THÈSE de DOCTORAT de l'UNIVERSITÉ PARIS 8
Spécialité :
Informatique
présentée

par Monsieur Jean-Christophe WEILL
pour obtenir le titre de DOCTEUR de L'UNIVERSITÉ PARIS 8
Sujet de la thèse :

PROGRAMMES D'ÉCHECS DE CHAMPIONNAT : ARCHITECTURE LOGICIELLE, SYNTHÈSE DE FONCTIONS D'ÉVALUATION, PARALLÉLISME DE RECHERCHE.

Soutenue le *27 Janvier 1995*

devant le jury composé de :

MM.	H.	WERTZ	Président
	M.	GONDRAN	Rapporteurs
	T.A.	MARSLAND	
	J.	PITRAT	
	C.	LAVAULT	Examinateurs
	P.	GREUSSAY	
	F.-Y.	VILLEMIN	

Résumé

Nous présentons nos travaux dans le domaine de la programmation des échecs. Ces travaux ont abouti à :

- une série de programmes dont le programme CUMULUS 2.0 qui a remporté le titre de vice-champion du monde d'échecs logiciel toutes catégories;
- la méthode de construction de fonctions d'évaluation que nous avons utilisée pour introduire de l'intelligence dans nos programmes combinatoires. Nous exposerons également des résultats de synthèse de fonctions d'évaluation par apprentissage;
- un nouveau schéma efficace de parallélisation du problème de la recherche de la valeur minimax d'un arbre de recherche;

En complément de ces études, nous décrivons l'état actuel des techniques dans la programmation des jeux de réflexion de compétition, dans le domaine des algorithmes de recherche, des heuristiques applicables à ces algorithmes, dans la parallélisation des algorithmes minimax, dans les schémas évaluatifs et dans l'apprentissage.

La programmation des jeux de réflexion fut considérée comme *the drosophilia melanogaster of machine intelligence*. Ce domaine devait permettre l'élaboration de techniques et d'algorithmes réutilisables dans d'autres domaines de l'intelligence artificielle. Selon C. Shannon, il s'agit d'un sujet sensible où l'avancée est facilement communicable au public. Nous avons abordé cette question dans le cadre de programmes de jeux devant répondre à un problème, c'est-à-dire proposer un coup, dans un temps limité, c'est-à-dire des conditions de tournois.

Nous donnons l'essentiel de l'état des techniques dans la programmation des jeux de réflexion en montrant les contributions que nous avons apportées au domaine. Nous comparons les différentes recherches minimax basées sur des élagages $\alpha\beta$ avec l'algorithme NegaC* que nous avons défini, après avoir donné les principaux résultats que nous avons établis sur la complexité de cet algorithme. Nous donnons une définition, dans le paradigme negamax, du nouvel algorithme de « recherche de nombre de preuves » que nous comparons avec notre programme d'échecs ECUME dans le cadre de la recherche de mats.

Nous présentons un ensemble d'heuristiques qui permettent de rendre les recherches negamax plus rapides et plus fiables quant au choix du bon mouvement, en explicitant les options que nous avons prises dans nos programmes d'échecs.

Après avoir énuméré l'ensemble des résultats connus jusqu'alors dans le domaine de la parallélisation de l'algorithme minimax, nous présentons les résultats de nos implantations sur une machine distribuée, la Connection Machine 5. Ces résultats nous ont permis de définir une nouvelle méthode de parallélisation que nous avons comparée aux meilleures méthodes connues jusqu'alors, sur des arbres simulés, sur un programme d'Othello de compétition et sur un programme d'échecs. Nous mettons en évidence les qualités de chaque méthode.

Nous continuons par la présentation de la méthode de construction de fonctions d'évaluation que nous avons choisie, en expliquant comment nous avons pu introduire la notion de plan stratégique dans nos programmes d'échecs. Nous montrons aussi comment un programme peut construire automatiquement une fonction d'évaluation par apprentissage et les conclusions de nos expériences dans la construction d'une fonction d'évaluation correcte pour la fin de partie roi et dame contre roi et dame.

Enfin, nous décrivons l'ensemble des caractéristiques de nos programmes d'échecs, et nous mettons en évidence, à l'aide de l'analyse de leurs parties en championnat, leurs forces et leurs faiblesses.

Remerciements

Je tiens à remercier

Harald Wertz d'avoir accepté la présidence de ce jury,

Michel Gondran, Tony Marsland et Jacques Pitrat d'avoir rapporté cette thèse; je leur suis reconnaissant de leur lecture attentive et de leurs critiques constructives.

Christian Lavault et François-Yves Villemin de l'intérêt qu'ils ont manifesté pour ce travail en acceptant de faire partie du jury.

Je remercie Patrick Greussay pour avoir su diriger ces travaux.

Les programmes de jeux d'échecs et par conséquent une bonne partie de ces travaux sont issus d'une petite équipe, je tiens particulièrement à remercier Marc-François Baudot pour sa collaboration fertile. Je le remercie ainsi que Christine Duflos pour leurs relectures qui m'ont aidé à essayer d'éradiquer de ce document les nombreux passages peu clairs et mal orthographiés.

Je tiens à remercier Michel Gondran, Hervé Augendre et Jean-Luc Seret du département IMA-TIEM d'EDF pour m'avoir donné l'accès à une connection machine 5 dans le cadre du projet FRENCHESS.

Je remercie aussi tous ceux qui nous ont permis de faire concourir nos différent programmes: Pascal Petit, Jean-Luc Deschamp, Pierre Nolot, David Levy, Tony Marsland, Corinne Msika (COMPAQ France), Olivier Granboulan (SUN France), Patrick Lefevre (SONY US), John Demco (Université de Vancouver), Jacques Vernadet, André Frisch et Jorge Orellana.

Sommaire

Chapitre 1

Introduction

1.1 Historique de la programmation du jeu d'échecs

Période 1994–1980

Le 29 août 1994, dans la première ronde du tournoi INTEL-PCA de Londres, le programme CHESS GENIUS 2.9 bat avec les noirs le champion du monde PCA, Garry Kasparov, dans une partie semi-rapide de 25 minutes. Lors de la deuxième partie opposant ces deux joueurs, le champion du monde commet une gaffe dans une position supérieure, ne peut qu'annuler et est donc éliminé d'un tournoi par un programme. Bien que ce résultat ne puisse pas être considéré comme significatif de la force des deux joueurs en présence, il n'en constitue pas moins un avertissement retentissant pour les joueurs d'échecs de haut niveau[13].

En mai 1994, dans le tournoi de l'AEGON, notre programme ECUME avait frôlé l'exploit dans une partie en cadence normale contre le Grand-Maître arménien et ancien candidat au titre mondial Rafael Vaganian en dominant complètement la partie. Pour conclure, le programme décida de rentrer dans une finale gagnante ♔+♗+♗ contre ♔+♘, mais, malheureusement pour lui, nous n'avions pas jugé bon de venir avec le CDROM contenant la base de données de cette fin de partie, et le Grand-Maître parvint à annuler grâce à la règle des 50 coups.

Le programme ECUME est le successeur du programme CUMULUS 2.0[1], qui a remporté le titre de « vice champion du monde des programmes d'échecs », au championnat de Madrid 1992.

Ces championnats du monde de programmes d'échecs ont lieu tous les trois ans et sont ouverts à toutes les catégories de machines. 1992 y confirma la montée en puissance des programmes sur micro-ordinateur avec la victoire de REBEL d'Ed Schroeder, un programme

1. En fait, le vrai nom du programme CUMULUS 2.0 était ECUME, mais comme lors de l'inscription au championnat du monde de Madrid 1992, nous ne savions pas si notre nouveau programme serait prêt à temps, nous avions inscrit deux autres de nos programmes, ECHEC 1.9B et CUMULUS 1.5, respectivement Champion et vice Champion de France de programmes d'échecs 1992. ECUME étant prêt à temps, il remplaça avec brio le programme CUMULUS.

hollandais [2]. Les précédentes éditions avaient vu les victoires de DEEP THOUGHT de F. Hsu, T. Anantharaman , M.Campbell et A. Nowatzyk en 1989, CRAY BLITZ de Robert Hyatt, A. Gower et H. Nelson, HITECH de Hans Berliner, BEBE de T. et L. Scherzer et SUN PHOENIX de J. Schaeffer ex-æquo en 1986, CRAY BLITZ en 1983, BELLE de Ken Thompson en 1980. Cette domination américaine avait commencé en 1977 avec la victoire du programme CHESS 4.6 de L.Atkin,K. Gorlen et D.Slate, alors que la première édition de cette compétition avait été remportée en 1974 par le programme Russe KAISSA.

Parallèlement, tous les ans depuis 1980 se tiennent les championnats du monde de programmes d'échecs sur micro-ordinateurs, dominés jusqu'en 1984 par la firme FIDELITY, avec les programmes de Kathe et Dan Spracklen, puis de 1984 à 1993 par les programmes de Richard Lang pour la firme MEPHISTO. En 1991, le titre est partagé avec le programme CHESS MACHINE d'Ed Schroeder. [3]

La troisième grande compétition d'ordinateurs d'échecs est les championnats nord-américains qui, depuis leur création en 1970, ont lieu chaque année. Ils sont organisés par l'ACM.

1989 fut une année symboliquement aussi importante que 1994, puisque DEEP THOUGHT remporta le challenge proposé par le magazine OMNI en battant David Levy sur le score de 4 à 0. Mais c'est un David Levy diminué par le manque de compétition qui s'inclina devant la machine, et non pas le jeune champion d'Écosse qui, en 1968, avait fait le pari avec John McCarthy et Donald Michie, rejoints plus tard par Seymour Papert et Ed Kowdrowicki [4], qu'il ne serait pas battu dans un match contre un programme d'ici dix ans. En août 78, alors que ce premier pari arrivait à expiration, un match fut organisé entre le meilleur programme de l'époque, CHESS 4.7, et David Levy, devenu Maître International. Le résultat fut de 3,5 - 1,5 pour le Maître écossais [5]. David Levy et le magazine OMNI offrirent alors un prix de 5000 dollars au premier programme qui le battrait en match. Et il prit aussi un autre pari sur 5 ans. En 1984, le programme CRAY BLITZ releva le défi et perdit 4 parties à 0. La tentative suivante, celle de DEEP THOUGHT, fut la bonne, mais David Levy avait arrêté la compétition depuis longtemps.

Période 1980–1950

Le premier programme d'un niveau raisonnable, MAC HACK 6 de Richard Greenblatt[70], fut conçu suite à la défaite du programme américain d'Alan Kotok[90] face à un programme russe dont les auteurs allaient plus tard écrire KAISSA, champion du monde 1974.

Le premier programme jouant vraiment aux échecs fut celui de Bernstein[26], et les premières compétitions de programmes eurent lieu dès le milieu des années 1960. Les travaux qui permirent à ces programmes de voir le jour, et qui donnèrent à la programmation des échecs l'impulsion décisive , sont principalement les premières simulations d'un programme

2. A la deuxième place arrivaient ex-aequo ZUGZWANG, un programme sur 1024 transputers, et CUMULUS 2.0, notre programme sur compatible PC.

3. Notre meilleur résultat dans ces championnats eut lieu en 1990 où un de nos programmes, ECHEC 1.9, obtint le titre de « Champion du monde sur Ordinateur Personnel 1990 » (les programmes MEPHISTO étant jusqu'alors des programmes sur machines spécialisées).

4. Ed Kowdrowicki est un des programmeurs du programme COKO.

5. Dans ce match eurent lieu la première nulle et le premier gain d'un programme face à un Maître International.

jouant aux échecs par Alan Turing[175], mais aussi les idées matérielles exposées par Tihamer Nemes[117] et Konrad Zuse[196], sans oublier la thèse d'Adriaan de Groot[47] et l'exposé fondateur et toujours d'actualité de Claude Shannon[160].

Période 1950–1770

L'intérêt pour le domaine n'était cependant pas neuf, car le premier automate jouant sans supercherie aux échecs fut une machine construite par Torres y Quevedo, qui jouait la finale ♖ et ♔ contre ♔. Une des dernières versions de cette machine fut présentée à l'exposition universelle de Paris en 1914.[6] En dépit du succès de cette machine au mécanisme ingénieux, il a donc fallu attendre les années 40 pour que la question des machines jouant aux échecs redevienne d'actualité.

Mais des premières machines jouant aux échecs, la plus connue reste certainement l'automate du Baron de Kempelen qui apparut vers 1770. Bien sûr, comme nous le laissent supposer les possibilités technologiques de l'époque, il ne s'agissait que d'une supercherie, un joueur de petite taille étant caché à l'intérieur[7] . Mais cela inspira au moins les réflexions de Charles Babbage, qui, vers 1840, se posa le problème de savoir comment configurer la machine analytique pour qu'elle joue aux échecs[97].

1.2 Modèle de fonction d'évaluation et parallélisation de la recherche

1.2.1 Modèle de fonction d'évaluation

L'un des reproches les plus justifiés fait aux programmes combinatoires est la faiblesse de leur évaluation : pauvreté des jugements positionnels, incompréhension du jeu, absence de plan et de vision globale, indigence du jeu en finales. Le compromis entre vitesse et connaissances se faisait le plus souvent au détriment des connaissances. Dans notre recherche du meilleur compromis possible, nous sommes partis de l'approche *minimaliste* du programme TECH de Gillogly[64, 65], d'après une idée de Berliner, dans laquelle la fonction d'évaluation appliquée pendant la recherche était réduite au matériel, une rapide analyse de la position de la racine ordonnant les mouvements. Ce modèle permettait déjà d'obtenir un jeu convenable, mais pas d'introduire de notions stratégiques. Gardant les avantages de l'approche de Gillogly (la vitesse et la simplicité) tout en nous affranchissant de ses plus grosses limitations, nous avons ensuite opté pour une évaluation fondée sur un Oracle et une table positionnelle. Dans cette approche, l'Oracle, qui n'intervient que sur la position de la racine, est un véritable planificateur qui peut se livrer à une analyse stratégique fine de la position et faire appel à une bibliothèque de plans. Une fois son analyse de la racine achevée, l'Oracle en consigne les résultats dans la table positionnelle ; cette table servira à mettre à jour incrémentalement

6. Le championnat du monde des ordinateurs d'échecs de 1990 se tenait à l'université polytechnique de Madrid où se trouve cette version de la machine de Torres y Quevedo.

7. Edgar Poe démontre, à sa façon, la fraude dans son livre *Nouvelles histoires extraordinaires* en utilisant une analyse globalement pertinente mais mettant en avant deux arguments erronés :« Une machine doit toujours gagner» (ce qui n'était pas le cas de l'automate) et « Une machine doit jouer à des intervalles réguliers.»

–donc rapidement– l'évaluation lors de la recherche. Nous avons ainsi introduit des plans et de la stratégie dans nos programmes, tout en leur gardant une bonne rapidité ; les parties de championnat que nous avons disputées, contre des joueurs humains ou contre des ordinateurs, témoignent des qualités de l'évaluation par Oracle et table positionnelle. Toutefois, certaines connaissances ne sont pas représentables à l'aide d'une table positionnelle, et l'étape suivante qui allait permettre à nos programmes d'augmenter encore leur compréhension du jeu, fut l'introduction du traitement dynamique de ces connaissances. Il fallut alors réellement sacrifier de la vitesse, mais avec une nette augmentation de la qualité du jeu de nos programmes qui traitaient ainsi correctement l'évaluation de la structure de pions, de la mobilité, de la sécurité du roi, de la paire de fous, etc. Le domaine des fins de partie, toujours considéré comme délicat pour les programmes, demanda également des évaluations spécifiques : Roi et pion contre Roi, application fine de la règle du carré dans les cas les plus complexes de courses de pions à dame, ou encore évaluation parfaite de certaines finales. Le niveau de jeu augmentant, le raffinement de la stratégie dans l'Oracle devint encore plus intéressant.

Nous pensons être arrivés à un excellent compromis entre vitesse et connaissances : nous pouvons rajouter de nombreuses connaissances en ne sacrifiant de la vitesse que dans quelques cas spécifiques bien choisis.

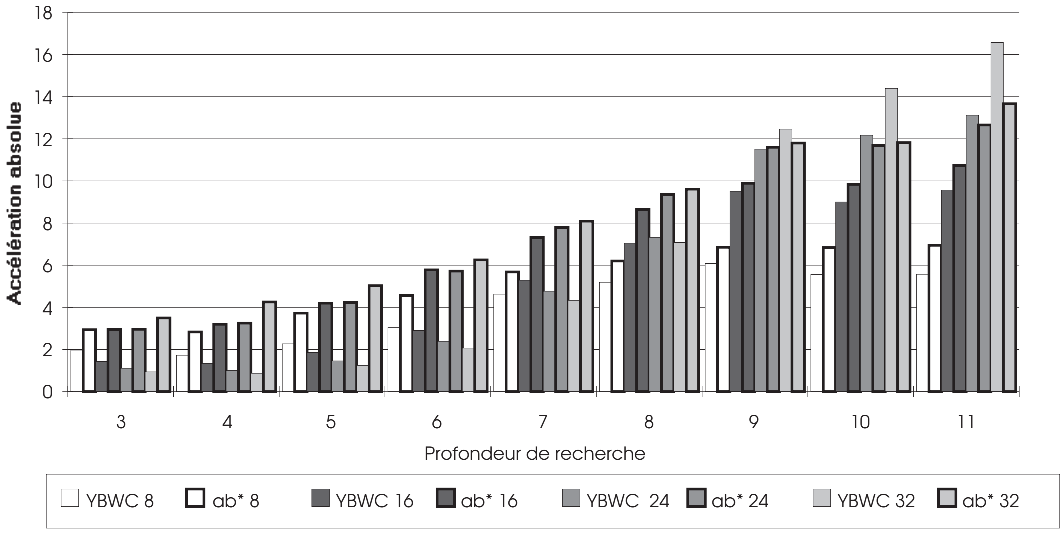
1.2.2 Parallélisation de la recherche

Un des facteurs pouvant augmenter la force des programmes de jeux est bien sûr la puissance de la machine sur laquelle le programme tourne. La définition d'algorithmes de recherche parallèles est donc une étape importante pour pouvoir tirer parti des nouvelles machines parallèles qui permettent de disposer de puissance beaucoup plus importante.

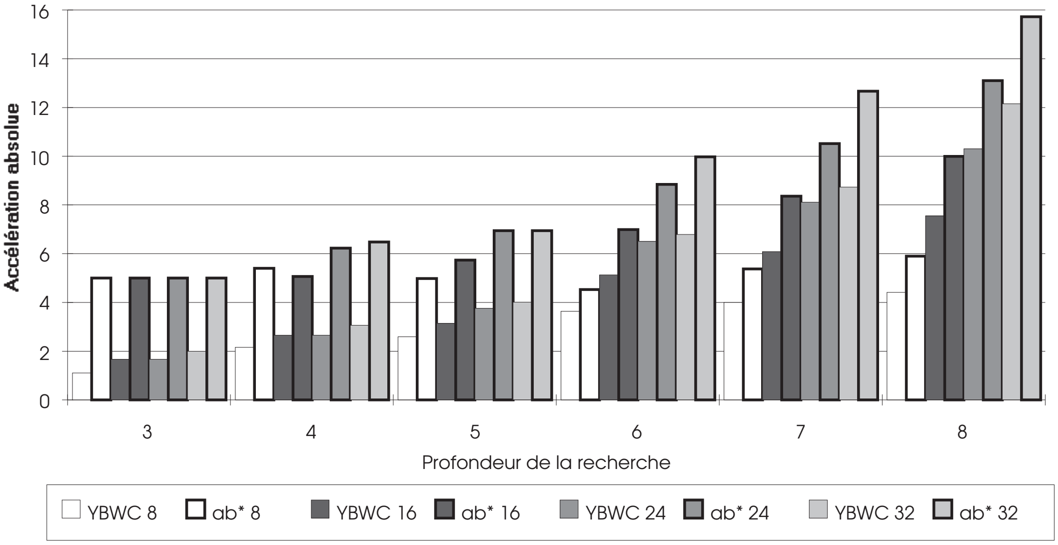
Jusqu'à peu, les algorithmes parallèles proposés avaient des accélérations relatives de l'ordre de la racine carrée du nombre de processeurs. Mais les travaux de Feldmann[51], Otto, Felten[126] et Kuszmaul[92] ont montré que cette barrière pouvait être levée. Principalement en définissant un concept de contrôle du parallélisme spéculatif que nous nommerons le *droit d'aînesse*. Ce concept stipule que deux processeurs qui analysent une même position doivent simultanément analyser le premier mouvement généré s'il n'a pas encore été totalement évalué.

Les algorithmes de Feldmann[51] et Kuszmaul[92] allient ce concept à une décomposition dynamique de l'arbre de recherche nommée le vol de tâche . Ce vol de tâche permet de partager dynamiquement le travail sur tous les processeurs. Un processeur inactif questionne les autres processeurs pour savoir s'ils ont un travail à lui sous traiter. Mais ces deux auteurs ont montré de manière indépendante qu'il fallait limiter les processeurs inactifs dans leur harcèlement des processeurs actifs pour permettre à ceux-ci de faire une recherche efficace.

Nous avons donc défini un nouvel algorithme basé sur le *droit d'aînesse* , qui s'acquitte de ce questionnement. Il se base sur des processus autonomes qui, pour ne pas tous rechercher les mêmes nœuds au même moment, utilisent une table globale connaissant, pour un nœud de l'arbre de jeu, le nombre de processeurs en cours d'analyse de ce nœud. Cette table globale peut être obtenue, à l'instar de la méthode de parallélisation définie par David[45], par une modification mineure de la table de transposition utilisée dans les programmes séquentiels et



Comparaison en accélération absolue de la méthode « Young Brothers Wait Concept » (YBWC) et de notre méthode, la décomposition $\alpha\beta$ couplée au contrôle spéculatif du droit d'aînesse (ab*), pour le programme d'Othello sur 8, 16, 24 et 32 processeurs.*



Comparaison en accélération absolue des deux méthodes de parallélisation pour le programme d'échecs.

FIG. 1.1 - *Comparaison de l'algorithme de Feldmann et de notre algorithme parallèle*

Comparaison effectuée sur une Connection Machine de 32 processeurs parallèles. Chaque fois qu'un processeur commence l'analyse d'un nœud, il va incrémenter le nombre de processeurs dans l'entrée de la table de transposition globale correspondant à la position de jeu. Nous avons exprimé le droit d'aînesse dans ce modèle ainsi : tout parallélisme est interdit au niveau d'un nœud N si le fils de gauche n'est pas totalement évalué ; de plus si le premier fils de gauche est totalement évalué, nous recherchons un fils libre, c'est à dire qui ne soit actuellement exploré par aucun processeur, et s'il n'existe pas de fils libre, nous explorons alors le premier fils non encore totalement évalué (qui existe car sinon la recherche au nœud N serait terminée).

Nous avons comparé ce nouvel algorithme de recherche parallèle avec le meilleur algorithme parallèle connu jusqu'alors, c'est à dire le « Young Brothers Wait Concept » (YBWC) de Feldmann qui conjugue le droit d'aînesse avec le « vol de tâche ».

La figure 1.1 nous montre la comparaison de ces deux méthodes pour deux programmes de jeu de compétitions : un programme d'Othello et un programme d'échecs.

Nous voyons que les deux approches ont des comportements différents sur les deux programmes. Que pour notre programme d'Othello, notre algorithme se laisse supplanter par l'algorithme de Feldmann pour les arbres profonds, mais qu'au contraire sur le programme d'échecs, notre algorithme est plus efficace. La taille maximale des arbres traités étant celle des arbres explorés dans les contraintes de temps de championnat.

Notre approche est de plus intéressante car elle ne requiert qu'un nombre minime de modifications à effectuer sur un programme séquentiel, au contraire des approches de Feldmann et Kuszmaul. Ainsi, les différentes heuristiques utilisées dans les programmes séquentiels peuvent être utilisées dans les programmes parallèles.

1.3 Comportement de notre programmes CUMULUS au championnat du monde de Novembre 1992

Les championnats du monde d'échecs sur ordinateurs n'ont lieu que tous les trois ans, et ne durent que cinq rondes. Evénements longtemps attendus au déroulement brutal, ils permettent à l'élite mondiale d'en découdre officiellement.

En 1992, seuls manquaient DEEP THOUGHT, champion du monde 1989, alors démonté selon ses concepteurs, et CRAY BLITZ, champion du monde 1983 et 1986, qui n'avait rien de neuf à offrir depuis plusieurs années. Il y avait 22 participants, dont nos programmes ECHEC et ECUME (sous le nom de CUMULUS car lors de l'inscription, deux mois auparavant, nous n'étions pas sûr que notre nouveau programme serait prêt).

23 Novembre 1992 : dans le but de ne pas nous laisser influencer une partie qui pourrait avoir lieu plus tard dans le tournoi, les organisateurs ont apparié nos deux programmes ensemble à la première ronde. Nous avons choisi de laisser les deux programmes avec une bibliothèque d'ouvertures plus variée que la bibliothèque prévue pour le tournoi, afin de ne rien laisser voir de notre préparation. CUMULUS 2 s'imposa logiquement.

24 Novembre 1992 : CUMULUS 2 rencontre ZUGZWANG, programme sur une machine de 1024 transputers, un des favoris de ce tournoi.

Zugzwang - Cumulus 2.0, Madrid, 24 Novembre 1992

1 e4 e5 2 ♘f3 ♘c6 3 ♗b5 a6 4 ♗a4 ♘f6 5 O–O ♗e7 6 ♖e1 b5 7 ♗b3 O–O 8 c3 d6 9 h3 ♘a5 10 ♗c2 c5 11 d4 ♕c7 12 ♘d2 c×d4 13 c×d4 ♗b7 14 ♘f1 ♖fc8 !? 15 ♗d3 e×d4 16 ♘×d4 ♘c4 17 ♘f5 g6 18 ♘×e7+ ♕×e7 19 ♗×c4 ♖×c4 20 f3 ♕e5 21 ♗e3 ♕×b2 22 ♕×d6 ♖e8 23 ♖ac1 ♖a4 24 ♖cd1 ♕e5 25 ♕b6 ♗c8 26 ♖d2 ♗e6 27 ♖c1 ♗c4 28 ♗d4 ♘d7 29 ♕b7 ♕e6 30 ♘e3 ♘f6 31 ♖c2 ♘h5 32 g4 ♘f4 33 ♘×c4 ♖×c4 34 ♖×c4 b×c4 35 ♗e3 ♘d3 36 ♖e2 c3 37 ♕b1 ♕d6 38 ♔f1 ♔g7? Considérant l'issue de la partie comme normale compte tenu du rapport de forces (1024 tranputers contre un PC), la plupart des commentateurs n'ont pas relevé cette faute, et tout leur jugement de la position et de la partie s'en est trouvé erroné. Il n'y a pourtant rien d'étonnant à ce que les noirs aient un coup gagnant dans cette position : [38 … ♖d8!! 39 f4 ♘×f4 40 ♖e1 ♘×h3 41 ♕c2 ♕f6+ 42 ♔g2 ♘f4+ 43 ♔g3 ♘d3−+] Au lieu de cela, le coup de la partie transforme une position gagnante en position difficile à défendre.

39 ♕b6 ♕×b6 40 ♗×b6 ♖b8 41 ♗d4+ f6 42 ♖c2 ♖b1+ 43 ♔e2 ♘f4+ 44 ♔e3 g5 45 ♗×c3 ♘×h3 46 ♗d4 ♖e1+ 47 ♔d2 ♖f1 48 ♖c7+ ♔g8 49 ♖c3 ♔f7 50 a3 ♘g1 51 ♖c7+ ♔g6 52 ♗×g1 ♖×g1 53 ♖a7 ♖b1 54 ♖×a6 ♔f7 Les finales avec ce type de structure de pions (pions des deux côtés à l'aile roi et pion blanc a passé) sont bien connues, mais difficiles à programmer. ECUME n'avait pas les connaissances nécessaires pour la défendre correctement. **55 a4 ♖b3 56 ♔e2 ♖a3 57 ♔f2 ♖a2 + 58 ♔g3 ♔e7 59 a5 h6 60 ♖a8 ♔d7** Le roi noir devait rester en g7 et h7 pour protéger les pions de l'aile roi, afin d'empêcher, une fois la position avec a7 Ta8 atteinte, Th8 Txa7 h7+- gagnant la tour noire. Sur une défense convenable, la partie était vraisemblablement nulle. **61 a6 ♔c7 62 a7 ♖a6 63 ♖h8 ♔b7 64 a8=♗+ ♖×a8 65 ♖h7+ ♔c8 66 ♖×h6 ♖a6** Le roi noir est trop éloigné de l'action pour que la partie soit défendable. **67 e5 ♔d7 68 e×f6 ♖a5 69 f4 ♖a3 + 70 ♔f2 g×f4 71 ♖h5 ♔e8 72 ♖f5 ♔f7 73 ♖×f4 ♔f8 74 g5 ♖a5 75 g6 ♖g5 76 g7+ ♔f7 77 ♔f3 ♖g6 78 ♔e4 ♖g1 79 ♔e5 ♖e1 + 80 ♖e4 ♖g1 81 ♖c4 ♖e1 + 82 ♔f5 ♖f1 + 83 ♖f4 ♖c1 84 ♖d4 ♖f1 + 85 ♔e5 ♖e1 + 86 ♔f4 ♖f1 + 87 ♔g5 ♖g1 + 88 ♖g4 ♖f1 89 ♖g3 ♖f2 90 ♖g1 ♖b2 91 ♖c1 ♖g2 + 92 ♔f5 ♖f2 + 93 ♔e5 ♖e2 + 94 ♔d4 ♖d2 + 95 ♔e3 ♖d8 96 ♖h1 ♖e8 + 97 ♔f4 ♖e6 98 ♖h6 ♖e1 99 ♖h8 ♖f1 + 100 ♔g5 ♖g1 1-0**

25 Novembre 1992 : ayant perdu un point, CUMULUS doit affronter le programme PROCHESS. CUMULUS prend l'avantage dès l'ouverture et ne parvient pas à concrétiser tout de suite, faute de trouver un plan convenable. Le gain survient après une magnifique combinaison en 12 demi-coups.

26 Novembre 1992 : CUMULUS doit jouer contre KALLISTO, fort programme Hollandais, qui deviendra en 1993 champion du monde des micro-ordinateurs en Blitz juste devant CUMULUS 2. CUMULUS 2 démontre dans cette partie ses capacités à tirer parti des fautes positionnelles de ses adversaire.

Cumulus 2 - Kallisto, Madrid, 26 Novembre 1992

1 d4 ♘f6 2 ♘f3 e6 3 e3 b6 4 ♘d2 ♗b7 5 ♗d3 ♘c6 6 c3 a5 7 O–O ♗a6?! 8 ♗×a6

♖×a6 9 e4 ♗e7 10 e5 ♘h5 11 ♘e4 a4 12 ♘g5 g6 [12 … ♗×g5 13 ♗×g5 f6 14 e×f6 ♘×f6 15 ♗×f6 g×f6 16 ♕h5+ ♔e7 17 d5] **13 ♕f3** Un bogue dans le programme ECUME lui faisait considérer le o-o noir comme un mauvais coup, aussi cherche-t-il à forcer les noirs à roquer. La position blanche n'en reste pas moins écrasante. [13 d5 e×d5 14 ♕×d5 O–O 15 f4] **13 … O–O 14 ♗e3 ♖a5 15 g4 ♘g7 16 ♕h3 h5 17 b4 a×b3 18 a×b3 ♕a8 19 ♖×a5 ♕×a5 20 ♘f6+ ♗×f6** [20 … ♔h8 21 g×h5 ♘×e5 (21 … g×h5 22 ♘×h5 ♗×g5 23 ♘f6+ ♘h5 24 ♕×h5+ ♔g7 25 ♕h7#) 22 h×g6+ ♘h5 23 ♕×h5+ ♔g7 24 d×e5 ♕×e5 25 ♘e8+ ♖×e8 26 ♕h7+ ♔f6 27 ♕×f7#] **21 e×f6 ♘e8 22 g×h5 g×h5 23 ♕g3 ♔h8 24 ♕f3 ♔g8 25 ♔h1 ♕d5 26 ♘e4 ♕f5 27 ♕×f5** [27 ♖g1+ ♔h8 28 ♕×f5 e×f5 29 ♖g5 f×e4 30 ♖×h5+ ♔g8 31 ♗h6 ♘a5 32 ♖g5+ ♔h7 33 ♗×f8 ♘×f6] **27 … e×f5 28 ♗h6 f×e4 29 ♖g1+ ♘g7 30 ♖×g7+** [30 ♖g5 ♔h7 31 f×g7 ♖g8 32 ♖×h5 ♘e7 33 ♔g2 ♔g6 34 ♖h4 ♘f5] **31 h4 d6 32 b4 b5** [32 … ♘d8 33 ♖g5 ♖e8 34 ♖×h5 ♔h7 35 ♗e3+ ♔g6 36 ♖h6+ ♔f5 37 d5] **33 ♔h2 ♘a7** [33 … ♘d8 △♘e6] **34 ♖g5 ♖e8** [34 … ♖g8 35 ♖×h5 ♔h7 36 ♗g7+ ♔g6 37 ♖g5+ ♔h7 38 ♔g3 d5 39 ♔g4 ♖×g7] **35 ♗g7+** [35 ♗g7+ ♔h7 36 ♖×h5+ ♔g6 37 ♖g5+ ♔h7 38 ♔g3 ♖g8 39 ♔g4 ♖×g7 (39 … ♘c6 40 ♔f5 ♘d8 41 ♖h5#) 40 ♖×g7+ ♔h6 41 ♖×f7] **1-0**

27 Novembre 1992 : Quand commence la 5e et dernière ronde, CUMULUS 2 a 3 points et occupe la quatrième place, derrière les programmes CHESSMACHINE[8], ZUGZWANG et KASPAROV SPARC, tous 3 à 3,5 points. Nous sommes appariés avec le programme HIARCS qui a 2,5 points. Mark Uniacke, qui a besoin d'un gain pour nous passer devant au classement et gagner le titre de meilleur amateur, paramètre le score de la partie nulle à -0,50 pion, ce qui aura pour effet que HIARCS refusera la nulle s'il estime avoir un score supérieur à -0,50 pion[9]. La partie fut longue et, dans la finale, HIARCS refusa plusieurs fois les répétitions proposées par ECUME, puis détruisit sa position pour échapper à la nulle. Au bout du 100e coup noir, l'arbitre Mike Valvo arbitra la partie et déclara le programme CUMULUS vainqueur.

Ainsi, avec 4 points sur 5, CUMULUS 2 (alias ECUME) devenait vice champion du monde des programmes d'échecs, devant des concurrents aussi prestigieux que HITECH, CHESS-GENIUS, THE KING, FRITZ, MCHESS et le KASPAROV SPARC. Le programme CHESSMACHINE remportait le championnat sans avoir rencontré les deux seconds ex-aequo : ZUGZWANG qui n'avait perdu aucunes parties et CUMULUS 2.

1.4 Jeux et Intelligence Artificielle

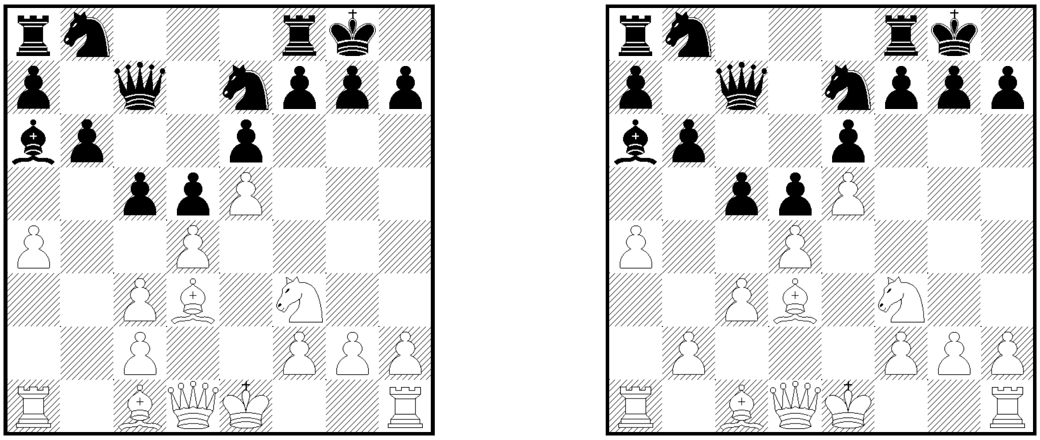
Lorsqu'Alan Turing proposa son test pour déterminer l' intelligence d'une machine, il inclut dans un exemple un problème d'échecs[176]. Ceci est révélateur d'un des défis que se posèrent les chercheurs d'alors, dans les termes de l'époque : construire une machine pour jouer au jeu d'échecs afin de démontrer qu'une machine pouvait penser. Dès 1948, Nor-

8. alias REBEL ou GIDEON.

9. Ceci se passait 2 ans jour pour jour après le championnat du monde des micro-ordinateurs 1990 où, le 27 novembre, nous avions mis le score de la partie nulle à 4 pions contre le programme MEPHISTO. Nous avions alors obtenus une position massacrante que notre programme ECHEC 1.9 transforma en partie nulle, laissant échapper le titre de champion du monde des micro-ordinateurs 1990.

bert Wiener[190] posait la question de savoir si la force au jeu d'échecs représente une différence essentielle entre les potentialités du cerveau humain et des machines. Claude Shannon suit en considérant le jeu d'échecs comme un moyen pour déterminer s'il est possible de construire une machine pensante[160]. Ainsi, la programmation des jeux de réflexion, et plus particulièrement les échecs, devint une part importante de la recherche en Intelligence Artificielle. Le jeu d'échecs était alors considéré comme la drosophile de l'Intelligence Artificielle[107] : un sujet simple à décrire, très riche, où les performances des programmes sont faciles à tester, et dont les techniques de résolution étaient censées être réutilisées dans d'autres domaines.

Quelques approches en complément des programmes de compétition sont à noter dans le sens de la construction de programmes plus intelligents. Ainsi le programme ROBIN de Jacques Pitrat, qui montrait d'excellentes capacités face aux problèmes de recherche de combinaisons et de mats aux échecs par l'application d'un langage de plans tactiques[132], ou D.Wilkins, toujours dans les problèmes de combinaisons en application de la recherche B*[19].



♗×h7 est un très bon coup

♗×h7 ne fonctionne pas

Diagramme 1.1: *Illustration de la nécessité de la recherche.[109]*

La nécessité d'une recherche est illustrée par les positions du diagramme 1.1. Dans les deux positions, nous nous trouvons dans le schéma d'attaque de mat classique 1. ♗×h7+ ♔×h7 2. ♘g5+ ♔g8 3. ♕h5, cette attaque fonctionnant dans le diagramme de gauche et ne fonctionnant pas dans le diagramme de droite (3. ... ♗d3!). Donc un programme ayant la connaissance de ce schéma d'attaque devra quand même effectuer une recherche pour vérifier la validité de l'application du schéma.

De là vient tout le problème de la programmation des échecs par des méthodes plus intelligentes : un minimum de recherche est nécessaire.

