pénalisant (94%) car toute manipulation de donnée globale ou tout passage de paramètre nécessite la manipulation d'un registre de segment. Ce modèle est donc à éviter à tout prix, ce qui est en général possible en ajoutant des « extra-segments » au modèle small, comme déjà expliqué.

• le modèle large autorise plusieurs segments de données et de code. Du point de vue performance c'est le plus défavorable, car il utilise à fond la segmentation, pour le code et les données.

• le modèle flat est le modèle 32 bits par excellence. Il permet l'adressage linéaire de 4 Go de code, données, pile, sans segmentation et sans la limitation artificielle des 64 Ko par variable. En réalité, les registres de segment (en fait des sélecteurs) sont toujours là, mais sont tous égaux et pointent vers un unique segment de 4 Go. Le modèle mémoire flat n'est donc en réalité qu'un modèle tiny 32 bits ! Le compilateur Watcom permet d'ailleurs de générer des programmes 32 bits en modèles medium, compact et large manipulant des pointeurs 48 bits (16 bits de sélecteur + 32 bits d'offset). Du point de vue performances, le modèle flat 32 bits n'apporte rien de particulier par rapport au modèle small 16 bits, mais d'une part il est lié au mode protégé des x86, qui est le plus rapide nous l'avons vu, et d'autre part il permet de s'affranchir de toute contrainte de taille de code ou de données, à condition d'avoir un compilateur et un système d'exploitation le supportant.

Influence du compilateur

Si l'on souhaite écrire un programme d'Othello dans un langage évolué, l'utilisation d'un bon compilateur est capitale. Je ne m'attarderai pas ici sur le choix du langage de programmation, mais dirai simplement qu'à mon avis il y a trois bons choix possibles : C, C combiné à C++, et Pascal (ceci dit celui qui écrira un bon programme d'Othello en COBOL ou FORTRAN méritera la reconnaissance éternelle du monde othellistique). Forest est entièrement écrit en C, mais vu la complexité croissante de l'interface utilisateur, je songe de plus en plus à une combinaison de C et C++, toute la partie calcul restant bien sûr en C.

En ce qui concerne les compilateurs C pour processeurs x86, ce n'est pas le choix qui manque et j'en vois au moins cinq bons sur le marché (Borland, Metaware, Microsoft, Symantec/Zortec, Watcom), et bien que ne les ayant pas tous évalués, je peux indiquer pourquoi et comment j'utilise le Borland et le Watcom pour Forest :

• Borland C++ 4.5

Forest, dans ses débuts, utilisait le Borland C++ 2.0, et c'est donc tout naturellement qu'il a suivi l'évolution vers le 4.5. Le compilateur Borland est très simple à utiliser, dispose d'un très bon débogueur, et compile vraiment très vite. C'est donc un bon outil de mise au point. Il y a toutefois eu rupture entre les versions 3.1 et 4.0 de ce compilateur, et l'optimiseur a hélas fait les frais de l'évolution. L'option -Oe (global register allocation), pourtant censée favoriser les performances, est par exemple inutilisable car elle a pour effet d'empêcher le compilateur d'utiliser les registres SI et DI pour stocker des pointeurs, d'où un désastre côté performances. Depuis la version 4.0, Borland fournit une version 32 bits de son compilateur, permettant de produire des exécutables 32 bits pour Windows NT ou WIN32S. Rappelons que WIN32S est une extension de Windows permettant de faire tourner des programmes 32 bits, destinés à Windows NT, sous Windows 3.1. Ce compilateur 32 bits produit hélas un code mal optimisé, plus lent que le code 16 bits correspondant, et nécessite de plus l'installation de WIN32S sur le système Windows 3.1 cible avant de pouvoir exécuter les programmes obtenus. La version 4.5 des compilateurs Borland, 16 et 32 bits, est néanmoins en léger progrès comme le montrent les chiffres.

• Watcom C++ 10.0

J'utilise ce compilateur d'origine canadienne depuis sa version 9.0, et comme les chiffres le montrent, il dispose d'un optimiseur redoutable. C'est certainement ce qu'il y a de mieux en la matière, avec le Metaware High C. Watcom livre deux versions de son compilateur, une version 16 bits et une version 32 bits, mais la version 16 bits est moins explosive. Ces compilateurs supportent une palette exceptionnelle d'environnements, aussi bien comme plate-forme de développement que comme cible : DOS, Windows 3.1, Windows NT, OS/2 2.0... et viennent avec un DOS extender et un Windows extender gratuits. Le DOS extender permet de faire tourner du code 32 bits sous DOS, alors que le Windows extender permet de le faire tourner sous Windows 3.1, sans WIN32S. La version compétition de Forest utilise ce Windows extender pour tourner en 32 bits sous Windows 3.1, alors que la version freeware se contente du Borland en mode 16 bits.

Il est important de noter que les résultats de l'optimisation produite par ces compilateurs varient suivant la quantité de mémoire disponible lors de la compilation. Plus il y a de mémoire, plus les optimiseurs disposent de place pour essayer des combinaisons dans l'arborescence représentant le code. En ce qui concerne Forest, 8 Mo sont toutefois suffisants. Les chiffres montrent par ailleurs que les options « optimisation maximale » fournis par les compilateurs ne donnent pas forcément les meilleurs résultats. Il est donc nécessaire d'étudier l'influence individuelle de chacune des options d'optimisation pour déterminer la combinaison d'options la plus performante.

Je n'utilise pas le compilateur Microsoft pour Forest, mais les essais effectués pour cette étude montre des performances plutôt décevantes, avec par ailleurs des temps de compilation excessifs.

Système d'exploitation

Tous les résultats précédents ont été obtenus sous DOS, ce qui est le cas le plus favorable, car DOS n'étant pas multitâche, il laisse pratiquement l'entièreté du processeur à l'application qui s'exécute, à condition bien sûr que quelque TSR ou virus ne se serve pas au passage.

Le tableau 4 donne quelques temps mesurés pour la position 27.

Les mesures Windows sont effectuées en mode 386 étendu, et sans aucune autre application DOS ou Windows active ou iconisée. Les temps du compilateur Borland sont donnés pour les versions 4.0 puis 4.5.

Ces chiffres appellent plusieurs remarques :

• Le code 32 bits produit par le compilateur Borland C 4.0 donne les plus mauvais résultats. Je vous avais prévenu, donc pas de surprise. La version 4.5 de ce compilateur est par contre en progrès, mais le code 32 bits reste tout de même en retrait par rapport au code 16 bits, ce qui n'est pas normal.

• Le code 16 bits sous Windows ne perd pratiquement rien par rapport à sa version DOS, alors que le code 32 bits perd 9% dans la bataille, tout en restant plus rapide que le code 16 bits. Le code généré par le compilateur pour les modules de calcul étant strictement indépendant de l'environnement, j'explique cela par une déperdition due au Windows extender adjoint par Watcom au code 32 bits, qui doit perdre beaucoup de temps à commuter entre mode réel et mode protégé à chaque fois que Windows prend la main.

• Le code 32 bits produit par le compilateur Watcom ne perd pratiquement rien lorsqu'il tourne nativement

Dos		fenêtre Dos		Windows		Windows + WIN32S	
16 bits Borland	32 bits Watcom	16 bits Borland	32 bits Watcom	16 bits Borland	32 bits Watcom	16 bits Borland	32 bits Watcom
100.1		105.6		100.8		109.4	
99.1	79.8	104.6	83.9	99.9	86.5	100.9	80.0

Tableau 4 : analyse du problème 27 suivant différents systèmes.

sous WIN32S plutôt qu'avec le Windows extender, ce qui conforte l'explication précédente. Vivement Windows 95, qui contiendra WIN32S et permettra donc de ne plus développer qu'en 32 bits...

• Les programmes DOS perdent environ 5% lorsqu'ils tournent dans une fenêtre DOS.

Si l'on souhaite tourner sous Windows avec le maximum de performances, il y a donc intérêt à développer un programme Windows natif plutôt que de faire tourner un programme DOS sous Windows.

Ceci devrait avoir un impact sur nombre de programmes d'Othello existants lorsque Windows se débarrassera de ce bon vieux DOS en tant que soubassement, comme l'a déjà fait OS/2.