Citons aussi l'approche prônée par Mikael Botvinnik: réaliser un programme simulant le raisonnement d'un Grand Maître. Malheureusement, si l'idée est stimulante, la réalisation est bien plus complexe, quoi qu'en écrive Botvinnik[30, 31, 32, 33] qui, depuis une vingtaine d'année, nous promet d'achever son programme PIONNEER, rebaptisé depuis CC SAPIENS, dans les prochains mois. De plus, comme l'a montré Hans Berliner[22], les analyses publiées, supposées être réalisées par le programme de Botvinnik dans des positions d'échecs, possèdent des caractéristiques troublantes nous poussant à croire qu'il s'agit tout au mieux d'un programme optimisé à la main pour bien faire sur ces positions. De toutes les façons, nos doutes et notre scepticisme disparaîtrons le jour où nous pourrons voir enfin le programme de l'ex-champion du monde jouer une partie entière, même si le niveau de jeu n'est pas celui d'un Grand Maître.

1.5 Plan de la thèse

Dans le chapitre 2, nous présentons des algorithmes séquentiels de recherche dans un arbre pour un jeu de réflexion. Nous commençons par une description du principe minimax, et ensuite nous donnons les principaux résultats sur l'algorithme $\alpha\beta$. Puis, nous voyons les différents algorithmes dérivés de l'algorithme $\alpha\beta$ dont le NegaC* que nous avons introduit, ainsi que les algorithmes de recherche de la valeur negamax basés sur un processus de recherche du type meilleur d'abord. Après avoir comparé ces algorithmes, nous montrons l'état de l'art dans les alternatives au principe minimax.

Dans le troisième chapitre, nous examinons les différentes heuristiques possibles et utilisables pour accélérer et améliorer une recherche basée sur le principe minimax. Nous classons les différentes heuristiques selon qu'elles sont ou non dépendantes du domaine c'est à dire du jeu de réflexion auquel elles s'appliquent. Les heuristiques présentées sont celles relatives à l'ordre des mouvements, celles concernant la réduction de l'effet horizon ainsi que les coupures sélectives. Ensuite, nous abordons le difficile problème du choix d'une extension et les inconsistances engendrées par ces heuristiques. Nous explicitons notamment les heuristiques que nous utilisons dans nos programmes d'échecs.

Le chapitre 4 traite de la parallélisation du principe minimax. Nous commençons par définir un certain nombre de quantités que nous utilisons par la suite pour mesurer la performance des algorithmes parallèles par rapport aux algorithmes séquentiels. Nous décrivons ensuite les différents travaux existant sur ce sujet, ainsi que, quand ils existaient, les résultats expérimentaux obtenus par les auteurs ayant utilisé ces méthodes.

Nous continuons par la présentation de nos travaux sur ce sujet. Dans un premier temps, nous développons les principales caractéristiques de la machine cible de notre étude, la CONNECTION MACHINE 5 ainsi que le modèle de programmation parallèle utilisé : les échanges de messages. Ensuite, nous décrivons les caractéristiques de l'implantation des méthodes que nous avons testées ainsi que leurs résultats expérimentaux face à un nouveau schéma de parallélisation du principe minimax que nous introduisons. Nous analysons aussi le comportement d'heuristiques définies dans le chapitre 3 sur notre programme d'échecs parallèle.

Le cinquième chapitre est réservé aux constructions de fonctions d'évaluation. Nous y montrons les problèmes auxquels nous sommes confrontés lors de la construction d'une fonction d'évaluation et les solutions retenues dans nos programmes d'échecs. Ensuite, nous examinons l'état des techniques dans la construction automatique par apprentissage d'une fonction d'évaluation.

Enfin, dans le chapitre 6, après avoir donné une description complète de nos programmes d'échecs de compétition, nous analysons les parties-clés qui leur ont permis de remporter les titres de « Vice-Champion du Monde des programmes d'échecs 1992 », « Champion du Monde des programmes d'échecs sur ordinateur personnel 1990 », « Champion du monde des programmes d'échecs amateurs 1990 ».

Chapitre 2

Les algorithmes de Recherche

Dans ce chapitre, après une révision détaillée des méthodes de recherche et de leurs propriétés, nous exposons nos contributions : l'algorithme NegaC*[1], une formalisation de l'algorithme « recherche du nombre de preuves » dans le modèle Negamax[2] plus une étude comparative des algorithmes minimax[3]. Considérons un jeu de réflexion à information complète à deux joueurs et à somme nulle. Nous entendons par information complète que les deux joueurs ont toutes les informations disponibles et qu'il n'existe pas d'informations cachées, et par somme nulle qu'il s'agit d'un conflit entre deux personnes où ce qui est gagné par l'un des joueurs est perdu par l'autre. De tels jeux sont typiquement : Puissance 4, Othello, Les Dames internationales ou anglaises, Awari, les échecs.

Nous allons, dans ce chapitre, décrire des algorithmes de recherche utilisés pour jouer à un jeu ayant les caractéristiques énoncées ci-dessus. Ces algorithmes sont basés sur le principe du minimax. Après avoir formalisé ce principe, nous analyserons les élagages $\alpha\beta$ et les améliorations possibles de l'$\alpha\beta$. Nous étudierons et comparerons les différents algorithmes basés sur le principe du minimax dont le NegaC* que nous avons défini. Et nous terminerons par la présentation d'alternatives au principe du minimax.

Dans la suite, nous nommerons MAX le joueur qui a le trait représenté par l'ordinateur, et MIN son adversaire.

La position de départ est nommée P_0. Les mouvements possibles depuis la position P_0 sont notés $C_{0.i}$, les positions résultantes $P_{0.i}$. Nous pouvons ainsi représenter par un arbre les possibilités de jeu à partir d'une position donnée. La position P_0 est représentée par la racine de l'arbre, les différents mouvements sont figurés par les branches de l'arbre, les feuilles représentent les positions terminales.

Nous allons décrire un principe de résolution pour un tel jeu.

1. Au § 2.3.3, nous exposons comment nous avons établis notre NegaC* à partir de C*[42]
2. Ainsi qu'une comparaison de notre programme ECUME face à la « recherche du nombre de preuves » pour les problèmes de mats au § 2.4.2
3. En y introduisant notre algorithme NegaC*au § 2.5.

2.1 Le principe Minimax

Nous allons expliquer ici le principe minimax : commençons par choisir une fonction heuristique f définie sur l'ensemble des positions légales $\mathcal{E}$ et à valeurs dans un ensemble ordonné $\mathcal{O}$. Cette fonction heuristique appelée ici *Fonction d'évaluation* est définie de manière à induire un ordre sur l'ensemble des position légales : $f(P)$ avec $P \in \mathcal{E}$, croît avec les possibilités de gain de MAX depuis la position P. De plus cette fonction est supposée *coïncidente*, c'est à dire, si P est une position terminale alors

$$f(P) = \begin{cases} -1_{\mathcal{O}} & \text{si } MAX \text{ a perdu} \\ 0_{\mathcal{O}} & \text{si la position } P \text{ est nulle} \\ +1_{\mathcal{O}} & \text{si } MAX \text{ a gagné} \end{cases}$$

La fonction *minimax* est alors définie par :

$$minimax(P) = \begin{cases} f(P) \text{ si } P \text{ est terminale} \\ \max_i(minimax(P_i)) \text{ si } MAX \text{ a le trait depuis } P \\ \min_i(minimax(P_i)) \text{ si } MIN \text{ a le trait depuis } P \end{cases}$$

La fonction d'évaluation étant uniquement appliquée aux positions terminales, Zermelo[192], Borel[133] et von Neuman[183] ont montré que la fonction *minimax* calcule la valeur théorique de la position P.

En pratique, la valeur théorique de la position P ne pourra généralement pas être calculée : cela signifierait qu'on peut déterminer la valeur théorique de la position ; en conséquence, la fonction d'évaluation sera appliquée dans des positions non terminales. On considérera que plus la fonction d'évaluation est appliquée loin de la racine, meilleur est le résultat du calcul[4]. C'est à dire qu'en examinant plus de coups successifs, nous supposons obtenir une meilleure approximation de la valeur théorique donc un meilleur choix de mouvement.

La figure 2.1 est un exemple de programme minimax où la fonction d'évaluation est appliquée dans des positions éloignées de N demi-coups[5] de la racine.

Supposons que l'ensemble $\mathcal{O}$ des valeurs prises par la fonction d'évaluation soit symétrique par rapport à 0, alors nous pouvons définir une fonction g telle que :

$$g(P) = \begin{cases} +f(P) & \text{Si } MAX \text{ a le trait en } P \\ -f(P) & \text{Si } MIN \text{ a le trait en } P \end{cases}$$

Cette nouvelle fonction va nous permettre de définir une fonction qui est appelée *negamax* de la façon suivante :

$$negamax(P) = \begin{cases} g(P) \text{ si } P \text{ est terminale} \\ \max_i(-negamax(P_i)) \text{ sinon} \end{cases}$$

4. Cette hypothèse est discutée dans le § 2.6.

5. Dans le vocabulaire usuel des jeux de stratégie un «coup» représente une succession de deux mouvements des deux joueurs. Le terme «demi-coup» représente donc un mouvement de la part d'un des deux joueurs.

$\underline{\textbf{funct}}\ minimax(Position, niveau)\ \equiv$
$\quad\ulcorner C \leftarrow ProchainCoup(Position);$
$\quad\ \underline{\textbf{if}}\ C = \textbf{null} \vee niveau = N\ \underline{\textbf{then}}\ \underline{\textbf{exit}}\ f(Position);\ \underline{\textbf{fi}}$
$\quad\ \underline{\textbf{if}}\ (niveau \text{ est pair})$
$\quad\quad \underline{\textbf{then}}\ Meilleur \leftarrow \infty;$
$\quad\quad\ \underline{\textbf{else}}\ Meilleur \leftarrow -\infty;$
$\quad\ \underline{\textbf{fi}}$

$\quad\ \underline{\textbf{while}}\ C \neq \textbf{null}\ \underline{\textbf{do}}$
$\quad\quad\quad valeur \leftarrow minimax(Position \bullet C, niveau + 1);$
$\quad\quad\quad \underline{\textbf{if}}\ (niveau \text{ est impair} \wedge valeur > Meilleur)$
$\quad\quad\quad\quad \vee\ (niveau \text{ est pair} \wedge valeur < Meilleur)$
$\quad\quad\quad\quad \underline{\textbf{then}}\ Meilleur \leftarrow valeur;$
$\quad\quad\quad \underline{\textbf{fi}}$
$\quad\quad\quad C \leftarrow ProchainCoup(Position)\ \underline{\textbf{od}};$
$\quad\ valeur\lrcorner.$

FIG. 2.1 - *L'algorithme* minimax

La fonction d'évaluation f est appliquée dans les positions éloignées de N demi-coups de la position de départ. L'appel initial est *minimax*(*Position*, 1). $-\infty$ et ∞ sont des valeurs telles que $\forall P, -\infty \leq f(P) \leq \infty$.

Il est facile de voir que $minimax(P_0) = negamax(P_0)$[6]. Cette fonction *Negamax* a un grand intérêt pratique car son écriture permet de rendre symétriques les rôles de *MIN* et de *MAX*. La figure 2.2 nous montre un exemple d'implantation du *negamax*.

$\underline{\textbf{funct}}\ negamax(Position, niveau)\ \equiv$
$\quad\ulcorner C \leftarrow ProchainCoup(Position);$
$\quad\quad \underline{\textbf{if}}\ C = \textbf{null} \vee niveau = N\ \underline{\textbf{then}}\ \underline{\textbf{exit}}\ g(Position);\ \underline{\textbf{fi}}$

$\quad\quad Meilleur \leftarrow -\infty;$

$\quad\quad \underline{\textbf{while}}\ C \neq \textbf{null}\ \underline{\textbf{do}}$
$\quad\quad\quad valeur \leftarrow -negamax(Position \bullet C, niveau + 1);$
$\quad\quad\quad \underline{\textbf{if}}\ (valeur > Meilleur)$
$\quad\quad\quad\quad \underline{\textbf{then}}\ Meilleur \leftarrow valeur;$
$\quad\quad\quad \underline{\textbf{fi}}$
$\quad\quad\quad C \leftarrow ProchainCoup(Position)\ \underline{\textbf{od}};$
$\quad\quad valeur\lrcorner.$

FIG. 2.2 - *L'algorithme* negamax *: la définition de la fonction d'évaluation g nous permet d'avoir une écriture symétrique de l'algorithme.*

Dans la figure 2.3, nous avons appliqué les recherches *minimax* et *negamax* sur une position d'Othello : la position initiale P_0 est représentée par la racine de l'arbre, les différents mouvements possibles sont figurés par les branches de l'arbre, les feuilles représentent les positions où est appliquée la fonction d'évaluation. A partir des valeurs de la fonction d'évaluation appliquée sur les feuilles de l'arbre, nous remontons les valeurs des positions par le schéma *negamax* vers la racine.

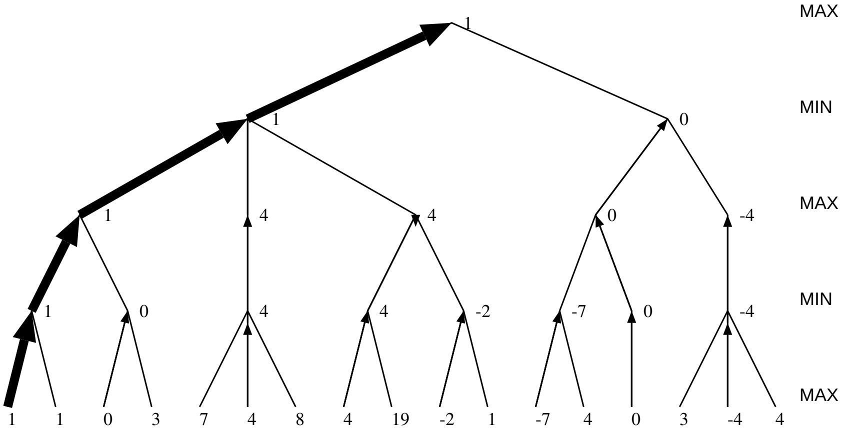
2.2 Les élagages $\alpha\beta$

2.2.1 Présentation de l'$\alpha\beta$

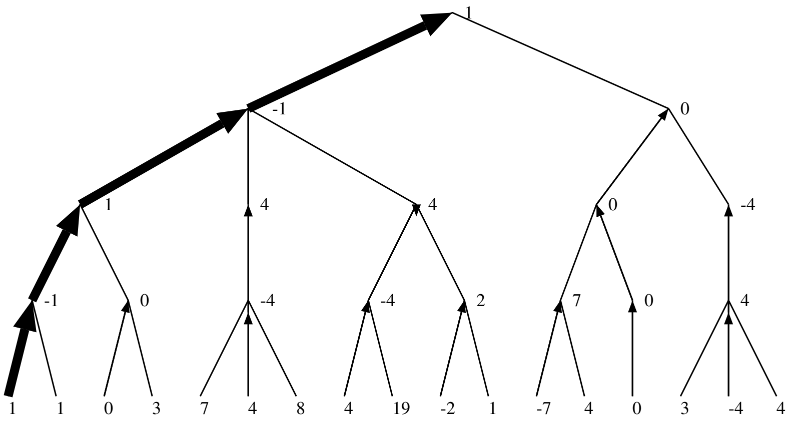
Il n'est pas nécessaire de calculer l'ensemble des valeurs *negamax* pour toutes les positions de l'arbre de recherche. En effet nous pouvons nous abstenir de les calculer pour des sous-arbres entiers de l'arbre de recherche ; les élagages $\alpha\beta$ sont une possibilité pour ce faire :

Soient α et β, deux valeurs appartenant au domaine d'arrivée de la fonction d'évaluation g supposé muni d'un ordre total et symétrique par rapport à 0 et telles que $\alpha < \beta$.

6. La démonstration de ceci se trouve dans [88]



Application du Minimax sur un arbre de jeu d'échecs. Dans un nœud *MAX* (resp. *MIN*), le maximum (resp. le minimun) de la valeur minimax des nœud fils est retournée.



Application du *Negamax* sur le même arbre. Nous retournons le maximun des négations des valeurs negamax des nœuds fils.

FIG. 2.3 - *les formalismes Negamax et Minimax sur un arbre de jeu*

Nous considérons maintenant la fonction :

$$\Phi(P,\alpha,\beta) = \begin{cases} g(P) \text{ si } P \text{ est terminale} \\ \min(\beta, \max(-\Phi(P_i,-\beta,-\alpha))) \text{ sinon} \end{cases}$$

Propriété 2.2.1 *Si nous considérons une position P, deux variables α et β telles que $\alpha < \beta$, $v = \Phi(P,\alpha,\beta)$ et $w = negamax(P)$, nous avons :*

$$\begin{cases} si\ v \leq \alpha\ alors\ w \leq v \\ si\ \alpha < v < \beta\ alors\ w = v \\ si\ \beta \leq v\ alors\ v \leq w \end{cases}$$

Preuve: Elle se fait par récurrence sur la profondeur de l'arbre de recherche :

- La propriété est vraie au rang 1 : dans ce cas $\Phi(P) = negamax(P) = f(P)$.
- Si la propriété est vraie pour tous les arbres de profondeur inférieure à n, soit P un arbre de profondeur $n > 1$, $v = \Phi(P,\alpha,\beta)$, nous distinguons alors 4 cas :

  1° P est terminale alors $\Phi(P) = negamax(P) = f(P)$.

  2° $v \leq \alpha$

$$\begin{aligned} &\Leftrightarrow \forall P_i, -\Phi(P_i,-\beta,-\alpha) \leq v \leq \alpha \\ &\Leftrightarrow \forall P_i, \Phi(P_i,-\beta,-\alpha) \geq -v \geq -\alpha \\ &\Leftrightarrow \forall P_i, negamax(P_i) \geq -v \geq -\alpha \\ &\Leftrightarrow -negamax(P_i) \leq v \leq \alpha \\ &\Leftrightarrow negamax(P) \leq v \leq \alpha \end{aligned}$$

  3° $\alpha < v < \beta$

$$\begin{aligned} &\Leftrightarrow \forall P_i, -\Phi(P_i,-\beta,-\alpha) \leq v < \beta, \exists P_i/\beta > -\Phi(P_i - \beta, -\alpha) = v > \alpha \\ &\Leftrightarrow \forall P_i, \Phi(P_i,-\beta,-\alpha) \geq v > -\beta, \exists P_i/-\beta < \Phi(P_i,-\beta,-\alpha) = -v < -\alpha \\ &\Leftrightarrow \forall P_i, negamax(P_i) \geq v > -\beta, \exists P_i/-\beta < negamax(P_i) = -v < -\alpha \\ &\Leftrightarrow \forall P_i, -negamax(P_i) \leq v < \beta, \exists P_i/\beta > -negamax(P_i) = -v > \alpha \\ &\Leftrightarrow \alpha < negamax(P) = v < \beta \end{aligned}$$

  4° $v \geq \beta$

$$\begin{aligned} &\Leftrightarrow \exists P_i/-\Phi(P_i,-\beta,-\alpha) \geq v \geq \beta \\ &\Leftrightarrow \exists P_i/\Phi(P_i,-\beta,-\alpha) \leq -v \leq -\beta \\ &\Leftrightarrow \exists P_i/negamax(P_i) \leq -v \leq -\beta \\ &\Leftrightarrow \max(-negamax(P_i)) \geq v \geq \beta \\ &\Leftrightarrow negamax(P) \geq v \geq \beta \end{aligned}$$

□

Corollaire 2.2.1.1

$$\Phi(P,-\infty,+\infty) = negamax(P)$$

En d'autres termes, la fonction Φ appelée avec $\alpha = -\infty$, $\beta = +\infty$ calcule la valeur negamax de la position P.

Pour calculer le terme $\min(\beta, \max(-\Phi(P_i, -\beta, -\alpha)))$, on peut s'arrêter dès que l'on a calculé un P_{i_0} tel que $\Phi(P_{i_0}, -\beta, -alpha)$ est supérieur à β, les fils P_i avec $i > i_0$ ne seront alors pas examinés.

La figure 2.4 représente l'algorithme $\alpha\beta$ qui utilise la propriété 2.2.1 pour calculer la valeur Negamax d'une position P. La figure 2.5 nous montre l'effet de la recherche $\alpha\beta$ sur l'arbre de la figure 2.3.

```
funct fab(Position, α, β) ≡
    ⌈M ← ProchainCoup(Position);
     if M = null then exit g(Position); fi
     Meilleur ← −∞;
     while M ≠ null do
         valeur ← −fab(Position • M, −β, −α);
         if valeur > Meilleur
           then Meilleur ← valeur;
                if Meilleur > α
                  then α ← Meilleur;
                       if α ≥ β then exit Meilleur; fi
                fi
         fi
         M ← ProchainCoup(Position) od;
     Meilleur⌋.
```

FIG. 2.4 - *l'algorithme Alpha-Beta*[a]

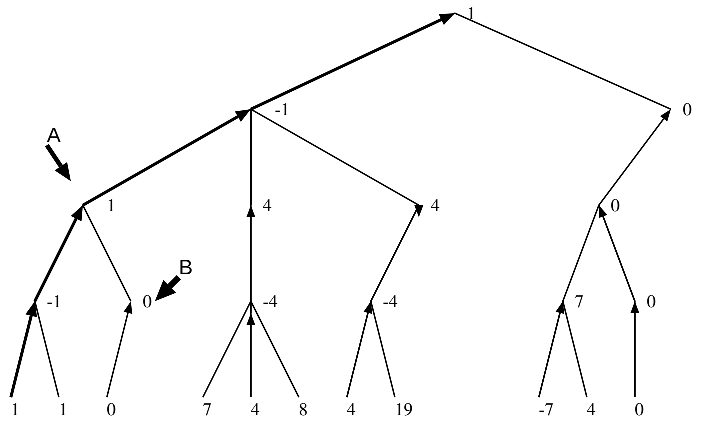
[a] En fait, il s'agit plus précisément de l'algorithme *fail-soft-$\alpha\beta$* définit par Fishburn[58] : au lieu de retourner α quand tous les mouvements ont une valeur inférieure à α, le maximum des valeurs retournées est à son tour retourné, et dans le cas où un mouvement a une valeur supérieure à β, cette valeur est retournée et non pas β. Ceci est compatible avec le principe 2.2.1 et donne plus d'informations sur le domaine de la fonction d'évaluation.

2.2.2 Principaux résultats sur l'$\alpha\beta$

Nous allons donner dans ce paragraphe les définitions et résultats importants concernant les élagages $\alpha\beta$.

Définition 2.2.1 *Nous appelons* **fenêtre** $\alpha\beta$ *le couple* $[\alpha, \beta]$ *où* α *et* β *sont les deux paramètres d'appel de la procédure* $\alpha\beta$.

Propriété 2.2.2 *[111] Soit deux fenêtre* $[\alpha_0, \beta_0]$ *et* $[\alpha_1, \beta_1]$ *telles que* $\alpha_0 \leq \alpha_1 < \beta_1 \leq \beta_0$, *l'algorithme* $\alpha\beta$ *appelé avec la fenêtre* $[\alpha_1, \beta_1]$ *parcourt moins de nœuds qu'avec l'appel avec la fenêtre* $[\alpha_0, \beta_0]$.



Après l'évaluation du premier fils du nœud **B**, on sait que la valeur negamax du nœud **B** sera supérieure à 0, l'algorithme $\alpha\beta$ ayant déjà évalué le premier fils du nœud **A**, sait que le nœud **A** choisira un fils dont la valeur negamax est inférieure à -1. Il est donc inutile d'analyser les autre fils du nœud **B** car le meilleur mouvement au nœud **A** ne peut être celui qui mène au nœud **B**.

FIG. 2.5 - *L'algorithme $\alpha\beta$ appliqué sur l'arbre de la figure 2.3*

Définition 2.2.2 *Nous appelons arbre de jeu* **uniforme de largeur** *l, un arbre de jeu où tous les nœuds non terminaux ont exactement l fils.*

Les nœuds élagués sont ceux qui seraient appelés avec une fenêtre telle que $\alpha \geq \beta$, il est alors clair que si les valeurs $-\infty$ et $+\infty$ ne font pas partie des valeurs retournées par la fonction d'évaluation, les nœuds appelés avec une des bornes à ces deux valeurs ne seront pas élagués ; 3 types de nœuds non élagables sont alors définis :

les nœuds de type 1 : la fenêtre d'appel est $[-\infty, +\infty]$;

les nœuds de type 2 : la fenêtre d'appel est $[-\infty, b]$ où $b \neq +\infty$;

les nœuds de type 3 : la fenêtre d'appel est $[a, +\infty]$ où $a \neq -\infty$.

La racine est un nœud de type 1 ainsi que tous les fils aînés des nœuds de type 1. Les fils cadets des nœuds de type 1 sont de type 2 et tous les fils des nœuds de type 3 sont de type 2. Les fils aînés des nœuds de type 2 sont de type 3.

Ces définitions induisent donc une typologie incomplète sur les nœuds : les cadets des nœuds de type 2 ne sont pas typés, et par conséquence tous leurs successeurs ne sont pas non plus typés.

Propriété 2.2.3 *[88] L'algorithme $\alpha\beta$ utilisé avec la fenêtre $(-\infty, +\infty)$ regarde au moins l'arbre critique, c'est-à-dire l'ensemble des nœuds de types 1, 2 et 3, et uniquement celui-ci dans le cas où l'arbre est parfaitement ordonné.*

Corollaire 2.2.3.1 *Le nombre de feuilles évaluées par l'algorithme $\alpha\beta$ est compris entre $l^{\lfloor n/2 \rfloor} + l^{\lceil n/2 \rceil} - 1$ et l^n pour un arbre uniforme de largeur l et de profondeur n.*

Preuve: Pour la borne inférieure, nous allons calculer le nombre de feuilles d'un arbre critique par récurrence: Soit u_n, le nombre de feuilles de type 2, v_n, le nombre de feuille de type 2 pour un arbre critique uniforme de largeur l et de profondeur n.

$$\begin{aligned}
&u_0 = 0, v_0 = 0 \\
&\forall n > 0, u_n = l v_{n-1} + l - 1, v_n = u_{n-1} \\
&\Rightarrow \forall n > 2, u_n = l u_{n-2} + l - 1 \\
&\Rightarrow \forall n > 2, u_n + 1 = l(u_{n-2} + 1) \\
&\Rightarrow \forall n > 2, u_n = \begin{cases} l^{\lfloor n/2 \rfloor} - 1 \text{ si } n \text{ est pair} \\ l^{\lfloor (n-1)/2 \rfloor} l - 1 \text{ si } n \text{ est impair} \end{cases} \quad \Rightarrow \forall n, u_n = l^{\lceil n/2 \rceil} - 1
\end{aligned}$$

Le nombre de feuilles de type 2 est donc $l^{\lceil n/2 \rceil} - 1$, le nombre de feuilles de type 3 est $l^{\lceil (n-1)/2 \rceil} - 1 = l^{\lfloor n/2 \rfloor} - 1$, et le nombre de feuille au total $l^{\lceil n/2 \rceil} - 1 + l^{\lfloor n/2 \rfloor} - 1 + 1$, qui est le résultat escompté. □

Corollaire 2.2.3.2 *Le nombre de nœuds de l'arbre minimal évalué par l'algorithme $\alpha\beta$ est*

$$S_n(l) = \frac{l+1}{l-1} l^{\lceil n/2 \rceil} + \frac{2l}{l-1}(l^{\lfloor n/2 \rfloor} - 2) - n + 2$$

.

Preuve: Soit U_n (resp. V_n), le nombre de nœuds de type 2 (resp. type 3) dans l'arbre minimal,

$$\begin{aligned}
U_n &= \sum_{i=0}^{n} u_i \Rightarrow \\
U_n &= \sum_{i=0}^{n} l^{\lceil i/2 \rceil} - 1 \Rightarrow
\end{aligned}$$

$$U_{2p} = 2l \frac{l^p - 1}{l - 1} - 2p \quad , \quad U_{2p+1} = U_{2p} + l^p - 1$$

Comme $V_n = U_{n-1}$ et que le nombre de nœuds de type 1 dans un arbre minimal de profondeur n est $n+1$, soit S_n, le nombre total de nœuds de l'arbre minimal :

$$
\begin{array}{l}
S_n = U_n + V_n + n + 1 \Rightarrow \\
\left\{\begin{array}{l} S_{2p} = U_{2p} + U_{2p-1} + 2p + 1 \\ S_{2p+1} = U_{2p+1} + U_{2p} + 2p + 2 \end{array}\right. \Rightarrow \\
\left\{\begin{array}{l} S_{2p} = 2l\frac{l^p}{l-1} - \frac{2l}{l-1} + 2l\frac{l^{p-1}}{l-1} + l^{p-1} - \frac{2l}{l-1} - 2p - 2p + 2 - 1 + 2p + 1 \\ S_{2p+1} = 2l\frac{l^p}{l-1} - \frac{2l}{l-1} + l^p + 2l\frac{l^p}{l-1} - \frac{2l}{l-1} - 2p - 1 - 2p + 2p + 2 \end{array}\right. \Rightarrow \\
\left\{\begin{array}{l} S_{2p} = \frac{2l}{l-1}(l^p - 2) + l^p\frac{2+l-1}{l-1} - 2p + 2 \\ S_{2p+1} = \frac{2l}{l-1}(l^p - 2) + l^{p+1}\frac{2+l-1}{l-1} - 2p - 1 + 2 \end{array}\right. \Rightarrow \\
S_n = \frac{2l}{l-1}(l^{\lfloor n/2 \rfloor} - 2) + l^{\lceil n/2 \rceil}\frac{l+1}{l-1} - n + 2
\end{array}
$$

□

Définition 2.2.3 *On appelle* **coefficient de branchement asymptotique** *la limite quand la profondeur p tend vers l'infini de la racine p*$^{\text{e}}$ *du nombre de feuilles évaluées.*

Propriété 2.2.4 *[12] Soit* $f_0(x) = x$, *et pour* $i = 1, 2, \ldots$,

$$
\begin{array}{rcl}
f_i(x) & = & 1 - \{1 - [f_{i-1}(x)]^l\}^l, \\
r_i(x) & = & \dfrac{1 - [f_{i-1}(x)]^l}{1 - f_{i-1}(x)}, \\
s_i(x) & = & \dfrac{f_i(x)}{[f_{i-1}(x)]^l}, \\
R_i(x) & = & r_1(x) \times \cdots \times r_{\lceil i/2 \rceil}(x), \\
S_i(x) & = & s_1(x) \times \cdots \times r_{\lfloor i/2 \rfloor}(x).
\end{array}
$$

Le nombre moyen $N_{l,n}$ *de feuilles évaluées par l'algorithme* $\alpha\beta$ *appelé avec une fenêtre d'appel complète pour un arbre uniforme de largeur l et de profondeur n avec une fonction d'évaluation continue est :*

$$N_{l,n} = l^{\lfloor n/2 \rfloor} + \int_0^1 R_n'(t).S_n(t)dt$$

Propriété 2.2.5 *[170][130] Le coefficient de branchement asymptotique de l'algorithme* $\alpha\beta$ *est optimal sur la classe des algorithmes de recherche minimax séquentiels. Il est donné dans le cas d'un arbre à évaluation continue par* $\frac{\zeta_l}{1-\zeta_l}$ *où* ζ_l *est la plus petite racine positive de* $x^l + x = 1$.

La propriété 2.2.5 est un résultat très important du point de vue de la recherche sur les algorithmes de type minimax : on ne peut pas faire beaucoup mieux que l'$\alpha\beta$ mais on peut quand même être un polynôme en l plus rapide ! Mais cet avantage est constant par rapport à la profondeur !

Le tableau 2.1 donne les valeurs de ζ_l, et du branchement asymptotique pour différentes valeurs du facteur de branchement de l'arbre de jeu l.

TAB. 2.1 - *Valeurs de ζ_l et $\frac{\zeta_l}{1-\zeta_l}$ en fonction de l*

l	1	2	3	5	15	20	30	40	50	100	500
ζ_l	0,50	0,62	0,68	0,75	0,87	0,89	0,92	0,93	0,94	0,97	0,99
$\frac{\zeta_l}{1-\zeta_l}$	1,00	1,62	2,15	3,08	6,81	8,42	11,42	14,20	16,85	28,92	106,41

Un autre résultat important pour la complexité de l'$\alpha\beta$ concerne l'optimalité des fenêtres minimales pour chercher si la valeur negamax de la racine est supérieure ou égale à une valeur donnée.

Propriété 2.2.6 *[185][130] Le nombre moyen de nœuds terminaux visités par l'appel de l'algorithme $\alpha\beta$ avec une fenêtre minimale $[v, v+1]$ pour un arbre uniforme de largeur l et de profondeur p est donné par :*

$$N_{l,p} = \prod_{i=1}^{p} \frac{P_i}{1 - P_{i-1}}$$

où $P_i = 1 - P_{i-1}^l$ et P_0 est la probabilité que la valeur des feuilles de l'arbre soit supérieure à v

Preuve: Dans un premier temps, nous considérons que la fonction d'évaluation a une image réduite à l'ensemble $\{-1, 1\}$, soit alors P_0, la probabilité que la valeur d'une feuille de l'arbre soit 1 et P_p, la probabilité que la valeur de la racine d'un arbre de profondeur p soit 1.

Si $p > 0$, la valeur de la racine d'un arbre est -1 si et seulement si tous les l mouvements partant de la racine mènent à des positions de valeur negamax 1, donc $1 - P_p = P_{p-1}^l$.

Soit alors, ξ_p, le nombre attendu de nœuds terminaux calculés par l'appel avec la fenêtre $[-1, 1]$, la probabilité d'avoir un élagage après le calcul du k^e successeur est $P_{p-1}^{k-1}(1 - P_d - 1)$. Une telle éventualité demande le calcul de $(k-1)\xi_{p-1} + \xi_{p-1}$ nœuds terminaux.

La probabilité que la valeur de la racine soit perdante est $1 - P_p$ et demande en moyenne $l\xi_{p-1}$ calculs de nœuds terminaux donc :

$$\begin{aligned}
\xi_{p+1} &= \sum_{k=1}^{l} P_p^{k-1}(1 - P_p)k\xi_p + l\xi_p P_p^l \\
&= \xi_p(lP_p^l + (1 - P_p)\sum_{k=1}^{l} kP_p^{k-1} \\
&= \xi_p \frac{1 - P_p^l}{1 - P_p} = \xi_p \frac{P_{p+1}}{P_p} \\
&= \prod_{k=0}^{p} \frac{P_k + 1}{1 - P_k}
\end{aligned}$$

Soit maintenant P_0, la probabilité que les feuilles de l'arbre soient supérieures ou égales à une valeur donnée v. Nous transformons cet arbre en modifiant les valeurs des feuilles de la façon suivante: si la valeur d'une feuille de l'arbre est supérieure ou égale à $-1^p v$, alors la nouvelle valeur de la feuille est 1, sinon la nouvelle valeur est -1. Et la valeur de la racine de l'arbre initial sera supérieure où égale à v si la valeur de la racine du nouvel arbre est 1. Le sous-arbre de l'ancien arbre examiné avec la fenêtre $[v, v+1]$ étant identique au sous-arbre du nouvel arbre examiné par la fenêtre $[-1, 1]$.

□

Propriété 2.2.7 *[170] Le nombre moyen optimal de nœuds terminaux examinés pour déterminer si la valeur de la racine d'un arbre uniforme de largeur l et de profondeur p est supérieure où égale à une valeur donnée v est*

$$N_{l,p} = \prod_{i=1}^{p} \frac{P_i}{1 - P_{i-1}}$$

où $P_i = 1 - P_{i-1}^l$ et P_0 est la probabilité que la valeur des feuilles de l'arbre soit supérieure à v et le facteur de branchement optimal est

$$\begin{cases} l^{1/2} & \text{si} \quad P_0 \neq \zeta_l \\ \frac{\zeta_l}{1-\zeta_l} & \text{sinon} \end{cases}$$

où ζ_l est la plus petite racine positive de $x^l + x - 1 = 0$ [7].

Corollaire 2.2.7.1 *La recherche avec des fenêtres minimales $[v, v+1]$ est séquentiellement optimale pour déterminer si la valeur de la racine d'un arbre uniforme de largeur l et de profondeur p est supérieure ou égale à v.*

Le corollaire 2.2.7.1 au contraire de la propriété 2.2.5 n'exprime pas une optimalité asymptotique mais simplement le fait que les recherches en fenêtre minimale sont optimales en complexité moyenne pour ce sous problème. On ne peut donc pas faire mieux qu'une recherche avec fenêtre minimale pour savoir si la valeur minimax de la racine d'un arbre est supérieure ou égale à une valeur donnée dans le cas général.

Mais les arbres de recherches générés par les programmes de jeux de réflexion se rapprochent des arbres minimaux pour l'algorithme $\alpha\beta$. Dans la prochaine suite, nous allons montrer comment utiliser cette propriété pour avoir des algorithmes encore plus efficaces.

2.3 Les alternatives à l'algorithme $\alpha\beta$

2.3.1 L'algorithme P-$\alpha\beta$

L'algorithme *P-$\alpha\beta$* [57] est basé sur l'utilisation de fenêtres minimales pour savoir si tous les « meilleurs » mouvements se trouvent dans la *variante principale* qui est la branche la plus

7. La discontinuité provient du fait que P_n converge quand n tends vers l'infini si et seulement si P_n est la suite constante $P_n = P_0 = \zeta_l$ sinon la suite oscille vers les valeurs 0 et 1.

à gauche de l'arbre[8].

Si tel est le cas, la recherche avec la fenêtre minimale réduit le temps de recherche, sinon, dans ce cas, les « meilleurs » mouvements ne sont pas tous dans la variante principale et donc, il faut effectuer une deuxième recherche pour connaître la valeur du nouveau « meilleur » mouvement.

<u>funct</u> $pab(Position)\ \equiv$
  $\lceil M \leftarrow ProchainCoup(Position)$;
    <u>if</u> $M =$ **null** **<u>then</u>** **<u>exit</u>** $g(Position)$; **<u>fi</u>**
    $Meilleur \leftarrow -pab(M)$;
    $M \leftarrow ProchainCoup(Position)$;
    <u>while</u> $M \neq$ **null** **<u>do</u>**
        $valeur \leftarrow -fab(Position \bullet M, -Meilleur - 1, -Meilleur))$;
        <u>if</u> $valeur > Meilleur$
          <u>then</u> $Meilleur \leftarrow -fab(Position \bullet M, -\infty, -valeur)$; **<u>fi</u>**
        $M \leftarrow ProchainCoup(Position)$ **<u>od</u>**;
    $Meilleur\rfloor$.

FIG. 2.6 - *l'algorithme P-$\alpha\beta$ [57]*

La figure 2.6 nous montre l'algorithme *P-$\alpha\beta$* qui peut être considéré comme une surcouche de l'algorithme $\alpha\beta$: après avoir déterminé de manière récursive la valeur de la branche la plus à gauche de l'arbre de recherche, l'algorithme *P-$\alpha\beta$* va tester les mouvements alternatifs le long de la branche la plus à gauche. La figure 2.7 nous montre le fonctionnement de l'algorithme *P-$\alpha\beta$* sur l'arbre de la figure 2.3.