Compilateur	Options de compilation		Temps de calcul
Watcom C++ 10.0	/5r /s /zp4 /oailrt	code Pentium, mode protégé (32 bits) passage paramètres par registres suppression stack checking alignement data sur 32 bits suppression alias checking fonctions mémoire inline optimisation boucles réarrangement instruction optimisation temps	79.8
	/5r /s /oailrt /zp8	(alignement data sur 64 bits)	79.6
	/5r /s /oailrt /zp2	alignement data sur 16 bits)	79.8
	/5r /s /oailrt /zp1	(alignement data sur 8 bits)	81.5
	/5s /s /oailrt /zp4	(passage paramètres par la pile)	83.3
	/4r /s /oailt /zp4	(sans optimisations Pentium)	80.9
	/4r /s /os /zp4	(optimisation taille du code)	89.6
Borland C++ 4.0	-4 -N- -a -ms -Z -k -Oabcilt	code 486, mode réel (16 bits) passage paramètres par la pile suppression stack checking alignement data sur 16 bits modèle small (pas de segmentation) élimination chargements redondants de registres suppression stack frame si inutile (sauvegarde/restoration BP) suppression alias checking élimination code mort optimisation locale fonctions mémoire inline optimisation boucles optimisation temps	100.1
	-Oxt	optimisation maximale	116.1
	-1	code 8086	107.4
	-mm	modèle medium	113.1
	-mc	modèle compact	194.0
	-ml	modèle large	195.6
Borland C++ 3.1	-3 -N- -a -ms -Z -k- -Oabcilt		108.4
	-Oabceilt		105.1
	-Oxt	optimisation maximale	108.2
Borland C++ 4.5	-4 -N- -a -ms -Z -k- -Oabcilt		99.1
	-Oabceilt		110.9
	-Oxt	optimisation maximale	114.1
Microsoft C 7.0	/G2 /Gs /Zp4 /AS /Oaceilt	code 286, mode réel (16 bits) passage paramètres par la pile suppression stack checking alignement data sur 32 bits modèle small suppression alias checking optimisation locale allocation globale des registres fonctions mémoire inline optimisation boucles optimisation temps	125.9
	/Oxt	optimisation maximale	127.1

Tableau 3 : analyse du problème 27 sur différents compilateurs.

Anthologie des tournois de programmes d'Othello

par Bruno de la Boisserie

Préambule

Ce texte est le premier jet d'un article que je souhaite complet sur l'histoire déja chargée des tournois de programmes informatiques jouant à Othello. Je pense n'avoir oublié aucun tournoi français et américain, mais je sais qu'il n'en est pas de même pour les tournois canadiens et néerlandais antérieurs à la fin 1988 et suis preneur de toute information à ces sujets.

Il s'attache pour l'instant davantage aux faits qu'aux idées. En particulier, une grosse partie de ce texte ne parle que de résultats de tournoi. J'espère avoir le temps d'ajouter une partie récapitulant les idées des meilleurs programmes, ce qui leur a permis de « faire la différence » avec leurs challengers, qui je pense intéressera tous les programmeurs... mais ce n'est pas encore fait.

Je tiens à disposition de ceux qui le souhaitent un fichier de 126 références bibliographiques sur le sujet, ainsi qu'un recensement détaillé de tous les programmes d'Othello sur Pc, Macintosh, Atari et Amiga que j'ai eu entre les mains (350 environ). Contactez-moi sur internet (`http://perso.wanadoo.fr/brunodlb/`) ou à l'adresse suivante :

Bruno de la Boisserie
3 rue François Millet
27180 ST SÉBASTIEN DE MORSENT

I. Les origines du jeu

Malgré son nom, Othello n'est pas un jeu d'origine maure : il a été « inventé » en 1971 par Goro Hasegawa, un Japonais. Il est très fortement inspiré de « Reversi », un jeu de la fin du 19e siècle inventé en Angleterre. On trouve d'ailleurs beaucoup de programmes à ce dernier nom (à cause du copyright !).

Une des principales caractéristiques d'Othello est la simplicité de ses règles. Il s'agit de remplir un « othellier » de 8 cases sur 8, à l'aide de pions noirs sur une face et blancs sur l'autre. Chaque joueur joue une couleur, le vainqueur étant celui dont la couleur domine lorsque la partie se termine (en général quand toutes les cases sont jouées). Chaque coup consiste en le retournement d'au moins un pion adverse. Il faut pour cela encadrer un ou plusieurs pions de la couleur opposée entre un pion ami déjà présent et le pion joué lors de ce coup. Les pions adverses « encerclés » sur une ou plusieurs lignes sont alors retournés. La prise peut s'effectuer verticalement, horizontalement ou en diagonale, mais seulement en ligne droite et lors de la pose du pion (le retournement d'un pion ne lui donne pas le droit de déclencher immédiatement d'autres retournements).

La simplicité de cette règle, en comparaison de celle des échecs, a séduit de nombreux programmeurs...

II. Les premiers programmes

Le plus vieux programme d'Othello connu est sans doute un certain *IAGO* écrit par l'université Caltech de Pasadena, en Californie. Plusieurs programmes portèrent ce nom (ce qui porte à confusion). Celui-ci est mentionné dans un article du *Time* du 22 novembre 1976, qui commentait le voyage aux USA du champion japonais d'alors, Fumio Fujita : *« À Pasadena, en Californie, Fujita battit facilement un programme nommé* IAGO *créé par des étudiants de Caltech. »*. (Source : Peter Michaelsen, *Othello Quarterly* Vol. 10 n°2, page 5.)

La revue américaine de micro-informatique *BYTE* a publié début 1978 un programme d'Othello en Fortran. Voici ce qu'en dit J.B. Touchard, de l'université Paris VI, auteur d'un célèbre programme télématique d'Othello nommé « Chiens et Chats » (source : *Éducation & Informatique* n°16, mai-juin 83) : *« La version d'Othello que nous avons mise au point était une adaptation d'un programme assez peu puissant décrit dans* BYTE *il y a environ 5 ans. Une adaptation judicieuse de la stratégie nous a permis d'aboutir à un produit convenable. »*.

En France, un autre programme aura un impact beaucoup plus grand : c'est le journal *L'ordinateur Individuel* qui le publia dans son numéro 1, en octobre 1978.

Aucun de ces programmes n'était fort, du moins comparé au niveau actuel. Le premier programme d'un bon niveau est peut-être *MAX* de Rob Phillips. Son auteur le décrit notamment dans le numéro d'*Othello Quarterly* (revue de la Fédération Américaine d'Othello) du printemps 1979. Mark Weinberg, l'un des meilleurs joueurs américains de l'époque (et premier président de la Fédération Américaine d'Othello) disait de lui : *« MAX est un terrible programme, du niveau des dix meilleurs joueurs américains »*. Ce programme, à l'origine écrit en Pascal, fut incorporé dans une machine à cristaux liquides diffusée par *Gabriel Industries*, le diffuseur américain d'Othello à l'époque, puis dans certaines cartouches de jeu (par exemple pour l'ordinateur Texas TI99-4a).

III. 1979-1980 : les premiers tournois

En France, le premier tournoi de programmes se déroula à l'initiative du journal *L'ordinateur Individuel*, le 26 mai 1979 à Paris. Six participants s'affrontèrent (dont deux programmes du journal) sur cinq rondes. C'est Philippe Keller, sur SWPTC 6800, qui remporta le tournoi.

Quelques mois plus tard, le 1er décembre 1979, se déroulait le « second tournoi de l'O.I. » des mêmes organisateurs. Douze participants, répartis par moitié entre la catégorie « interprété » et la catégorie « compilé/assemblé » firent parler la poudre tout au long des cinq rondes. Noël Revoil et son TRS-80 remportèrent la victoire de la première catégorie, tandis qu'Adrien Holliger remportait celle de la seconde, sans avoir subi de défaite.

Le mois d'avril 1980 est à marquer d'une pierre blanche, à cause de la parution d'un article de David Levy dans le numéro 16 de *L'ordinateur Individuel*. C'était le premier d'une série d'articles sur les jeux de réflexion qui allait s'étendre sur vingt numéros et près de deux ans... L'auteur, alors Maître International d'échecs, s'était rendu célèbre à la fin des années 60 pour avoir parié qu'aucun ordinateur ne le battrait avant 10 ans. Ce pari,

qu'il gagna, l'amena à devenir l'un des meilleurs spécialistes du sujet. Il dirigea longtemps l'*International Computer Chess Association.* Cette série d'articles eut le mérite de présenter au grand public les techniques issues de la recherche universitaire américaine, utilisées avec succés par les meilleurs programmes d'échecs et de dames de l'époque. En France, les premiers à l'utiliser allaient assez vite « faire la différence »...

Dès lors, les compétitions allaient se succéder à intervalles réguliers : le 19 avril 1980 se tint le premier tournoi « décentralisé », à Rouen. Six participants s'affrontèrent et le tournoi fut remporté haut la main par Jean Maingourd, sur TRS 80, avec 10 points sur 10 et 257 pions, soit une moyenne de gain de 51 à 13 ! Il est vrai que le programme était l'un des premiers à utiliser une recherche sur plusieurs niveaux grâce à la technique du « minimax à élagage alpha-bêta ».

Moins d'un mois plus tard, le 10 mai 1980, 24 concurrents se pressent au troisième « tournoi d'Othello-Reversi de l'O.I. ». Et parmi eux, pour la première fois, deux étrangers : l'espagnol Joan Ludevid et surtout l'anglais David Levy et son équipe. Celui-ci utilise pour la circonstance un programme en Pascal sur IBM 370, et remporte le tournoi (10 points sur 10, 242 pions) en catégorie « compilé/assemblé », suivi d'Adrien Holliger, Vincent Leroux et Jean Maingourd.

De l'autre côté de l'Atlantique, aux États-Unis, on s'active aussi... mais semble-t-il plus sérieusement. Le professeur Peter Frey de l'université de Northwestern organise le premier tournoi d'Othello hommes-machines le 19 juin 1980. Deux humains y participent : le Japonais Hiroshi Inoue, champion du monde 1979, et l'Américain Jonathan Cerf qui devint quelques mois plus tard le champion du monde 1980. Côté ordinateurs, trois micros et trois gros systèmes sont présents. Parmi les auteurs on peut citer David Levy, les époux Spracklen (auteurs de *Sargon*), Tom Truscott (auteur d'un excellent programme de dames), Peter Frey... C'est Hiroshi Inoue qui, logiquement, remporta le tournoi. Mais il eut aussi le désagréable privilège de perdre sa partie contre le programme de David Levy. À ma connaissance, c'est la première fois qu'un champion du monde perdait une partie contre un ordinateur dans un jeu de réflexion pure. Il faut cependant souligner que ce résultat est dû à une grosse erreur en finale, que peu de joueurs de club actuels commettraient... La seconde place du tournoi fut prise par le programme des époux Spracklen, sur Apple II, grâce à sa victoire contre Jonathan Cerf. Il faut savoir que ce dernier avait testé une version préliminaire du programme quelque temps plus tôt, et que c'est lui qui avait fourni les « conseils d'expert » pour l'améliorer. Si vous possédez un *Reversi Sensory Challenger*, vous pourrez vous-même tester la différence : les niveaux « Novice » sont ceux du programme initial, les « Expert » sont ceux de la version corrigée...

Retour en France le 20 septembre 1980, pour le premier tournoi international d'Othello-Reversi, toujours organisé par le journal *L'Ordinateur Individuel.* Cinq nationalités et vingt concurrents s'y affrontèrent. Les Anglo-saxons ne se déplacèrent pas pour rien, puisque parmi les quatre premiers de la catégorie « compilé/assemblé », on trouve le programme *Microthello* de l'Américain Mike Riley, puis *The Moor* de l'Anglais David Levy, *Otelic* du Français Jean Maingourd et *Odin* de l'Américain Peter Frey...

L'année 1980 se termine avec l'apparition dans les boutiques françaises d'une machine à cristaux liquides, d'origine japonaise mais diffusée par Dujardin. Elle joue, bien sûr, à Othello, mais le niveau reste moyen.

IV. 1981-1983 : L'expansion

En janvier 1981 se tint aux USA un tournoi de programmes organisé par le professeur Peter Frey, à l'université de Santa Cruz, en Californie. Vingt programmes étaient présentés, dont plusieurs déjà connus. Les quatre premiers connurent tous une carrière commerciale : *IAGO*, de Paul Rosenbloom, *Aldaron* de Charlie Heath (diffusé sous le nom *Master Reversi*), le programme de Dan et Kathe Spracklen (celui de la machine dédiée *Reversi Sensory Challenger* alias *Reversal* sur Apple II) et enfin *Odin* de Peter Frey et Larry Atkin.

De plus en plus fort : 34 participants s'affrontent le 23 mai 1981 à Lyon, en France, sur cinq rondes. Huit programmes participent à la catégorie « compilés/assemblés » dont *Reversi Sensory Challenger*. Celui-ci pulvérise tout le monde et établit un record de domination : 10 points sur 10, 274 pions, soit une moyenne de 55 pions à 9 ! Douze programmes s'affrontent en catégorie « interprété », remportée par Maurice et Jean-Michel Claverie, avec *Otelion* sur HP 9825. Innovation : quatorze programmes participaient à la catégorie « Poche ».

Même domination du *Reversi Sensory Challenger* au 3e tournoi international d'Othello-Reversi de l'O.I. en septembre 1981, avec aucune défaite sur les cinq rondes et 265 pions (soit 53-11 de moyenne) en catégorie « compilé/assemblé », forte de seize programmes. En catégorie « interprété », 26 programmes sont présents, et c'est une nouvelle fois Maurice et Jean-Michel Claverie qui gagnent. Du coté des 35 ordinateurs de poche, Jean-Pierre Leroy gagne les trois rondes. Incontestablement, la programmation d'Othello devient un loisir à la mode : ils étaient 77 en tout !

Cette année 1981 voit l'introduction en France du *Reversi Sensory Challenger* de Fidelity Electronics, tout auréolé de ses précédents succès, et de l'*Odin Encore* d'Applied Concepts (alias *Odin Edition* de la *Great Game Machine*). Ces machines contiennent de bons programmes et ont longtemps été les partenaires d'entraînement de plusieurs grands noms de l'Othello français dont Paul Ralle, champion du monde 1984.

Le 15 et 16 mai 1982, une nouvelle édition du *« North American Computer Othello Championship Tournament »* a lieu. Quatorze programmes, dont huit sur micro-ordinateurs, jouèrent les huit rondes. Aux trois premières places, on retrouve Charlie Heath et son *Aldaron*, devant Anders Kierulf avec *Brand* et Peter Frey avec *Odin*, respectivement sur TRS-80, Commodore 3032 et TRS-80.

DEUX CENTS ! C'est à peu de choses près le nombre de participants du 4e tournoi international d'Othello de l'O.I, le 26 septembre 1982. Jamais, à ma connaissance, compétition de programmes de jeu de réflexion n'a eu pareil succès. En catégorie « compilé/assemblé », seul Bruno Bras remporte ses cinq parties, grâce à sa bonne programmation mais sans doute aussi à sa machine, un kit Intel à base de 8086. Il est suivi de Sylvain Quin et de Vincent Baillet. Le seul invaincu de la catégorie « interprété » est Maurice Barret-Catan, suivi de Christian Saussard et de M. Ernotte.

Un an plus tard, les 24 et 25 septembre 1983, a lieu la 5e édition du tournoi international de l'O.I. 96 concurrents sont là, dont 20 en « compilé/assemblé », 28 en « interprété » et 48 en « poche ». On retrouve à la première place de la catégorie « compilé/assemblé » *Inthello*, le programme de Bruno Bras. Mais pour la première fois depuis longtemps, le vainqueur n'a pas remporté toutes ses parties : un certain François Aguillon et son programme *Comp'oth* lui ont arraché le match nul (il ne finira malgré tout que 10e). Le second est Vincent Baillet et son *Reversi Champion* (qui connut une carrière commerciale), suivi de Maurice et Jean-Michel Claverie, nouvellement acquis à cette catégorie. Du côté des « interprétés », victoire de Gilles Bisson devant P. Kaplan et Philippe Picou. En « poche », le tiercé gagnant est Cayre, Vernot et Tordjmann.

V. 1984-1989 : en Europe, le règne de Comp'oth

Février 1984 : Un tournoi se déroule à l'initiative de l'antenne belge du journal *L'Ordinateur Individuel*. Seulement quatre programmes concourent en catégorie « compilés/assemblés », dont deux inscrits par François Aguillon. L'un d'eux, baptisé du nom chatoyant de *Speedy Moulard,* est censé permettre au frère de François de ne pas trop s'ennuyer et a été écrit pour l'occasion quelques jours plus tôt (sur un Commodore Pet 8Ko !). Surprise, il remporte le tournoi ! Inutile de préciser que les versions ultérieures de *Comp'oth* descendent beaucoup plus de ce *Speedy Moulard* que de son homonyme de l'époque !