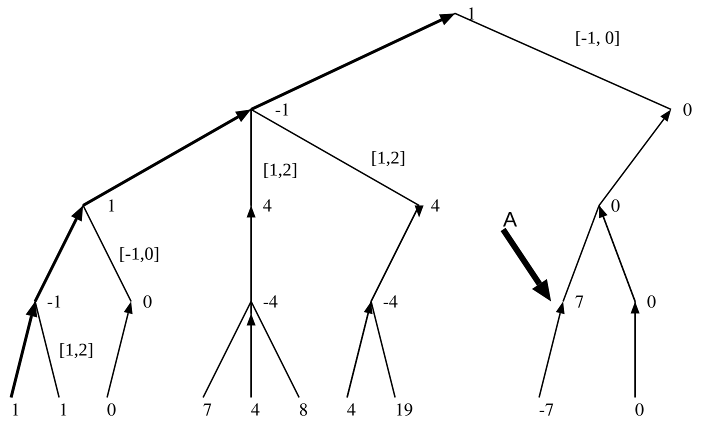
2.3.2 L'algorithme Scout

La figure 2.8 nous présente l'algorithme *Scout*[129]. Les différences avec l'algorithme *P-$\alpha\beta$* sont les suivantes :

1° L'algorithme *Scout* s'appelle de manière itérative en cas de nouveau « meilleur mouvement », les appels avec la procédure test ne sont donc pas réservés aux seuls frères des nœuds de la branche la plus à gauche.

2° Au lieu d'appeler *fail-soft-$\alpha\beta$* pour tester si la valeur minimax d'une branche est supérieure à une valeur donnée, la procédure TEST est appelée, c'est l'équivalent de la procédure $\alpha\beta$ appelée avec une fenêtre minimale avec toutefois le désavantage de ne pas utiliser les bornes que l'on peut obtenir en utilisant le *fail-soft-$\alpha\beta$*.

8. C'est l'ensemble des nœud de type 1.



Après avoir calculé la valeur la plus à gauche de l'arbre, l'algorithme *P-$\alpha\beta$* remonte cette valeur jusqu'à la racine en comparant les valeurs des nœuds de droite à la valeur du nœud de gauche grâce à l'utilisation d'une recherche $\alpha\beta$ avec une fenêtre minimale. Après le retour de l'évaluation du premier frère du nœud **A** (-7) et comme l'appel de ce nœud est fait avec la fenêtre $[-1, 0]$, les autres fils du nœud **A** sont élagués.

FIG. 2.7 - *Exemple d'application de l'algorithme P-$\alpha\beta$.*

Initialement, l'algorithme *Scout* a été inventé pour calculer le coefficient de branchement asymptotique de l'$\alpha\beta$. En effet du fait de sa structure, il est plus facile de faire ce calcul sur l'algorithme Scout qui était supposé par construction plus coûteux que l'algorithme $\alpha\beta$; mais, *Scout* s'est montré plus efficace que l'$\alpha\beta$ dans certains cas : par exemple il parcourt l'arbre de la figure 2.3 de la même façon que l'algorithme *P-$\alpha\beta$*, donc dans le cas de cet arbre, il est plus efficace car ces deux algorithmes regardent un nœud de moins sur cet exemple que l'algorithme $\alpha\beta$.

L'algorithme *Scout* a été reformulé sous une forme récursive avec une seule procédure avec une fonction de TEST utilisant les principes du *fail-soft-$\alpha\beta$* par Marsland[99] en *PVS* (figure 2.9), puis en *NegaScout*[136]. Le *NegaScout* apportant en plus du *PVS*, une modification dans le cas où une recherche est nécessaire près des feuilles.

2.3.3 L'algorithme NegaC*

Coplan[42] a défini l'algorithme *C** pour résoudre des problèmes combinatoires aux échecs. L'algorithme *C** utilise deux procédures analogues à la procédure TEST pour calculer par un procédé de dichotomie la valeur minimax de la racine. Nous avons, à partir de cet algorithme, défini un nouvel algorithme que nous avons appelé *NegaC**[185, 186]. La figure 2.10 montre une implantation possible de l'algorithme *NegaC**.

```
funct Scout(Position) ≡
      ⌈M ← ProchainCoup(Position);
       if M = null then exit g(Position); fi
       Meilleur ← −Scout(Position • M);
       M ← ProchainCoup(Position);
       while M ≠ null do
             if ¬Test(Position • M, −Meilleur − 1)
               then Meilleur ← −Scout(Position • M);
             fi
             M ← ProchainCoup(Position); od
       Meilleur⌋.


funct Test(Position, Valeur) ≡
      ⌈M ← ProchainCoup(Position);
       if M = null then exit g(Position > valeur); fi
       M ← ProchainCoup(Position);
       while M ≠ null do
             if ¬Test(Position • M, −valeur − 1)
               then
                    exit true;
             fi
             M ← ProchainCoup(Position; od
       false⌋.
```

FIG. 2.8 - *L'algorithme Scout*

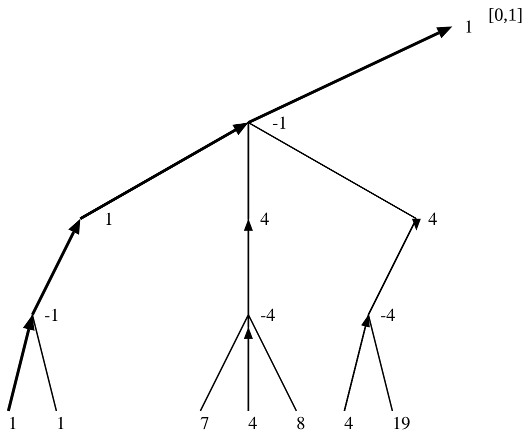
```
funct PVS(Position, α, β) ≡
        ⌈M ← ProchainCoup(Position);
         if M = null then exit g(Position); fi
         Meilleur ← −PVS(position • M, −β, −α);
         M ← ProchainCoup(Position);
         while M ≠ null do
                if Meilleur > α then α ← Meilleur; fi
                if α ≥ β then exit Meilleur; fi
                Valeur ← −PVS(Position • M, −α − 1, −α);
                if Valeur > α ∧ Valeur < β
                   then Valeur ← −PVS(Position • M, −β, −Valeur); fi
                if Valeur > Meilleur then Meilleur ← Valeur; fi
                M ← ProchainCoup(Position); od
         Meilleur⌋.
```

FIG. 2.9 - *L'algorithme PVS: Une reformulation plus efficace de l'algorithme Scout*

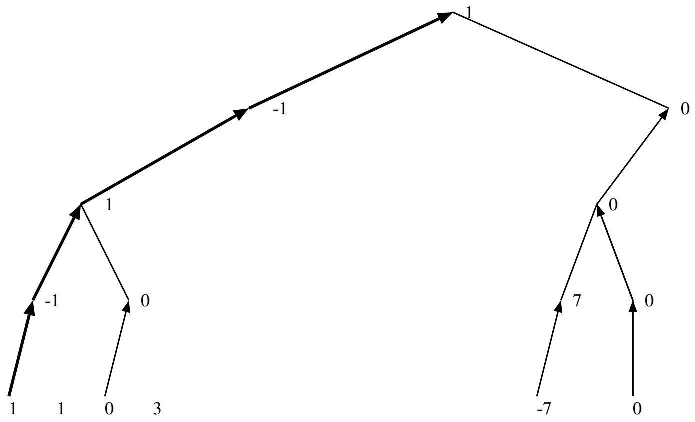
```
funct negac*(Position) ≡
        ⌈Inf ← −∞;
         Sup ← +∞;
         while Inf ≠ Sup do
                v ← [(Inf+Sup)/2];
                t ← fab(Position, v, v + 1);
                if t > v
                   then Inf ← t;
                    else Sup ← t;
                fi od
         t⌋.
```

FIG. 2.10 - *L'algorithme NegaC**

On suppose que la fonction d'évaluation a son image dans $[-10000, 10000]$, le premier appel est donc fait avec la fenêtre $[0, 1]$.



L'appel de la fonction *fail-soft-$\alpha\beta$* ayant retourné $+1$, la valeur negamax de la racine est donc supérieure ou égale à 1; la nouvelle fenêtre est $[+5001, +5002]$.



Le retour est 1, la valeur negamax de la racine est donc inférieure ou égale à 1, c'est donc 1.[a]

FIG. 2.11 - *Application de l'algorithme* NegaC* *sur l'arbre de la figure 2.3*

[a] Si un appel à TEST avait été utilisé plutôt qu'un appel à fail-soft-$\alpha\beta$ du dernier appel on aurait seulement pu déduire que la valeur negamax de la racine était inférieure à 5001.

La figure 2.11 nous montre le comportement de l'algorithme *NegaC** sur l'arbre de la figure 2.3. Nous avons établi les résultats suivants [185, 186] :

Propriété 2.3.1 *[42, 185] L'union des arbres parcourus par l'algorithme* NegaC* *est un sous-arbre de l'arbre parcouru par l'algorithme* $\alpha\beta$.

Preuve: Chaque sous-arbre $\mathcal{A}_i$ parcouru par l'algorithme *NegaC** correspond à un sous-arbre parcouru par l'algorithme $\alpha\beta$ avec une fenêtre minimale, donc par application de la propriété 2.2.2, $\mathcal{A}_i$ est un sous-arbre du sous-arbre $\mathcal{B}$ parcouru par l'algorithme $\alpha\beta$ avec une fenêtre $[-\infty, \infty]$. Donc $\cup_i \mathcal{A}_i$ est un sous arbre de $\mathcal{B}$. □

Propriété 2.3.2 *[185, 186] La complexité moyenne $I_{l,p}$ de l'algorithme* NegaC* *pour un arbre uniforme de largeur l et de profondeur p et une évaluation aléatoire dont l'ensemble image est de cardinal C est bornée par*

$$\lceil \log_2(C) \rceil \left(\frac{\zeta_l}{1-\zeta_l}\right)^p$$

où ζ_l est la plus petite racine positive de $x^l + x = 1$.

Preuve: Si l'ensemble image de la fonction d'évaluation est de cardinal C, alors il faut au plus $\lceil log_2(C) \rceil$ appels de la fonction *fab* pour connaître par recherche dichotomique la valeur negamax de la racine. D'après la propriété 2.2.6, la complexité moyenne de chaque appel de *fab* est bornée par $\frac{\zeta_l}{1-\zeta_l}$. □

Corollaire 2.3.2.1 *L'algorithme* NegaC* *est donc asymptotiquement optimal pour la recherche de la valeur minimax.*

Preuve: $I_{l,p}^{1/p} \leq (\lceil \log_2(C) \rceil)^{1/p} (\frac{\zeta_l}{1-\zeta_l})$ donc $lim_{p \to \infty} I_{l,p} \leq \frac{\zeta_l}{1-\zeta_l}$ et la propriété 2.2.5 nous donne le résultat attendu. □

La figure 2.12 montre le rapport de la complexité de la recherche $\alpha\beta$ avec borne supérieure et de la complexité moyenne de l'algorithme *NegaC**. Nous y voyons que la recherche *NegaC** peut être supérieure à la recherche $\alpha\beta$ dans le cas de grands problèmes.

Lors d'une recherche avec l'algorithme *NegaC** au cours des différents appels avec les fenêtres minimales, certains nœuds sont évalués plusieurs fois. Nous pouvons dès lors pour économiser la recherche utiliser au maximum les informations collectées lors des recherches précédentes en utilisant la procédure *informed-fail-soft-*$\alpha\beta$ donnée par la figure 2.13[185, 186, 187] : pour chaque nœud visité, les bornes connues sur les valeurs negamax du nœud sont gardées et comparées à notre valeur cible. De plus si un élagage se produit lorsqu'on examine un nœud P sur le fils i, il n'est pas nécessaire dans les recherches suivantes de considérer les fils P_j de P quand $i < j$, cela nous permet d'économiser de la mémoire[42, 140]. Comme nous l'avons montré dans [186], il n'est pas nécessaire d'utiliser la procédure *informed-fail-soft-*$\alpha\beta$ pour toutes les profondeurs, en effet, les informations sur les nœuds proches de la racine

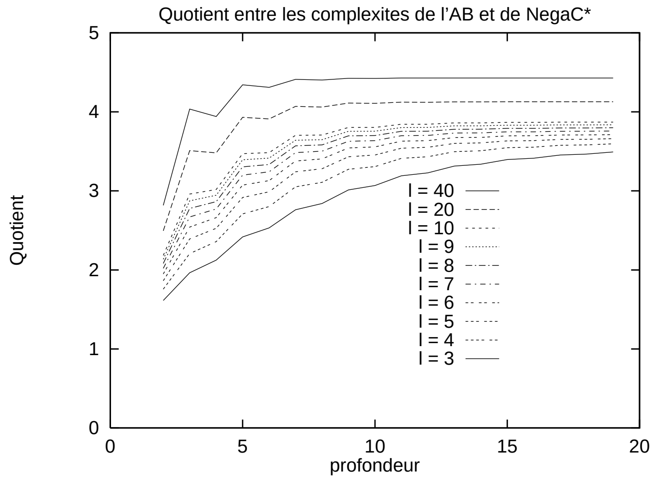


FIG. 2.12 - *Comparaison de la complexité de l'$\alpha\beta$ et de NegaC*.*

En ordonnée, nous avons le quotient de la complexité de l'$\alpha\beta$ avec la complexité d'une recherche avec une fenêtre minimale, en fonction de la profondeur de l'arbre de recherche. Nous avons tracé les courbes correspondantes à différentes largeurs (l).

sont plus importantes car elles permettent d'élaguer des sous-arbres plus grands. La mémoire nécessaire est alors de $(l^{\lceil P_0/2 \rceil} + 1 - 1)/(l - 1)$, si on décide de stocker les nœuds jusqu'à la profondeur P_0 [9].

2.4 Les algorithmes minimax non basés sur les élagages $\alpha\beta$.

Jusqu'à présent, nous avons montré des algorithmes minimax faisant appel à un processus de recherche du type « profondeur d'abord », c'est à dire que l'arbre de recherche était exploré de haut en bas puis de gauche à droite. Dans cette partie, nous allons présenter les algorithmes minimax faisant appel à une recherche du type « meilleur d'abord ».

Définition 2.4.1 *A chaque pas d'un processus de recherche* **meilleur d'abord** *, nous choisissons le plus prometteur des nœuds que nous avons générés jusqu'à présent. Ceci se fait en appliquant une fonction heuristique appropriée à chacun d'eux. Nous élargissons alors le nœud choisi en utilisant les règles pour générer ses successeurs qui sont ajoutés à l'ensemble des nœuds déjà générés. Le processus se termine dès que la solution est trouvée ; dans notre cas précis, elle se termine dès que la valeur negamax de la racine est connue.*

9. De plus, l'utilisation de l'heuristique de la table de transposition (cf § 3.1.2.3) permet de stocker efficacement en temps et en mémoire les bornes sur la valeur negamax d'un nœud. La table de transposition étant nécessaire pour les programmes de jeu réel, le surplus de mémoire pour le NegaC* informé est alors bien moindre. (Un entier par entrée dans la table de transposition).

```
funct ifab(Position, α, β) ≡
    ⌈M ← ProchainCoup(Position);
     if M = null then exit g(Position); fi

     if (PremierVisite(Position))
       then
             CreationEnregistrement(Position);
             Position.min ← −∞;
             Position.max ← ∞;
        else
             if (Position.min > α)
               then α ← position.min fi
             if (Position.max < β)
               then β ← position.max fi
             if (α ≥ β)
               then exit β; fi
     fi
     Meilleur ← −∞;
     while M ≠ null do
           valeur ← −ifab(Position • M, −β, −α);
           if valeur > Meilleur
             then Meilleur ← valeur;
                  if Meilleur > α
                    then α ← Meilleur;
                  fi
           fi
           if α ≥ β
             then
                  Position.min = Meilleur;
                  DetacheFilsPrecedant(Position, M)
                  exit Meilleur; fi

           M ← ProchainCoup(Position) od;
     Position.max = Meilleur
     Meilleur⌋.
```

FIG. 2.13 - *La procédure Informed-Fail-Soft-$\alpha\beta$.*

2.4.1 L'algorithme SSS*

L'algorithme *SSS**[168] est un algorithme non directionnel pour calculer la valeur minimax d'arbres ET/OU. Les arbres de jeu étant des arbres ET/OU, nous allons pouvoir y appliquer l'algorithme *SSS**.

Un arbre de jeu est un arbre ET/OU où la racine est un nœud de type ET et tous les fils d'un nœud de type ET (resp. OU) sont de types OU (resp. ET). Un arbre solution $\mathcal{T}$ d'un arbre de jeu $\mathcal{S}$ est un arbre où :

1° La racine de $\mathcal{S}$ est la racine de $\mathcal{T}$

2° Pour tout nœud N interne de $\mathcal{S}$ inclus dans $\mathcal{T}$, tous les fils N de $\mathcal{S}$ sont dans $\mathcal{T}$ si N est de type ET, sinon exactement un fils de N de $\mathcal{S}$ est dans $\mathcal{T}$ si N est de type OU.

On notera $F_{\mathcal{T}}(p)$ la valeur de l'arbre solution $\mathcal{T}$ définie comme le minimum de la valeur des feuilles de $\mathcal{T}$.

Propriété 2.4.1 *[168] Quel que soit l'arbre solution $\mathcal{T}$ de racine p, $minimax(p) \geq F_{\mathcal{T}}(p)$ et il existe $\mathcal{T}_\prime$ tel que $minimax(p) = F_{\mathcal{T}_\prime}(p)$.*

La figure 2.14 nous montre la définition de l'algorithme *SSS** telle qu'elle est donnée dans [40], de telle sorte que l'algorithme *SSS* domine* l'algorithme $\alpha\beta$: chaque nœud de l'arbre examiné par l'algorithme *SSS** est examiné par l'algorithme $\alpha\beta$ [10].

*SSS** est particulièrement efficace comparé aux algorithmes précédents quand la meilleure suite de mouvements se trouve dans la partie droite de l'arbre de recherche. L'inconvénient principal à l'utilisation de *SSS** provient du fait de l'utilisation de la liste OUVERT qui demande beaucoup de ressources mémoire. En effet, cette liste peut contenir jusqu'à $l^{\lceil p/2 \rceil}$*nœuds* pour un arbre de largeur l et de profondeur p. De plus, par rapport à l'$\alpha\beta$, comme deux nœuds consécutifs ne sont pas forcément voisins, il y a un surcoût supplémentaire de gestion de la position de jeu. C'est pourquoi de nombreuses variantes sont apparues pour remédier à ces problèmes.

A ce jour les plus efficaces sont :

DUAL*[105] qui est basée sur des compléments de l'opérateur Γ nécessite moins de mémoire et examine les nœuds de la racine de gauche à droite comme le ferait un algorithme basé sur l'$\alpha\beta$. Mais le surcoût dû à l'utilisation intensive de la mémoire est encore trop grand[105]($l^{\lfloor p/2 \rfloor}$ nœuds pour un arbre de largeur l et de profondeur p).

RecSSS*[28, 137] est une re-formulation de l'algorithme SSS*. D'après Reinefeld[137], les algorithmes recSSS* et surtout recDUAL* sont plus rapides en temps d'exécution que l'algorithme $\alpha\beta$ sur des arbres aléatoires et des arbres fortement ordonnés. Mais ces

10. La version initiale de l'algorithme *SSS** donnée par [168] ne dominait pas toujours l'$\alpha\beta$.

```
funct SSS*(Position) ≡
    push(racine, EN_VIE, +∞);
    while (true) do
        pop(noeud, etat, h);
        if (etat = EN_VIE)
          then
             if (noeud est terminal)
               then insere(noeud, RESOLU, min(f(noeud, h)));
             elsif (noeud est un nœud ET)
                 then
                       push(noeud_1, EN_VIE, h);
                 else
                       soit n tel que noeud_n
                        soit le dernier fils de noeud
                       for j = n to j = 1 do
                           push(noeud_j, EN_VIE, h);
                       od
             fi
        elsif (noeud = racine)
             then exit h;
        elsif (noeud est un nœud ET)
             then
                 if (noeud a un frère non exploré)
                   then push(frere, EN_VIE, h);
                    else push(pere, RESOLU, h); fi
             else
                 purge(parents, noeud);
                 push(pere, RESOLU, h);
        fi
    od
```

FIG. 2.14 - *L'algorithme SSS* donné par [40]*

algorithmes requièrent une capacité mémoire très importante : $(l^{\lceil p/2 \rceil+1} - 1)/(l - 1)$ nœuds pour recSSS* et $(l^{\lfloor p/2 \rfloor+1} - 1)/(l - 1)$ nœuds pour recDUAL* pour un arbre de profondeur p et de largeur l.

2.4.2 Nombres de conspirateurs et nombres de preuves

Une approche différente est utilisée dans les recherches de type minimax issues de la recherche des « nombres de conspirateurs[106] ». Ces recherches du type du meilleur d'abord utilisent elles aussi beaucoup de mémoire. La valeur d'un arbre de jeu est considérée plus *sûre*[11] s'il faut que plusieurs nœuds « conspirent »[12] pour changer la valeur minimax de la racine. L'application directe de cette recherche dans le programme d'échecs ARACHNE de Marten van der Meulen fut loin d'être concluante[72], mais des résultats plus encourageants furent obtenus pour un programmes uniquement tactique[149]. Cet algorithme a en outre permis, dans sa variante « $\alpha\beta$-conspiracy number search », de déterminer la valeur théorique du jeu *Puissance IV*[4, 177][13].

Les nombres de preuves[7, 5] est un algorithme dans le même ordre d'idée que les nombres conspirants. L'idée de départ est de remplacer la recherche fail-soft-$\alpha\beta$ pour déterminer si la valeur minimax de la racine est supérieure ou égale à une valeur donnée v dans la recherche NegaC*. Ici, nous allons étendre le nœud qui requiert le moins de calculs a priori; à chaque nœud, nous maintenons le nombre de nœuds a priori nécessaires selon la branche, pour confirmer ou nier que la valeur de la racine est supérieure à v. Cette recherche a eu de bons résultats sur un programme d'Awari[7] et pour la recherche de mats profonds aux échecs[6]. Mais en vertu du corollaire 2.2.7.1, la recherche du nombre de preuves ne peut être supérieure à la recherche fail-soft-$\alpha\beta$ avec des fenêtres minimales dans le cas général. En fait, pour que la recherche des nombres de preuves soit efficace, il faut, d'après les expériences menées pas Allis[5], que l'arbre de recherche contienne deux types de non uniformité :

1° Le nombre de mouvements légaux par nœud ne doit pas être constant;

2° Il doit exister des lignes courtes dans l'arbre. (C'est-à-dire : la profondeur de l'arbre de jeu ne doit pas être uniforme.)

Nous avons redéfini la recherche du nombre de preuves dans le modèle *negamax*, ce qui permet une implantation plus concise comme le montre la figure 2.15.

A l'instar de [6], dans le cadre des problèmes de recherche de mats aux échecs, nous avons comparé la recherche du nombre de preuves avec notre programme d'échecs ECUME et un programme utilisant une fenêtre minimale et une recherche $\alpha\beta$ sans extensions(cf le chapitre 3). Nous avons choisi de compter le nombre de mats trouvés en 3 minutes sur la même machine par les différents programmes; le programme de recherche du nombre de preuves est écrit en langage C, et les deux autres programmes sont écrits pour la partie recherche en assembleur. Les mats à trouver étaient dans la base des 300 problèmes du livre « Win at

11. En ce sens que la probabilité que la valeur change est faible.
12. Les nœuds conspirent s'ils changent simultanément de valeurs.
13. Il est à noter que ce jeu fut indépendamment résolu par l'algorithme $\alpha\beta$ en même temps.[3]

```
funct pnsearch(racine, v0) ≡
                                                        Initialisation de la racine
    racine.proof ← racine.disproof ← 1;
    v ← v0;
    while (racine.proof > 0 ∧ racine.proof ≠ ∞) do
        noeud ← racine;
                                              Recherche le nœud le plus « prouvant ».
        while (noeud déjà étendu ) do
            i ← 1;
            while noeud.fils[i].disproof ≠ noeuds.proof do
                i ← i + 1;
            od
            noeud ← noeud.fils;
            v ← −1 − v;
        od
                                                     Etend le nœud le plus prouvant
        if noeud est terminal
          then
              if g(noeud) > v
                then
                    noeud.proof ← 0; noeud.disproof ← ∞;
                else
                    noeud.disproof ← 0; noeud.proof ← ∞;
              fi
              noeud ← noeud.pere
          else
              GenereLesFils(noeud);
              for i ← 1 to noeud.nombredefils do
                                  Initialisation des fils du nœud le plus « prouvant ».
                  noeud.fils[i].proof ← noeud.fils[i].disproof ← 1;
              od
        fi
                                         Met à jour les nombres de preuves de la branche
        while noeud ≠ 0 do
            noeud.proof = min_{N∈fils(noeud)} N.disproof;
            noeud.disproof = Σ_{N∈fils(noeud)} N.proof;
            noeud ← noeud.pere
        od
    od
```

FIG. 2.15 - *La recherche du nombre de preuves dans le modèle* negamax

Chess »[141] (certains problèmes sont des mats, d'autres des combinaisons). La recherche du nombre de preuves était légèrement modifiée dans l'initialisation des nombres de preuves en donnant pour un nouveau nœud 1 dans le nombre de preuves et le nombre de coups légaux pour le nombre de « disproof » ; de plus lors du choix du nœud le plus prouvant, on considérait, à nombre de « disproof » égal, le nœud racine du sous-arbre le moins profond.

TAB. 2.2 - *Comparaison de la recherche du nombres de preuves, du programme* ECUME *et d'un programme de recherche de mats sur les positions de « Win At Chess »[141]*

Mats identifiés par au moins un des algorithmes	1 4 5 6 9 12 14 27 35 49 50 51 54 55 57 60 61 64 79 84 88 97 99 102 103 104 105 132 134 136 138 139 143 154 156 158 159 160 161 167 168 172 173 177 179 182 184 186 188 191 197 201 203 211 212 215 217 218 219 222 225 241 244 246 250 251 252 253 260 263 266 267 278 281 282 283 285 290 293 295 298
Non trouvés par la recherche du nombre de preuves	105
Non trouvés par le programme ECUME	6 159 222 250 281
Non trouvés par la recherche de mats	6 49 51 105 159 168 182 201 211 217 218 222 241 250 251 252 260 266 278 281

Le programme à base de recherche du nombre de preuves a identifié 80 mats, le programme ECUME en a identifié 76 et le programme de recherche de mats basé sur l'$\alpha\beta$ en a identifié 60. En tout 81 mats ont été identifiés par au moins un des programmes. Dans cette base de problèmes, nous connaissons deux autres mats qu'aucun de nos programmes n'a trouvé en 3 minutes, les problèmes 96 et 213[14].

La table 2.2 nous résume les résultats des différents programmes. Nous voyons que le meilleur résultat est obtenu par la recherche du nombre de preuves, puis par le programme ECUME et enfin par la recherche spéciale de mats.

2.5 Comparaison qualitative des algorithmes minimax

Nous avons comparé expérimentalement les algorithmes $\alpha\beta$, SSS* *NegaScout* et *NegaC** informé.

Les arbres de recherche qui nous ont servi étaient définis par les caractéristiques suivantes : uniforme de largeur l et de profondeur p. L'évaluation appliquée aux feuilles était soit aléatoire, soit de telle sorte que les arbres soient *fortement ordonnés*[102][15].

14. Le problème 96 est trouvé en 4 minutes par la recherche du nombre de preuves et le problème 213 est un mat en 18 coups.

15. Le premier fils d'un nœud de l'arbre ayant 70% de chance d'être le meilleur fils.

Chaque mesure correspond à la moyenne arithmétique de 30 mesures réelles.

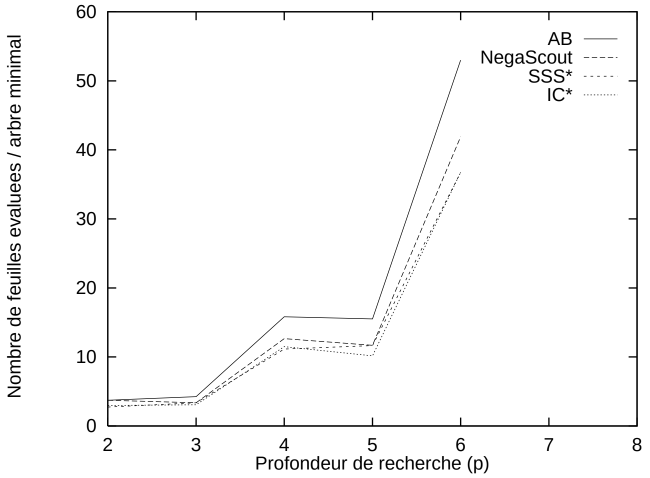


FIG. 2.16 - *Comparaison avec des arbres de largeur 15 à évaluation aléatoire*

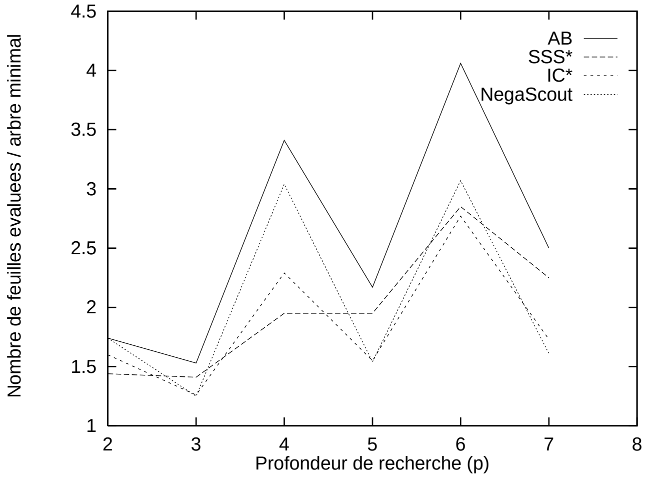
Pour les algorithmes $\alpha\beta$(AB), SSS, NegaScout et NegaC* informé (IC*)*

Dans la figure 2.16, nous avons comparé ces algorithmes sur des arbres aléatoires de largeur 15. La recherche $\alpha\beta$ est la moins performante en nombre de nœuds et les recherches SSS* et *NegaC** informé[16] ont des performances équivalentes en nombre de nœuds. Mais, comme le surcoût de temps dû à l'utilisation de la mémoire est plus important pour l'algorithme SSS*[137], en temps comme en nombre de nœuds, l'algorithme *NegaC** informé est nettement supérieur dans le cadre d'arbres à évaluation aléatoire.

Les mêmes algorithmes sont alors comparés dans la figure 2.17 sur des arbres de jeux plus proches de ceux que l'on rencontre dans les programmes de jeux de réflexion : les arbres fortement ordonnés. Ici, la recherche *NegaScout* montre sa supériorité face aux algorithmes SSS* et *NegaC** informé dans les niveaux impairs. Dans les niveaux pairs les recherches *NegaC** et SSS* donnent des résultats semblables en nombre de nœuds, donc l'avantage est pour *NegaC** en temps. De plus, cette même figure nous montre l'importance du tri des mouvements : en effet, la comparaison des figures 2.16 et 2.17 montre que ces algorithmes examinent 10 fois moins de nœuds sur des arbres de largeur 15 et de profondeur 5 et 6, ce qui infirme la conclusion de [137] qui stipule que le fait d'utiliser recSSS*(qui examine plus de nœuds que SSS* mais avec une gestion de la mémoire moins pénalisante en temps) permet d'obtenir de bons résultats même sur des arbres non ordonnés.

Dans la figure 2.18, nous avons fait varier la largeur de l'arbre de recherche à profondeur fixée paire. Nous voyons donc que même dans les profondeurs paires, les algorithmes *NegaC** et *NegaScout* peuvent faire mieux que l'algorithme SSS* pourvu que la largeur de l'arbre soit suffisamment grande.

16. Utilisant la procédure Informed-fail-soft-$\alpha\beta$.



En ordonnées, nous avons le nombre de feuilles nécessaires pour calculer la valeur minimax de la racine divisé par le nombre de feuilles de l'arbre minimal.

FIG. 2.17 - *Comparaison avec des arbres fortement ordonnés de largeur 15*

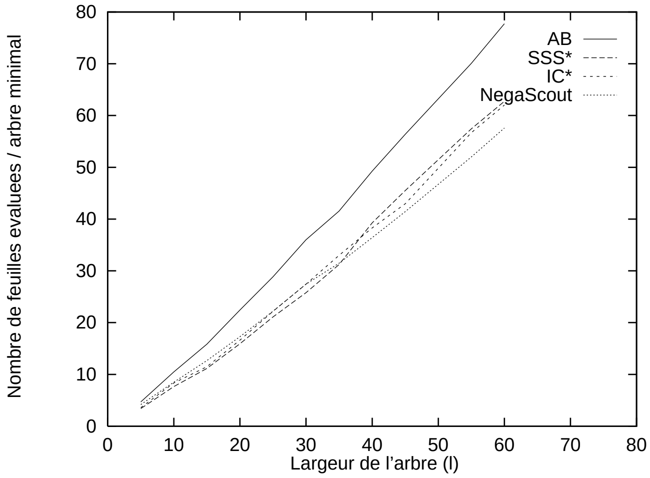


FIG. 2.18 - *Comparaison en fonction de la largeur de l'arbre pour des arbres de profondeur de 4 demi-coups.*

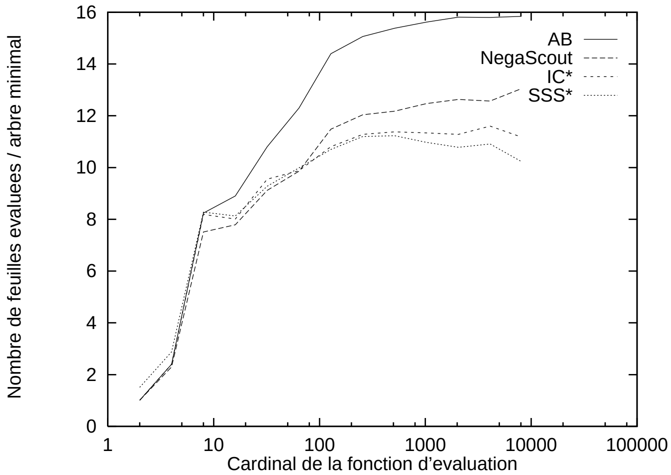


FIG. 2.19 - *Comparaison en fonction du cardinal de l'ensemble image de la fonction d'évaluation*

La borne que nous avons établie pour la complexité moyenne de l'algorithme *NegaC** prend en compte le cardinal de l'image de la fonction : nous avons calculé l'influence de ce cardinal sur les efficacités des algorithme minimax dans le cas d'arbres aléatoires[17]. Nous voyons que tous les algorithmes sont pareillement sensibles à cette valeur à l'exception du *NegaScout* qui est le meilleur pour une fonction d'évaluation prenant peu de valeurs.

2.6 Comportements pathologiques de la recherche minimax

Nous avons supposé jusqu'ici que plus la recherche est profonde, meilleur est le choix du mouvement. Cette supposition est basée sur le fait qu'une recherche profonde est plus à même de s'occuper des combinaisons et des pièges. Mais les analyses mathématiques de Beal[15, 16] utilisant un arbre de jeu uniforme à évaluation aléatoire, reprises par Nau[112, 113, 114] ont montré que dans bien des jeux une recherche minimax plus profonde ne mène pas toujours à un meilleur choix du mouvement. En analysant la propagation des erreurs de la fonction d'évaluation en utilisant le schéma minimax, ils ont montré que l'erreur d'évaluation à la racine était croissante avec la profondeur d'analyse. Les jeux où cela se produit sont dits *pathologiques*.

La pratique avec les programmes de jeux de réflexion tels que les échecs ou Othello a montré au contraire que le niveau de jeu du programme était croissant avec la profondeur des arbres évalués. Pour comprendre ce résultat, Beal[16], Nau[113, 115, 116], Bratko et Gams[34] ont donné des critères de non- pathologie basés sur les dépendances entre les évaluations des

17. L'algorithme que nous utilisons pour générer les arbres fortement ordonnés ne nous permettant pas de restreindre le nombre de valeurs pouvant être prises par la fonction d'évaluation.

feuilles terminales. Pearl[131] a montré qu'un arbre de jeu qui contient un nombre de courtes séquences de mouvements conduisant à des positions terminales appelées les « pièges », n'est pas pathologique. De plus Schrüfer[155] a montré que les arbres tels que chaque nœud gagnant a au moins deux fils, ne sont pas pathologiques. Bien que les évaluations des feuilles ne sont pas indépendantes et qu'il existe des pièges dans le cas de l'approche standard minimax, il n'est pas clair que de tels programmes jouant aux échecs ne soient pas amenés à évaluer un arbre de jeu pathologique.

2.7 Les alternatives au paradigme minimax

2.7.1 L'algorithme B*

Berliner[19] propose l'algorithme B*. Dans sa forme primitive, la fonction d'évaluation retourne deux bornes pour une position donnée. La valeur réelle de la position est supposée être alors entre la borne optimiste et la borne pessimiste.

Ces valeurs sont remontées vers les mouvements de la racine. Un mouvement sera choisi si sa borne pessimiste est supérieure aux bornes optimistes des autres mouvements[18].

Palay[127, 128] propose l'algorithme PB^* qui utilise non pas un intervalle mais plutôt une distribution de la probabilité que le nœud soit supérieure à une valeur donnée x. Il remonte ces distributions vers la racine en utilisant une fonction proche du *Negamax* :

$$P^{(pere)}(v \leq x) = \prod_{i=1}^{b} P^{(fils_i)}(v \geq -x)$$

Expérimentalement muni de ces informations complémentaires, Palay a montré que PB^* était plus efficace que B^* pour la résolution de positions d'échecs tactiques.

Cet algorithme a été implanté dans le programme HITECH B*[25] en simplifiant le modèle mathématique donné par Palay[128]. Comparé au programme HITECH[19] qui utilise le principe minimax et contient les mêmes connaissances positionnelles, HITECH B* a montré que l'algorithme B^* convenait mieux à des positions tactiques qu'à des positions faisant plus appel aux connaissances stratégiques et positionnelles (les fins de parties par exemple). Confrontées entre elles, la version HITECH l'emporte sur HITECH B* dans 60 % des parties.

2.7.2 Approximation Min/Max

Rivest[143] propose de remplacer dans l'algorithme *Negamax* l'opérateur *max* par une approximation $(\frac{1}{n}\sum_{i=1}^{n} a_i^p)^{1/p}$. L'arbre est ensuite étendu d'une façon « meilleur d'abord ». En

18. Cela suppose quand même qu'un tel mouvement existe : si deux mouvements sont égaux, par exemple s'ils mènent à la même position après transposition, l'algorithme B^* ne se termine pas.

19. HITECH développé par l'équipe de Hans Berliner, ancien champion du monde d'échecs par correspondance, est construit autour d'un circuit VLSI spécialisé.

utilisant une fonction d'évaluation naïve pour le jeu de *Puissance IV*, Rivest a trouvé que son algorithme était expérimentalement meilleur que l'$\alpha\beta$ pour des arbres de profondeur égale. Mais, comme son algorithme requiert plus de calculs par nœuds, il est expérimentalement inférieur à l'$\alpha\beta$ si on fait jouer les deux algorithmes sous la contrainte du temps. Le grand avantage de l'algorithme de Rivest est sa simplicité, sa vitesse relative et son comportement robuste face aux arbres de jeux à grands nombres de conspirateurs.