En juillet 1984, un petit tournoi se tient à Gagny, en banlieue parisienne. Seulement douze participants, mais quasi-exclusivement des programmes de catégorie « compilé/assemblé », dont plusieurs anciens champions. Sept rondes devaient, a priori, permettre de les départager. Malgré cela, cinq programmes se partagèrent la première place, avec cinq victoires sur sept : *Reversi Sensory Challenger* (niveau expert 6) ; *Inthello 2.9* et *Inthello 3.2*, deux versions du programme champion en titre ; la machine dédiée *Odin Encore* ; *Comp'oth*, enfin, dont c'est le premier grand succès. Derrière, on trouve entre autres plusieurs versions de *Reversi Champion* et *Modot*, un programme anglais qui malgré une analyse à seulement trois coups de profondeur gagne quatre parties sur sept.

Deux mois plus tard, se tint le 6e tournoi international d'Othello de l'O.I. Cette fois-ci, c'est durant six rondes que les concurrents s'affrontèrent. C'est François Aguillon et son programme *Comp'oth* sur Apple II qui remportait une nouvelle fois la victoire en catégorie « compilé/assemblé », devant Jean-Christophe Weill et Bruno Bras. En « interprété », le premier était M. Prudhomme, suivi de l'équipe Kombar/Camus et de M. Lereverend. En « poche compilé », deux versions du programme de M. Kombar remportent les deux premières places, Didier Ernotte prenant la troisième. Enfin, en « poche interprété », victoire de M. Bornes devant M. Cambien et Laurent Tordjmann.

Le 21 septembre 1985, 84 programmes viennent essayer de conquérir le podium du 7e tournoi international d'Othello de l'O.I. Et en catégorie « compilé/assemblé », c'est toujours *Comp'oth* de François Aguillon qui tient le haut de la marche et de loin : il termine invaincu sur les cinq rondes, avec 245 pions soit un gain moyen de 49 à 15. Il faut dire qu'il dispose depuis peu d'une carte 68000 à 10 MHz, une vraie bête de course pour l'époque, dans son Apple II. Son dauphin est Claude Hell, sur Apple Macintosh (une autre machine à base de 68000 !). En « interprété », victoire de Patrick Audinet sur CPC 464 devant Denis Joubert et son compatible PC. En « poche compilé », c'est Antoine Capron qui gagne, suivi de Didier Ernotte. Enfin, en « poche interprété », le CANON X-07 remporte les deux premières places, grâce à Laurent Tordjmann et Etienne Colella.

Le 9 novembre 1985, coup de tonnerre sur le sol canadien : le programme *Bill*, de l'université de *Carnegie Mellon* (USA) fait sa première apparition et ne perd aucune de ses quatre parties. Il termine premier, devant neuf autres programmes.

Retour aux USA les 15 et 16 février 1986. Onze programmes s'y affrontèrent sur huit rondes, dont seulement quatre sur « gros-systèmes ». C'est une nouvelle fois *Aldaron*, de Charlie Heath, sur TRS-80 modèle 1, qui gagne le tournoi, son plus mauvais résultat étant une nulle. La qualité du programme a encore une fois fait la différence face à des machines beaucoup plus rapides. Derrière, *Bill*, sur Vax 11/785, prend la seconde place. *Brand*, d'Anders Kierulf, sur IBM PC, et *Excalibur*, de Lance Fortnow et Chris Eisnaugle, sur ATT 6300 (= OLIVETTI M 24) prennent la troisième place.

En France, il fallut attendre le 22 mars 1987 (soit près d'un an et demi) pour qu'un nouveau tournoi ait lieu, à l'initiative de François Aguillon. L'équipe du journal *L'Ordinateur Individuel* avait en effet complètement changé, et il n'était plus question ni d'organiser un tournoi, ni même d'assurer un quelconque soutien médiatique. Coup dur, donc, qui se ressentit sur le nombre de participants : dix seulement, tous en « compilé/assemblé », et dont deux fois le même auteur ! *Reversi Sensory Challenger expert 5* et *Odin Niveau 9* (version Apple II) furent engagés à titre de « témoin » de niveau. *Comp'oth 68000* concervait sa couronne, son plus faible score étant de 42 à 22 contre le *Reversi Sensory Challenger* qui pris la troisième place, la seconde étant détenue une nouvelle fois par Claude Hell. À noter que ce dernier programme a depuis été versé dans le domaine public.

Nouvelle victoire de *Comp'oth 68000* le 13 mars 1988, au 2e tournoi de St-Michel sur Orge organisé par l'auteur, François Aguillon. Douze programmes étaient présents en tout et la bataille fit rage pour la seconde place. C'est finalement *Reversi Sensory Challenger expert 5* qui se l'adjugeait, suivi de *Inthello* de Bruno Bras, d'une version de *Microb* (la famille Claverie) et du programme de Claude Hell.

Le 12 novembre 1988 se tint le 5e tournoi annuel de programmes de l'université de Waterloo, au Canada. Sept programmes s'affrontèrent mais, comme le règlement stipulait que le premier coup était blanc, beaucoup de programmes ne purent utiliser leur bibliothèque d'ouverture. *Chung hoa* de Duy-Minh-Nhieu, sur VAX, remporta le tournoi devant *Zoth*, sur AMIGA 2000.

Retour en Europe le 11 février 1989, pour le tournoi d'ordinateurs d'Utrecht, aux Pays-Bas. Vingt participants étaient présents, dont *Comp'oth*, *Brand* et de nombreux programmes néerlandais. *Comp'oth 68000*, de François Aguillon, remportait les six rondes et la finale, les places d'honneur étant prises par **VRT* de Roemer Lievaart, *Mast 87* de Ron Kroonenberg et *VERS2* de Ben De Woolf.

Une semaine plus tard, c'est aux États-Unis que se déroulait le tournoi de Northridge, réunissant l'élite

nationale. *Bill*, de Kai Fu Lee et Sanjoy Mahajan remportait le tournoi, avec une seule défaite contre *Peer Gynt*, d'Anders Kierulf, une version remaniée de *Brand*.

Le 12 mars 1989, la troisième édition du tournoi de St-Michel sur Orge, en France, eut lieu. Seize programmes y participaient, dont *Comp'oth* sur ATARI ST, *Reversi Sensory Challenger*, *Peer Gynt*... Cinq rondes ne suffirent pas aux adversaires du premier cité pour lui faire perdre une partie, avec toutefois une sévère alerte à la dernière ronde, avec le match nul obtenu par *Badia* de Marcel Van Tien. *Thor*, de Sylvain Quin, s'adjugeait la seconde place, suivi de *Badia* et du programme de Jean-Christophe Weill. Il faut noter que ces trois derniers programmes étaient en langage C, ce qui laissait augurer des versions sur machines plus puissantes dans le futur...

C'est le 11 juin 1989 que se réunirent à Pérenchies, dans le Nord de la France, douze programmes dont *Comp'oth ST*, *Reversi Sensory Challenger*, *Thor* et *Badia*. Après sept rondes disputées, *Comp'oth ST* restait le seul invaincu, suivi de *Badia* et *Théole* (deux défaites chacun).

Londres, 9 août 1989 : le plus long tournoi de programmes d'Othello jamais organisé commence. L'événement a lieu dans le cadre des premières olympiades informatiques, organisées par David Levy. 28 parties seront jouées par les quinze programmes d'Othello présents, au cours d'un double toutes-rondes. À l'issue de ce marathon ne laissant aucune place au hasard des couleurs et des appariements, c'est un programme anglais, *Polygon*, écrit par Alex Selby en assembleur sur Acorn Archimède, qui remporte le tournoi avec 26 points sur 28 possibles. Derrière, on trouve *Comp'oth* (23/28), *Badia* (21/28), *Jonathan* (18,5/28) et *Thor* (17,5/28). Le niveau de ce tournoi était tel que lors du tournoi hommes-machines qui clôtura l'événement, le 15 août 1989, l'équipe des ordinateurs, constituée de ces cinq programmes, gagna par 12 points sur 20 contre celle des humains qui comprenait pourtant trois des meilleurs joueurs du monde ! On ne peut que regretter l'absence de *Bill* qui, tournant sur VAX, ne pouvait concourir, le règlement stipulant que la machine devait être physiquement présente sur le lieu du tournoi...