2.7.3 Modèle probabiliste naïf

Dans le modèle probabiliste naïf, la fonction d'évaluation retourne pour les feuilles la probabilité que la position Q soit un gain pour le joueur au trait, que nous noterons $P(Q \in \mathcal{G})$. Une position Q est un gain pour le joueur au trait s'il possède un coup gagnant, et donc une perte si tous les coups sont perdants. Donc sa probabilité de gain est, si on considère l'indépendance de la probabilité de gains des P_i, de

$$P(Q \in \mathcal{G}) = 1 - \prod_{i=1}^{p} P(Q_i \in \mathcal{G}).$$

Donc si deux positions sœurs sont gagnantes avec une probabilité $\frac{1}{2}$, la probabilité de gain pour le joueur au trait dans la position père de ses deux positions est d'après le schéma probabiliste naïf de $0,75$.

Ce schéma nous permet de déterminer, pour un arbre de profondeur donnée, les probabilités de gain des positions filles de la racine de l'arbre de recherche, et le joueur au trait dans la racine choisira le mouvement qui maximise sa probabilité de gain.

Mais ce modèle souffre deux sérieux problèmes :

1° le fait de calculer la probabilité de gain d'un joueur ne donne pas la probabilité que le joueur sera capable de gagner sa position;

2° la supposition d'indépendance des positions se montre trop forte : l'expérience montre qu'une amplification exponentielle apparaît souvent menant à la conclusion erronée qu'un mouvement procurant un faible avantage est un gain avec la probabilité 99,999999 %[14].

Des expériences menées sur le jeu Puissance IV ont montré que le modèle *Negamax* est très nettement supérieur au modèle stochastique naïf sur des arbres de jeu de même profondeur, mais que le rapport se renverse avec un petit avantage pour le modèle stochastique dans les mêmes conditions pour le jeu de Kalah (variante Slagle et Dixon)[165].

2.7.4 Best Play for Imperfect Player (BPIP)

En prenant un modèle mathématique proche de l'algorithme PB*, Baum et Smith[14] proposent l'algorithme BPIP pour utiliser des fonctions d'évaluation retournant une distribution de probabilité.

La définition du « Best Play for Imperfect Player » est la suivante :

1° La valeur de la position à la racine est le maximum des valeurs de ses fils;

2° la valeur d'une position interne P (non racine et non feuille) est la somme pour chaque mouvement M issu de P de la probabilité que l'algorithme BPIP, appliqué à la racine P, choisisse M comme mouvement, multipliée par la valeur de la position $P \bullet M$;

3° la valeur d'une position terminale est l'espérance de la valeur obtenue par l'algorithme BPIP appliqué à cette position.

Le mouvement retenu sera le fils de la racine ayant la plus grande valeur.

Les résultats obtenus jusqu'ici[165] montrent que pour le jeu de Kalah (variante Slagle et Dixon) :

1° à arbres de jeux égaux et à évaluations égales, BPIP est légèrement meilleur que le principe *Negamax* (7% de parties gagnées en plus par BPIP) quelle que soit la profondeur des arbres de recherche;

2° en utilisant des heuristiques adaptées à BPIP pour étendre l'arbre de recherche, l'avance de BPIP s'accentue (13% de gains en plus);

3° mais le traitement de l'algorithme $\alpha\beta$ étant moins coûteux que celui de l'algorithme BPIP ($\alpha\beta$ étant 3,5 fois plus rapide en nœuds par seconde que l'algorithme BPIP), l'algorithme $\alpha\beta$ est bien plus performant que BPIP sous des contraintes de temps.

2.8 Conclusion

De l'étude exposée au § 2.5, nous pouvons voir que l'algorithme *Negascout*, ou dans sa forme un peu affaiblie *PVS*, est confirmé comme le meilleur choix pour les arbres fortement ordonnés. C'est l'algorithme le plus utilisé dans la pratique pour les programmes d'échecs. Dans les programmes d'Othello, nous avons noté que les arbres étaient souvent moins bien ordonnés du fait de la non stabilité des fonctions d'évaluation utilisées ; cela favorise plus des recherche du type $\alpha\beta$ et NegaC*, et de plus NegaC*[186] est particulièrement adapté à la fin de partie.

Enfin, nous pensons que les recherches de type B* et plus particulièrement les recherches telles que BPIP, sont très probablement sous-exploitées à l'heure actuelle, et que leurs potentiels sont énormes.

Chapitre 3

Améliorations de l'enveloppe de la recherche

Dans ce chapitre, nous abordons les heuristiques permettant d'améliorer la recherche, c'est à dire la rendre plus performante. En outre, nous exposons les heuristiques que nous utilisons dans nos programmes d'échecs [1] et nous comparons expérimentalement les heuristiques de classement des mouvements aux échecs. [2]

Les heuristiques utilisées pendant le processus de recherche permettent de gagner en performance, c'est à dire ordonner les mouvements [3] et donner une évaluation plus juste notamment en approfondissant les variantes intéressantes.

Les heuristiques sont classées en deux grandes familles : les heuristiques dépendantes du domaine et celles indépendantes du domaine. Nous entendons par domaine le jeu auquel s'applique ces heuristiques. Certaines sont donc générales, comme l'approfondissement itératif, les mouvements meurtriers, les extensions singulières ; d'autres, au contraire, sont limitées à des jeux possédant des caractéristiques particulières comme l'heuristique du mouvement nul, voire limitées à un seul jeu comme les extensions statiques.

Dans la suite de ce chapitre, nous supposons que la recherche negamax choisie est un algorithme basé sur un processus de recherche en profondeur d'abord tel que par exemple le *Negascout* ou l'$\alpha\beta$. Nous présentons donc un certain nombre d'heuristiques et nous donnons des exemples d'application empruntés à différents jeux.

Nous décrivons notamment les heuristiques de recherche que nous avons implantées dans nos différents programmes d'échecs.

Finalement, nous terminons le chapitre par un exposé des problèmes que pose l'utilisation des heuristiques et sur les méthodes utilisées pour choisir une bonne combinaison.

1. Les extensions statiques d'ECUME au § 3.2.4, la recherche de la tranquillité de CUMULUS 1.5 au § 3.3.4 et les extensions entropiques de JOKER au § 3.2.5.
2. Au § 3.1.3.
3. *With the literature as rich as it is now in move-ordering and pruning schemes, the time seems ripe for a balanced scientific comparison, identifying their strengths and weaknesses. I recommend it as a splendid subject for a Master's thesis or even for a Doctoral Dissertation.* Jonathan Schaeffer[152].

3.1 Heuristiques ayant trait à l'ordre des mouvements

L'étude comparative entre différents algorithmes faite au chapitre 2 nous a montré que les algorithmes de recherche étaient très sensibles à l'ordre des mouvements dans l'arbre de recherche. En fait, pour obtenir un arbre minimal, il faut que le premier fils des nœuds de type 1 et 2 soit leur meilleur fils. L'ordre des mouvements n'a pas d'effet pour les nœuds de type 3.

De nombreuses heuristiques furent développées pour obtenir un bon ordre. Slagle et Dixon[162] proposent « l'ordre fixe » : classer les mouvements en suivant la valeur de la fonction d'évaluation prise après que le mouvement soit effectué. Ils proposent aussi « l'ordre dynamique » qui termine la recherche d'un sous-arbre en fonction de la valeur de la fonction d'évaluation prise dans les nœuds internes et en reprenant la recherche sur un nouveau nœud.[4]

Ces deux heuristiques sont très efficaces comparées à un ordre aléatoire mais elles ont été abandonnées dans la pratique car le coût en temps des générations et évaluations nécessaires est trop important. Il existe des méthodes plus rapides à mettre en œuvre, qui donnent les mêmes résultats sur l'ordre des mouvements pour un coût moindre. Ces méthodes peuvent se classer en deux catégories : les méthodes qui dépendent du domaine, donc du jeu auquel est appliquée l'heuristique et celles, plus générales, qui utilisent des informations venant de manière dynamique de la recherche elle-même et qui seront indépendantes du jeu choisi.

3.1.1 Ordre induit par des heuristiques dépendantes du domaine

Pour ordonner les mouvements, on utilise dans un premier temps des heuristiques tirées de la connaissance humaine sur le jeu traité, c'est à dire le domaine d'application de l'algorithme.

Le but est d'ordonner les coups selon leurs possibilités d'être des bons coups. Nous allons prendre des exemples pour différents jeux :

- Aux échecs par exemple, les mouvements de capture auront une importance particulière. Bettadapur[27] a montré que les captures doivent être ordonnées dans les sens « MinorToMajor », c'est à dire les captures des pièces les plus importantes par les pièces les plus faibles en premier, le critère principal étant la valeur de la pièce capturée. Uniacke[179] modifie cette heuristique en utilisant une fonction donnant une approximation du résultat des échanges successifs qui se produisent sur la case de la pièce capturée et en accordant une importance particulière aux mouvements qui capturent la dernière pièce bougée. Ke et Parng[84] ordonnent les mouvements autres que les captures en fonction des menaces exercées par l'adversaire sur les cases d'arrivée des mouvements.
- A Othello, les mouvements seront ordonnés selon le type des cases[59, 85, 185]. La figure 3.1 nous donne la nomenclature des cases à Othello et l'ordre de génération sera, par exemple, les coins, puis les cases $A - D - B - E - G - F - C - X$.

4. Si l'on effectue ceci sur l'arbre global après chaque génération/évaluation de mouvement, on transforme la recherche en profondeur en recherche du meilleur d'abord[82].

	C	A	B	B	A	C	
C	X	F	G	G	F	X	C
A	F	D	E	E	D	F	A
B	G	E	○	●	E	G	B
B	G	E	●	○	E	G	B
A	F	D	E	E	D	F	A
C	X	F	G	G	F	X	C
	C	A	B	B	A	C	

FIG. 3.1 - *Nomenclature des cases à Othello*

- Dans les jeux dérivés des jeux d'Awari, mettre les mouvements tactiques en tête accélère considérablement la recherche. Ainsi, Rivin, Smith et Tudor ont résolu la variante du KALAH donnée par Slagle et Dixon en ordonnant les mouvements de sorte que les mouvements tels que la dernière pierre tombe dans le Kalah du joueur soient examinés en premier, suivis des mouvements qui capturent les pierres adverses[165].

3.1.2 Ordre induit par des heuristiques indépendantes du domaine

On peut aussi induire l'ordre des mouvements par des statistiques recueillies au fur et à mesure que la recherche se déroule. Les heuristiques de tri ainsi réalisées seront indépendantes du domaine, donc du jeu.

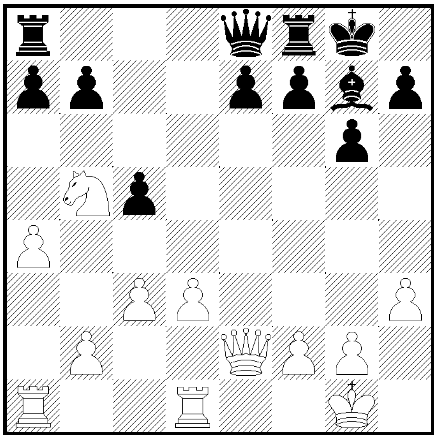
3.1.2.1 Les mouvements meurtriers

Les mouvements meurtriers représentent l'heuristique la plus commune utilisée pour avoir comme premier mouvement étudié un mouvement qui a de fortes chances d'être le meilleur.

Les bases de l'heuristique des mouvements meurtriers sont posées par Huberman[79] :

Pendant l'exploration de l'arbre de recherche, après la fin de l'examen d'une position, le meilleur mouvement obtenu pour cette position sera stocké dans une table indexée par la distance de la position à la racine (la profondeur). Alors, pour une position de même profondeur examinée ensuite, le premier mouvement envisagé sera le mouvement stocké dans la table à l'entrée correspondante à la profondeur de la position. Cette table est appelée la table des « mouvements meurtriers ».

L'idée sous–jacente est le fait que dans beaucoup de positions au même niveau de l'arbre de recherche, le meilleur mouvement sera le même. Ainsi, si nous prenons un exemple emprunté au jeu d'échecs, dans la figure 3.1, les blancs menacent de jouer ♘c7 , fourchettant la ♖a8 et la ♕e8. C'est aux noirs de jouer. Si l'on suppose que la recherche examine 1 ... e5 en premier, et si la profondeur de la recherche est supérieure à 4 demi-coups, après l'analyse de 1 ... e5 , la meilleure réponse sera 2. ♘c7 . Cette réponse sera stockée en tant que mouvement meurtrier, et ce mouvement sera testé en priorité sur les autres premiers mouvements, jusqu'au moment où la recherche sélectionnera un mouvement qui ne permettra pas la fourchette.



Trait aux noirs.

Diagramme 3.1: *Exemple d'utilisation des mouvements meurtriers*

Pour l'implantation des mouvements meurtriers, il est possible de choisir d'en stocker plusieurs. Le programme CHESS 4.5[164]par exemple, en stocke deux par niveau et le programme HIARCS 5.0[179] en stocke trois par niveaux (deux pour les mouvements causant des élagages, un pour les meilleurs mouvements); on peut aussi choisir d'essayer comme premiers mouvements les mouvements meurtriers de niveaux inférieurs (avec toujours le même joueur au trait).

3.1.2.2 Approfondissement itératif / Tables de réfutations / Fenêtres aspirantes

L'approfondissement itératif est introduit par Scott[157] afin de contrôler le temps pris par la recherche : plutôt que de commencer la recherche à la profondeur N, en ne sachant pas combien de temps elle va prendre, on effectue une succession de recherches aux profondeurs 1, 2, 3, ... demi-coups, jusqu'à ce que le temps alloué soit atteint. Le meilleur coup à la profondeur i étant évalué en premier à la profondeur $i + 1$.

Au fur et à mesure que les années passèrent, cette heuristique fut raffinée et améliorée jusqu'au point où l'approfondissement itératif jusqu'à la profondeur N devint plus rapide que la recherche directe au niveau N[164].

La plus simple de ces améliorations est de combiner l'approfondissement itératif avec des tables de réfutations[2] : lors de la recherche à la profondeur d, pour chaque mouvement de la racine, la meilleure variante de ce mouvement (variante qui mène au score *negamax*, même si celui-ci n'est qu'une borne) est stockée dans une table. La variante stockée pour le meilleur mouvement issu de la racine à la profondeur d est appelée la *variante principale*. Les autres variantes montrent les suites de mouvements qui permettent à l'adversaire de « réfuter » les autres mouvements, c'est à dire rendre leur valeur negamax inférieure à la valeur negamax du meilleur mouvement. Lors de la recherche à la profondeur $d+1$, pour chaque mouvement, la variante correspondant à ce mouvement stockée dans la table sera essayée en premier.

Une autre amélioration de l'approfondissement itératif est de faire une supposition sur la valeur *negamax* de la racine après une recherche avec une profondeur d en fonction des valeurs *negamax* obtenues pour les profondeurs 1, 2, ..., $d-1$. Cette approximation prenant la forme d'un intervalle $[\alpha_0, \beta_0]$ dans lequel est supposée se trouver la valeur de la racine. L'heuristique consiste alors à limiter l'intervalle de recherche de l'algorithme *negamax* à cet intervalle plutôt que de prendre l'intervalle complet $[-\infty, +\infty]$. Trois cas peuvent alors se produire selon la valeur v de la recherche avec l'intervalle $[\alpha_0, \beta_0]$:

1° $\alpha_0 < v < \beta_0$ alors, l'approximation est correcte, et la recherche prend moins de temps qu'avec la fenêtre $[-\infty, +\infty]$;

2° $v \leq \alpha_0$, alors pour connaître la valeur *negamax* de la racine, il faut effectuer une recherche à la profondeur d avec une fenêtre $[-\infty, v]$;

3° $v \leq \beta_0$, alors pour connaître la valeur *negamax* de la racine, il faut effectuer une recherche à la profondeur d avec une fenêtre $[v, +\infty]$.

3.1.2.3 La table de transposition

La table de transposition est une façon plus générale de traiter les réfutations, c'est à dire de trouver les coups susceptibles d'être les meilleurs dans une position donnée. Dans sa forme la plus simple, elle consiste en une table des résultats des positions explorées pendant la recherche, une même position pouvant être obtenue par plusieurs chemins différents, les deux chemins sont appelés des transpositions.

Après la fin de l'exploration d'une position, les paramètres suivants sont stockés dans la table[5] :

1° Le meilleur mouvement trouvé pour cette position[6];

2° la valeur negamax calculée pour cette position;

3° un drapeau booléen indiquant si la précédente valeur est une borne sur la valeur negamax (s'il y a eu une coupure dans l'arbre, la borne est inférieure à la valeur

5. Comme la taille de la table est finie, il existe plusieurs stratégies disponibles pour choisir si oui ou non, une position doit remplacer une autre position dans la table. Notre préférence reste de garder les positions les plus proches possible.

6. On dispose d'un bon mouvement quand ce mouvement est consécutif à une valeur exacte de la position (i.e. la valeur est entre α ou β) ou quand il s'agit d'une borne inférieure (i.e. la valeur est supérieure à β), dans les autres cas, on gardera le mouvement stocké dans la table s'il existait déjà une entrée pour la position.

negamax de la position si la valeur est supérieure ou égale à β, sinon il s'agit d'une borne supérieure inférieure ou égale à α) ou une valeur exacte;

4° la profondeur du sous-arbre dont la position est racine.

Alors, au début de l'examen d'une position P, on recherche la position P dans la table de transposition. Si cette position existe dans la table et si la profondeur d'analyse est égale à la profondeur stockée, alors, on utilise la valeur x negamax de la table pour ne pas faire une recherche supplémentaire. S'il s'agit d'une valeur exacte, l'examen de la position P est fini, sinon, si la valeur est une borne inférieure de la valeur negamax de P, α est changé en $\max(\alpha, x)$, et s'il s'agit d'une borne supérieure, la valeur de β est changée en $\min(\beta, x)$. Si la profondeur stockée dans la table est supérieure à la profondeur de recherche, on utilise l'hypothèse que la recherche n'est pas pathologique en faisant comme si la profondeur de la table était égale à la profondeur d'analyse.

Si, de toutes façons, la position P doit être examinée, on prendra comme fils de gauche la position issue du mouvement stocké dans la table.

Pour pouvoir accéder rapidement à ces informations quand la recherche se retrouve dans la même position (par transposition des mouvements ou par le fait de l'approfondissement itératif), on utilise une clé de hachage[87], dont on précisera par la suite le mode de calcul. La clé de hachage est aussi contenue dans l'entrée correspondante pour authentifier les résultats stockés.

Nous allons détailler le calcul de la clé de hachage car la littérature sur ce sujet reste vague ou erronée[184, 51].

Calcul de la clé de hachage La méthode moderne de calcul des clés de hachage vient de celle proposée par Zobrist[193] pour le jeu de Go. Dans le cas du jeu d'échecs, on représente l'échiquier à l'aide d'un vecteur de bits $c = (c_1, c_2, ..., c_{749})$[184].

La longueur du vecteur, 749, provient des considérations suivantes : les cases de l'échiquier sont numérotées de 0 à 63, les différentes pièces de 0 à 11 (par exemple, pion blanc = 0, cavalier blanc = 1, ... pion noir = 6, ..., roi noir = 11). Ainsi, on peut définir la valeur des coordonnées c_l du vecteur c pour $1 \leq l \leq 768$ par

$$c_{64i+j+1} = 1 \Leftrightarrow \text{La pièce } i \text{ est sur la case } j$$

En fait en considérant le fait que les pions ne peuvent pas se trouver sur les lignes 1 et 8, on peut ramener le codage à $i \leq 736$. On utilise les bits $c_{737}, c_{738}, c_{739}, c_{740}$ pour les droits aux roques petits et grands des blancs et des noirs, et 8 bits supplémentaires pour la case éventuelle de prise en passant (codage de la colonne si le dernier mouvement est un mouvement de pion permettant une prise en passant). Le bit c_{749} étant utilisé pour stocker la couleur du joueur au trait. Ce codage présente l'avantage de réduire la distance de Hamming entre les vecteurs représentant deux positions séparées par un mouvement simple (mouvement qui ne met en cause qu'une seule pièce).

La clé de hachage est alors déterminée par[7] :

$$H = \bigoplus_{i=0}^{i=749} c_i H_i$$

où les H_i sont des vecteurs à coefficients binaires de dimension P, P étant le nombre de bits de la clé de hachage.

Dans la théorie des codes correcteurs d'erreurs, les H_i peuvent être vus comme les colonnes d'une matrice de parité définissant un code linéaire $\mathcal{C}$. H sera alors, le syndrome du message c[98]. Un message m valide de $\mathcal{C}$ est tel que $Hm^{tr} = 0$. Deux positions P et P' auront la même clé de hachage si et seulement si $Hc^{tr} = Hc'^{tr}$ où c et c' sont les représentations des positions P et P', c'est à dire $H(c - c')^{tr} = 0$, donc $c - c'$ est un message valide du code linéaire défini par la matrice de parité H. Donc deux positions auront même clé de hachage si et seulement si elles sont identiques ou si la différence de leur représentation est un code valide. Le code linéaire $\mathcal{C}$ étant défini par sa longueur, ici 749, sa dimension, et la distance minimale entre deux codes d. Ce qui peut être reformulé en : deux positions auront mêmes clés de hachage si et seulement si elles sont identiques ou si leurs représentations sous forme de c diffèrent en au moins d positions.

TAB. 3.1 - *Nombre de bits protégés en fonction de la dimension des H_i[184]*

Nombre de bits protégés	Dimension des H_i		
	Borne théorique inférieure	BCH	GV
2	10	11	10
4	18	21	26
6	26	31	41
8	34	41	55
10	41	51	67
12	48	61	80
14	55	71	91
16	61	81	103
18	67	91	114

Le tableau 3.1 donne, en fonction de la dimension des codes, le nombre de bits protégés pour des codes linéaires Bose-Chaudhuri-Hocquenghem(BCH), (la longueur du codes étant 1023, ce calcul prévaut pour toute longueur de code comprise entre 512 et 1023, bornes incluses) et les codes Gilbert-Varshamov (GV) où les H_i sont choisis de manière aléatoire comme indiqué par Zobrist[193]). Le choix raisonnable est donc de prendre un code BCH de dimension supérieure à 41, pour nous assurer d'une bonne répartition des clés de hachage.

7. $\bigoplus$ représentant un *ou exclusif* bit à bit, c'est à dire l'addition des vecteurs H_i à coefficient dans $\mathbb{Z}/2\mathbb{Z}$

3.1.2.4 La table historique

Schaeffer[146] définit les tables historiques qui vont servir à trier les mouvements qui ne sont ni mouvements meurtriers, ni mouvements de réfutations ni mouvements de table de transposition.

A chaque niveau, lorsqu'un mouvement est le meilleur ou lorsqu'il provoque un élagage dans une position P, on incrémente l'entrée dans la table historique indexée par la couleur du joueur au trait dans la position P_i et par le mouvement lui-même (aux échecs, on prendra donc une double indexation : la case de départ $\times$ la case d'arrivée). L'incrément étant fonction de la profondeur de la position P_i, il est plus important quand la position P_i est proche de la racine. Schaeffer donne comme exemple d'incrément 2^p où p est la profondeur du sous-arbre ayant pour racine P_i.

Quand une nouvelle position est examinée, les mouvements sont ordonnés suivant le score qu'ils obtiennent dans la table historique.

3.1.2.5 Le contre-mouvement

De manière indépendante, Uiterwijk[178] et Hartmann[71, 73] ont défini l'heuristique du contre-mouvement (appelée l'heuristique des tableaux papillons par Hartmann).

Cette heuristique consiste à stocker le meilleur mouvement ou le mouvement $\mathcal{C}$ qui a causé un élagage dans la position P dans une table indexée par le mouvement $\mathcal{D}$ précédant la position P. Le mouvement ainsi stocké est un mouvement de « réfutation » du mouvement précédent. Ainsi, si l'algorithme examine une position Q obtenue immédiatement après le mouvement $\mathcal{D}$, le mouvement $\mathcal{C}$ sera examiné parmi les mouvements de tête.

Une heuristique similaire : « la réponse meurtrière » avait déjà été définie dans le programme d'Othello BILL [93].

3.1.3 Comparaison expérimentale des heuristiques ayant trait à l'ordre des mouvements

Nous avons comparé les différentes heuristiques concernant les ordres des mouvements sur un programme d'échecs. La base de la comparaison est l'ensemble du temps pris pour résoudre à une profondeur donnée l'ensemble des 24 positions dites de Bratko et Kopec[35].

La version initiale correspond à une version où les coups sont ordonnés selon l'ordre *MinorToMajor*, c'est-à-dire les captures des pièces les plus importantes par les pièces les plus faibles en premier.

Quand la table de transposition est active, le mouvement qu'elle propose est toujours placé en tête. Le mouvement meurtrier, lorsqu'il est actif, sera placé avant les captures négatives ou nulle (la valeur de la pièce capturée est inférieure ou égale à la valeur de la pièce capturant, dame = 9, tour = 5, cavalier = fou = 3, pion = 1).

Le tableau 3.2 nous montre les résultats de différentes combinaisons des heuristiques de tri de coups par rapport à la version initiale. Ainsi, nous remarquons :

- que les heuristiques de tri des coups sont plus efficaces quand l'arbre de recherche est le plus profond;
- que le surcoût engendré par l'approfondissement itératif est effacé dès lors que celui-ci est couplé avec des mouvements meurtriers ou la table de transposition;
- que de toutes les heuristiques, la table de transposition à elle seule permet d'obtenir de bons résultats;
- que les effets ne sont pas purement additifs : en effet, appliquées simultanément certaines heuristiques se contrarient;
- que le meilleur résultat est obtenu pour l'application simultanée de l'approfondissement itératif, de l'heuristique du mouvement meurtrier, de l'usage de la table de transposition, de la table historique et du contre mouvement.

A la profondeur de 7 demi-coups, un bon choix des heuristiques de tri des mouvements permet d'obtenir un processus de recherche 5 fois plus rapide.

3.2 Réduction de l'effet horizon

Un des problèmes principaux rencontrés par les recherches de type *negamax* est l'apparition de « l'effet horizon »[18]. Après l'avoir défini, nous montrerons les heuristiques qui permettent de le combattre.

3.2.1 Définitions des types d'effet horizon

Théoriquement, les arbres de recherche utilisés sont de profondeur uniforme, la fonction d'évaluation étant appelée à une profondeur fixée par rapport à la racine de l'arbre. Ce schéma conduit rapidement à ce qui a été nommé, par Berliner[18], « l'effet horizon » : les problèmes non identifiés sont repoussés au delà de la profondeur d'évaluation. Ceci est souvent causé par le fait que la profondeur des feuilles est fixée, mais aussi par l'interaction entre la fonction d'évaluation et l'algorithme de recherche.

Berliner[18], puis Hartman[73] classent les différents effets horizons en 4 catégories :

Effet horizon Normal : dans le diagramme 3.2[73], les noirs au trait font nulle. Le seul mouvement qui annule dans cette position est 1 ... ♔e4, la ligne pouvant être par exemple 1 ... ♔e4 2. a5 ♔d5 3. a6 ♔c6 4. a7 ♔b7 5. a8♕+ ♔×a8 . Un programme

TAB. 3.2 - *Comparaisons des heuristiques ayant trait à l'ordre des coups dans un programme d'échecs.*

Les chiffres les moins élevés dénotent les versions les plus rapides

Version	Profondeur 4	Profondeur 5	Profondeur 6	Profondeur 7
Initiale	100	100	100	100
HI	104	102	80	94
MM	98	84	79	71
TT	83	78	64	68
MM HI	102	84	88	74
MM TT	83	70	60	52
HI TT	64	63	56	74
MM HI TT	69	64	57	57
AI	101	127	105	126
AI HI	107	120	83	127
AI CM	90	100	84	86
AI MM	72	71	54	57
AI MM CM	68	72	59	59
AI MM DC	68	73	58	62
AI CM HI	91	102	72	78
AI MM HI	67	81	51	48
AI CM TT	52	45	26	24
AI MM TT	50	47	27	23
AI MM TT HI	41	41	25	21
AI MM TT CM	51	46	26	22
AI MM TT DC	58	52	30	26
AI TT HI CM	45	40	25	22
AI MM HI CM	68	78	57	58
AI MM TT DC HI	42	42	30	26
AI MM TT HI CM	**42**	**41**	**23**	**20**
AI MM TT HI CM DC	43	42	26	21

(AI = Approfondissement itératif, MM = Mouvement meurtrier, TT = Table de transposition, HI = table historique, CM = Contre Mouvement, DC = Capture de la pièce qui vient de bouger)

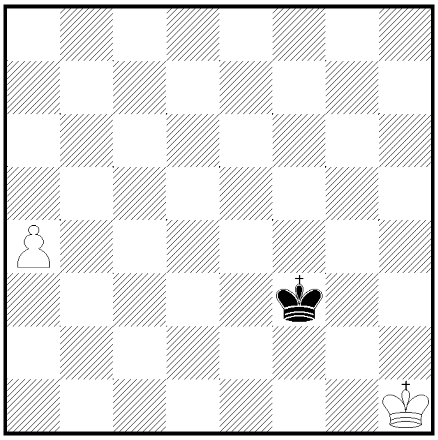


Diagramme 3.2: *exemple de position où se produit un effet horizon normal*

d'échecs qui appliquerait la fonction d'évaluation uniquement sur des positions situées à une profondeur fixée de la racine nécessiterait ici une profondeur de 9 demi-coups pour voir cette ligne. De même la variante 1 ... ♔f4 2. a5 ♔e5 3. a6 ♔d6 4. a7 ♔c7. 5. a8♕ ♔b6, nécessite une profondeur de 9 demi-coups pour montrer que le mouvement 1 ... ♔f4 est perdant. Aux profondeurs inférieures, le programme ne sera pas capable de distinguer ces deux lignes; à ces profondeurs, le programme souffrira alors d'un effet horizon normal, caractérisé par le fait que la meilleure ligne n'est pas reconnue comme telle car elle ne se distingue pas des autres lignes.

Effet horizon négatif : dans le diagramme 3.3[73], les noirs ont le trait et doivent parer l'attaque des blancs sur la huitième rangée.

Considérons les lignes, toutes deux de profondeur de 7 demi-coups, 1 ... ♕g8 2. ♖e8 h6 3. ♖×g8+ ♖×g8 4. ♖e8+ ♔h7 et 1 ... ♘g3+ 2. h×g3 ♕g8 3. ♖e8 h6 4. ♖×g8+ ♔×g8 . D'un point de vue purement matériel la première ligne est la meilleure, donc un programme effectuant sa recherche à la profondeur de 7 demi-coups en conclura que 1 ... ♕g8 est un meilleur mouvement que 1 ... ♘g3+ . Or, à une profondeur de 5 demi-coups, la deuxième variante aurait été préférée : c'est un cas d'effet horizon négatif, qui se caractérise par le fait que la meilleure ligne est rejetée parce qu'il semble qu'il en existe de meilleures à des profondeurs inférieures. Ici, le sacrifice du cavalier repousse de deux demi-coups, l'inexorable perte de la dame.

Effet horizon partiellement positif : dans la position du diagramme 3.4[18], les blancs sont au trait. Une recherche à la profondeur fixe de 3 demi-coups montre que 1. ♘e5 n'est pas un bon mouvement, la ligne 1. ♘e5 d6 2. ♘ef3 ayant pour résultat de développer pour les noirs le pion d7. Mais les blancs, voulant occuper la case e5 avec leur cavalier, peuvent à une profondeur de 5 demi-coups choisir la ligne 1. a×b5 a×b5 2. ♖×a8 ♗×a8 3. ♘e5 avec l'illusion que leur cavalier restera en e5 : les blancs ont inséré des mouvements et ne voient pas qu'à l'instar de la première variante, le cavalier peut

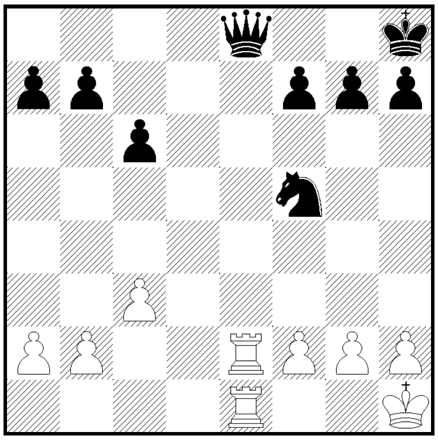


Diagramme 3.3: *exemple de position où se produit un effet horizon négatif*

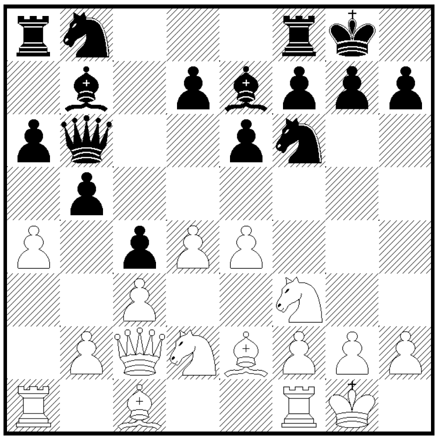


Diagramme 3.4: *exemple de position où se produit un effet horizon partiellement positif*

très bien être chassé de la case e5 par le pion d7. C'est un effet horizon semi-positif, caractérisé par le fait que la réfutation d'un avantage illusoire est repoussée au delà de l'horizon par l'insertion de mouvements dans la variante.

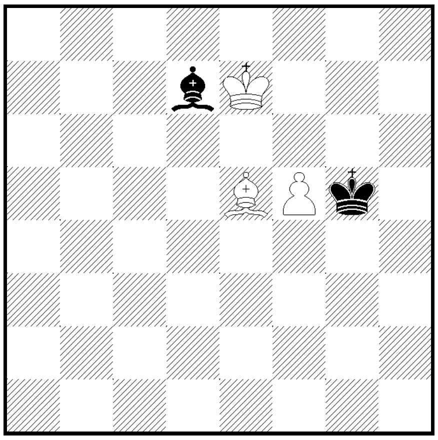


Diagramme 3.5: *exemple de position où se produit un effet horizon positif*

Effet horizon totalement positif : dans la position du diagramme 3.5, les blancs au trait gagnent par 1. f6 ♔f5 2. ♔×d7 ♔×e5 3. f7 ♔d4 4. f8♕ . Un programme analysant cette ligne jusqu'à la profondeur fixe de 4 demi-coups la considérera, d'un point de vue matériel, plus mauvaise que 1. ♔×d7 ♔×f5 2. ♔d6 ♔e4 = , et préférera donc le mouvement ♔×d7. Il s'agit d'un effet horizon positif : l'avantage obtenu est réel, mais il existe une meilleure ligne de jeu.

Pour contrecarrer ces effets horizons, et pour permettre une évaluation plus correcte des positions, nous allons essayer de donner une évaluation stable des positions, c'est à dire une évaluation qui ne change pas trop si on augmente la recherche d'un niveau. En effet, les fonctions d'évaluation utilisées pour les programmes de jeux peuvent changer brusquement après un mouvement ou une séquence de mouvements. Par exemple, dans un programme d'échecs, la différence de l'évaluation de deux positions avant et après une capture de dame va être grande. L'évaluation sera donc plus fiable si elle est appliquée dans une position consistante, c'est à dire une position où il n'existe pas de mouvements profitables au joueur au trait.

3.2.2 La recherche de la consistance

Beal[15, 16, 63] a défini « la recherche de la consistance ». Une position est dite « consistante », si son évaluation est égale à sa valeur *negamax* calculée sur une profondeur de un demi-coup. Cette recherche a été construite dans le but de réduire les erreurs d'évaluation et de rendre la recherche *negamax* moins pathologique[15].

```
funct consistance(Position, α, β)
    ⌈pos_eval ← g(Position);
     C =← PremierCoup(Position);

     Meilleur ← −∞;
     while C ≠ null do
         valeur ← val[C] ← −g(Position • C);
         if (valeur > Meilleur)
           then Meilleur ← valeur;
         fi
         C ← ProchainCoup(Position) od;

     if Meilleur = pos_eval then exit Meilleur; fi
     tri(C, val[C]);                          Trie les coups selon le tableau val
     C =← PremierCoup(Position);
     Meilleur ← −∞;
     while C ≠ NULL do
         valeur ← −consistance(Position • C, −β, −α);
         if valeur > Meilleur
           then Meilleur ← valeur;
                if Meilleur > α
                  then α ← Meilleur;
                       if α ≥ β then exit Meilleur; fi
                fi
         fi
         if Meilleur = pos_eval then exit Meilleur; fi
         C ← ProchainCoup(Position) od;

     Meilleur⌋.
```

FIG. 3.2 - *Recherche de la consistance[15]*

La figure 3.2 montre la définition générale de la recherche de la consistance : dans un premier passage, on calcule la valeur *Negamax* à un demi-coup de profondeur de la position, que l'on compare avec la valeur de la fonction d'évaluation de la position. Si ces deux valeurs sont identiques, la position est consistante. Sinon, on applique récursivement la recherche de la consistance, qui s'arrête dès que l'on a trouvé un fils tel que la valeur retournée par la recherche de la consistance est égale à la valeur de l'évaluation, la position sera alors dite semi-consistante, ou dès que l'évaluation est supérieure à β.

L'enveloppe de cette recherche n'étant pas forcément bornée, nous pourrons, dans les positions très profondes, définir une pseudo-consistance en s'arrêtant non plus sur l'égalité stricte, mais plutôt sur des valeurs proches.

Une bonne façon de voir les recherches de consistance est de considérer que l'on remplace la fonction d'évaluation par une fonction plus complexe qui encapsule une recherche. On peut donc supposer que l'on dispose d'une fonction d'évaluation plus stable, mais aussi plus coûteuse en temps de calcul.

3.2.3 La recherche de la consistance aux échecs : la recherche de la tranquillité

Dans le cas du jeu d'échecs, les mouvements qui modifient la valeur *negamax* de la position de la manière la plus conséquente sont les mouvements tactiques tels que les échecs, les captures et les promotions. Ce sont donc les mouvements susceptibles de conduire à la non stabilité de la position, et la recherche de la consistance se traduira alors par la classique recherche de la tranquillité si l'on considère une évaluation purement matérielle.

Le concept de position tranquille est définit par Shannon[160] et par Turing[175][8] indépendamment. Une position sera dite tranquille s'il n'existe pas de capture/promotion susceptible de faire augmenter la fonction d'évaluation, et si le roi au trait n'est pas en échec.

La recherche de la tranquillité donnée dans la figure 3.3 est donc premièrement une instance de la recherche de la consistance en considérant que la fonction d'évaluation est uniquement matérielle.

3.2.4 Extensions statiques

Un autre moyen pour diminuer les effets horizons est d'identifier les mouvements qui repoussent les problèmes au delà de l'horizon de la recherche. Une fois ces mouvements identifiés, la profondeur de la recherche est modifiée en ne comptant pas ces mouvements comme des demi-coups entiers. Ce qui permet d'aller plus loin dans les variantes critiques.

Le processus de ne pas compter des mouvements comme des demi-coups entiers est appelé faire une **extension**. La profondeur des feuilles où sera appliquée la recherche de la consistance

8. Turing les nomme les positions mortes.

```
funct tranquillite(Position, α, β) ≡
    ⌈if echec(Position) then exit fab(Position, α, β); fi
     Best ← g(Position);
     if Best > α then α ← Best; fi
     if α ≥ β then exit Best; fi
     M ← ProchaineCaptureouPromotion(Position);
     while M ≠ null do
         value ← −tranquillite(M, −β, −α);
         if value > Best then Best ← value; fi
         if Best > α then α ← Best; fi
         if α ≥ β then exit Best; fi
         M ← ProchaineCaptureouPromotion(Position) od;
     Best⌋.
```

FIG. 3.3 - *Recherche de la tranquillité*

(ou une recherche de la tranquillité, ou directement une fonction d'évaluation selon les cas) ne sera plus uniforme mais dépendra des mouvements des variantes menant aux feuilles.

L'approche générale consiste alors à définir le coût des mouvements (en demi-coups) en fonction de leurs types, définis heuristiquement en fonction du domaine d'application (donc du jeu). Il faut aussi se prémunir d'une explosion combinatoire et limiter le nombre de demi-coups maximum d'une branche de l'arbre, ou même définir le coût (en demi-coups) en fonction de la profondeur réelle du mouvement depuis la racine.

Dans le cas du jeu d'échecs, on considère par exemple qu'un mouvement faisant échec au roi adverse ne compte pas de demi-coup.

Voici un exemple complet de définition d'extensions statiques telles que nous les avons heuristiquement définies dans le programme d'échecs ECUME :

1° Si le dernier mouvement est une prise, on fait une extension de $\frac{1}{16}$ demi–coup si la pièce prise est un pion, $\frac{3}{16}$ demi–coup si c'est un fou ou un cavalier, $\frac{1}{4}$ demi–coup si c'est une tour, $\frac{3}{8}$ demi–coup si c'est une dame;

2° Si l'avant dernier et le dernier mouvements sont des prises et si la différence de matériel sur l'échiquier est la même, à un demi–pion près que la différence de matériel à la racine, l'extension de recherche est d'un demi–coup;

3° Si le dernier mouvement est un mouvement de pion l'amenant sur sa 7^{e} rangée, l'extension de recherche est d'un demi–coup;

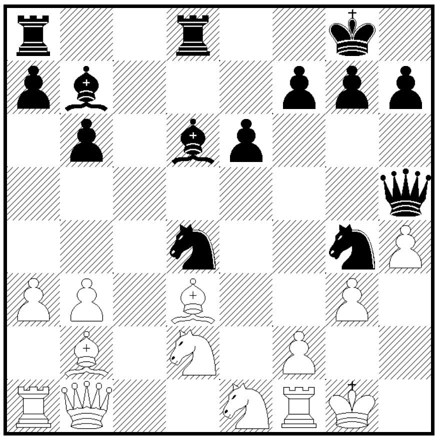
4° Si le dernier mouvement est un mouvement de pion l'amenant sur sa 6^{e} rangée, l'extension de recherche est de $\frac{max(17, 1700 - \frac{\text{matériel} - 1000}{300})}{16}$ où le matériel représente la somme du matériel sans les pions (♘= ♗= 300, ♖= 500, ♕= 900);

5º Si le roi au trait est en échec double, l'extension de recherche est de $1,5$ demi-coup;

6º Si le roi au trait subit un échec à la découverte, l'extension de recherche est de $1,25$ demi-coup;

7º Si le roi au trait est en échec provoqué par la pièce qui vient de bouger, l'extension de recherche est de 1 demi-coup;

8º Si le dernier mouvement pare un échec en capturant la pièce qui faisait échec, l'extension de recherche est de $-0,5$ demi-coup.



Les noirs jouent et gagnent.

Diagramme 3.6: *105e position du livre « Win at Chess »[141]*

Dans la position du Diagramme 3.6, les noirs jouent et font mat en 6 coups. Il faut une recherche nominale de 7 demi-coups, 16 secondes[9] et 210560 nœuds examinés pour que le programme ECUME annonce l'existence d'un mat forcé. La ligne critique est 1 . . . ♕×h4 ($\frac{+1}{16}$) 2. ♘ef3 ♘×f3+ ($\frac{3}{16}+1$) 3. ♘×f3 ($\frac{3}{16}$–0, 5) ♗×f3 ($\frac{3}{16}$) 4. ♗×h7+ ($\frac{1}{16}+1$) ♔f8 5. ♗×g7+ ($\frac{1}{16}+1$) ♔e8 6. . . . ♕h2# (ce coup est généré par la recherche de la tranquillité qui inclut les échecs pour le premier niveau) , nous avons mis entre parenthèses les valeurs en demi-coups des extensions de recherche.

3.2.5 Extensions entropiques

Les extensions entropiques, apparues dans la thèse de David[45], constituent une méthode originale de définir les extensions.

Il s'agit de la méthode duale des extensions statiques; plutôt que de considérer les mouvements, on considère les changements intervenus entre la position qui précède le mouvement

9. L'ordinateur utilisé pour les tests est un PC 486DX2 66Mhz.

et la position qui le suit. Nous définissons pour ce faire une grandeur S qui définit l'entropie d'une position. La variation de cette grandeur va nous désigner les mouvements à ne pas compter comme demi-coups.

S peut être définie comme étant la différence de l'instabilité du joueur au trait et de celle de l'adversaire, ce que nous notons $S(P) = S_{trait}(P) = I_{trait}(P) - I_{adv}(P)$. L'instabilité étant la somme de critères dépendants de la position. Pour simplifier notre propos, nous allons considérer que les blancs examinent la variation de S_{blanc} provoquée par le mouvement C_i depuis la position P (où les blancs ont donc le trait). On définit alors $\Delta(P, C_i) = S_{trait}(P) - S_{trait}$ en $_P(P_i)$. Nous sommes dans l'un des 3 cas suivants :

1° $\Delta(P, C_i) > 0$ où le mouvement C_i sera dénommé critique. Ceci peut être provoqué par une augmentation de l'instabilité des noirs, c'est à dire une attaque ou par une diminution de l'instabilité des blancs, c'est à dire un mouvement de défense;

2° $\Delta(P, C_i) = 0$ où le mouvement C_i sera dénommé neutre;

3° $\Delta(P, C_i) < 0$ où le mouvement C_i sera dénommé négatif.

On fera une extension pour la suite d'un mouvement C_i critique d'attaque et d'un mouvement C_{i+1} critique (d'attaque ou de défense).[45, 46]

Nous avons implanté, pour notre part, dans le programme JOKER, ce schéma d'extension en utilisant les définitions données par David dans son programme Reza Tarazed[46].

L'instabilité pour un des joueurs est définie comme la somme :

1° du nombre d'échecs subis;

2° du nombre de ses pièces en prise (une dame valant deux pièces);

3° du nombre de clouages sur son roi;

4° du nombre de découvertes possibles (attaque masquée par une pièce ennemie);

5° du double du nombre de pions adverses en 7^{e} rangée;

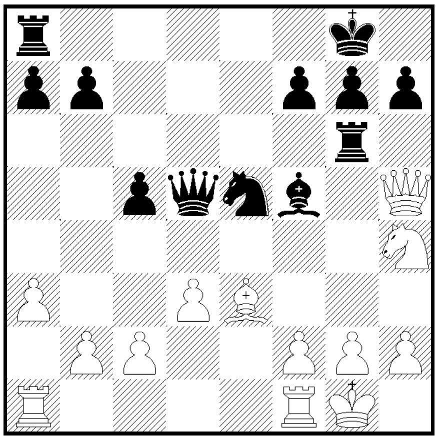
6° du nombre de pions adverses passés (doublé si le camp n'a plus de pièce);

7° de l'instabilité du roi : le roi est considéré instable s'il existe une case que l'adversaire attaque en surnombre, où il peut installer une tour ou une dame faisant ainsi échec au roi. Cette instabilité compte alors pour 2 points d'instabilité auxquels on ajoute 1 point dans le cas où le roi n'a pas de cases de fuites.

De plus, dès que le roi est en échec, la recherche est systématiquement étendue d'un demi-coup.

Dans la position du diagramme 3.7, JOKER à la suite d'une recherche nominale de 5 demi-coups, en ayant examiné 157864 nœuds, trouve en 25 secondes[10] le gain suivant : 1. (2 − 10 = −8) ... ♖×g2+ (13 + 1 −11 = 3) 2. ♘×g2 (4 − 6 − 5 = −5) ♗g4 (13 − 6 = 7) 3. ♘f4 (10 −

10. Sur un PC 486DX50 Mhz.



Les noirs jouent et gagnent.

Diagramme 3.7: *Combinaison 5 de la grille de tests de Pierre Nolot[124]*

19 = −9) ♕d6 (10 − 10 = 0) 4. ♗×c5 (14 − 19 = −5) ♕×c5 (15 − 10 = 5) , entre parenthèses figurent les instabilités des blancs et des noirs[11].

3.2.6 Extensions singulières

Le fait d'avoir remarqué que souvent les programmes entrent dans des lignes forcées sans en déterminer la valeur résultante, a poussé l'équipe de Chiptest[11, 10] à définir les extensions singulières : un mouvement est singulier si sa valeur est supérieure d'une certaine marge S à la valeur de ses frères, et dans ce cas ce mouvement n'est pas décompté dans la recherche.

Définition 3.2.1 *Un mouvement C_i dans une position P est S* singulier *si la valeur* Negamax *de la position P_i est inférieure à celle de $P_j + S$ pour tous les i différents de j, S étant une valeur positive fixée.*

En pratique, il ne sera pas possible de faire des tests de singularité pour tous les coups de la recherche, car cela produirait un surcoût de recherche trop important. On essaye donc de se limiter aux mouvements pouvant influencer le score *negamax* de la position, c'est à dire les mouvements de la variante principale et ceux qui produisent un élagage dans la position P_0.

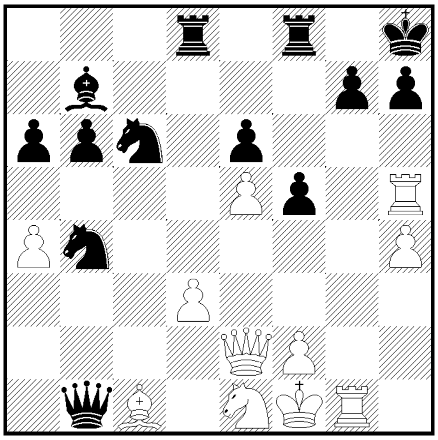
Dans le cas d'un mouvement de la variante principale (caractérisée par le fait que P et P_i sont des positions de type 1), comme on connaît la valeur *Negamax* de la position P_i, la singularité du mouvement C_i sera vérifiée en testant si les valeurs des positions P_j pour $j \neq i$

11. A titre de comparaison, le programme ECUME trouve la même combinaison en 58" à la suite d'une recherche nominale de 7 demi-coups et 550000 nœuds examinés.

sont supérieures à celle de $P_i - S$. Dans ce cas, le mouvement C_i sera le meilleur dans la position P et singulier, on ne comptera donc pas ce mouvement comme un demi-coup.

L'autre cas de test de singularité est dans les positions P_i telles que $-negamax(P_i) > \beta$. On teste alors si $(v - S_h) > -negamax_r(P_j)$ pour $j \neq i$, où $negamax_r(Q)$ est la valeur negamax de Q calculée sur une profondeur de r demi-coups moindres que $negamax(Q)$, S_h est une valeur positive supérieure à S et v est la valeur *negamax* de la position.

Pour plus de détails dans l'implantation des extensions singulières, le lecteur est invité a se référer à [10].



Les blancs jouent et font mat en 18 coups.

Diagramme 3.8: *Exemple d'utilisation des extensions singulières : la position 213 du livre « Win at Chess »[141]*

Dans la figure 3.8, le programme CHIPTEST muni des extensions singulières trouve un mat en 18 coups (35 demi-coups) sur une recherche nominale de 8 demi-coups : 1. ♖×h7+ ♔×h7 2. ♕h5+ ♔g8 3. ♖g7+ ♔g7 4. ♗h6+ ♔h7 5. ♗g5+ ♔g7 6. ♕h6+ ♔f7 7. ♕f6+ ♔g8 8. ♕g6+ ♔h8 9. ♗f6+ ♖f6 10. ef6 ♕e1+ 11. ♔e1 ♘c2+ 12. ♔f1 ♘e3+ 13. fe3 ♖d7 14. ♕e8+ ♔h7 15. ♕d7+ ♘e7 16. ♕e7+ ♔h6 17. ♕g7+ ♔h5 18. ♕g5# .

L'implantation la plus simple consiste à considérer qu'une position est singulière dès lors qu'un seul mouvement est légal dans cette position. Cette réduction permet déjà d'étendre les positions où le roi est en échec et n'a qu'un seul mouvement possible comme le feraient les extensions singulières. Cette réduction des extensions singulières ne trouve pas le mat précédent.

3.3 Coupures sélectives

Pour accélérer le temps de la recherche, on peut décider d'avoir recours à la sélectivité qui va choisir les mouvements à explorer et en ignorer certains, les « élagués a priori ».

De nombreuses heuristiques ont été ainsi définies[164] : *l'algorithme Gamma*[118], *le rasoir*[29].

Ces coupures peuvent soit être dépendantes du domaine, soit utiliser les connaissances disponibles dans l'algorithme de recherche, c'est à dire les valeurs α et β des bornes de la fenêtre et la valeur de la fonction d'évaluation ou d'une fonction plus simple à calculer dans les nœuds internes.

3.3.1 Élagage de futilité

De toutes les heuristiques d'élagage a priori définies, la plus populaire, car la moins dangereuse, est l'heuristique définie et analysée par Schaeffer[147, 191], *l'élagage de futilité* : un mouvement C_i issu d'une position P sera considéré comme futile, donc pas examiné, s'il est à un demi-coup de la frontière, que l'évaluation de la position P est inférieure à $\alpha - \delta$, δ étant une constante positive, et si le mouvement C_i n'est pas un coup tactique tel qu'une capture, une promotion ou un échec.

Si P est une position située à un demi-coup de l'évaluation (ou de la recherche de la consistance), si $g(P) < \alpha - \delta$ [12], la définition de la valeur de P par élagage de futilité est :

$$\text{futil}(P) = \min(\beta, \max(v_0, \max_{C \in \text{tactique}} -g(P \bullet C)))$$

où v_0 peut être soit α, soit $\alpha - \delta$ soit $g(P) + \delta$ selon les stratégies d'implantation. (La valeur α étant la moins risquée).

3.3.2 Coupures stochastiques

Si la fonction d'évaluation est suffisamment stable, les *coupures stochastiques* peuvent être utilisées : dans le programme d'Othello LOGISTELLO, Buro[38] a calculé pour un ensemble de positions l'écart type σ de la variable $\mid v_8(P) - v_4(P) \mid$ où v_8 (resp. v_4) représente la valeur negamax de la position pour une recherche de profondeur nominale 8 (resp. 4). Ainsi, lorsqu'il est nécessaire d'examiner une position P à une profondeur 8 avec une fenêtre $[\alpha, \beta]$, à l'aide de fenêtre minimale, on vérifiera si $\alpha - \sigma.p < v_4(P) < \beta + \sigma.p$ où p est une constante fixée (par exemple si on considère que la variable $\mid v_8(P) - v_4(P) \mid$ suit une loi normale de paramètre σ, une valeur de p de 1,80 correspond à 96,4% de l'échantillon observé), et dans le cas contraire, on effectuera un élagage indiquant que la valeur $v_8(P)$ est très probablement inférieure à α ou à β.

12. La fonction $g(P)$ désigne comme au chapitre 2, la fonction d'évaluation.

3.3.3 Heuristique du mouvement nul

L'heuristique du mouvement nul[17, 67, 48] repose sur la supposition que les mouvements joués par un joueur améliorent sa position, et donc que le fait de passer son tour lui serait préjudiciable. Ce qui est une supposition qui dépend du domaine traité. Ainsi à Othello, cette heuristique n'est pas applicable car, généralement, le fait de jouer un coup augmente la mobilité de l'adversaire donc détériore la position.

Aux échecs, cette supposition est vérifiée dans le cas normal, mais il existe des positions, surtout situées dans la fin de partie[13], où le fait de jouer détériore la position. Ces positions dans lesquelles la supposition n'est pas vérifiée, sont appelées des positions de *Zugzwang*.

Un *mouvement nul* est caractérisé par le fait qu'un des joueurs « passe son tour », c'est à dire donne le trait à son adversaire sans effectuer de mouvement.

Quand commence l'exploration d'une nouvelle position P avec la fenêtre $[\alpha, \beta]$, le mouvement nul C_0 est considéré avant tous les autres mouvements possibles comme s'il s'agissait d'un mouvement normal. Soit alors $v_{nul} = -fab(P \bullet C_0, -\beta - \alpha)$[14], l'heuristique du mouvement nul consiste à considérer que la valeur de la position P, c'est à dire $fab(P, \alpha, \beta)$, est supérieure à v_{nul}. Donc si $v_{nul} \geq \beta$ les autres mouvements de la position P ne seront pas explorés.

Pour élaguer les mouvements de la position P, la fenêtre n'est pas nécessairement entière. En effet, pour tester si v_{nul} est supérieur à β, il suffit d'effectuer une recherche avec une fenêtre minimale $[\beta, \beta + 1]$.

Dans le cas du jeu d'échecs on appliquera l'heuristique du mouvement nul si l'on n'est pas dans un des cas suivants:

1° Le mouvement précédent n'est pas un mouvement nul;

2° Le joueur au trait dans la position P est en échec. En effet le fait de passer provoquerait la prise du roi, donc un gain de l'adversaire et une valeur de v_{nul} inférieure à α;

3° La position P est trop près de l'horizon où sera appliquée la recherche de consistance (ou la fonction d'évaluation);

4° La position P est telle que le joueur au trait n'est pas en zugzwang;[15]

5° Si $\beta = \infty$, c'est à dire la position P est dans la variante principale.

On améliore l'entropie du mouvement nul en considérant par exemple que le mouvement nul coûte plus qu'un demi-coup, Donninger[48] propose 2 demi-coups.

La différence entre la valeur de la position $v = fab(P, \alpha, \beta)$ et la valeur de v_{nul} représente

13. On nomme fin de partie, les positions où le matériel est réduit.

14. fab désignant, à l'instar du chapitre 2, le *fail soft* $\alpha\beta$.

15. Il est difficile de savoir si une position P est telle que le joueur au trait est en zugzwang ou non. Comme aux échecs, cela arrive le plus souvent dans les positions de finales, l'heuristique du mouvement nul n'est pas utilisé dans de telles positions. Donninger[48] l'interdit uniquement dans les cas extrêmes où le joueur au trait n'a qu'une pièce mineure (fou ou cavalier).

l'entropie de la position. Nous avons vu que v_{nul} peut être considéré comme une valeur pessimiste de la position P. C'est ainsi que sont définies les bornes pessimistes nécessaires aux algorithmes B*[128, 25].

Cette valeur pessimiste mesure en partie les menaces que possède l'adversaire. Ainsi, l'heuristique du mouvement nul permet de définir deux extensions sélectives :

1° Dans le cas où un mouvement C_i est tel que $V = -fab(P \bullet C_i, -\beta, -\alpha) > \beta$ (le mouvement C_i provoque normalement un élagage), Anantharaman[10] propose de tester si $v_{nul} < V - \delta_0$ où δ_0 est une valeur positive et d'effectuer une extension d'un demi-coup.

2° Si $v_{nul} < g(P) - \delta$ et $g(P) > \beta$ où δ est une valeur positive alors Donninger[48] définit pour les positions proches de l'horizon une extension de recherche en repoussant l'horizon d'un demi-coup.

3.3.4 Recherche de tranquillité et élagages a priori

La recherche de la tranquillité peut être considérée comme une forme spéciale d'élagage a priori. Dans le programme BOBBY[156], Schrüfer montre comment définir une recherche de la tranquillité non plus uniquement tactique mais en rajoutant des mouvements à caractères positionnels qui ont statistiquement été « intéressants », c'est à dire qui ont été susceptibles d'augmenter la valeur *negamax* de la position, dans les précédentes recherches de tranquillité effectuées pendant la recherche.

D'autres approches consistent à essayer de faire résoudre par la recherche de la tranquillité les problèmes tactiques courants tels que les pièces en prise ou les attaques de mats[81].

Dans le programme CUMULUS, nous avons défini une recherche de la tranquillité sur 6 demi-coups : à chaque nœud de la tranquillité, le programme calcule les pièces en prises du camp au trait. S'il n'existe pas de pièce en prises ou si le camp au trait est en échec, la tranquillité est appliquée normalement de manière récursive.

Sinon, le programme génère deux valeurs qui encadrent la valeur tactique de la position, $\underline{S}$, valeur pessimiste telle que :

$$\underline{S} = g(P) - \bar{P} - 50$$

où g est la fonction d'évaluation de la position P et $\bar{P}$ est la valeur de la plus grande pièce en prise et 50 représente la moitié de la valeur d'un pion. Si cette valeur pessimiste est supérieure à β, la recherche de la tranquillité retourne cette valeur. Sinon, l'ensemble des captures possibles est généré. Chaque capture est évaluée à l'aide d'un évaluateur des résultats possibles des échanges statiques sur une case. Ainsi, est définie la valeur optimiste

$$\bar{S} = g(P) + \sup_{c \in \mathrm{Captures}} echanges(c) + 50.$$

La recherche est appliquée ensuite récursivement sur tous les mouvements tactiques, c'est à dire les captures et les promotions. Si aucun mouvement tactique M ne résout le problème lié à la pièce en prise, c'est à dire si

$$tac(P) = \sup_{M \in \text{Tactique}} -T(P \bullet M, -\beta, -\alpha) < g(P),$$

et si dans les mouvements précédents de la variante en cours le joueur au trait ne s'est pas déjà trouvé dans cette situation, les autres mouvements sont générés et la valeur $T(P)$ est alors $\min(\beta, \max(-T(P \bullet M)))$ où M parcourt l'ensemble des mouvements possibles, sinon

$$T(P) = \begin{cases} g(P) - \mathcal{P} + 50 & \text{si la plus grande pièce en prise est clouée} \\ & \text{(dans ce cas, } \mathcal{P} = \bar{P}\text{)} \\ & \text{ou s'il existe au moins deux pièces en prises} \\ & \text{(dans ce cas, } \mathcal{P} \text{ est la valeur de la deuxième} \\ & \text{plus grande pièce en prise)} \\ (2 * \max(tac(P), \underline{S}) + g(P))/3 & \text{Si l'ordinateur est au trait} \\ (\max(tac(P), \underline{S}) + g(P))/2 & \text{Sinon} \end{cases}$$

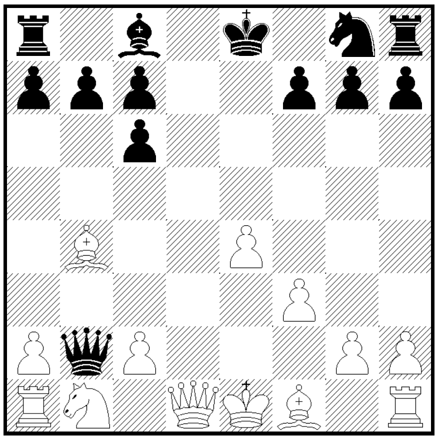


Diagramme 3.9: *Les Noirs jouent et gagnent.*

Dans la position du diagramme 3.9 les Noirs doivent éviter le mouvement 1 ... ♕×a1?, car les blancs ont dès lors l'attaque 2 ♗c3! ♕×a2 2. ♗×g7. CUMULUS 1.5C identifie le piège à partir d'une profondeur nominale de 2 demi-coups. Son analyse sur 1 ... ♕×a1?est alors 1 ... ♕×a1? 2. ♗c3 ♕×a2 3. ♗×g7 ♗e6 4. ♗×h8 < –329 en ayant analysé 408 noeuds, et il trouve la bonne variante 1 ... ♕×b4+ 2. ♕d2 ♕×d2+ 3. ♘×d2 = 103 en 945 nœuds[16].

16. A titre de comparaison, le programme ECUME avec une recherche de tranquillité classique nécessite une recherche de 6 demi-coups et 33358 nœuds pour sélectionner le bon mouvement

3.4 Critère de choix d'une extension

Le choix des extensions dans un programme de jeux est une question difficile, il est en effet peu aisé de déterminer si telle ou telle extension augmente ou non le niveau de jeu d'un programme.

Le seul critère fiable est le résultat d'un grand nombre de parties contre des joueurs variés. Malheureusement, il n'est pas envisageable de faire un grand nombre de parties pour chaque combinaison possible d'extensions.

Il existe deux méthodes classiques pour comparer deux programmes :

leur faire jouer un grand nombre de parties en imposant les ouvertures : Chaque programme jouant alternativement avec le premier joueur, c'est une solution fiable pourvu que les programmes utilisés soient assez variés, le nombre de parties suffisamment élevé et la cadence (nombre de mouvements par heure) de jeu pas trop rapide. Malheureusement, cette méthode est trop lente;

compter le nombre de réponses correctes sur un ensemble de positions standards : Les jeux de positions disponibles ne reflétant pas l'éventail des situations possibles dans les parties, ils ne font pas ressortir la capacité à jouer correctement une partie d'échecs du premier au dernier mouvement.

Ces deux méthodes étant insatisfaisantes, Anantharaman[9] propose, pour comparer deux programmes n'ayant que des aspects de recherche sélective qui diffèrent, d'utiliser un jeu de positions construit en extrayant toutes les positions de 867 parties de Maîtres. Ainsi, une base de 9000 positions différentes représentatives de tous les stades d'une partie d'échecs est obtenue. Le nombre de réponses communes avec un programme de référence ayant 8 fois plus de temps que les programmes testés, est utilisé comme mesure de « force » du programme.

Anantharaman considère que le même programme ayant 8 fois plus de temps pour analyser les positions joue avec une force supérieure de 300 points Elo, s'appuyant sur les résultats établis par Thompson[174], et, ainsi, il calcule une différence Elo à partir des résultats obtenus sur les positions de tests.

Mais il est admis que le fait d'opposer deux programmes quasiment identiques fait entrer en résonance leurs différences et donne un résultat exagéré pour la meilleure des deux versions. C'est tout aussi vrai d'un programme jouant 8 fois plus vite, ou d'un programme examinant une profondeur de plus que l'autre. De plus les résultats de Thompson ont été établis pour des profondeurs faibles par rapport au standard actuel et leur validité pour des grandes profondeurs est hautement improbable.

3.5 Inconsistances de la recherche dues à l'application des heuristiques

Les heuristiques modifiant l'enveloppe de la recherche peuvent amener à des inconsistances dans la recherche. Prenons le cas d'une heuristique coupant des branches de l'arbre de recherche en fonction des bornes α et β. Comme ces heuristiques ne sont pas exactes, qu'elles coupent des branches utiles, susceptibles de modifier la valeur negamax du sous-arbre, pour certaines fenêtres de recherche, le résultat retourné sera erroné. De plus, comme dans les algorithmes à base de fenêtres minimales, comme le *Negascout*, une même position peut être analysée dans la même recherche par des fenêtres différentes, on peut arriver à des situations absurdes : prenons un exemple avec la position du diagramme 3.10. L'évaluation du programme ECUME après le mouvement 1. ♖b5 est de 0,24 (pour les noirs). Si lors de l'examen du mouvement 1. ♖b5 que nous supposons à 2 demi-coups de l'application de la recherche de la tranquillité, la borne α vaut 1,00. L'heuristique de futilité ne génère que les mouvements tactiques réponses à 1. ♖b5 , c'est à dire ♗×d4 ♖×f3 ♖e1+ . Si le programme ECUME applique, après chacun de ces coups, une recherche de tranquillité, il s'en suit les valeurs negamax suivantes :

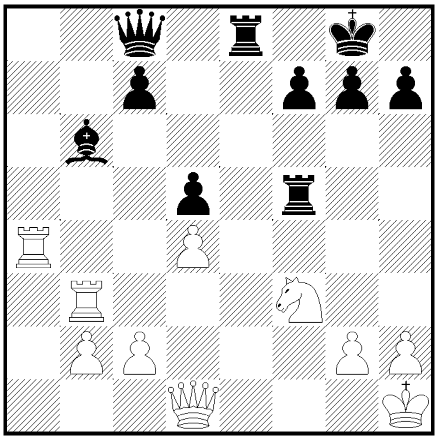
- 1 ... ♗×d4 ($< -2,44$) 2. ♘×d4 ($> 2,44$);
- 1 ... ♖×f3 ($< -1,06$) 2. P×f3 ($> 1,06$);
- 1 ... ♖e1+ ($< -5,10$) (2. ♘×♖ ($= -99,99$) ♖f1 # ($= 99,99$)) 2. ♕×e1 ($> 5,10$)

Supposons, ce qui est nécessaire pour obtenir un exemple simple, que la définition de la futilité soit erronée et qu'elle retourne ici la valeur de la fonction d'évaluation, c'est-à-dire 0,24 plutôt que $1,00 - \delta$, la recherche retournera alors que le mouvement 1. ♖b5 à une valeur supérieure à -0,24 pour les blancs. Or si, le programme ECUME analyse à nouveau ce mouvement avec une valeur de α bien inférieure à 0.24, (par exemple si 1 ♖b5 est devenu le meilleur mouvement, si on utilise l'algorithme PVS) le programme découvre la réponse 1 ... ♖e4 , qu'il estime à $0,32$, donc la valeur du mouvement 1. ♖b5 sera inférieure à $-0,32$. Nous avons donc une contradiction causée par l'élagage de futilité.

Un autre facteur d'inconsistance est l'utilisation des valeurs inexactes stockées dans la table de transposition pour une position donnée à une profondeur supérieure à celle voulue. En effet, ces valeurs inexactes sont des bornes sur la valeur à une profondeur supérieure et vont provoquer des coupures en fonction des valeurs des bornes α et β, coupures qui ne seront pas forcément validées si un nouvel examen de la position est nécessaire avec une nouvelle valeur de la fenêtre[184].

Il est donc nécessaire lors de l'implantation de telles heuristiques de prendre en compte la possibilité d'apparition d'inconsistances. En particulier dans le cas où un mouvement recherché avec une fenêtre minimale $[\alpha, \alpha+1]$ est donné comme supérieur à α au premier passage et comme inférieur à α aux passages suivants. Dans ce cas, plusieurs stratégies sont possibles comme celle consistant à faire un troisième examen avec une fenêtre la plus large possible[17]

17. Malheureusement, la table de transposition interfère dans ce cas en rétablissant les bornes α, β aux valeurs



Trait aux Blancs

Diagramme 3.10: *Position où peut se produire une inconsistance.*

ou à carrément ignorer cette inconsistance en supposant que le résultat faux est celui de la première recherche.

3.6 Conclusion

Les différentes heuristiques de recherche, que ce soit les extensions, la recherche de la consistance ou les coupures sélectives, transforment l'enveloppe du processus de recherche. Ainsi, comme le montre la figure 3.4, on peut considérer qu'une première partie de la recherche est réservée à l'exploration systématique des branches, puis vient une partie sélective, elle même suivie d'une recherche de la consistance. Bien sûr, certaines parties de l'enveloppe peuvent être réduites à néant, et la recherche de la consistance et la partie sélective peuvent fusionner en une recherche de la consistance sélective.

Pour choisir les méthodes sélectives employées, aussi bien que les extensions de recherche, nous nous aidons d'une batterie de tests constitués par les ensembles de tests classiques tels que les positions du livre « Win at Chess »[141], les positions de Pierre Nolot[124, 125] auxquelles nous ajoutons des positions glanées au fur et à mesure des parties jouées par nos programmes. De plus, nous faisons jouer les différentes versions contre elles-mêmes et contre des programmes extérieurs pour diminuer les effets de résonance. Le cas typique est un ensemble de 102 parties jouées contre une version de référence de notre programme, et, contre le programme ZARKOV 3.0. Ces 102 parties correspondent à 51 ouvertures différentes jouées alternativement avec les blancs et avec les noirs. Si les résultats des deux tests montrent

que l'on aurait pu déduire des précédentes recherches, donc il faut dans le cas de cette troisième recherche limiter l'effet de la table de transposition.

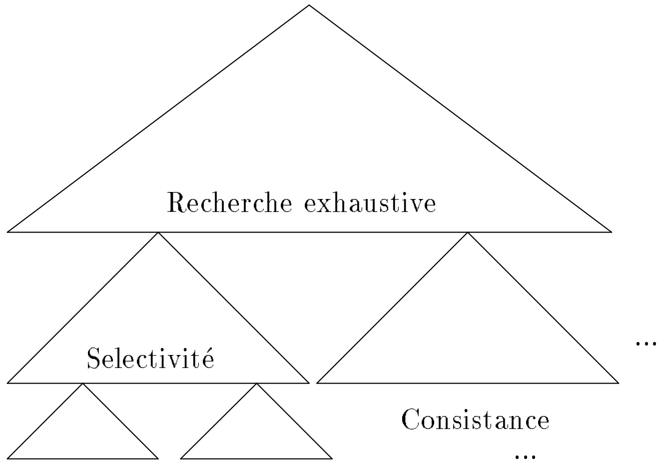


FIG. 3.4 - *Enveloppe de la recherche*

une amélioration, nous prenons la nouvelle version comme version de référence et ainsi de suite, sinon, soit les résultats sont plus mauvais dans les deux cas et nous améliorons ou abandonnons le concept que nous voulons introduire, soit le résultat n'est pas consistant avec les deux adversaires, auquel cas, nous analysons les parties pour déterminer la cause de ce résultat inconsistant.

Notre choix pour la sélectivité s'est finalement porté, au cours des années, sur le couple heuristique du mouvement nul et élagage de futilité qui nous permettent de gagner jusqu'a 50% de temps dans nos programmes CUMULUS, ECUME et JOKER.

Certains programmeurs ont fait des choix extrêmes, pour aller le plus vite possible en profondeur. Par exemple, le programme commercial FRITZ[110] dans ses versions 1 et 2 ne possède qu'une heuristique d'extension de recherche, sur les échecs quand on ne reprend pas la pièce qui fait échec et une sélectivité basée sur l'heuristique du mouvement nul. Tout cela, lié à une fonction d'évaluation très rapide, donne un programme sur micro-ordinateur atteignant de grandes profondeurs de recherche. Le résultat est un programme capable du pire, comme du meilleur, qui quelquefois va plus loin que n'importe quel programme classique, mais qui dans certaines lignes, où se trouvent des successions de captures/recaptures et ou échec, ne « voit » la valeur de la variante que bien après les programmes classiques.

Il est à noter que les heuristiques de recherche ont permis des transferts de technologie de la programmation des jeux de réflexion à d'autre thèmes. Ainsi, Newborn a utilisé avec succès la technique de la table de transposition dans les démonstrateurs de théorème et l'algorithme A* a bénéficié de l'approfondissement itératif[89], de la table de transposition[138], des heuristiques d'ordre des coups comme la table historique[139]. Ces heuristiques doivent aussi être transposables à d'autres problèmes de parcours de graphes[69].

Chapitre 4

Algorithmes minimax parallèles

Dans ce chapitre, nous allons montrer les résultats que nous avons obtenus dans la parallélisation des programmes de jeux de réflexion. Nous y présentons une nouvelle méthode [1] que nous comparons aux meilleures méthodes existantes [2].

L'évolution récente des machines parallèles ainsi que des outils de programmation dont elles disposent, conduit à s'intéresser à ces machines pour la programmation du minimax.

En effet, les progrès des niveaux de jeu des programmes de jeux de réflexion passent, entre autres choses, par l'utilisation de machines de plus en plus performantes. Ceci est vrai au niveau des machines séquentielles, et l'utilisation de machines parallèles est une manière de disposer d'une puissance beaucoup plus importante. Reste cependant le problème majeur sur de telles machines : il est nécessaire d'arriver à exploiter les ressources des machines parallèles. Cela passe évidemment par la parallélisation efficace de schémas séquentiels et par l'introduction et la définition de nouveaux algorithmes intrinsèquement parallèles dans leur conception.

Nous présentons ici l'état de l'art en ce qui concerne la parallélisation des algorithmes minimax : différentes voies ont été explorées avec plus ou moins de succès. De plus, jusqu'à récemment, le terme de machine parallèle signifiait des machines comportant un nombre faible de processeurs. L'arrivée de machines comme les CONNECTION MACHINES permet de passer à une autre échelle, avec des machines de plus de 65000 processeurs synchrones ou de 1024 processeurs asynchrones.

Nous décrivons les résultats de nos implantations d'algorithmes parallèles, et plus particulièrement la méthode de parallélisation que nous avons introduite.

1. Dans le section 4.3.4
2. L'ensemble de nos expériences est présenté dans la partie 4.3.1.

4.1 Définitions des accélérations et des sources de perte d'efficacité

Nous allons dans cette section définir un certain nombre de quantités qui vont servir à évaluer les performances des algorithmes parallèles et à les comparer avec celles des algorithmes séquentiels.

Pour mesurer les performances des algorithmes parallèles, nous définissons *l'accélération absolue* comme étant

$$\textit{accélération absolue} = \frac{\text{Temps du meilleur algorithme séquentiel}}{\text{Temps de l'algorithme parallèle}}$$

C'est en effet ce qui doit permettre en théorie d'évaluer l'efficacité d'un algorithme parallèle. Donc dans le cas des algorithmes minimax, les algorithmes parallèles devraient être comparés avec l'algorithme *NegaScout*, qui est le meilleur dans le cas des arbres fortement ordonnés(cf. le chapitre 2). Mais comme ce n'est pas toujours possible et pour pouvoir comparer avec les résultats déjà obtenus, nous définissons *l'accélération relative*:

$$\textit{accélération relative} = \frac{\text{Temps de l'algorithme séquentiel}}{\text{Temps de l'algorithme parallèle}}$$

De la même façon, nous définissons les efficacités absolue et relative par :

$$\textit{efficacité} = \frac{\text{accélération}}{\text{Nombre de processeurs}}$$

L'efficacité parfaite est de 1.0 ; nous dirons dans ce cas que l'accélération est *idéale*. Nous dirons que l'accélération est *sur-linéaire* si l'efficacité est supérieure à 1.0, *sous-linéaire* si l'efficacité est inférieure à 1.0.

En pratique, les algorithmes rencontrés sont d'accélération sous-linéaires du fait de la perte d'efficacité due à la parallélisation. Nous allons considérer trois sortes de pertes :

1° **Les surcoûts de recherche ($RE = \frac{\text{Nombre de nœuds visités par } N \text{ processeurs}}{\text{Nombre de nœuds visités par un processeur}} - 1$) :** l'algorithme parallèle parcourt plus de nœuds que l'algorithme séquentiel ou parcourt plusieurs fois le même nœud. Ce surcoût est généré par le *parallélisme spéculatif* qui commence des recherches sur des nœuds qui seraient élagués par l'algorithme séquentiel;

2° **Les surcoûts de synchronisation (SY) :** certains processeurs sont inactifs pendant un certain laps de temps, réduisant ainsi la *charge moyenne*;

3° **Les surcoûts de communication (CO) :** ils traduisent le temps passé par les processeurs pour préparer les communications, les envoyer, les recevoir.

Nous pouvons aussi définir le **surcoût en temps (TO) :**

$$TO = \frac{\text{Temps parallèle avec } N \text{ processeurs} * N}{\text{Temps séquentiel}} - 1 = \frac{N}{\text{accélération relative}} - 1$$

On peut alors approcher ce surcoût par la formule $TO = RE + SY + CO$[103]. Les surcoûts facilement calculables sont TO, RE et SY, dont nous pourrons déduire CO.

4.2 Travaux précédents

Dans cette partie, nous allons présenter les principaux travaux et résultats existants pour la parallélisation des algorithmes de recherche minimax. Il existe en effet diverses approches que nous allons analyser brièvement.

4.2.1 Approche naïve de la parallélisation de la recherche minimax

Une des approches pour paralléliser l'algorithme *minimax* est de distribuer la responsabilité de calcul des différents mouvements issus de la racine à des processeurs différents. Un processeur est chargé de collecter les résultats des différents mouvements.