Côté américain, le tournoi de l'Université de Waterloo, en novembre 1989, vit la nette victoire d'un nouveau venu, *Harold*, sur DecStation 3100, devant *Parallelo* (sur un réseau de 85 compatibles PC !) et *Thumper* (sur VAX 8600). L'examen des parties montre cependant que la majorité des 14 programmes présents jouait franchement mal — parfois même en finale !

VI. 1990-1992 : La montée en puissance

Le premier tournoi de la décennie 90 fut le 4e tournoi international de St-Michel sur Orge, le 11 mars 1990. Seize joueurs dont cinq Hollandais étaient présents, prêts à en découdre sur sept rondes. Quelques favoris manquaient cependant à l'appel, comme *Polygon*, *Thor* ou *Othello Master*. Une fois de plus, *Comp'oth* s'imposait, devant *Badia* et *VERS2*. On notera la forte progression de programmes comme *Jacp'oth* ou *Othel du Nord*, qui viennent juste derrière.

Quelques jours plus tard eu lieu le tournoi d'Utrecht. *REV 90* remporta le tournoi devant *Mast 90* et *My Turn*. *Badia* dut se contenter de la quatrième place. Quand à *Comp'oth*, il dut partager une inhabituelle cinquième place avec *VERS2* et *Prothello*...

Nouvelle victoire des Néerlandais au tournoi de Pérenchies, en juin 90. *Dumbo* arrachait à la surprise générale la première place, suivi de *Jacp'oth* et de *Comp'oth*, ex aequo.

La montée en puissance des programmes néerlandais se confirma à la seconde *Computer Olympiad* de Londres, en août 90. Seulement six programmes d'Othello firent le déplacement, au lieu de quinze l'année précédente, sans doute à cause des 50£ de droit d'inscription. *Dumbo* confirma son nouveau leadership, suivi de *VERS2* et de *Microb*.

De l'autre côté de l'Atlantique, *Harold* confirmait sa domination en remportant le 17 novembre le tournoi annuel de l'université de Waterloo, devant sept autres programmes. Il termina cependant ex aequo avec *Peer Gynt* et dut concéder une défaite au troisième, *Desdemona's Revenge*.

Le 10 mars 1991, fin d'une époque : au 5e tournoi international de St-Michel sur Orge, *Comp'oth* perdait sa première partie à « domicile » contre *Thor*. Mais surtout, *Jacp'oth* terminait invaincu sur les sept rondes ! *Othel du Nord et Comp'oth* se partageait la seconde place, suivi de *Thor*. On put noter lors de cette compétition que les meilleurs programmes ne fonctionnaient pas sur les machines les plus puissantes (un Archimède A 310 et un Tandon 486/25), ce que divers tournois récents auraient pu laisser craindre...

L'édition 1991 du tournoi de Pérenchies, qui se tint le 9 juin, accueillit seulement neuf participants. Mais aucun ténor ne manquait à l'appel, excepté *Dumbo*. A l'issue de neuf rondes très disputées, c'est finalement *Othel du Nord* qui s'adjugea la première place, suivi de *Jacp'oth*, *Comp'oth* et *Thor*. L'analyse des finales, comme au précédent tournoi, révéla que certaines parties gagnées (par exemple par *Othel du Nord*) étaient perdantes ou nulles au coup précédent. Ce résultat semble prouver que l'écart de niveau est de plus en plus réduit entre les meilleurs programmes français.

Les programmeurs néerlandais se retrouvèrent quant à eux nombreux à la troisième *Computer Olympiad* qui se tint à Maastricht (NL) du 22 au 28 août 1991. Un seul programme français, *Purée*, y participait, contre huit néerlandais, un finlandais et un russe. Il finit cinquième sur son Atari 1040 ST, derrière *Prothello* (Atari TT), *Mast 91* (PC 486), *REV 91* (PC 486) et *VERS2* (Acorn Archimède).

L'université de Waterloo accueillit le 28 novembre 1991 douze programmes et un joueur humain (non-officiel) pour son 8e tournoi annuel. Une nouvelle fois, c'est *Harold* (DecStation 5100) qui s'imposait, devant le joueur humain Colin Springer, suivi de cinq ex aequo dont *Desdemona's Revenge* (SparcStation), *REV 5.6* et *Jonathan* (PCs 386).

Pour la première fois depuis longtemps, un tournoi hommes-machines eut lieu le 8 mars 1992 à Paris. Ce tournoi réunissait sept excellents programmes français contre sept joueurs français de haut niveau, chaque joueur affrontant tous les programmes présents (pas de matches entre ordinateurs ou entre humains). *Thor*, sur Pc 486/33, termina le tournoi invaincu, devant *Othel du Nord* (Pc 386/25c) et *Comp'oth* (Atari 68000/8) à un point. À noter la bonne prestation de Dominique Penloup et de Paul Ralle, qui dépassèrent la moyenne contre les programmes. Au total, victoire des machines par 29,5 points à 19,5. L'équipe humaine, auquel il manquait de nombreux bons joueurs français (notamment les numéros 2, 3 et 4 au classement national), s'est bien comportée, d'autant plus que diverses parties ont été perdues dans les

derniers coups : si les joueurs humains n'avaient pas fait d'erreur en finale, ils auraient gagné 23 parties. La rumeur selon laquelle les ordinateurs sont imbattables à Othello est visiblement exagérée !

Le sixième tournoi de St-Michel, le 15 mars 1992, accueillit huit programmes. Les participants décidèrent donc de faire un toutes-rondes. C'est *Othel du Nord* (PC 386/25c) qui l'emporta avec six victoires sur sept, devançant d'un point *Théole* (PC 486/33). *Comp'oth* (ST 68000/8) et *Spock* (PC 486/25) se partagèrent la troisième place, mais avec seulement 3,5 points sur 7 !

C'est une nouvelle fois *Othel du Nord* (PC 386/25c) qui remporta le quatrième tournoi de Pérenchies, le 12 juillet 1992, devant onze autres participants. *Théole,* malgré son PC 486/50, dut partager la deuxième place avec *Jacp'oth* (Atari Mega STE 68000/16) et *Spock* (PC 486/25).

En août 1992 se tint la quatrième édition des *Computer Olympiad*, à Londres. *Othel du Nord* (486SX/20 et parfois 486/33) et *Jacp'oth* (ST 68000/8) y étaient présents et remportèrent respectivement la médaille d'or et celle de bronze. Le programme néerlandais *Aida* (SparcStation 2) remporta quant à lui la médaille d'argent.

Affluence inhabituelle au tournoi de l'université de Waterloo (Canada), le 22 novembre 1992 : 25 programmes étaient présents, dont quatre français. C'est justement le français *Cassio* (Macintosh), de Stéphane Nicolet, qui avec 4,5 points sur 5 remportait le tournoi, suivi de *Fugazi*, *REV 6.1* et *Rodionov* à 4 sur 5, puis des français *Spock* et *Gthor* à 3,5/5. Détail amusant, aucun de ces programmes ne tournait sur les stations de travail hyper-rapides habituellement utilisées par les gagnants de l'épreuve. On peut cependant regretter qu'avec un tel « plateau », il n'ait pas été possible de jouer davantage de parties.

VII. 1993-???? : Les universitaires se rebiffent

C'est un tournoi d'un genre nouveau qui se tint le 4 avril 1993 aux Ulis, dans la banlieue de Paris : il s'agissait d'un tournoi en cadence « blitz » (dix minutes par joueur). *Othel du Nord* remporta une nouvelle fois ce tournoi, suivi de *Spock* (en nette progression) puis de *Dumbo* et *Comp'oth*. Il faut noter l'avant-dernière place de la machine dédiée *Reversi Challenger* (niveau expert 4). Cela montre bien les énormes progrès accomplis depuis dix ans...

Le 12 juin 1993 se tint la cinquième édition du tournoi de programmes de Courchelettes (anciennement de Pérenchies). L'excellent niveau de celui-ci avait cette année attiré deux néerlandais, un canadien et deux américains, dont le mythique *Bill*, dans sa version 4. Celui-ci participait par téléphone depuis l'université de Carnegie-Mellon, à Pittsburgh aux USA ! C'est finalement les deux français *Spock* et *Othel du Nord* qui terminaient en tête le tournoi, suivi de près par *Dumbo*. *Bill* dut se contenter de la quatrième place ex aequo avec *Fugazi* et *Jacp'oth*.