L'avantage de cette approche est que les processeurs n'ont pas besoin de communiquer entre eux.

Si nous considérons que l'arbre à évaluer est un arbre uniforme de profondeur p et de largeur l, chaque processeur recevra un arbre de profondeur $p - 1$. L'accélération observée est donc au maximum de l'ordre du facteur de branchement asymptotique pour des arbres suffisamment profonds, c'est à dire $\frac{\zeta_l}{1-\zeta_l}$ où ζ_l est la plus petite racine positive de $x^l + x - 1 = 0$ (cf. la propriété 2.2.5). Si l'arbre est de largeur 40, l'accélération théorique observée pour 40 processeurs sera de $14,2$. Expérimentalement, il est montré que pour des arbres fortement ordonnés l'accélération observée est inférieure à 6 [12].

4.2.2 Découpe de la fenêtre de recherche

Une autre approche possible est de diviser la fenêtre d'analyse en intervalles disjoints et de donner un intervalle par processeur. La valeur *minimax* de la racine sera comprise dans un seul des intervalles, donc seul un processeur finira sa recherche sans échec en largeur.

Encore une fois, l'avantage de cette recherche proposée par Baudet[12] est qu'il n'est pas nécessaire aux processeurs de communiquer entre eux.

Expérimentalement, Baudet a montré que l'accélération maximale est comprise entre 5 et 6 et ceci indépendamment du nombre de processeurs.

Ceci s'explique par le fait que chaque fenêtre de recherche étant plus grande qu'une fenêtre minimale, l'accélération moyenne est inférieure au quotient de la complexité moyenne de l'algorithme $\alpha\beta$ par la complexité moyenne d'une recherche $\alpha\beta$ avec une fenêtre minimale. D'après la figure 2.12, l'accélération moyenne pour des arbres de largeur 40 et de profondeur inférieure à 12 est inférieure à 4,43. Ainsi, Baudet a observé une accélération relative sur-linéaire pour 3 processeurs[12] pour l'algorithme $\alpha\beta$.

4.2.3 Mandatory Work First

Akl, Barnard et Dorand[1] ont défini l'algorithme parallèle *Mandatory Work First* (MWF) basé sur une pré-recherche parallèle de l'arbre minimal. Le but du schéma MWF est de réduire le surcoût de recherche dû au parallélisme spéculatif.

L'arbre minimal considéré ne prend pas en compte les élagages profonds et se résume donc uniquement aux nœuds critiques de type 1 et 2[12].

L'algorithme MWF contient 2 phases :

1° Les nœuds de type 1 et 2 sont explorés en parallèle;

2° Les fils de gauche des nœuds de type 2 sont explorés suivant la règle suivante : soit P un nœud de type 2, la recherche MWF ne va pas explorer son fils P_i tant que les nœuds les plus à gauches P_j où $j < i$ n'ont pas été explorés. L'exploration du sous-arbre ayant pour racine P_i se fait en utilisant une recherche $\alpha\beta$ séquentielle.

La figure 4.1 nous montre un exemple d'application de l'algorithme MWF. Après la première phase, la valeur temporaire (valeur dérivée par minimax des feuilles déjà évaluées à cet instant) du nœud P_l est comparée avec les négations des valeurs temporaires des nœuds P_i et P_j. Si elle est plus grande alors on ne doit pas explorer les nœuds P_i et P_j, sinon ils sont explorés en parallèle (chacun séquentiellement). Ainsi, la valeur du nœud P_l est connue et comparable à la valeur temporaire du nœud P_m. Si la valeur du nœud P_l est plus grande, le sous-arbre ayant pour racine le nœud P_m peut être élagué, sinon les sous-arbres cerclés en pointillé sont explorés en parallèle (chacun séquentiellement).

Fishburn[57] a montré que pour des arbres très ordonnés, MWF est pratiquement optimal si l'on dispose d'un très grand nombre de processeurs. Des simulations sur des arbres à évaluation aléatoire de profondeur 4 ont montré que l'accélération croît au début rapidement avec le nombre de processeurs mais ne dépasse pas la valeur 6. Ainsi, à partir de 10 processeurs, les processeurs supplémentaires sont inactifs.

4.2.4 L'algorithme Tree-Splitting

L'algorithme *Tree-Splitting* est décrit par Fishburn[57] : il définit un arbre statique de processeurs. Le père d'un nœud de l'arbre de processeurs est appelé un *maître* et ses fils des *esclaves*.

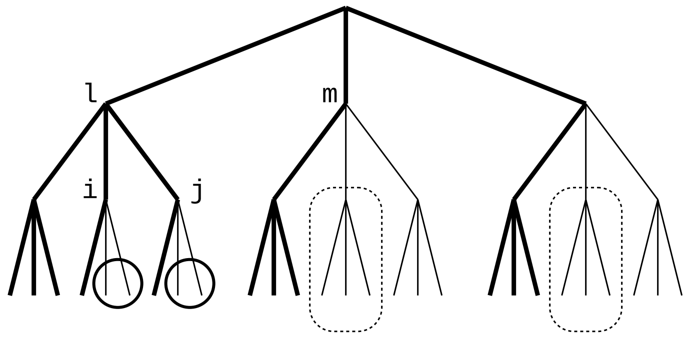


FIG. 4.1 - *Exemple d'application de l'algorithme* MWF.

Un nœud *maître* est chargé de donner du travail à ses *esclaves* et de les tenir informés des évolutions de la fenêtre $\alpha\beta$: il distribue les fils du sous-arbre dont il a la charge à ses esclaves quand ceux-ci sont disponibles. La racine de l'arbre des processeurs est un nœud maître, les nœuds internes sont à la fois maîtres et esclaves, les processeurs feuilles sont des nœuds uniquement esclaves.

Les processeurs feuilles exécutent un $\alpha\beta$ séquentiel. Quand un processeur esclave a fini sa recherche, il communique son résultat à son maître qui communiquera, s'il y a lieu, une nouvelle fenêtre de recherche à ses esclaves et si possible une nouvelle tâche au processeur qui vient de finir sa recherche.

Fishburn a montré que l'accélération est en $O(k^{1/2})$ pour k processeurs dans le cas d'arbres de jeux parfaitement ordonnés et presque idéale dans le cas d'arbres de jeux parfaitement mal-ordonnés. En simulation, il a été obtenu une accélération de $3,12$ pour 8 processeurs et de $5,31$ pour 27 processeurs.

4.2.5 Principal Variation Splitting

Le *Principal Variation Splitting* (*PVSplit)*[102] est une variante de l'algorithme *Tree Splitting* pour diminuer le surcoût de recherche dans le cas des arbres fortement ordonnés.

L'algorithme *PVSplit* parcourt la branche la plus à gauche de l'arbre de recherche (aussi appelée la variante principale) jusqu'à ce que la hauteur (distance jusqu'aux feuilles) du nœud courant soit égale à la profondeur de l'arbre de processeurs. A ce niveau, l'algorithme de *Tree Splitting* est appliqué; à sa terminaison, il est appliqué sur les frères du nœud courant, puis à son père et ainsi jusqu'à la racine de l'arbre de recherche.

Ainsi, tout le long de la variante principale, les nœuds le plus à gauche sont parcourus avant leurs frères, permettant à ceux-ci d'avoir une valeur de référence pour pouvoir faire des élagages, ce qui diminue le surcoût de recherche dû au parallélisme spéculatif, mais aussi la charge moyenne des processeurs.

Les expériences menées par Marsland et Campbell[102] montrent que l'algorithme *PVSplit* est 41% plus rapide que le *Tree Splitting*. Ils mirent en évidence une accélération de 3, 27 pour 4 processeurs[104] pour le programme PARABELLE sur des profondeurs de recherche de 6 demi-coups.

Cet algorithme a été implanté dans de nombreux programmes : Newborn[119] avec le programme OSTRICH a obtenu une accélération de 5, 03 pour 8 processeurs en redistribuant uniformément les successeurs des nœuds de type 1 en attente si un processeur devient inactif. Marsland, Olafsson et Schaeffer[103] décrivent une implantation pour le programme SUNPHOENIX qui donne une accélération de 3,75 pour 4 processeurs pour des recherches de profondeur de 7 demi-coups.[3]

Schaeffer[150] propose le *Dynamic PVSplit* : un processeur particularisé, le *contrôleur*, modifie dynamiquement l'arbre des processeurs restants, nommés les *chercheurs*. Quand un chercheur C est nommé directeur d'une tâche par le contrôleur, il applique *PVSplit* sur le sous-arbre dont il a la charge. Quand il reçoit un message du contrôleur lui proposant de l'aide, il répond favorablement s'il reste du travail à faire et le contrôleur lui indique alors l'identification du processeur D qui, à son tour, va appliquer *PVSplit* à un sous-arbre de celui de C. Quand un processeur est inactif, il le signale au contrôleur qui demande du travail aux processeurs actifs.

Hyatt, Suter et Nelson[80] proposent le *Enhanced PVSplit* qui, pour augmenter la charge moyenne, utilise la stratégie de distribution des processeurs suivante : si un processeur finit sa recherche et si son maître ne dispose pas de nœuds supplémentaires à lui proposer, alors tous ses frères esclaves sont interrompus et il se redistribue avec eux deux niveaux plus bas dans la recherche, sur un nœud de l'arbre pas encore complètement évalué. Cette stratégie permet de passer d'une accélération de 4,57 à une accélération de 5,93 pour 16 processeurs.

Kuszmaul[92] rapporte, lui, une accélération relative de 50 sur une CM5 de 512 processeurs et de 13 sur une CM5 de 32 processeurs pour l'algorithme *Jamboree* qui est une version de l' algorithme PVSplit combiné avec une décomposition de l'arbre de recherche du type « vol de tâches ».

4.2.6 Algorithmes de droit d'aînesse

Les algorithmes *Bound and Branch*[56] et *Young Brother Wait Concept* (YBWC) [55, 53, 54, 52, 51] développés indépendamment stipulent que l'on peut permettre le parallélisme à un nœud si et seulement si le premier fils du père a déjà été totalement évalué. Otto et Felten[126] ont implanté un algorithme parallèle qui permet le parallélisme à un nœud si et seulement si il existe déjà une entrée dans la table de transposition pour ce nœud ; si de plus il existe un mouvement dans l'entrée, ce mouvement est envisagé avant de permettre le parallélisme

3. Pour 6 demi-coups, les performances sont inférieures à celles obtenues par Marsland sur le programme PARABELLE. Cela soulève un point important : l'accélération obtenue n'est pas aussi importante que la qualité du jeu. Il s'avère plus facile d'obtenir de bonnes accélération avec des fonctions d'évaluation plus simples donc moins discriminantes. Ainsi, le programme SUNPHOENIX avait une fonction d'évaluation plus complète que le programme PARABELLE et était bien plus performant.

spéculatif sur ses frères cadets. Si par contre, il n'existe pas de mouvement dans la table de transposition, tous les mouvements peuvent être envisagés en parallèle.

Il a été mesuré une accélération relative de 344,49 pour 1024 transputers pour l'algorithme YBWC et 101 pour 256 transputers pour l'algorithme de Otto et Felten.

4.2.7 Delayed Branching Scheduled Strategy

Hsu[77] propose l'algorithme suivant : un processeur, nommé l'hôte, est chargé d'explorer l'arbre de recherche jusqu'à un niveau p. Les nœuds du niveau p sont alors empilés dans deux listes FIFO. Une des listes est chargée des nœuds de l'arbre minimal. Les nœuds sont dépilés pour être transmis à des processeurs esclaves très rapides ; les nœuds de l'arbre minimal sont examinés en priorité.

L'algorithme *Delayed Branching Scheduled Strategy* permet de ne pas rater d'élagages. HSU a montré, sur des simulations, une accélération de 250 pour 1000 processeurs esclaves.

4.2.8 L'algorithme CABP

Développé par Cung, l'algorithme CABP[44, 43] reprend les idées de Hsu dans une implantation parallèle asynchrone sur machine à mémoire partagée. Les deux listes de nœuds sont réparties dans une file critique et une file non-critique. Les sommets critiques sont parcourus en premier pour permettre une exploration plus efficace des sommets non critiques. Les listes sont partagées entre les processeurs de manières à diminuer les contentions d'accès aux données. Il obtient une accélération de 4,8 pour des arbres simulés fortement ordonnés. Il montre, dans le cas d'arbres à évaluation aléatoire, l'intérêt de fixer le degré d'élagages concurrents, c'est-à-dire le nombre de processus explorant en même temps un nœud de l'arbre de recherche, à une valeur proche du facteur de branchement.

4.2.9 SSS* parallèle

L'algorithme *SSS** a lui aussi motivé des modèles de parallélisation d'algorithmes minimax : le principal problème rencontré est le besoin de maintenir la liste des nœuds à explorer. Kraas[91] montre un exemple où cette liste est gérée par un seul processeur ; il obtient une accélération relative de 19 avec 40 processeurs. Hiromoto et al. quant à eux, obtiennent une accélération de 45 pour 128 processeurs sur des arbres simulés. Ils montrent aussi que l'algorithme $\alpha\beta$ parallèle n'est pas dominé par l'algorithme *SSS** parallèle.

Ces deux implantations mettent bien en évidence le goulot d'étranglement du *SSS** parallèle, c'est à dire l'accès à la liste *OUVERT* des nœuds à explorer.

4.2.10 L'algorithme Evaluation-Réfutation

Steinberg et Solomon[166, 167] proposent l'algorithme *Evaluation Réfutation* ($\mathcal{ER}$) : les nœuds de l'arbre de recherche sont divisés en deux groupes, les nœuds de type e et les nœuds de type r. Tous les fils d'un nœud de type e vont être parcourus, alors qu'un seul fils des nœuds de type r suffira. Chaque nœud interne a exactement un seul fils de type e.

Pour choisir le fils de type e d'un nœud interne z, $\mathcal{ER}$ évalue les petits-fils aînés (les fils aînés de ses fils) en parallèle et choisit le fils dont le fils aîné a la plus grande valeur. Le fils de type e est ensuite évalué en prenant soin de ne pas réévaluer son fils aîné, puis les autres fils sont réfutés les uns après les autres.

Les petits-fils aînés peuvent être évalués en parallèle car ils sont inclus dans l'arbre visité par l'algorithme $\mathcal{ER}$ séquentiel. Comme ces petits-fils sont eux-mêmes des nœuds de type e, leurs petits-fils aînés peuvent être récursivement évalués en parallèle. Pour éviter d'avoir à réfuter séquentiellement, les uns après les autres, les frère d'un nœud de type e, $\mathcal{ER}$ introduit deux autres types de parallélisme :

1° Réfutation parallèle : quand y, nœud de type e d'un nœud z, est fini d'évaluer, la réfutation des frères de y est faite en parallèle.

2° Définition de fils de type e multiples : après l'évaluation d'un nœud de type e, on choisit le meilleur fils, excepté y comme deuxième nœud de type e. Si le premier nœud de type e n'est pas le meilleur fils de z, alors il y aura un autre fils de type e qui devrait élaguer les autres fils de z.

$\mathcal{ER}$ a été parallélisé sur un ordinateur à mémoire partagée, le tas des problèmes en cours étant partagé entre les processeurs. Les auteurs ont obtenu une accélération relative de 11 pour 16 processeurs sur 5 positions d'Othello [167]. D'autres expériences menées par Rezaie et Finkel[142] ont montré sur des positions d'Othello 6×6 que les algorithmes MWF et *PVSplit* ont une meilleure efficacité relative que l'algorithme $\mathcal{ER}$.

4.2.11 L'algorithme $\alpha\beta$*

David[45] propose une méthode originale dans le but de s'affranchir des relations maître-esclave, sources de surcoût de synchronisation. Chaque processeur participant à la recherche se voit affecter de la résolution du problème de la racine. La recherche est finie pour tous dès qu'un processeur a fini sa recherche. Il évite la duplication des recherches par le partage de l'information du nombre de processeurs par nœud par l'intermédiaire d'une table de transposition partagée. Il introduit un partage de tâche pour les nœuds de type 3 qui limite le nombre de processeurs par fils à

$$k \times \textit{le nombre de processeurs pour le nœud de type 3}$$

où $k = 0,5$ pour 8 processeurs et $k = 0,25$ pour 32 processeurs[46].

Expérimentalement, il a été observé une accélération relative de $6,5$ pour 8 processeurs (sur une machine de 16 transputers, les 8 autres processeurs gérant la table de transposition partagée).

4.2.12 Résultats théoriques

Il existe plusieurs résultats indépendants sur l'accélération théorique des algorithmes minimax parallèles :

1° Althöfer[8] donne un algorithme minimax qui a une accélération relative moyenne linéaire en N (le nombre de processeurs), si les profondeurs des arbres de recherche sont au moins de $O(N.\log N)$.

2° Karp et Zhang[83] montrent que leur algorithme atteint au moins une accélération linéaire en N si les profondeurs des arbres de recherche sont au moins $O(N)$.

3° Cung[43] définit deux bornes : si P est le nombre de processeurs pour un arbre de largeur l et de profondeur n, T_P le temps pour P processeurs, alors :

- $T_P \geq 3+n(l+1)$ si l'arbre est réduit à l'arbre minimal et si $P \geq l^{\lceil n/2 \rceil} + l^{\lfloor n/2 \rfloor} - 1$
- $T_P \leq 3+n(l+1)$ si l'arbre de recherche est parfaitement mal ordonné, donc contient l^n nœuds si $P \geq l^n$. Cette borne restant valable en fait quel que soit l'arbre de recherche.

Dans [43], l'auteur suppose que les extrêmas de T_P sont obtenus pour les arbres réalisant les extrêmas de T_1. Ce qui est abusif : il aurait dû déduire, à l'aide du corollaire 2.2.3.2, la borne de l'accélération suivante :

$$\frac{\frac{l+1}{l-1} l^{\lceil n/2 \rceil} + \frac{2l}{l-1}(l^{\lfloor n/2 \rfloor} - 2) - n + 2}{3 + n(l+1)} \leq \frac{T_1}{T_P}$$

si $P \geq l^n$.

4° Kuszmaul[92] donne une borne supérieure pour l'accélération dans le cas des algorithmes de « droit d'aînesse » à l'aide d'une formule analogue à celle du corollaire 2.2.3.2, dans le cas où l'arbre est parfaitement ordonné (arbre critique) et si l'on utilise un algorithme de droit d'aînesse alors, quel que soit le nombre de processeurs, l'accélération est bornée par :

$$\frac{S_N(l)}{S_N(2)}$$

.

4.2.13 Conclusions

Il est difficile de comparer les résultats obtenus par les différentes équipes. Les problèmes sont différents, les accélérations bien souvent relatives, les machines de types divers et de modèles différents. Comment en effet comparer les résultats d'un programme d'Othello sur une machine à mémoire partagée avec les résultats d'un programme d'échecs sur une machine distribuée? De plus, nous sommes en droit, dans le cas de machines distribuées, de nous demander quel est l'effet de la granularité des nœuds de la machine : sur une machine plus lente, les communications peuvent apparaître plus rapides relativement que sur une machine basée sur un processeur rapide.

Néanmoins, nous notons que les meilleurs résultats ont été obtenus pour les algorithmes qui utilisent le droit d'aînesse.

4.3 Expérimentations

Dans cette partie, nous allons comparer les résultats que nous obtenons pour les algorithmes YBWC et $\alpha\beta*$ sur des arbres simulés et sur deux programmes de compétition, l'un d'Othello et l'autre d'échecs. Cette comparaison nous amènera à la définition d'un nouvel algorithme de parallélisation de recherche minimax.

Au commencement de nos travaux, les résultats donnés par l'équipe de Feldmann[51] étaient contestés par d'autres équipes[148, 151]. D'autres équipes encore n'obtenaient pas, sur des implantations du YBWC, des résultats similaires à ceux de Feldmann[101].

Dans un premier temps, nous allons donc décrire notre implantation du YBWC et ses performances. Puis nous décrirons l'implantation que nous avons réalisée de l'algorithme $\alpha\beta*$, et, enfin, nous comparerons ces deux algorithmes à un nouvel algorithme de recherche minimax parallèle.

4.3.1 Conditions de la parallélisation

Nous présentons ici la machine sur laquelle sont réalisées nos expériences, et décrivons les arbres de jeu utilisés pour les mesures.

4.3.1.1 Présentation de la Connection Machine 5

La CM-5 est une machine dont le modèle d'exécution est le MIMD avec une facilité de synchronisation. La machine supporte le modèle de programmation data parallèle qui est traditionnellement mais non exclusivement supporté par des machines SIMD. En effet lors de sa conception, il a été prévu de conserver les meilleurs concepts du modèle SIMD, à savoir un partage des données efficace sur les processeurs et une capacité de synchronisation rapide de l'ensemble des processeurs. Elle est donc optimisée pour exécuter de manière efficace des programmes SIMD.

Elle est composée de différents types de processeurs : processeurs de calcul, processeurs de contrôle et d'unités d'entrée/sortie ainsi que de différents réseaux, voir la figure 4.2.

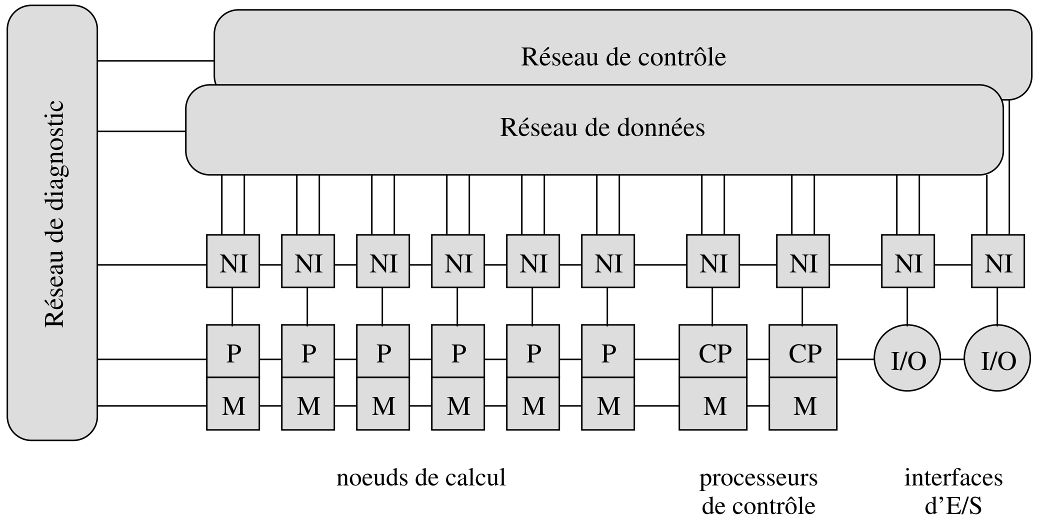


FIG. 4.2 - *Composition du système.*

Le processeur de base de cette machine est un processeur SPARC. Nous distinguerons plusieurs classes de processeurs selon leurs fonctions: les processeurs de contrôle (Control Processor : CP) et les processeurs de calcul (Processor Node : P). Mais tous les processeurs sont à l'origine du même type et selon leurs fonctions possèdent la capacité de privilégier certains types de tâches. Les processeurs de contrôle ont en effet les mêmes possibilités que les processeurs de calcul mais ils sont organisés pour optimiser des tâches de gestion (allocation de ressources, gestion de répartition des tâches, requête d'entrée/sortie, etc.) plutôt que du calcul. Les processeurs sont reliés aux trois réseaux de la machine par une interface (Network Interface : NI). Diverses configurations de machines existent : les plus petites contiennent 32 P et les plus grosses existant actuellement sont de 1024 P. La configuration maximale pouvant être réalisée comprend théoriquement 16384 processeurs, mais le coût et la réalisation d'une machine de cette taille sont assez prohibitifs. La machine peut être partagée en différentes partitions indépendantes les unes des autres.

La machine est prévue pour calculer en 64 bits et donc permet de bonnes performances pour le calcul numérique notamment sur des applications de grande taille.

Contrairement à de nombreuses autres machines et à l'idée de minimiser les coûts en matériels, la Connection Machine CM-5 comporte trois réseaux que nous allons brièvement décrire. Seuls deux d'entre eux sont visibles à l'utilisateur, il s'agit du réseau de données et du réseau de contrôle. Le troisième, moins intéressant du point de vue de l'utilisateur, est le réseau de diagnostic. Ces différents réseaux ont été conçus pour que la machine puisse être agrandie aisément, c'est-à-dire ait une bonne scalabilité.

Le réseau de données est organisé en arbre : c'est un fat-tree quaternaire, la figure 4.3 nous décrit un exemple de tel arbre. Les processeurs de calcul, de contrôle et d'entrée-sortie constituent les feuilles de l'arbre. Le réseau est composé de puces de routage ayant chacune quatre fils et deux ou quatre pères en fonction de la profondeur dans l'arbre. En effet pour avoir l'extension (scalability) de la machine, les noeuds proches de la racine ont quatre pères afin de maintenir un grand débit. Une partition de la machine correspond à un sous-arbre. Le routage de données d'un processeur à un autre se fait en remontant jusqu'à un ancêtre commun le plus proche puis en redescendant dans l'arbre. Il y a sur la CM-5 des communications générales entre les processeurs et des opérations de réduction et/ou distribution sur une classe de processeurs.

Le réseau de contrôle a une structure plus classique, c'est un arbre binaire complet dont les feuilles sont les différents types de processeurs énumérés précédemment. Il est notamment chargé de connecter chaque partition de la machine à un noeud de contrôle. C'est aussi lui qui gère les opérations de diffusions/réductions.

Le réseau de diagnostic est un arbre binaire non nécessairement complet. Nous ne nous étendrons pas sur sa description. Il est décrit ainsi que les deux autres réseaux et les motivations de ces choix dans l'article [95].

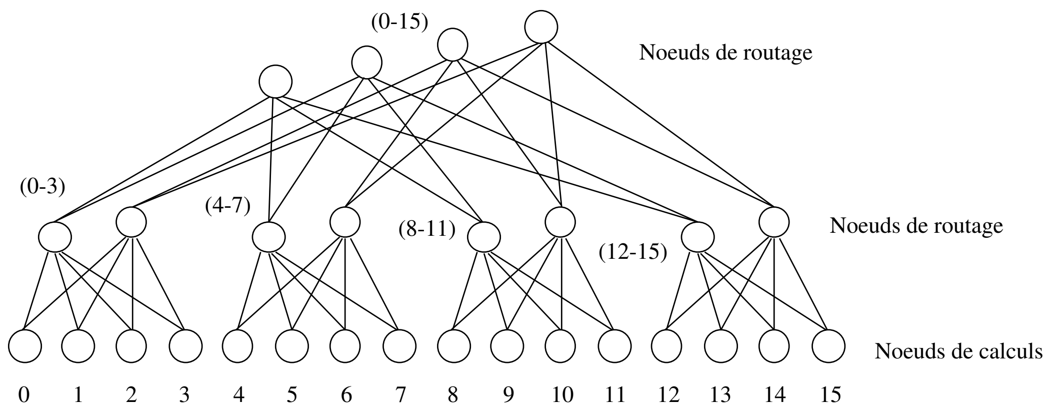


FIG. 4.3 - *Réseau de communication de la* CM-5.

Les processeurs de calcul possèdent aussi des unités vectorielles : 4 par processeur, une pour chaque banc mémoire, qui permettent d'accélérer les calculs en exécutant des instructions vectorielles. Elles servent aussi de contrôleur mémoire et permettent d'optimiser les requêtes de lecture et d'écriture vers le banc mémoire qui leur est associé.

La machine peut fonctionner selon deux modèles : message passing (envoi de message) ou data parallélisme avec possibilité de mixer les deux modes. Pour le fonctionnement en message passing, la CM-5 dispose d'une librairie de communication CMMD; c'est ce mode de fonctionnement qui donne accès à toute la généralité de l'architecture de la machine en exploitant les particularités du mode MIMD. Les échanges de messages sont aussi réalisables sur la CM-5 par l'intermédiaire de PVM (Parallel Virtual Machine voir [61]) qui est un outil permettant de considérer un réseau de machines comme un seule ressource de calcul. PVM permet de relier des machines hétérogènes et tend à devenir une librairie d'échange de message standard.

La bibliothèque PVM est implantée en tant que surcouche de la bibliothèque CMMD, elle même implantée en tant que surcouche de la bibliothèque de *messages actifs*[4]. Pour avoir une implantation la plus portable possible, PVM constituerait le bon choix, mais les performances en étant trop dégradées, nous avons choisi d'implanter nos programmes en CMMD.

Pour tous ces points, nous référons au manuel technique de la CM-5 [172] ainsi qu'à l'article [95].

4.3.1.2 Arbres utilisés lors des expérimentations

Nous avons utilisé différents modèles d'arbres de jeu pour comparer les implantations des différents algorithmes parallèles et déterminer leur champ d'application :

Arbres de jeu uniformes : à évaluation aléatoire ou de telle sorte que les arbres soient fortement ordonnés ;

Arbres de jeu d'Othello : cela correspond à des arbres de jeux fortement ordonnés avec un facteur de branchement faible et une profondeur élevée. Le programme séquentiel utilisé ayant la propriété d'avoir une évaluation non stable, entraînant une variante principale instable;

Arbres de jeu d'échecs : arbres de jeu très fortement ordonnés avec une fonction d'évaluation stable.

Les programmes d'échecs et d'Othello utilisent une table de transposition que nous avons implantée dans une simulation très efficace de mémoire partagée sur la CM5 en utilisant les messages actifs[173]. Pour tous les résultats qui suivent, le nombre d'entrées dans la table de transposition est constant, quel que soit le nombre de processeurs, sauf mention indiquant le contraire.

Pour les arbres simulés, chaque résultat correspond à la moyenne de 30 mesures. Les résultats sur Othello correspondent à la moyenne des 50 premiers mouvements d'une partie. Les résultats échiquéens correspondent à la moyenne sur les 24 positions Bratko Kopec (cf Annexe A).

4.3.2 Implantation de l'algorithme YBWC

Dans cette partie, nous allons détailler l'implantation et les résultats que nous avons obtenus avec l'algorithme YBWC. Nous allons séparer les aspects liés à la composition dynamique

4. Implantée elle-même directement en CMNA, la bibliothèque de gestion directe des réseaux de communications.

Les messages actifs représentent une implantation du protocole RPC (Remote Procedure Call) et se présentent sous la forme très simple (nœud, adresse de fonction sur le nœud, arg1, arg2, arg3, arg4) où les arguments sont des entiers signés sur 32 bits. Cet appel réalise l'appel de la fonction donnée par son adresse sur le nœud récepteur avec les arguments arg1, arg2, arg3, arg4.

de l'arbre de processeurs et ceux liés au contrôle du parallélisme spéculatif lié au principe de droit d'aînesse.

Il est à noter qu'à l'instar de la littérature existante nous nommons algorithme YBWC l'ensemble, alors qu'en fait le YBWC concerne plutôt le principe de droit d'aînesse.

4.3.2.1 Composition dynamique de l'arbre de processeurs

Dans les algorithmes de type *PVSplit*, l'arbre de processeurs est fixé ou est modifié dynamiquement au cours de la recherche[150, 80]. Feldmann a choisi de reprendre ce type de réagencement dynamique tout en restreignant l'effet des réorganisations et en décentralisant le système de l'algorithme *Dynamic PVSplit*. Cet algorithme de composition dynamique, proche de l'algorithme de *vol de tâche*[92], est régi par les principes suivants :

1° Au début de la recherche, nous affectons un processeur à la recherche de la valeur *negamax* de la racine;

2° Tout processeur affecté à la recherche de la valeur *negamax* d'un sous-arbre finira sa tâche quand il aura fini le calcul ou quand son maître le lui commandera;

3° Tout processeur inactif va choisir aléatoirement [5] un processeur (actif ou non) et lui demander du travail;

4° Tout processeur actif peut donner du travail à un processeur qui le lui demande en lui donnant une partie du sous-arbre dont il a la charge. Le processeur donnant sera appelé le maître et le recevant l'esclave.

Cette décomposition est mise en œuvre grâce à l'échange des messages suivants :

Demande de Travail : Un processeur inactif envoie ce message vers un processeur choisi aléatoirement;

Réponse à une Demande de Travail : Ce message contiendra les informations nécessaires pour que le processeur qui a demandé du travail puisse commencer une recherche si le processeur qui a reçu le message *Demande de Travail* peut partager son arbre de recherche, sinon la réponse sera négative et le processeur demandeur devra à nouveau renvoyer un message *Demande de Travail*;

Nouvelle fenêtre (α, β) : Un processeur maître envoie ce message à un de ses processeurs esclaves quand il est en mesure de communiquer une fenêtre (α, β) mise à jour pour le sous arbre de son esclave [6];

Fin de Travail : Le maître signifie à son esclave que sa tâche est finie.

5. On peut aussi choisir un processeur dans son voisinage pour réduire le coût des communications, mais cela dépend alors de l'architecture de la machine cible.

6. Ce type de message est la différence principale entre l'algorithme de décomposition de Feldmann et le vol de tâche tel que le définit Leiserson[92].

Résultat : Un processeur esclave ayant fini sa recherche renvoie à son maître la valeur *negamax* du sous-arbre dont il avait la tâche.

Les messages *Nouvelle fenêtre*, *Fin de Travail*, *Résultat* sont estampillés avec le numéro de travail de l'esclave qui lui a été fourni lors du message *Réponse à une Demande de Travail* pour l'authentifier du fait du croisement possible des messages.

Pour limiter les effets des processeurs inactifs sur le travail des processeurs actifs, un processeur actif ne répond qu'à un message par itération. [7]

La figure 4.4 montre un exemple d'échange de messages entre deux processeurs. Dans ce système de message, un processeur sera inactif soit parce qu'il n'a pas de tâche à effectuer, soit parce qu'il attend la terminaison de ses esclaves. Dans le dernier cas, le concept *Helpfull Master*[53] qui stipule qu'un processeur maître qui attend le résultat d'un de ses esclaves pour terminer sa recherche doit proposer à son esclave de devenir à son tour son esclave, n'est en pratique pas implanté du fait de sa complexité.

4.3.2.2 Algorithmes $\alpha\beta$ itératifs

Pour qu'un processeur puisse passer un sous-arbre de son arbre de recherche, il est nécessaire qu'il connaisse l'état total de son arbre de recherche, c'est à dire, par niveau, la fenêtre (α, β), les mouvements déjà explorés, le mouvement en cours, les mouvements en attente.

Dans l'algorithme $\alpha\beta$ sous sa forme parallèle, ces informations sont contenues dans la pile d'exécution, d'où la nécessité de dérécursifier l'algorithme $\alpha\beta$. La figure 4.5 nous montre la dérécursification que nous avons utilisée dans notre implantation de l'algorithme *YBWC*.

Il est à noter pour la suite que dans nos implantations, ainsi que dans celles que nous avons trouvées dans la littérature[36], le fait de dérécursifier les algorithmes $\alpha\beta$ et *Negascout* entraîne une baisse de performance de l'ordre de 10% à 30%.

Ainsi, à un instant donné, un processeur connaît les sous-arbres non encore évalués et peut les affecter à un esclave lui demandant du travail.

Dans la figure 4.6, nous voyons un exemple d'exécution de la composition dynamique de l'arbre de recherche : au moment où le processeur 1 reçoit la demande de travail du processeur 2, il explore le sous-arbre issu de la position $A12$; interrompu dans son exploration par la requête du processeur 2, il lui sous-traite la recherche du sous-arbre issu de la position $A2$, marque que cette position est sous-traitée par le processeur 2 et reprend sa recherche. Le processeur 2 enverra au processeur 1 un message *Résultat* quand il aura finit sa recherche ou alors il sera interrompu par un message *Fin de Travail* envoyé par le processeur 1.

Les autres messages envoyés par le maître à son esclave rapportent le fait que la fenêtre

7. Kuszmaul[92] prend un schéma semblable en autorisant un seul message par laps de temps entre deux « barrières » de synchronisation des processeurs. Ces barrières sont réparties régulièrement durant le temps de recherche.

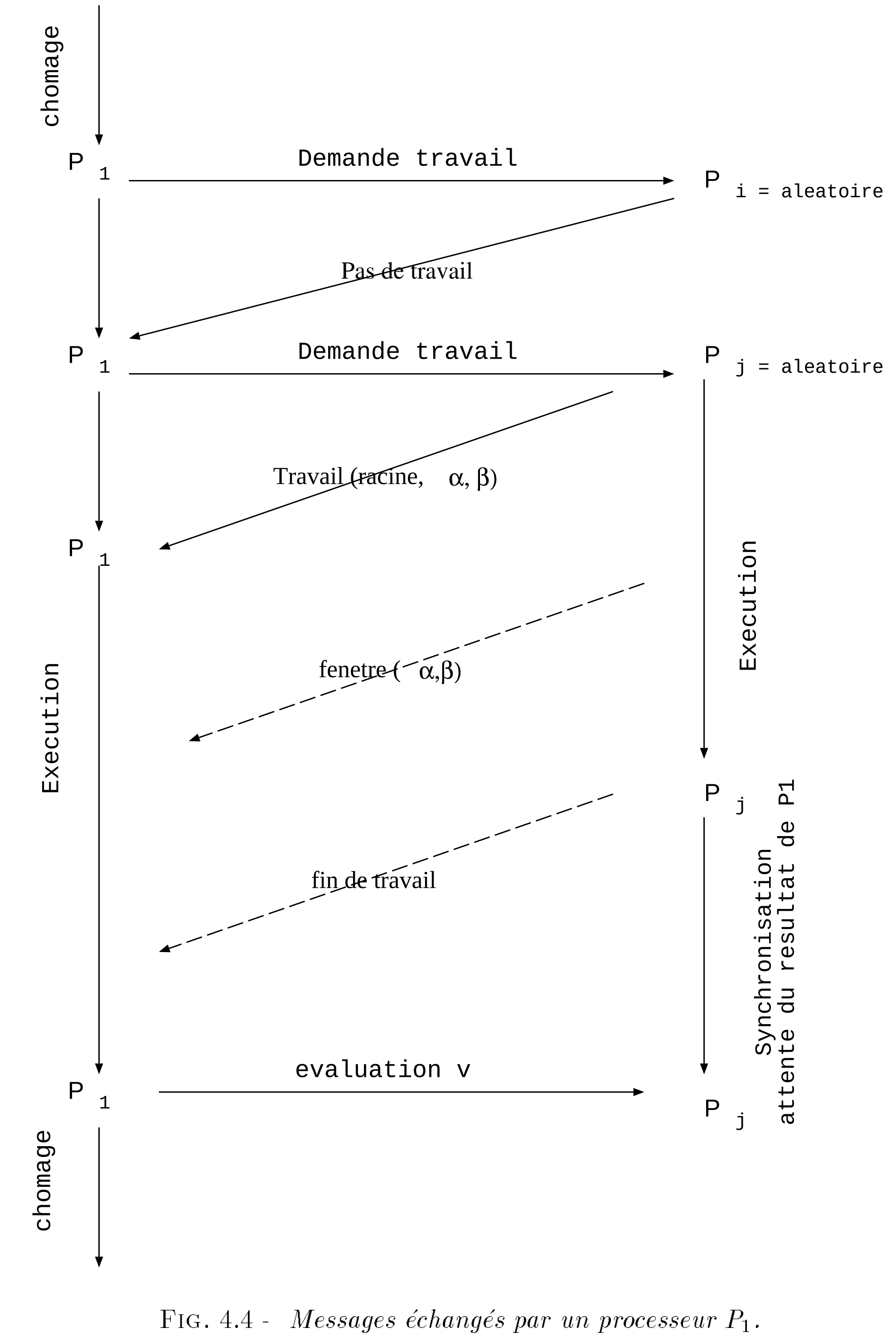


FIG. 4.4 - *Messages échangés par un processeur P_1.*

```
funct fab(Position[0], α[0], β[0]) ≡
        ⌈Niveau ← 0;
         indice[0] = 0;
         while Niveau ≥ 0 do
                 M ← ProchainCoup(Position[Niveau]);
                 while ¬M = 0 do
                         Position[Niveau + 1] ← Position[Niveau] • M;
                         α[Niveau + 1] ← −β[Niveau];
                         β[Niveau + 1] ← −α[Niveau];
                         Niveau ← Niveau + 1;
                         Valeur[Niveau] ← −∞;
                 od;
                 Valeur[Niveau] ← g(Position);
                 while Niveau ≥ 0 do
                         v ← −valeur[Niveau];
                         Niveau ← Niveau − 1;
                         if Niveau > 0
                            then if v > Valeur[Niveau]
                                       then
                                               Valeur[Niveau] ← v;
                                               if v > α[Niveau]
                                                  then
                                                          α[Niveau] ← v;
                                                          if v ≥ β[Niveau]
                                                             then
                                                                     β[Niveau] ← v;
                                                                     continue fi fi fi
                                 if ¬ProchainCoup(Position[Niveau]) = 0
                                    then break; fi
                         fi
                 od

         od
         Valeur[0]⌋.
```

FIG. 4.5 - *Algorithme $\alpha\beta$ dérécursifié*

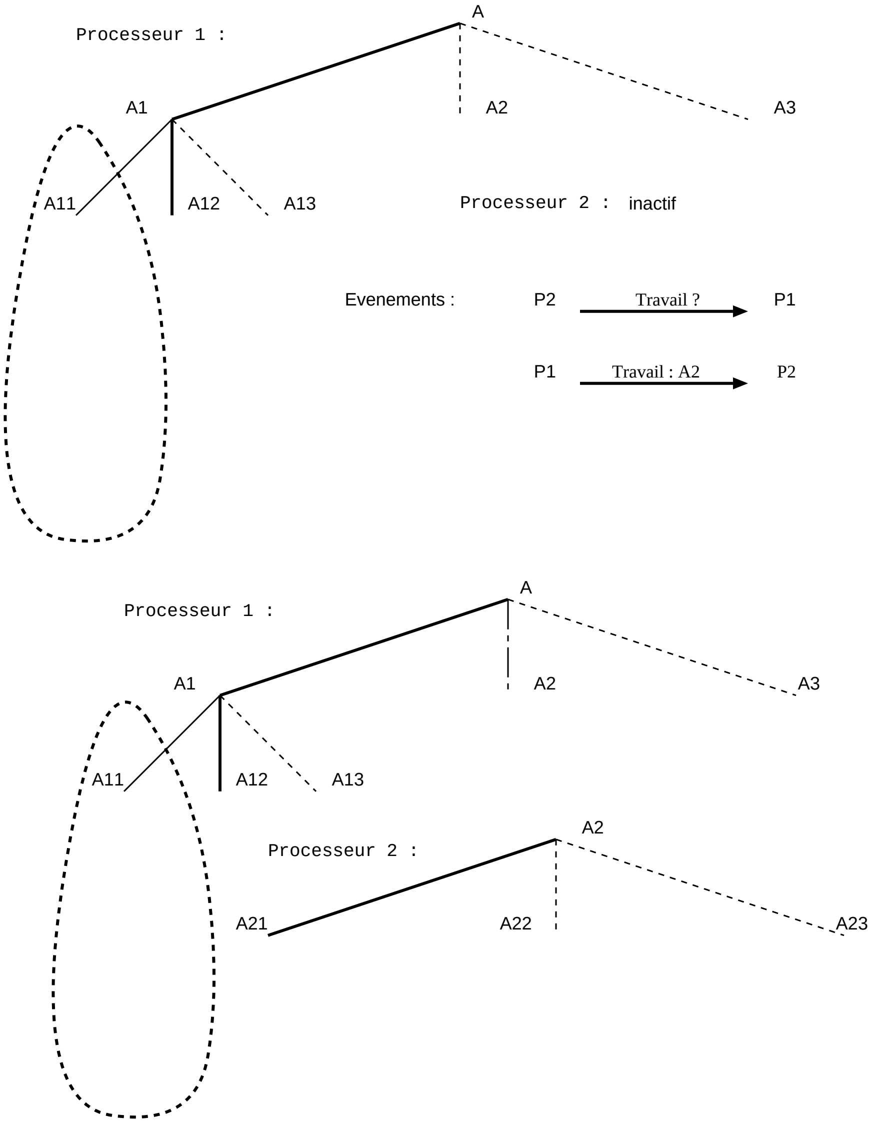


FIG. 4.6 - *Exemple d'exécution de la composition dynamique de l'arbre de processeurs*

$(\alpha[niveau], \beta[niveau])$ a changé au niveau de la racine du sous-arbre sous-traité à son esclave. Quand un esclave renvoie une valeur, cette valeur permet la modification des fenêtres au niveau de l'ex-racine de l'esclave et dans les niveaux supérieurs (en dessous dans le dessin des arbres). Cette nouvelle valeur peut même provoquer un élagage de la branche explorée par le processeur maître qui devra alors reprendre la recherche au niveau où la coupure s'est produite.

4.3.2.3 Résultat de la composition dynamique

Pour bien séparer les différents aspects de l'algorithme *YBWC* tel que Feldmann[51] le décrit, nous avons mesuré les accélérations relatives à l'algorithme $\alpha\beta$ en utilisant uniquement la composition dynamique de l'arbre de recherche, ce que nous nommons par la suite « vol de tâche » d'après la terminologie de Leiserson[92] [8].

Dans la figure 4.7, nous pouvons voir les résultats que nous avons obtenus en appliquant le vol de tâches sur des arbres théoriques. Nous remarquons que le vol de tâches à lui seul permet d'obtenir de très bons résultats sur les arbres à évaluation uniforme (ou aléatoire) atteignant une accélération relative de 19 pour des arbres de largeur 40 et de profondeur 6 sur 32 processeurs, le surcoût principal étant alors le surcoût de recherche. Pour les arbres fortement ordonnés, nous obtenons au contraire une accélération relative de 8,7 pour 32 processeurs avec un surcoût de recherche prédominant de 210,67% et nous voyons que les accélérations obtenues pour la profondeur de 7 demi-coups sont inférieures à celles obtenues pour la profondeur de 6 demi-coups.

Dans le cadre des arbres de jeux réels, ici Othello et les échecs, les résultats de la figure 4.8 confirment les résultats obtenus avec les arbres simulés ordonnés : l'accélération relative n'est que de 5,81 avec un surcoût de recherche de 451% pour le jeux d'Othello pour des arbres de profondeur 11 et de 4,24 avec un surcoût de recherche de 592% pour les arbres d'échecs de profondeur 8 demi-coups. De plus, pour les arbres d'échecs, les résultats des profondeurs de 7 et 8 demi-coups sont inférieurs à ceux de la profondeur de 6 demi-coups, phénomène déjà remarqué dans le cadre des arbres simulés fortement ordonnés mais qui n'apparaît pas dans le cadre des arbres d'Othello; cela peut s'expliquer par le fait que le facteur de branchement à Othello est moindre donc les processeurs se répartissent mieux dans l'arbre de recherche, mais aussi par le fait que, pour nos programmes, les arbres d'Othello sont moins bien ordonnés.

Il apparaît donc que dans le cas des arbres réels, et plus généralement pour les arbres ordonnés, un contrôle du parallélisme spéculatif est nécessaire.

Ces résultats sont à rapprocher du modèle de parallélisation naïf (cf. paragraphe 4.2.1). En fait si le processeur maître fournit toujours à ses esclaves les modèles les plus grands, ces deux approches deviennent similaires à ceci près qu'ici, en ayant fini son travail, un esclave se met au service des autres esclaves.

8. En fait, il s'agit d'un vol de tâche étendu puisque les processus maîtres avertissent leurs esclaves des variations de la fenêtre de recherche.

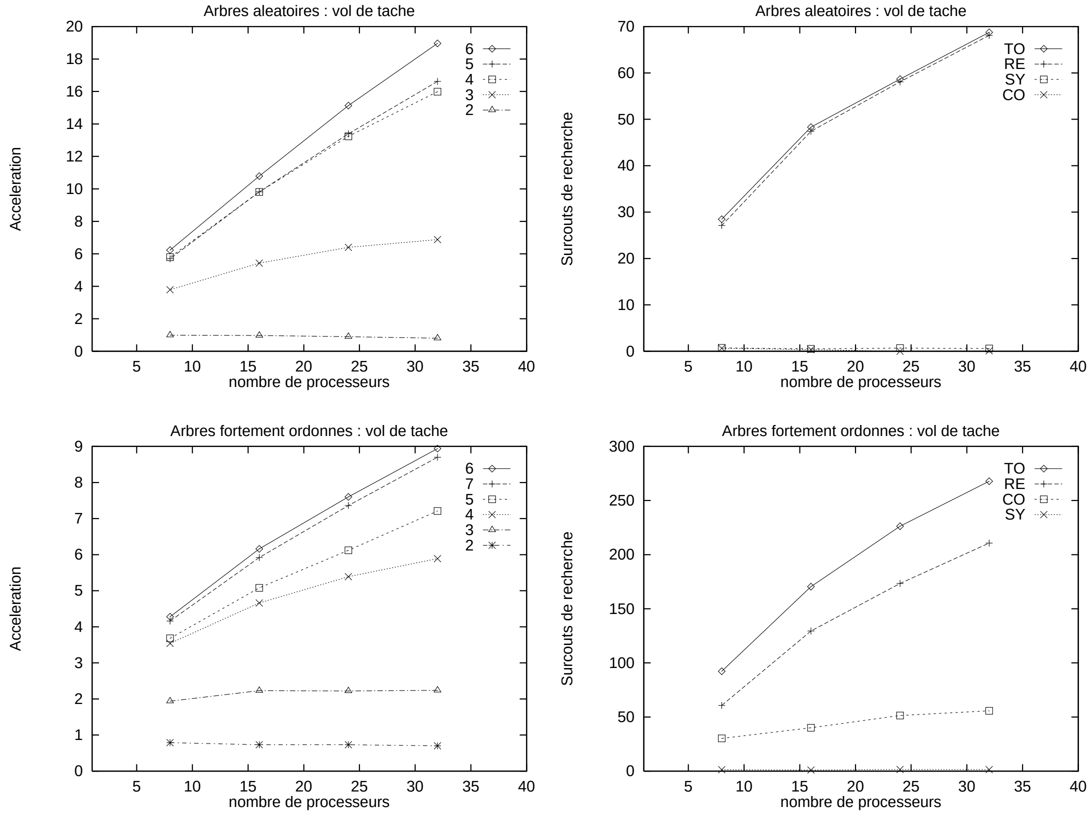


FIG. 4.7 - *Résultats de la décomposition dynamique sur des arbres théoriques*

Les accélérations sont données pour différentes profondeurs en fonction du nombre de processeurs. Les surcoûts sont ceux définis au § 4.1 et sont donnés pour la profondeur la plus grande.

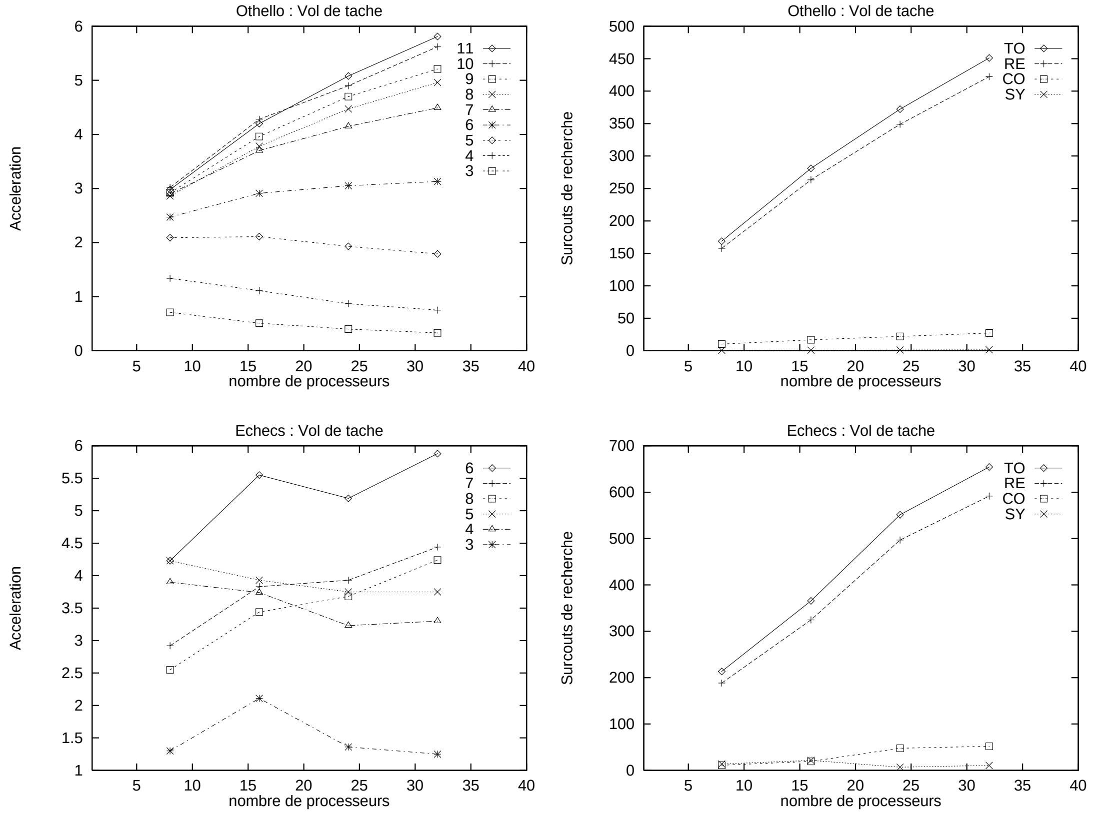


FIG. 4.8 - *Résultats de la composition dynamique sur des arbres réels.*

4.3.2.4 Le concept « Young Brothers Wait »

Le concept « Young Brothers Wait » est l'expression du droit d'aînesse dans cette composition dynamique de l'arbre de recherche : un processeur maître ne fait pas sous-traiter un nœud si son frère aîné n'est pas totalement évalué. Ce concept va diminuer le parallélisme spéculatif, donc diminuer le surcoût de recherche mais aussi augmenter le surcoût de synchronisation.

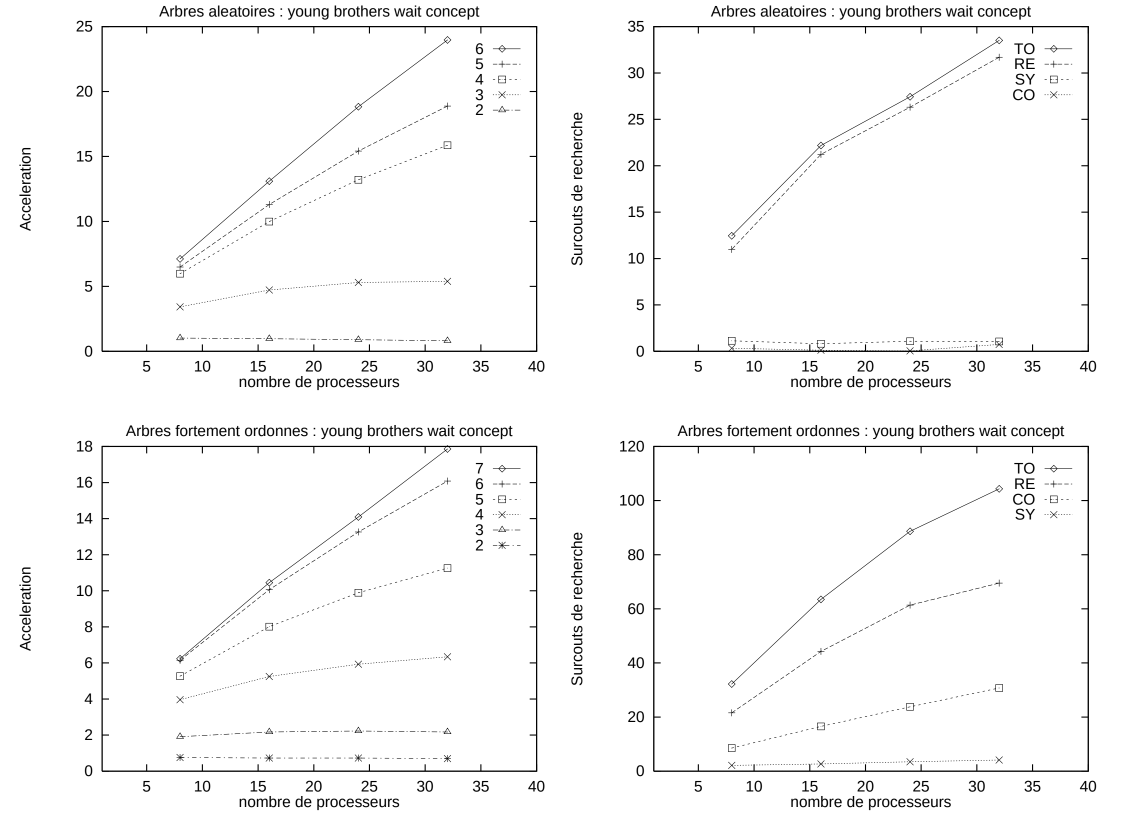


FIG. 4.9 - *Résultats du « Young Brothers Wait Concept » sur les arbres simulés.*

Dans la figure 4.9, nous donnons les résultats que nous obtenons sur les mêmes arbres simulés que ceux de la figure 4.7. Les résultats sont conformes à notre attente, le surcoût de recherche a notablement diminué pour les arbres aléatoires et fortement ordonnés. Nous obtenons alors une accélération relative de 24 pour 32 processeurs sur les arbres aléatoires de largeur 40 et de profondeur 6 et une accélération de 18 pour les arbres fortement ordonnés. Nous notons l'augmentation des effets dûs aux surcoûts de communications et de synchronisation dans le cas des arbres fortement ordonnés (Ils diminuent en valeur absolue, mais augmentent en valeur relative).

Dans la figure 4.10, nous avons appliqués le Young Brothers Wait Concept sur les arbres de jeux réels. Ici, nous obtenons un bien meilleur résultat avec les arbres d'Othello qu'avec les arbres d'échecs. Ainsi, sur le programme d'Othello, nous obtenons une accélération relative

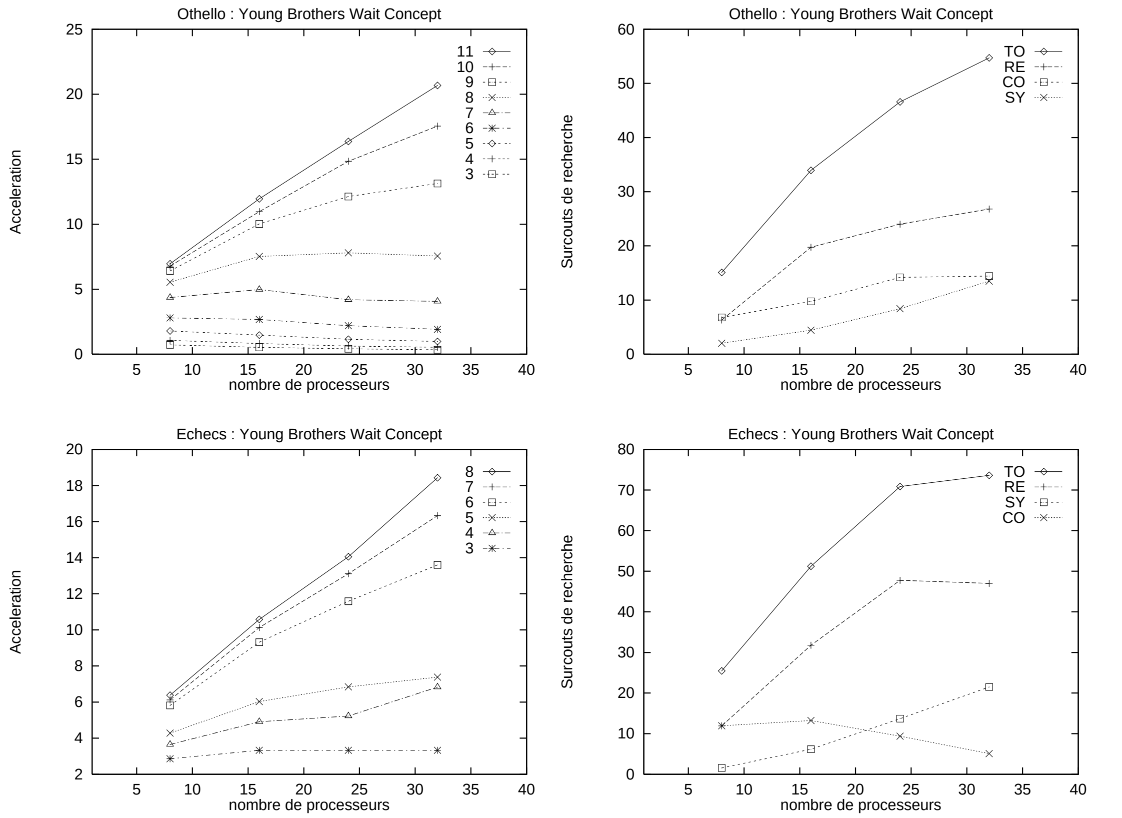


FIG. 4.10 - *Résultats du « Young Brothers Wait Concept » sur les arbres réels.*

de 20,68 pour des arbres de profondeur 11, le parallélisme spéculatif est bien contrôlé et nous notons l'apparition d'un surcoût de communication dû à la gestion de la table de transposition.

Nous avons observé aussi que jusqu'à une certaine taille de problème, le surcoût prépondérant est le surcoût de synchronisation. Ainsi, pour les arbres de profondeur 7, sur 32 processeurs, le surcoût de recherche est de 37 et le surcoût de synchronisation de 69, la charge moyenne des processeurs étant de 70%. Cette charge moyenne augmente avec la profondeur, donc avec la taille du problème, ainsi pour les arbres de profondeurs 8, la charge moyenne est de 88% et l'accélération relative obtenue est de 21,78 pour 32 processeurs.

4.3.3 Implantation de l'algorithme $\alpha\beta$*

Suite aux résultats de la thèse de David[45], nous avons implanté l'algorithme $\alpha\beta$* pour analyser ses performances par rapport à celles de l'algorithme YBWC. Du fait de l'utilisation de la table de transposition, nous n'avons pas pu tester l'algorithme $\alpha\beta$* sur les arbres simulés. Nous nous sommes donc restreints à l'analyse de cet algorithme sur les programmes d'Othello et d'échecs.

Pour implanter l'algorithme $\alpha\beta$*, nous avons ajouté un champ dans la table de transposition distribuée reflétant le nombre de processeurs qui, à un moment donné, explorent un nœud donné.

David introduit un « facteur de confluence », k, qui limite le nombre de processeurs explorant le même nœud s'il s'agit du fils d'un nœud de type 3 : le nombre de processeurs explorant en même temps un fils P_i d'un nœud de type 3 est limité à k fois le nombre de processeurs explorant P. Il fixe k à $0,5$ pour 8 processeurs et k à $0,25$ pour 32 processeurs[45, 46]. Expérimentalement, nous avons obtenu les meilleurs résultats pour k fixé à $0,25$ quel que soit le nombre de processeurs pour les deux programmes de jeux.

La figure 4.11 montre les résultats que nous avons obtenus en utilisant l'algorithme $\alpha\beta$* sur les programme de jeux réels. De ces résultats, nous pouvons induire que l'accélération est bien inférieure à celle reportée dans [45]. Le surcoût dominant est le surcoût de recherche. Ce qui montre que le contrôle spéculatif proposé n'est pas suffisamment efficace. Néanmoins, ces résultats, au contrôle spéculatif près, valident le schéma de parallélisation, en ce sens qu'aucun autre surcoût ne prédomine quand nous augmentons le nombre de processeurs. En effet, nous aurions pu nous attendre à avoir un goulot d'étranglement dû à la table de transposition, ce qui se serait traduit sur nos mesures par une augmentation du surcoût de communication. Il n'en est rien ici, du fait de l'implantation de la table par les messages actifs sur la CM5[9]. C'est pourquoi, nous avons choisi de combiner le schéma proposé par David avec un contrôle du parallélisme spéculatif de type droit d'aînesse.

9. Ceci sera vérifié dans le § 4.3.6.

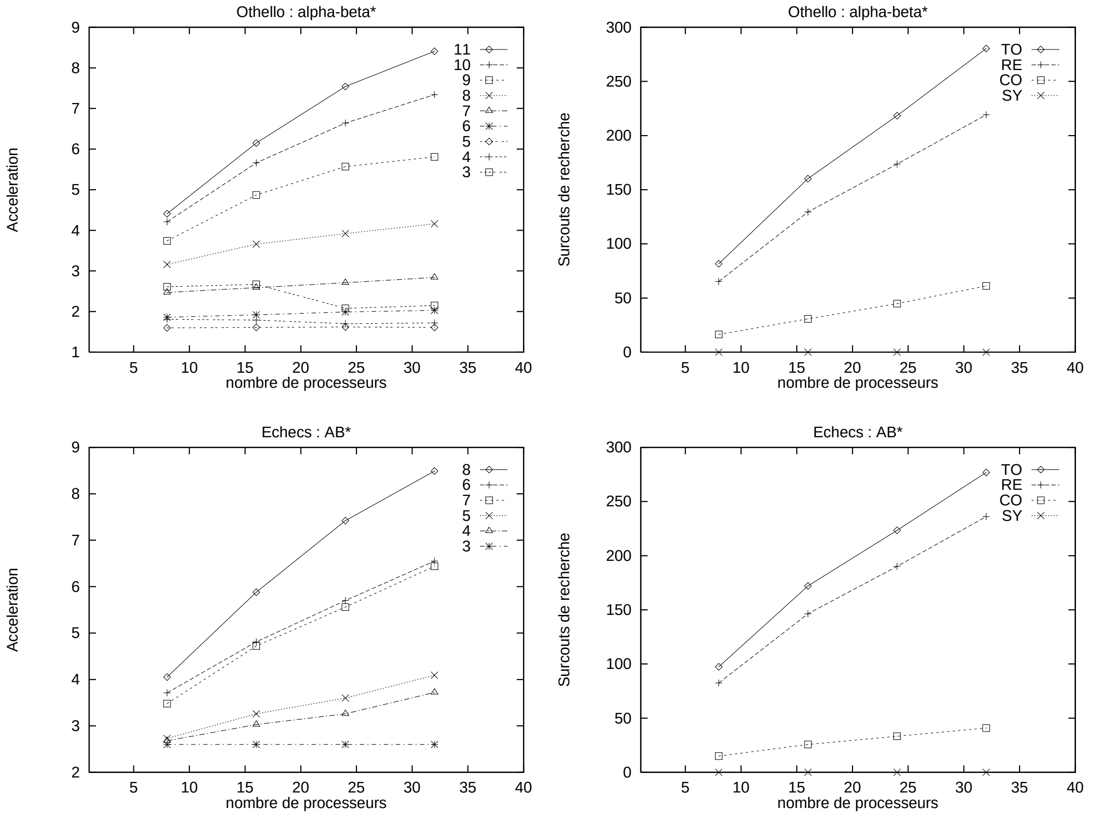


Fig. 4.11 - *Résultats obtenus en appliquant l'algorithme $\alpha\beta*$*

4.3.4 Résultat du Schéma $\alpha\beta$* avec le contrôle de droit d'aînesse.

Dans cette partie, nous allons montrer les résultats que nous avons obtenus en modifiant le schéma $\alpha\beta$* en y adjoignant un contrôle du parallélisme spéculatif de type droit d'aînesse.

Le contrôle du parallélisme spéculatif que nous avons défini est le suivant : tout parallélisme est interdit au niveau d'un nœud si le fils de gauche n'est pas totalement évalué, de plus si le premier fils de gauche est totalement évalué, nous recherchons un fils de ce nœud non en cours d'exploration, et s'ils sont tous déjà en cours, nous prenons un nœud non totalement évalué.

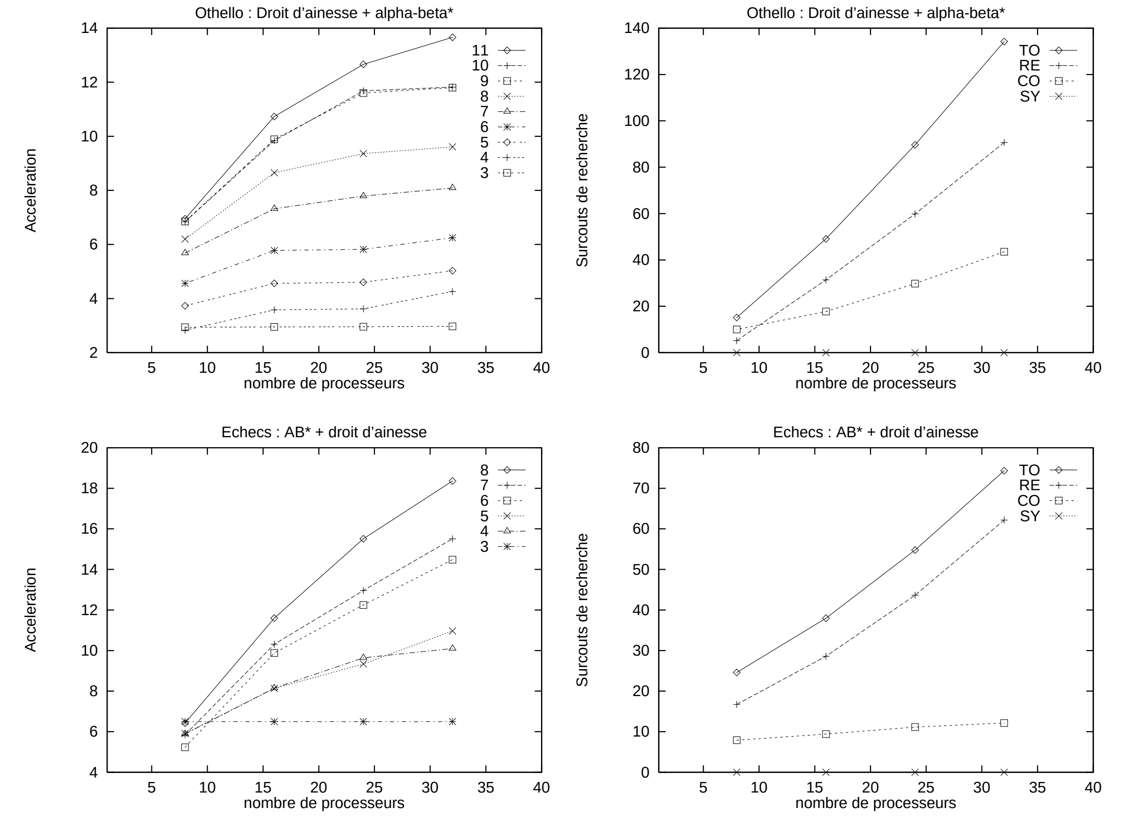


FIG. 4.12 - *Accélérations obtenues en appliquant l'algorithme $\alpha\beta$* combiné au droit d'aînesse.*

Dans la figure 4.12, nous voyons les résultats obtenus sur les programmes de jeux par la combinaison du droit d'aînesse et du schéma de parallélisation de l'algorithme $\alpha\beta$*. Pour le programme d'Othello, nous pouvons observer que l'accélération relative est de 12 pour 32 processeurs, le principal surcoût étant le surcoût de recherche, cela ne suffit donc pas à compenser le gain de 10% obtenu par l'utilisation d'une version récursive de l'$\alpha\beta$. Par contre, pour les échecs, l'accélération relative obtenue est de 18,36 et le gain obtenu par l'utilisation de la version récursive est supérieur à 30%. Donc, cette méthode donne sur ce programme d'échecs un résultat nettement supérieur au couple vol de tâches et droit d'aînesse (YBWC).

4.3.5 Faut-il paralléliser l'$\alpha\beta$ ou bien Negascout ?

Jusqu'ici, nous avons uniquement montré les résultats obtenus en parallélisant l'algorithme $\alpha\beta$. Nous avons vu dans le chapitre 2 que l'algorithme le plus rapide pour traiter les arbres fortement ordonnés est l'algorithme *Negascout*. Cet algorithme est plus efficace en séquentiel, et de ce fait, il est plus difficile à paralléliser car plus intrinséquement séquentiel.

Donc, nous avons étudié le comportement de l'algorithme *Negascout* suivant les schémas de parallélisation les plus performants, c'est-à-dire les deux schémas de décomposition de l'arbre de processeurs alliés au droit d'aînesse.

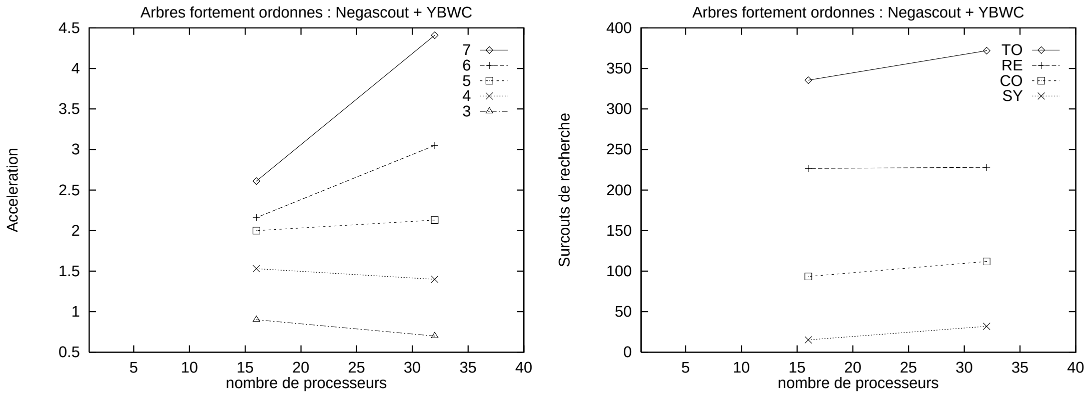


FIG. 4.13 - *Accélérations obtenues en appliquant le schéma « Young Brothers Wait Concept » à l'algorithme* NegaScout.

Dans la figure 4.13, nous voyons les résultats obtenus pour la parallélisation de l'algorithme *NegaScout* dans le schéma YBWC. Nous constatons que dans les arbres fortement ordonnés simulés, la parallélisation est nettement moins efficace que pour l'algorithme $\alpha\beta$. Ce qui n'est pas compensé par le fait que pour ces arbres sur un processeur (c'est à dire la référence pour ces accélérations relatives), l'algorithme *Negascout* est 2 fois plus rapide que l'algorithme $\alpha\beta$.

Ceci provient du fait que les recherches avec des fenêtres minimales sont dans le cas parallèle exécutées en pure perte dans le cas où un nœud P_i qui n'est pas de type 1 a une valeur *negamax* supérieure à celle de son frère P_1 de type 1. Si les autres frères de P_i étaient évalués en parallèle en même temps que P_i, après le retour de la valeur *negamax* de P_i, leurs recherches seraient interrompues et elles devraient alors redémarrer de zéro avec une nouvelle fenêtre disjointe de celle de la première recherche. Donc les recherches spéculatives sur les nœuds P_i ne serviraient à rien.

Feldmann[51] et Kuszmaul[92] justifient leurs algorithmes [10] en précisant que le cas décrit ci-dessus intervient rarement, mais les résultats obtenus sur les arbres simulés nous montrent le contraire.

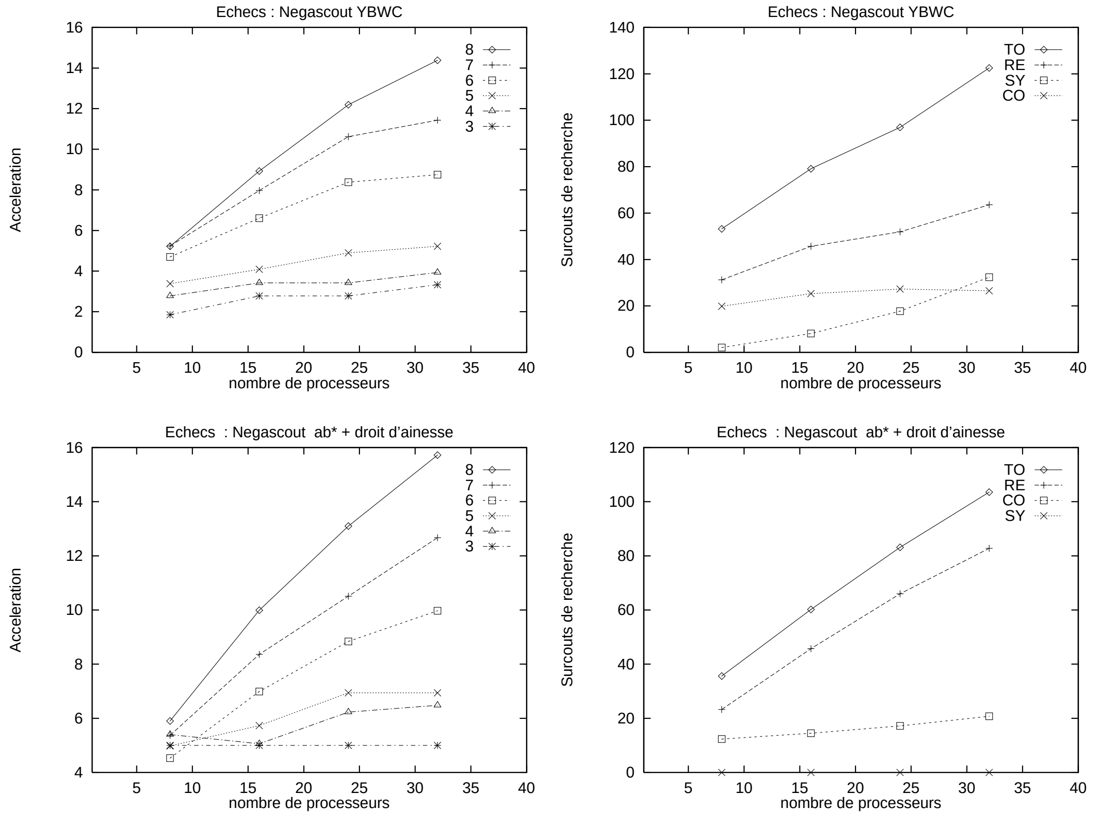


FIG. 4.14 - *Accélérations obtenues en appliquant les schémas de parallélisation au programme d'échecs sous Negascout.*

Dans la figure 4.14, nous voyons que dans le cas du jeu d'échecs, nous obtenons des résultats moins bons que pour la parallélisation de l'algorithme $\alpha\beta$ avec le même schéma, mais que le résultat est moins catastrophique qu'avec les arbres simulés. Ceci provient du fait que la table de transposition globale conserve les résultats intermédiaires des recherches interrompues et donc lorsque les processeurs reçoivent une nouvelle fenêtre, ils utilisent ces résultats pour ne pas les recalculer. Le manque d'efficacité de la parallélisation de la recherche *Negascout* est ici compensé par le fait que l'algorithme *Negascout* est, sur ce programme, 1,52 fois plus rapide que l'algorithme $\alpha\beta$ pour des arbres de profondeur de 8 demi-coups. [11]

Nous avons résumé dans la table 4.1, l'effet de l'algorithme *Negascout* parallélisé par

10. L'algorithme *Jamboree*[92] allant même jusqu'à ne pas interrompre les frères de P_i mais plutôt, il attend leurs terminaisons avant de poursuivre avec l'exploration de P_i avec une nouvelle fenêtre.