Le tournoi annuel de la *Computer Olympiad* ayant été annulé, l'université allemande de Paderborn, en Allemagne, organisa du 5 au 7 octobre 1993 un tournoi de programmes pour le remplacer. Douze programmes s'affrontèrent et *Logistello*, jusqu'alors inconnu, remporta largement la victoire, avec 16,5 points sur 18 possibles, suivi de *Keyano* (13,5) et de *REV* (12).

La dixième édition du tournoi de Waterloo, le 13 novembre 1993, accueillit une nouvelle fois plus de vingt participants. Ce fut hélas au préjudice du nombre de parties jouées : seules quatre rondes purent être terminées. À l'issue de celles-ci, *Logistello* et *Stell* se partageaient la première place (avec 3,5 points chacun), suivi à un demi-point par *Brutus*, *Keyano*, *REV*, *Nicksrev*, *VERS2* et *Chigorev*. Compte tenu du faible nombre de parties jouées pendant le tournoi principal, les six premiers jouèrent par réseau interposé une finale en double toutes-rondes, dans la dernière semaine de novembre 1993. C'est une nouvelle fois *Logistello* qui remportait le tournoi, avec 6.5 points sur 10, suivi de *Keyano* à 5.5. À noter l'extraordinaire domination de la couleur blanche dans cette finale à six joueurs, puisque seulement quatre parties (sur 30 !) furent perdues par cette couleur !

Sur une idée de Sylvain Quin, le 9 janvier 1994 vit l'organisation d'un tournoi blitz à ouverture imposée : il s'agissait pour cette première édition de l'ouverture Rose. La cadence choisie (10 mn par partie) permit la tenue d'un double toutes-rondes entre les programmes présents. C'est une nouvelle fois *Spock* qui empocha la première place avec 11,5 points sur 14, suivi par *Cassio* à 10/14 et *Keyano* à 8,5/14.

C'est le 8 mai 1994 que se tint la troisième édition du tournoi annuel hommes-machines entre les joueurs et les programmes français. La forte équipe cybernétique s'imposa sans mal face à une équipe humaine démotivée, sur le score cinglant de 32 à 4. Les trois programmes de tête furent *Keyano*, *Spock* et *Othel du Nord*, tous les trois auteurs d'un sans-faute. Ironie du sort, le jour précédent avait eu lieu un tournoi traditionnel ou précisément trois des quatre programmes présents avaient déçu par leurs mauvais scores...

C'est à un tournoi de très haut niveau que les participants de la sixième édition du tournoi de Courchelettes, le 19 juin 1994, furent conviés. Y participaient en effet *Logistello*, sur SunSparc 10/M20, *Polygon*, sur Archimède A 310, *Othel du Nord* sur 486DX2/66 et *Spock* sur Pentium 66, qui terminèrent dans cet ordre devant douze autres programmes. *Logistello*, qui ne perdit aucune partie, fit une nouvelle fois la preuve de son actuelle domination.

Lors du tournoi joué à l'aide du réseau Internet en Juillet 1994, *Logistello*, sur SunSparc 10/M51 confirmait ce résultat, devançant *Eclipse*, le nouveau programme de Colin Springer, sur IBM RS/6000 et *Pee Wee*, sur Pentium 90.

La deuxième édition du tournoi blitz à ouverture imposée se tint le 4 septembre 1994 et fut remportée par *Spock*, suivi de *Cassio* et de *Thor* ; l'ouverture imposée étant la « Tigre Tamenori ».

Les 1 et 2 octobre 1994 eut lieu la seconde édition du tournoi de l'université de Paderborn, joué une nouvelle fois en grande partie à l'aide du réseau Internet. Le tenant du titre, *Logistello*, mais aussi le programme *Eclipse* remportèrent haut la main le tournoi, avec onze points sur douze possibles. Les huit autres participants durent se contenter des miettes, le premier d'entre eux, *Kitty* (le nouveau nom du *REV* d'Igor Durdanovic) n'obtenant que 6,5 points sur 12. Signalons enfin que la machine la moins puissante du tournoi était déjà un 486/50, la majorité des autres utilisant des SunSparc 10/M20 et M30.

Format de la base Wthor

par Sylvain Quin et Stéphane Nicolet

Généralités

La base de Wthor est d'origine PC/Intel. Aussi, les données de type *Word* et *Longint* sont-elles stockées au format Intel, c'est-à-dire l'octet de poids faible devant.

En-tête

Tous les fichiers de la base de données de Wthor possèdent un en-tête de 16 octets, suivi d'un certain nombre d'enregistrements ayant tous une taille identique. L'en-tête est constitué comme suit :

Libellé	Taille	Type
Siècle de création du fichier	1	Byte
Année de création du fichier	1	Byte
Mois de création du fichier	1	Byte
Jour de création du fichier	1	Byte
Nombre d'enregistrements N1	4	Longint
Nombre d'enregistrements N2	2	Word
Année des parties	2	Word
Paramètre P1 : taille plateau de jeu	1	Byte
Paramètre P2 : type de parties	1	Byte
Paramètre P3 : profondeur	1	Byte
X	1	(réservé)

• Les quatre premiers octets représentent une signature permettant d'éviter l'écrasement d'un fichier par une version plus ancienne. Pour cela, les programmes doivent permettre une mise à jour du répertoire (ou dossier) autrement que par une copie système.
• Le nombre d'enregistrements N_1 contient le nombre de parties (fichier de parties) ou de positions (fichier de solitaires) dans le fichier. Il vaut 0 pour les fichiers de joueurs et de tournois. Le nombre d'enregistrements N_1 est limité à 2 147 483 648 parties par année pour les fichiers de parties, à 2 147 483 648 positions pour les fichiers de solitaires.
• Le nombre d'enregistrements N_2 contient le nombre de joueurs (fichier de noms de joueurs), de tournois (fichier de libellés de tournois) ou le nombre de cases vides des solitaires (fichier de solitaires) dans le fichier. Il vaut 0 pour les fichiers de parties. Le nombre d'enregistrements N_2 est limité à 65535 pour les libellés des tournois et les noms des joueurs, à 64 pour les fichiers de solitaires.
• L'année des parties vaut 0 dans les fichiers de joueurs, de tournois ou de solitaires.
• Le paramètre P_1 (dans un fichier de parties ou de solitaires) indique la taille du plateau de jeu :

0 : plateau de jeu standard 8x8
8 : plateau de jeu standard 8x8
10 : plateau de jeu 10x10

Il vaut 0 dans tous les autres cas.
• Le paramètre P_2 vaut 1 dans les fichiers de solitaires, et 0 dans tous les autres cas.
• Le paramètre P_3 (dans un fichier de parties) indique la profondeur pour laquelle est calculé le score théorique (la valeur 0 est équivalente à la valeur 22 pour les fichiers postérieurs au 01/01/2001).

Fichiers des parties sur plateau 8x8

Nom du fichier : `WTH_####.WTB`
Chaque enregistrement (68 octets) contient:

Libellé	Taille	Type
Numéro du libellé de tournoi	2	Word
Numéro du joueur noir	2	Word
Numéro du joueur blanc	2	Word
Nombre de pions noirs (score réel)	1	Byte
Score théorique	1	Byte
Liste des coups	60	Byte[]

• Il existe un fichier de parties par année. Dans un fichier de parties, celles-ci sont stockées dans un ordre quelconque, mais normalement regroupées par tournoi.
• Les #### du nom du fichier correspondent au numéro de l'année.
• Le score théorique contient le score (en nombre de pions) du joueur noir sur une finale parfaite. Cette finale est calculée sur la position dont le nombre de cases vides est égal au paramètre P_3 (profondeur). Par exemple, pour une profondeur de 22, la finale parfaite commence au coup 39, c'est-à-dire une fois le coup 38 joué.
• La liste des coups commence au coup 1.f5.
• Les coups sont stockés dans l'ordre chronologique de la partie selon le format suivant : numéroter les lignes et colonnes de 1 à 8 et effectuer l'opération *colonne + (10*ligne)*. Exemple : a1 devient 11, h1 devient 18, a8 devient 81 et h8 devient 88.
• Taille du fichier en octets : $16 + N_1 * 68$.

Fichiers des parties sur plateau 10x10

Nom du fichier : `WTH_####.WTD`
Chaque enregistrement (104 octets) contient :

Libellé	Taille	Type
Numéro du libellé de tournoi	2	Word
Numéro du joueur noir	2	Word
Numéro du joueur blanc	2	Word
Nombre de pions noirs (score réel)	1	Byte
Score théorique	1	Byte
Liste des coups	96	Byte[]

• Il existe un fichier de parties par année. Dans un fichier de parties, celles-ci sont stockées dans un ordre quelconque, mais normalement regroupées par tournoi.
• Les #### du nom du fichier correspondent au numéro de l'année.
• Le score théorique contient le score (en nombre de pions) du joueur noir sur une finale parfaite. Cette finale est calculée sur la position dont le nombre de cases vides est égal au paramètre P_3 (profondeur). Par exemple, pour une profondeur de 22, la finale parfaite commence au coup 75, c'est-à-dire une fois le coup 74 joué.
• La liste des coups commence au coup 1.g6.
• Les coups sont stockés dans l'ordre chronologique de la partie selon le format suivant : numéroter les lignes et colonnes de 1 à 10 et effectuer l'opération *colonne + (12*ligne)*. Exemple : a1 devient 13, j1 devient 22, a10

devient 121 et j10 devient 130.
• Taille du fichier en octets : 16 + N_1*104.

Fichiers des solitaires sur plateau 8x8

Nom du fichier : `SOLITAIRES_##.PZZ`
Les solitaires sont des positions de finales intéressantes extraites de la base de Wthor dans lesquelles le joueur ayant le trait a, à chaque coup, un seul coup menant au gain ou à la nulle sur la suite parfaite. Tous les solitaires d'un même fichier ont le même nombre de cases vides, indiqué dans l'en-tête et le ## du nom du fichier.

À la suite de l'en-tête principal standard de 16 octets de la base Wthor, chaque fichier de solitaires 8x8 comprend un en-tête secondaire de 512 octets constitué comme suit :

Libellé	Taille	Type
Table de présence	512	Longint []

La table de présence est un tableau de 64 entiers longs dans laquelle toutes les entrées valent 0, sauf la N_2^e qui contient le nombre d'enregistrements N_1 du fichier. Cette table de présence permet une vérification de cohérence des informations lues dans l'en-tête principal : N_1 est le nombre de solitaires et N_2 le nombre de cases vides des solitaires.

Chaque enregistrement (36 octets) est constitué comme suit :

Libellé	Taille	Type
Année du solitaire	2	Word
Numéro du tournoi	2	Word
Numéro du joueur noir	4	Longint
Numéro du joueur blanc	4	Longint
Position	16	Byte []
Nombre de cases vides	1	Byte
Trait	1	Byte
Score de la solution	1	Signed Byte
1er coup de la solution	1	Byte
Score réel de la partie	1	Byte
Coup 25 de la partie	1	Byte
X	2	(réservé)

• La position est stockée ligne par ligne, avec 2 octets par ligne. L'octet 0 code les cases a1-d1, l'octet 1 les cases e1-h1, etc., jusqu'à l'octet 15 codant les cases e8-h8. Dans chaque octet, la couleur de chaque case est codée par la combinaison de bits suivante :
case vide : 00
pion noir : 11
pion blanc : 10
• Le trait vaut 1 pour Noir, 2 pour Blanc.
• Le score de la solution contient le score, en différence de pions avec son adversaire, du joueur ayant le trait dans la position du solitaire, sur une finale parfaite.
• Le premier coup de la solution est stocké selon le format suivant : numéroter les lignes et colonnes de 1 à 8 et effectuer l'opération *colonne* + *(10* ligne)*.
• Le score réel de la partie contient le score réel (en nombre de pions) du joueur noir dans la partie dont est extrait le solitaire.
• Le coup 25 de la partie contient celui qui a eu lieu dans la partie dont le solitaire est tiré, si la partie apparaît dans la base Wthor. Il est stocké selon le format suivant : numéroter les lignes et colonnes de 1 à 8 et effectuer l'opération *colonne* + *(10* ligne)*. Il contient 0 si cette information n'est pas disponible.
• Taille du fichier en octets : 16 + 512 + N1*36.

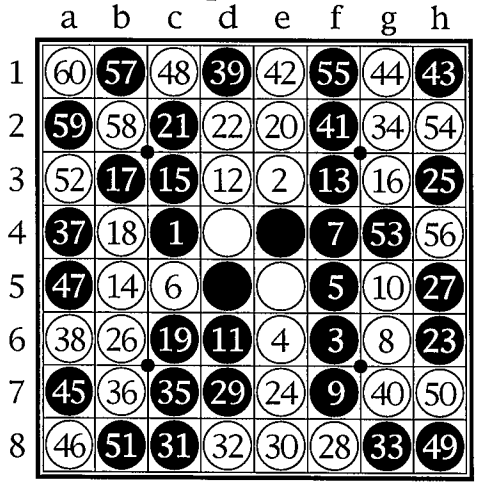
Fichier des libellés des tournois

Nom du fichier : `WTHOR.TRN`
Chaque enregistrement (26 octets) est un tableau de caractères terminé par un zéro binaire. La longueur utile est de 25 caractères.
• Taille du fichier en octets : 16 + N2*26.

Fichier des noms des joueurs

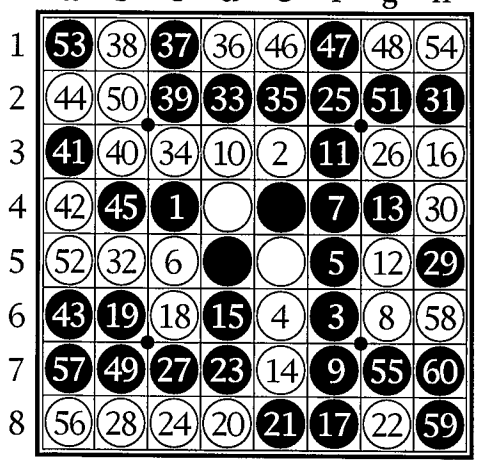
Nom du fichier : `WTHOR.JOU`
Chaque enregistrement (20 octets) est un tableau de caractères terminé par un zéro binaire. La longueur utile est de 19 caractères.
• Taille du fichier en octets : 16 + N2*20.

Diverses parties 2001



Zebra 32-32 Balder

Thématique semi-rapide 2001



Thor 33-31 Brutus

Bibliographie

par Stéphane Nicolet

Les grands fondateurs

J. von Neumann, « Zur theorie der gesellschaftsspiele », *Math. Annalen* **100** (1928), 295-230. Repris dans : A. H. Taub (ed.), *John von Neumann Collected Works* vol. **VI**, Pergamon Press, Oxford (1963), 1-26.

J. von Neumann et O. Morgenstern, *Theory of games and Economics Behavior*, Princeton University Press, Princeton N.J. (1947).

C. E. Shannon, « Programming a computer for playing chess », *Philosophical Magazine* **41** (1950), 256-75.

A. M. Turing, « Digital computer applied to games », *in* B.V. Bowden (ed.), *Faster than thought: a symposium on digital computing machines*, Pitman, London, (1953), 286-310.

Algorithmes de recherche dans les arbres de jeux

D. E. Knuth et W. Moore, « An analysis of alpha-beta pruning », *Artificial Intelligence* **6** (1975), 293-326.

G. Baudet, « The branching factor of the alpha-beta pruning algorithm », *Artificial Intelligence* **10** (1978), 173-199.

T. A. Marsland, A. Reinefeld et J. Schaeffer, « Low overhead alternative to SSS* », *Artificial Intelligence* **31** (1987), 185-199.

C. Ferguson et R. E. Korf, « Distributed tree search and its application to alpha-beta pruning », *Proceedings AAAI-88, Seventh National Conference on Artificial Intelligence* **2** (1988), 128-132.

T. Ibaraki et Y. Katoh, « Searching minimax game trees under memory space constraint », *Annals of Mathematics and Artificial Intelligence* **1** (1990), 141-153.

T. Ibaraki, « Generalization of alpha-beta and SSS* search procedures », *Artificial Intelligence* **29** (1986), 73-117.

J. C. Weill, « The NegaC* search », *ICCA Journal* **15-1**, (1992), 3-7.

L. V. Allis, M. van der Meulen et H. J. van der Herik, « Proof-number search », *Artificial Intelligence* **66** (1994), 91-124.

M. Shijf, « *Proof-number search and transpositions* », thèse de M.Sc., Université de Leiden, Pays-Bas (1993). Extraits dans : *ICCA Journal* **17-2** (1994), 63-74.

A. Reinefeld et P. Ridinger, « Time-efficient state space search », *Artificial Intelligence* **71** (1994), 397-408.

A. de Bruin, W. Piljs et A. Plaat, « Solution trees as a basis for game tree search », *ICCA Journal* **17-4** (1994), 207-219.

R. M. Hyatt, B. W. Suter et H. L. Nelson, « A parallel alpha-beta tree searching algorithm », *Parallel Computing* **10** (1989), 299-308.

J. Schaeffer, « Distributed game-tree searching », *Journal of Parallel and Distributed Computing* **6** (1989), 90-114.

Heuristique

J. Pearl, *Heuristics — Intelligent search strategies for problem solving*, Addison Welsey, Reading, Massachussetts (1984). Traduction française : *Heuristique — Stratégies de recherche intelligente pour la résolution de problèmes par ordinateur*, Cepadues éditions, Toulouse (1990).

R. E. Korf, « Depth-first iterative-deepning: an optimal admissible tree search », *Artificial Intelligence* **27** (1985), 97-109.

K. Lee et S. Mahajan, « A pattern classification approach to evaluation function learning », *Artificial Intelligence* **36** (1988), 1-25.

T. Anantharaman, M. S. Campbell et F. H. Hsu, « Singular Extensions, adding selectivity to brute-force searching », *Artificial Intelligence* **43** (1990), 99-109.

I. Althöfer, « On telescoping linear evaluation functions », *ICCA Journal* **16-2** (1993), 91-95.

A. L. Zobrist, « *A new hashing method with application for game playing* », rapport technique 88, Computer Science Department, The University of Wisconsin, Madison (1970). Repris dans : *ICCA Journal* **13-2** (1990), 69-73.

D. M. Breukler, J. W. H. M. Uiterwijk et H. J. van der Herik, « Replacement schems for transposition tables », *ICCA Journal* **17-4** (1994), 183-193.

Othello

E. Cali (ed.), *Othello par l'exemple : le championnat du monde de Paris 1988*, L'Impensé Radical, Paris (1989). On le trouve encore dans les bonnes boutiques de jeux...

P. Frey, « Machine Othello », *Personal Computing* **4-7** (1980), 89-90.

A. Kierulf, « Brand: an Othello program », *in* M. A. Bramer (ed.), *Computer game playing: theory and practice*, Ellis Horwood, Chichester (1983), 197-203.

P. S. Rosenbloom, « A world-championship level Othello program », *Artificial Intelligence* **19** (1982), 279-320.

E. Lazard, « Othello : faites votre programme », *Jeux et Stratégie* **35** (1985), 34-37.

K. Morita, K. Kunieda et N. Tsuda, « *Thinking game programming: algorithms of Othello game and practice* » (en japonais), ASCII Publ. Co., Tokyo (1986).

C. Hewlett, « Report on a hardware computing system dedicated to the game of Othello », *Othello Quarterly* **8 (2)** (1986), 6-7.

K. Lee et S. Mahajan, « *Bill: a table-based knowledge-intensive Othello program* », Carnegie-Mellon University, Pittsburgh, PA (1986). Voir aussi : « The development of a world class Othello program », *Artificial Intelligence* **43** (1990), 21-36.

G. M. Gupton, « Genetic learning algorithm applied to the game of Othello », *in* D. N. L. Levy et D. F. Beal (eds.), *Heuristic programming in artificial intelligence: the 1st Computer Olympiad*, Ellis Horwood, Chichester (1989), 241-254.

A. Kierulf, « New concepts in computer Othello: corner value, edge avoidance, access and parity », *in* D. N. L. Levy et D. F. Beal (eds.), *Heuristic programming in artificial intelligence: the 1st Computer Olympiad*, Ellis Horwood, Chichester (1989), 225-240.

A. Kierulf, « *Smart Game Board: a workbench for game playing programs, with Go and Othello as case studies* », thèse de Ph.D, ETH Swiss Federal Institut of Technology, Zurich (1990).

L. Jansen, « The Kings of Othello », *in* J. van der Herik et V. Allis (eds.), *Heuristic programming in artificial intelligence 3: the 3rd Computer Olympiad*, Ellis Horwood, Chichester (1992), 53-55.

J. Feinstein, « Forty billion nodes under the tree: Othello 6x6 is solved », *The Newsletter of the British Othello Federation, July 1993* (1993), 6-8.

J. Gnodde, « *Aïda: new search techniques applied to Othello* », thèse de MSc, Université de Leiden, Pays-Bas (1993).

M. Buro, « *Techniken für die Bewertung von Spielsituationen anhand von Beispielen* », thèse de Ph.D, Fachbereich Mathematik/Informatik – Universität GH Paderborn, 1994.

D. Moriarty et R. Miikkulainen, « *Evolving complex Othello strategies using marker-based genetic encoding of neural networks* », Technical Report AI93-206, University of Texas at Austin, September 1993.

D. Moriarty et R. Miikkulainen, « Evolving neural networks to focus minimax search », in *Proceedings AAAI-94, Twelfth National Conference on Artificial Intelligence*, 1994.

D. Billman et D. Shaman, « Strategy knowledge and strategy chance in skilled performance: a study of the game Othello », *American Journal of Psychology* **103** (1990), 145-166.

D. Levy, « Les jeux et l'ordinateur : Othello /Reversi un jeu à double face », *l'Ordinateur Individuel* **30** (septembre 1981), 93-97.

J. C. Weill, « *Programmes d'échecs de championnat : architecture logicielle, synthèse de fonctions d'évaluation, parallélisme de recherche* », thèse de Ph.D, Université Paris VIII, janvier 1995.

Autres jeux de réflexion

A. L. Samuel, « Some studies in machine learning using the game of checkers », *IBM Journal of Research and Development* **3** (1959), 210-229. Repris dans E.A. Feigenbaum et J. Feldman (eds), *Computers and Thougth*, McGraw-Hill (1963).

A. L. Samuel, « Some studies in machine learning using the game of checkers II », *IBM Journal of Research and Development* **11** (1967), 210-229.

H. J. Berliner, « Backgammon computer program beats world champion », *Artificial Intelligence* **14** (1980), 205-220.

J. Schaeffer, « Checkers: a preview of what will happen in chess? », *ICCA Journal* **14-2** (1991), 71-78.

P. W. Frey, *Chess Skill in Man and Machine*, Springer-Verlag, 1983, 2^{e} édition.

Pour aller plus loin...

L. V. Allis, H. J. van der Herik et I. S. Herschberg, « Which games will survive », *in* D. N. L. Levy et D. F. Beal (eds.), *Heuristic programming in artificial intelligence 2: the 2nd Computer Olympiad*, Ellis Horwood, Chichester (1991), 232-243.

D. Michie, « Game playing programs and the conceptual interface », *in* M. A. Bramer (ed.), *Computer game playing: theory and practice*, Ellis Horwood, Chichester (1983), 11-25.

J. Nievergelt, « Experiments in computational heuristics and their lessons for software and knowledge engineering », *Advances in Computers* **37** (1993), 167-205.

D. S. Nau, « Pathology on games trees revisited and an alternative to minimaxing », *Artificial Intelligence* **21** (1983), 221-244.

G. Tesauro, « Neuragamon — A neural-network backgammon learning program, » *in* D. N. L. Levy et D. F. Beal (eds.), *Heuristic programming in artificial intelligence: the 1st Computer Olympiad*, Ellis Horwood, Chichester (1989), 78-80.

A. Junghanns, « Fuzzy numbers as a tool in game programs », *ICCA Journal* **17-3** (1994), 141-148.

H. Iida, J. W. H. M. Uiterwijk, H. J. van der Herik et I. S. Herschberg, « Potential applications of opponent model search », *ICCA Journal* **16-4** et suivants (1993), 201-207.

H. Iida et Y. Kotani, « A strategy of game tree search modelling experts' thinking process » (en japonais), *The Transactions of information processing of Japan* **33-11** (1992), 1296-1305.