11. Comme l'algorithme *Negascout* est moins performant que l'algorithme $\alpha\beta$ sur le programme d'Othello, nous n'avons pas jugé utile de faire figurer les résultats obtenus sur le programme d'Othello.

TAB. 4.1 - *Accélérations relatives des algorithmes* Negascout *et $\alpha\beta$ parallélisés pour le programme d'échecs par rapport à l'algorithme $\alpha\beta$ sur un processeur sur les arbres de profondeur de 7 et 8 demi-coups.*

nombre de processeurs	accélérations relatives par rapport à $\alpha\beta$ séquentiel				*Negascout* parallèle par rapport à $\alpha\beta$ parallèle	
	$\alpha\beta$ parallèle		*Negascout* parallèle			
	prof 7	prof 8	prof 7	prof 8	prof 7	prof 8
vol de tâches + droit d'aînesse (YBWC)						
1	1	1	1,35	1,52	1,35	1,52
8	5,71	6,38	7,72	7,91	1,35	1,24
16	9,89	10,58	11,95	13,53	1,20	1,28
24	13,06	14,05	14,65	18,46	1,12	1,31
32	14,69	18,43	16,46	21,78	1,12	1,18
$\alpha\beta$* **+ droit d'aînesse**						
1	1	1	1,35	1,52	1,35	1,52
8	6,79	6,42	7,70	8,99	1,13	1,40
16	11,29	11,60	13,35	15,22	1,18	1,31
24	13,89	15,51	17,63	19,97	1,26	1,29
32	15,29	18,36	19,83	23,97	1,30	1,31

rapport à l'algorithme $\alpha\beta$ sur le programme d'échecs. En comparant les accélérations relatives des algorithmes $\alpha\beta$ et *Negascout* parallélisé par rapport à l'algorithme $\alpha\beta$, nous voyons que pour le schéma de vol de tâche, l'algorithme *Negascout* est de moins en moins intéressant face à l'algorithme $\alpha\beta$ quand le nombre de processeurs augmente. Pour le schéma $\alpha\beta$*, au contraire le rapport est plus stable.

4.3.6 Effet de la simulation de la mémoire partagée

Dans les deux schémas de parallélisation sur les jeux réels, nous avons utilisé l'heuristique de la table de transposition pour bien ordonner les mouvements et dans le cas de l'$\alpha\beta$* pour connaître le degré de confluence à un nœud de l'arbre (c'est-à-dire le nombre de processeurs qui analysent simultanément un nœud donné). Or, nous avons vu que sur les jeux réels, le surcoût de communication est plus important, du fait des communications supplémentaires engendrées par la gestion de la table de transposition. Si, dans l'$\alpha\beta$*, il n'est pas possible de se passer de table de transposition car elle règle le parallélisme, dans le schéma de vol de tâches, elle n'est pas nécessaire. C'est pourquoi Brockington[36] a montré sur un BBN que le schéma distribué proposé par Feldmann[51], consistant à traiter les messages concernant la table de hachage une fois par itération, n'est pas toujours optimal dans l'application du YBWC. En effet, Brockington obtient une accélération relative légèrement supérieure avec des tables locales et non pas une table distribuée pour des arbres d'Othello.

Sur la CM5, nous avons géré la table de transposition à l'aide des messages actifs. Ce type de messages permet, entre autres choses, de simuler facilement et de manière efficace la mémoire partagée. La figure 4.15 montre les résultats obtenus avec la mémoire partagée simulée sans messages actifs comme dans le schéma proposé par Feldmann. De plus, dans cette expérience, les 32 processeurs de la CM5 servent tous pour la simulation de la mémoire partagée et non uniquement les processeurs actifs pour la recherche. Nous voyons que le surcoût de communication devient rapidement un facteur limitant et que finalement, sans messages actifs, il est préférable dans ce cas précis de ne laisser que 24 processeurs chargés de la recherche pour ne pas encombrer trop le réseau de communication.

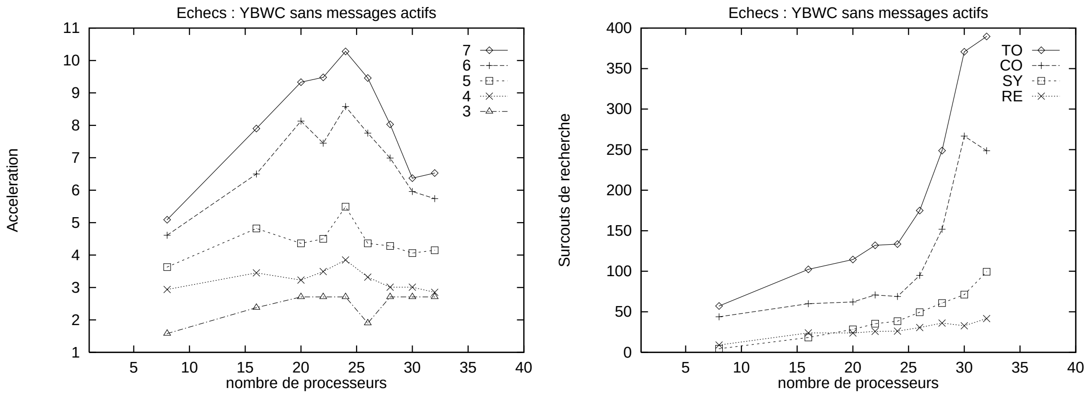


FIG. 4.15 - *Effet des messages actifs en « Young Brothers Wait Concept ». Algorithme $\alpha\beta$ Connection Machine de 32 processeurs.*

Cette précédente expérience a motivé les deux stratégies de simulation de la table de transposition partagée suivantes. On considère une machine de 32 processeurs dont P sont chargés de la recherche; nous définissons alors les deux stratégies suivantes :

1° La table de transposition est distribuée sur les $32 - P$ processeurs non chargés de la recherche;

2° La table de transposition est distribuée sur les 32 processeurs.

Les résultats que nous avons obtenus avec ces deux stratégie de distribution de la table de transposition sont dans la figure 4.16. Nous y voyons le fait que dans les deux stratégies, le surcoût de communication augmente avec le nombre de processeurs chargés de la recherche. Dans le cas du programme d'échecs, la meilleure solution est donc de prendre 30 processeurs dédiés à la recherche et 2 processeurs dédiés à la table de transposition[12].

4.3.7 Effet de la taille de la table de transposition

12. Pour le programme d'Othello, la meilleure solution est de distribuer la table de transposition sur les 32 processeurs.

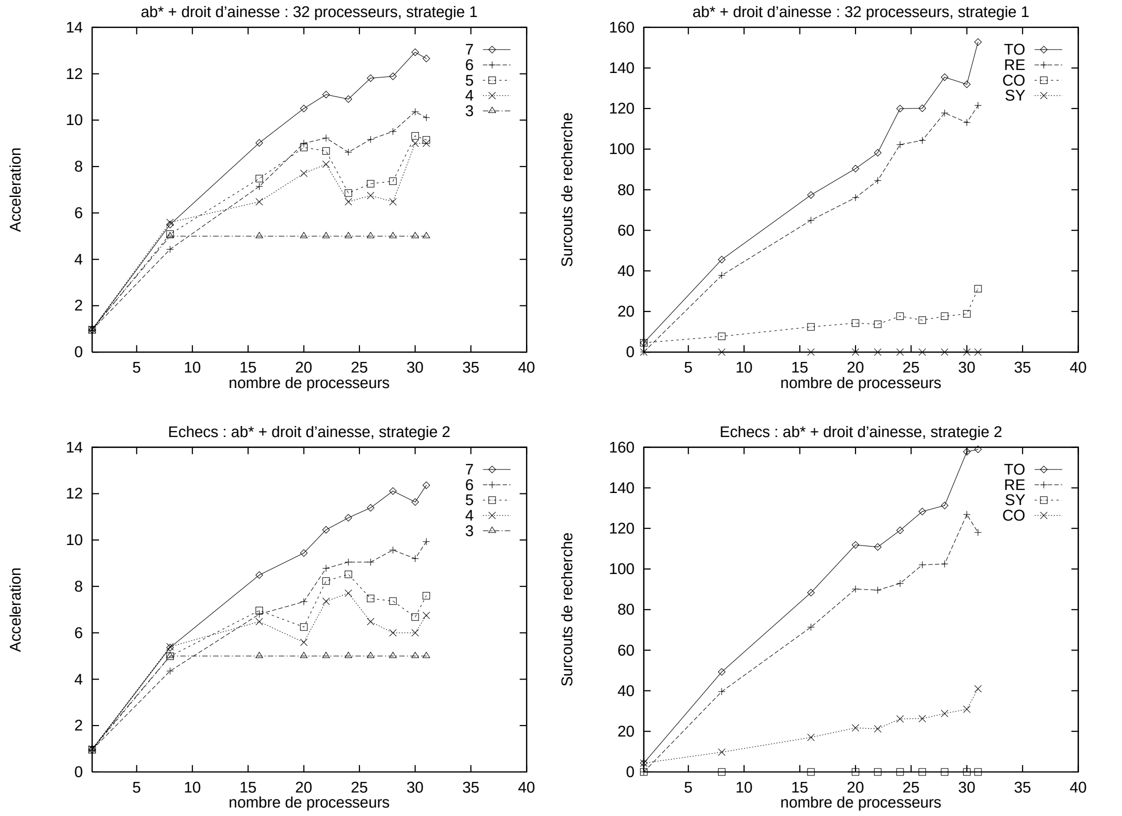


FIG. 4.16 - *Comparaison de deux stratégies de distribution de la table de transposition. Algorithme* Negascout, *Connection Machine de 32 processeurs.*

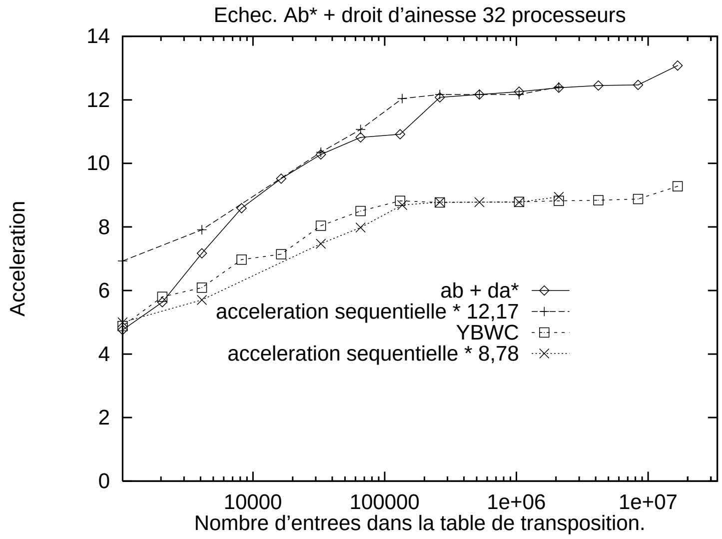


FIG. 4.17 - *Influence du nombre d'entrées dans la table de transposition sur les performances des schémas* YBWC *et $\alpha\beta$* avec droit d'aînesse.*

Algorithme Negascout, *Connection Machine de 32 processeurs. Les arbres sont de profondeurs de 7 demi-coups. Les accélérations sont relatives à la version pour un processeur avec 524288 entrées dans la table de transposition.*

Dans la figure 4.17, nous avons fait varier le nombre d'entrées dans la table de transposition partagée dans les schémas YBWC et $\alpha\beta$* avec droit d'aînesse. Nous comparons ces deux schémas directement à l'aide de l'accélération obtenue par le programme séquentiel du fait du changement du nombre d'entrées dans la table de transposition que nous avons multiplié avec l'accélération absolue des deux schémas obtenue pour une table de transposition de 524288 entrées.

Nous voyons que les accélérations obtenues pour les deux schémas parallèles en variant le nombre d'entrées dans la table de transposition sont identiques à celles du programme séquentiel sauf pour le schéma $\alpha\beta$* pour un nombre d'entrées inférieur à 16384, ce qui est particulièrement bas puisque le schéma $\alpha\beta$* utilise la table de transposition pour le contrôle du parallélisme spéculatif. Ainsi, le schéma YBWC devient le plus efficace pour un nombre d'entrées dans la table de transposition inférieur à 2048.

4.3.8 Effet de l'heuristique du mouvement nul

Pour analyser l'effet de la parallélisation sur certaines heuristiques, nous avons choisi d'étudier le comportement de l'heuristique du mouvement nul pour le programme d'échecs dans le modèle $\alpha\beta$* avec droit d'aînesse. Nous avons donc calculé l'accélération obtenue par rapport à la version sans l'heuristique dans le tableau 4.2. Nous voyons que l'effet de cette heuristique est atténué avec le nombre de processeurs.

TAB. 4.2 - *Effet de la parallélisation sur l'heuristique du mouvement nul*

nombre de processeurs	accélérations relatives sans mvt nul *Negascout*	avec mvt nul *Negascout*	Version avec mvt nul par rapport à version sans
1	1	1,89	1,89
8	5,26	8,61	1,63
16	8,29	12,30	1,48
20	9,72	13,78	1,42
24	10,63	14,64	1,38
32	11,77	15,85	1,35

4.4 Synthèse des résultats

Pour comparer les différentes décompositions d'arbres de jeu parallèles, il est nécessaire d'utiliser une référence commune qui est idéalement la meilleure implantation séquentielle pour pouvoir calculer une accélération absolue. Dans le cas du programme d'échecs, nous choisissons donc le programme séquentiel qui utilise le *Negascout* récursif, pour le programme d'Othello, au contraire, nous utilisons l'$\alpha\beta$ récursif.

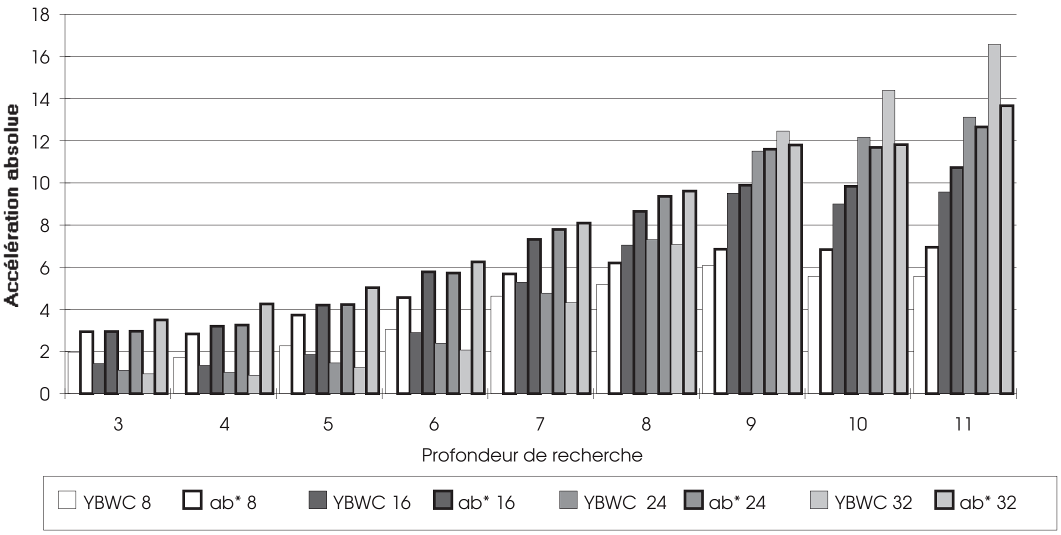


FIG. 4.18 - *Comparaison en accélération absolue des deux méthodes de décomposition de l'arbre de recherche pour le programme d'Othello.*

La figure 4.18 nous résume les résultats pour le programme d'Othello. Nous voyons dans cette figure qu'en dessous d'une certaine taille du problème pour des profondeurs de recherche inférieures à 8 demi-coups, la décomposition $\alpha\beta$* est plus efficace que la décomposition vol de tâches, mais celle-ci devient plus efficace pour les profondeurs supérieures.

Dans la figure 4.19, nous voyons que dans le cas de notre programme d'échecs, la décomposition $\alpha\beta$* donne de meilleurs résultats que la décomposition YBWC, pour des problèmes de tailles identiques à ceux rencontrés en temps de tournoi. Les différences de résultats pour les deux programmes de jeux peuvent s'expliquer par le fait que :

1° Les arbres de jeu ne sont pas de la même taille : le facteur de branchement pour un arbre d'Othello de milieu de partie est de 10 et celui d'un arbre d'échecs est de 40. Les arbres d'Othello étant plus profonds du fait du plus faible facteur de branchement;

2° La fonction d'évaluation du programme d'Othello étant peu stable, les arbres d'Othello sont moins bien ordonnés que ceux du programme d'échecs. Ce qui, nous l'avons vu, rend l'algorithme *Negascout* et le schéma de parallélisation YBWC moins efficaces.

Ces différences se traduisent pour le schéma YBWC par de plus grands surcoûts de synchronisation et de recherche pour le jeu d'échecs par rapport aux surcoûts constatés pour le jeu d'Othello. Le schéma $\alpha\beta$* permet de supprimer le surcoût de synchronisation qui est remplacé par de plus grands surcoûts de communication (engendrés par les contentions d'accès à la table de transposition partagée) et de recherche. Nous expliquons l'augmentation du surcoût de recherche sur l'exemple suivant. Considérons des processus P_1, P_2, ..., P_n analysant les mouvements issus d'une position $\mathcal{P}$ (Le contrôle spéculatif du droit d'aînesse étant appliqué, le premier mouvement de cette position a déjà été analysé). Supposons alors que

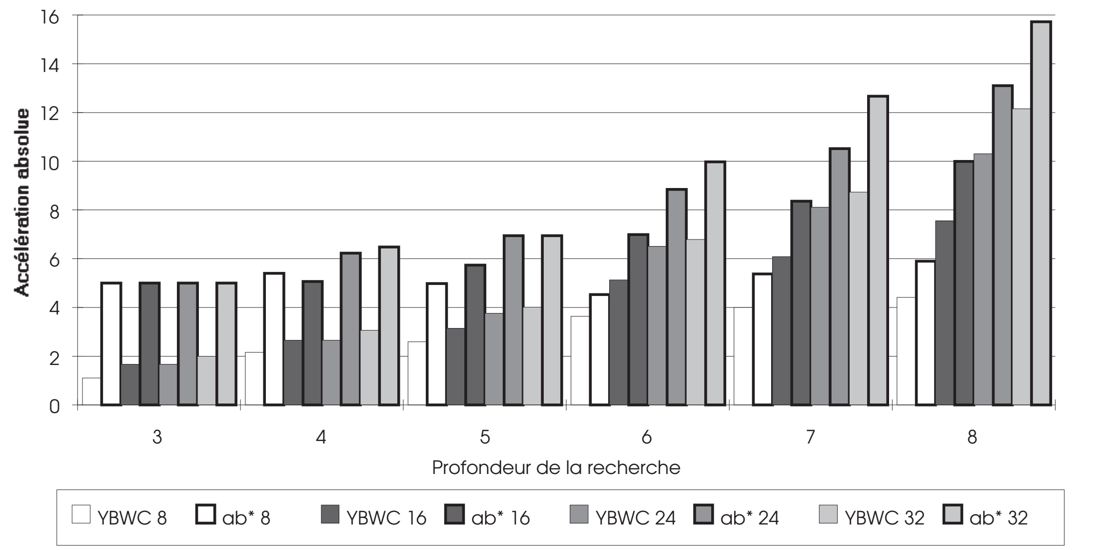


FIG. 4.19 - *Comparaison en accélération absolue des deux méthodes de décomposition de l'arbre de recherche pour le programme d'échecs.*

l'un des processus P_i se termine avec une valeur provoquant l'élagage de ses frères, le schéma YBWC répercute cet élagage sur les processus P_j pour $j \neq i$ alors que le schéma $\alpha\beta$* laisse les processus finir leur travail. Ceci se produit uniquement si le premier mouvement n'est pas le meilleur mouvement depuis la position $\mathcal{P}$ car sinon le premier mouvement aurait lui même provoqué un élagage. Les arbres d'Othello étant moins bien ordonnés que les arbres d'échecs, ce cas s'y produit plus fréquemment. À Othello, l'augmentation de ces surcoûts n'est pas compensée par la perte du faible surcoût de synchronisation alors qu'aux échecs la compensation est réelle [13].

13. De plus, il ne faut pas oublier que le schéma YBWC est pénalisé par la dérécursification de l'$\alpha\beta$ ou du *Negascout*.

De tous ces résultats, nous pouvons tirer les conditions d'application de ces deux décompositions quand elles sont alliées à un contrôle du parallélisme spéculatif du type droit d'aînesse.

vol de tâches :

Avantages :
- Fonctionne sans table de transposition;
- Fonctionne avec des tables de transposition locales;
- Efficace pour des problèmes de grande taille;
- Optimisable suivant la topologie de la machine cible[51].

Inconvénients :
- Relativement complexe à implanter : requiert de dérécursifier l'algorithme negamax de recherche, d'où une difficulté supplémentaire pour implanter les heuristiques de recherche.
- Peu ou pas efficace pour les problèmes de petite taille.

$\alpha\beta$* :

Avantages :
- Facile à implanter : l'algorithme est le même que celui utilisé pour les programmes séquentiels, donc peu de modifications sont nécessaires aux programmes existants (pour les algorithmes, mais aussi pour les heuristiques).
- Efficace pour les problèmes de petite et grande taille.
- Résistant aux pannes : si un processeur ne répond pas, cela n'influe pas le résultat des autres processeurs.

Inconvénients :
- Requiert une table de transposition à accès très rapide.

4.5 Conclusion

Nous avons donné une nouvelle méthode de parallélisation du calcul de la valeur negamax d'un arbre de jeu : la méthode de décomposition de l'arbre de recherche $\alpha\beta$* alliée au droit d'aînesse. Cette méthode s'est montrée plus rapide que la méthode YBWC proposée par Feldmann[51] sur des programmes d'échecs.

Nous avons montré que les comparaisons d'algorithmes parallèles établies sur un problème donné et un nombre de processeurs donné ne valent que sur les conditions d'expérimentation. Ce qui nous semble évident, mais semble novateur dans le domaine de ces algorithmes. En effet, jusqu'à présent dans la littérature, il était considéré que dès que l'on donnait une bonne accélération relative sur un problème donné, le résultat était généralisable sur l'ensemble des recherches minimax. Nos expériences montrent le contraire.

De plus, il faut noter que les résultats que nous avons obtenus sur les diverses classes d'algorithmes sont bien inférieurs aux résultats obtenus par Feldmann[51] et David[45] sur des machines à bases de transputers. Ceci est très certainement dû à plusieurs raisons :

- Ces auteurs considèrent des accélérations relatives ou font apparaître des coupures heuristiques qui perturbent les résultats;

- le processeur de base de leurs machines, le transputer, est 10 fois plus lent que le processeur de base des CM5. Donc les temps de communications deviennent négligeables en face des temps de calcul.

Chapitre 5

Évaluation et Apprentissage

Dans ce chapitre, nous abordons les problèmes relatifs à l'évaluation d'une position. En outre, nous présentons la méthode de construction de fonctions d'évaluation que nous avons choisie avec Marc-François Baudot, en montrant comment nous avons pu introduire la notion de plan stratégique[1]. Nous décrivons aussi le langage CHEVAL qui a été défini en collaboration avec l'aide de Marc-François Baudot et Jean-Luc Seret pour l'écriture de fonctions d'évaluation aux échecs[2].

Nous montrons aussi la problématique particulière des fins de parties à matériel réduit.

Enfin, nous donnons l'essentiel de l'état des techniques relatives à la synthèse automatique de fonctions d'évaluation et les conclusions de nos expériences dans la construction d'une fonction d'évaluation correcte pour la fin de partie roi et dame contre roi et dame[3].

5.1 Fonctions d'évaluation

5.1.1 Fonctions d'évaluation linéaires et non linéaires

Dans le jeu d'échecs, il a été observé que le joueur qui a obtenu un avantage matériel tend à gagner la partie. Une des premières notions données aux joueurs débutants est la valeur matérielle des pièces : 1 pour un pion, 3 pour un cavalier ou un fou, 5 pour une tour et 9 pour une dame. Le total du matériel pour un joueur fait partie de ce qui sera nommé dans la suite « un critère ». D'autres critères dans le jeu d'échecs sont par exemple la mobilité des pièces (c'est à dire le nombre de mouvements possibles par pièce), le contrôle du centre et le développement des pièces.

Historiquement, les fonctions d'évaluation étaient mises sous la forme d'une combinaison

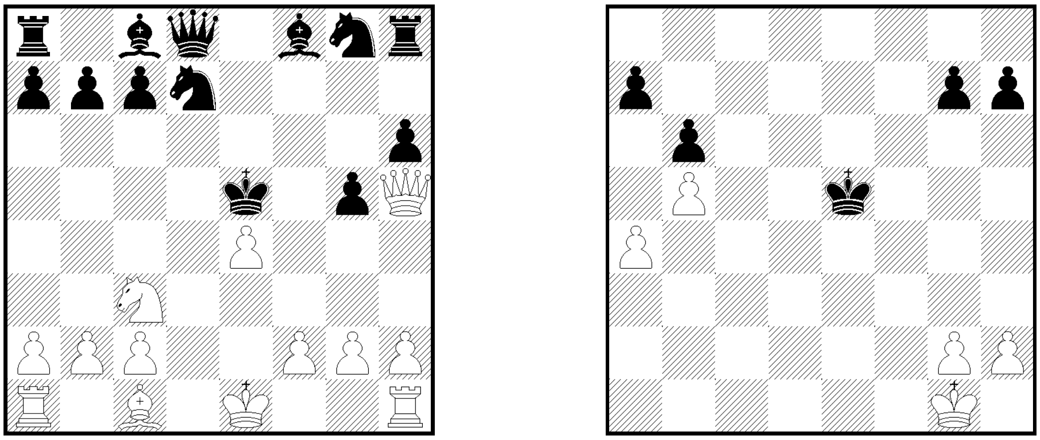
1. Dans la section 5.1.3, nous montrons comment à partir de la notion d'Oracle définie par Berliner, nous introduisons les plans stratégique.
2. Au § 5.1.4.
3. En utilisant les algorithmes ID3, ID5R, IDL, dans le § 5.2.3

linéaire :

$$f = \sum_{c \in C} A_c * c$$

où C est l'ensemble des critères utilisés et les A_c représentent la pondération du critère c.

Mais les importances relatives de certains critères ne sont pas les mêmes selon les différentes phases de la partie, ou plus généralement selon les différentes classes de positions. Dans les diagrammes 5.1, nous voyons que le critère de placement du roi change selon la phase de la partie aux échecs. Dans le premier, il s'agit d'une position de milieu de partie et le roi noir est trop exposé au centre de l'échiquier ; au contraire, dans le deuxième diagramme le roi noir est très bien placé car il s'agit d'une position de finale où le roi devient une pièce active.



E. Vassioukov – Lebedev, Moscou 1963 Aux blancs de jouer. Le ♔ noir est très mal placé (Les blancs gagnent par 10. ♗×g5!).

Le ♔ noir est très bien placé.

Diagramme 5.1: *Illustration du changement de la valeur des critères selon les classes de positions.*

D'où la nécessité de construire des fonctions d'évaluation différentes selon les classes, mais ce n'est pas sans apporter des problèmes de transitions.

Cela se produit si nous considérons les positions situées à la frontière de deux classes C_1 et C_2 et si l'évaluation est très favorable selon la classe C_1 et défavorable selon la classe C_2. Pour les positions situées sur la frontière, un programme sera poussé à jouer de façon à mener la position dans la classe C_1 et à y rester le plus longtemps possible, pour ne pas traverser la frontière qui conduit à la classe C_2. Cet effet, dont le comportement est proche de « l'effet horizon », a été nommé par Berliner « the blemish effect »[21].

Il est clair que même pour des positions se trouvant dans une même classe, les combinaisons linéaires ne permettent que de juger des importances moyennes des différents critères et qu'il est souhaitable d'introduire une non-linéarité. Mais, comme l'a montré Berliner[21], ceci peut introduire deux nouveaux effets indésirables : une plus grande instabilité et « l'effet suicide ». En effet, si nous considérons que deux critères sont compris entre 0 et 50, alors leur produit sera compris entre 0 et 2500 : la non-linéarité engendre une plus grande instabilité. De plus, si nous nommons I un critère mesurant l'intensité d'une douleur et D un critère mesurant la durée de la douleur, alors la douleur totale est de $I \times D$. Si ce terme apparaît dans une fonction d'évaluation, le programme, pour minimiser $I \times D$, peut choisir la solution de minimiser D, en le mettant à zéro par exemple ce qui correspond à un suicide. Ainsi un programme peut choisir de perdre rapidement plutôt que de perdre de beaucoup, c'est « l'effet suicide ».

Pour éviter ces problèmes Berliner[21] remplace les coefficients constants A_c par des variables spéciales à variations lentes ; ces nouveaux coefficients déterminent à eux seuls les classes de parties donc les transitions; il les appelle les « coefficients d'applicabilité ». Comme ces coefficients d'applicabilité varient lentement et sans à-coups, ils peuvent tenir compte du contexte tout en évitant les frontières abruptes entre des positions différentes. Cette approche du problème est nommée la méthode SNAC (Smoothness, Non-linearity, Application Coefficients) et a été utilisée avec succès dans le programme de Backgammon BKG[20].

5.1.2 Le compromis : Vitesse contre Connaissances

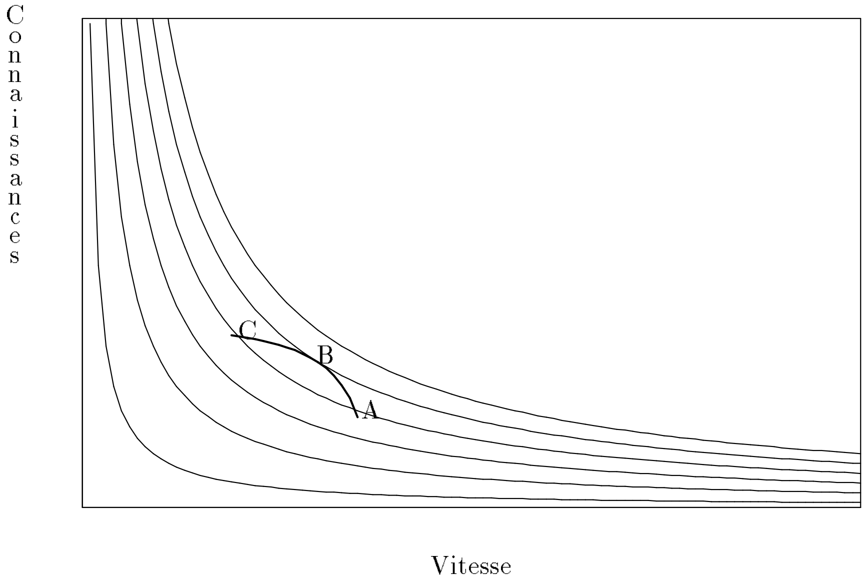


FIG. 5.1 - *Effet de la quantité de connaissances et de la vitesse sur le niveau de jeu du programme.*

Les courbes représentent un niveau de jeu constant, la qualité du jeu étant plus élevée pour les courbes les plus éloignées de l'origine.

Un des compromis importants à trouver lors de la programmation d'un jeu de réflexion est le rapport entre les connaissances contenues dans la fonction d'évaluation et la vitesse de calcul de cette fonction. En effet, si la fonction d'évaluation est trop lente, la vitesse de la recherche s'en trouve amoindrie et la qualité du jeu tactique devient plus faible. Au contraire, une évaluation trop rapide sera synonyme de jeu positionnel faible. Ceci est illustré par la figure 5.1 : tout ajout de connaissances dans la fonction d'évaluation correspond à une perte de vitesse. La courbe ABC illustre ce qui se passe quand on ajoute des connaissances à une fonction d'évaluation : le programme est au point A, et le but est d'implanter une nouvelle connaissance. Une première approximation de cette connaissance est implantée rapidement et nous mène au point B ; cette approximation n'est pas parfaite et ne traite qu'une faible partie de ce que l'on cherche à ajouter. Dans un deuxième temps, on implante la partie restante de la connaissance, ce qui nous mène au point C. Dans cet exemple, il est clair que l'approximation de la connaissance est le meilleur choix, et donc que, bien souvent, il est préférable de donner seulement une idée de la connaissance à ajouter plutôt que d'implanter la connaissance entièrement. Le problème résultant est donc de trouver un schéma qui va nous permettre d'augmenter les connaissances sans diminuer la vitesse.

Une des premières approches possible, que nous ne considérerons pas ici, est de paralléliser l'évaluation comme dans le cas du programme BELLE[41]. L'autre approche que nous avons utilisée avec succès dans nos programmes d'échecs est l'approche de type « Oracle »[49, 24, 185].

5.1.3 Fonction d'évaluation par Oracle

L'approche de type « Oracle »[49, 24, 185] va nous permettre d'ajouter des connaissances sans nuire à la rapidité du programme, la vitesse obtenue par cette approche étant indépendante de la quantité et du raffinement des connaissances. Nous définissons une table de valeurs positionnelles $\mathcal{T}$, indexée par les pièces, les couleurs et les cases, représentant la valeur positionnelle d'une pièce d'une couleur donnée sur une case. Par exemple $\mathcal{T}$[♘][blanc][f3] représente la valeur d'un cavalier blanc sur la case f3. L'évaluation d'une position est alors définie par :

$$f(P) = \sum_{case} \mathcal{T}[\text{pièce}(P, case)][\text{couleur}(P, case)][case]$$

Si, depuis la position P, les blancs jouent le mouvement ♘f3–e5,

on peut déduire la valeur de la position résultante $P_1 = P \bullet$ ♘f3-e5 par

$$f(P_1) = f(P) + \mathcal{T}[\text{♘}][\text{blanc}][\text{e5}] - \mathcal{T}[\text{♘}][\text{blanc}][\text{f3}]$$

.

L'évaluation est donc mise à jour incrémentalement pendant l'exploration de l'arbre de recherche, en utilisant uniquement quelques additions et soustractions de valeurs de la table positionnelle $\mathcal{T}$, et son coût est alors négligeable. Le remplissage judicieux de la table nous assure la qualité de la fonction d'évaluation : c'est le rôle de l'Oracle. Il intervient une seule fois, avant la recherche de la valeur negamax de la position. Il peut donc se livrer à une analyse très fine de la position, en tout cas plus fine qu'une fonction d'évaluation standard qui serait appelée des millions de fois pendant le processus de recherche.

Néanmoins, il est clair que l'on ne peut pas représenter toutes les connaissances sous forme de table, c'est pourquoi nous sommes amenés à définir des notions de complexité d'évaluation.

5.1.3.1 Complexité d'évaluation

Définition 5.1.1 *Un critère c est un* **critère du premier ordre** *si et seulement si il existe une fonction $\mathcal{G}_c$ telle que*

$$c(P) = \sum_{case} \mathcal{G}_c[pi\grave{e}ce(P, case)][couleur(P, case)][case]$$

où case parcourt l'ensemble des cases de l'échiquier.

La valeur d'un critère du premier ordre peut être calculée incrémentalement pendant la recherche. Si, dans la position S_1, le joueur de couleur j joue la pièce p de la case k à la case l pour obtenir la position S_2, alors :

$$c(S_2) = c(S_1) - \mathcal{G}_c[p][j][k] + \mathcal{G}_c[p][j][l]$$

.

Nous pouvons donner des exemples de critères positionnels du premier ordre :

Le matériel : Il suffit de parcourir l'échiquier et de compter la valeur des pièces présentes sur chaque case;

le contrôle du centre par les cavaliers et par les pions : Les cases contrôlées par un cavalier, un pion (ou un roi) sont uniquement fonction de la case sur laquelle se trouve le cavalier ou le pion, et le centre de l'échiquier est défini de manière indépendante de la position des pièces;

l'occupation de la septième traverse : les cases constituant la septième traverse (la ligne 7 pour les blancs, la ligne 2 pour les noirs) sont connues indépendamment de la position des pièces. La valeur du critère est la somme des pièces lourdes (dames et tours) qui s'y trouvent.

Si une fonction d'évaluation est définie comme combinaison linéaire de critères du premier ordre, alors cette fonction d'évaluation est elle-même du premier ordre, donc calculable à l'aide d'une table positionnelle. De plus, l'évaluation calculée à l'aide d'une table positionnelle nous définit un critère du premier ordre.

Définition 5.1.2 *Nous appelons* **critère du second ordre** *un critère dont le calcul fait intervenir les relations entre deux cases (ou plus).*

De tels critères ne peuvent pas se calculer incrémentalement à l'aide d'un tableau positionnel. En voici quelques exemples :

La structure des pions : les notions de pions doublés, passés, etc. sont définies par les relations entre les pions situés sur des colonnes voisines ;

les tours doublées sur une colonne ouverte : la notion de colonne ouverte repose sur la présence de pions sur la colonne, et la notion de tours doublées implique de s'intéresser à deux pièces simultanément;

la mobilité : le nombre de mouvements possibles dépend des interactions entre les pièces.

la sécurité du roi : dépend de la position du roi, de ses pièces et des pièces adverses;

la paire de fous.

Dans la pratique, l'évaluation fine réalisée par l'Oracle permettra d'approcher des critères du second ordre par un critère du premier ordre. Mais dans certains cas, cette approximation est impossible. Dans la définition de la fonction d'évaluation de nos programmes d'échecs, nous avons eu les dilemmes suivants :

- On peut supposer qu'une colonne ouverte a de grandes chances de le rester, et qu'il est possible de prévoir quelles colonnes peuvent le devenir ; comme encourager une tour à occuper cette colonne va nécessairement encourager les deux tours à y venir, il n'est pas trop gênant d'utiliser un tableau positionnel pour encourager le programme à doubler ses tours sur les colonnes ouvertes. La perte du bonus dû au doublement (impossible à détecter autrement que par une évaluation du second ordre) sera le prix à payer pour obtenir un gain en vitesse.
- Pour encourager le programme à garder la paire de fous dans les positions ouvertes, l'Oracle devra attribuer à chacun un gros bonus ; la perte d'un des deux fous entraînera la disparition de seulement la moitié du bonus, alors que la paire de fous aura complètement disparu : cette approche est très problématique pour une gestion correcte des échanges matériels.
- L'apparition d'un pion passé ou d'un pion doublé est impossible à détecter par une évaluation du premier ordre, et leur importance dans la conduite d'une partie est bien trop grande pour qu'un programme de niveau décent puisse utiliser des approximations dans ce domaine.

Dans tous ces cas, l'utilisation d'une évaluation dynamique sera d'un grand bénéfice pour le niveau de jeu. L'évaluation dynamique de critères positionnels n'exclut pas l'utilisation d'un Oracle. On peut, par exemple, laisser à l'Oracle la responsabilité du choix de la valeur à attribuer à la paire de fous. Cette approximation est justifiée par le coût prohibitif du calcul dynamique du degré d'ouverture de la position et de son potentiel à s'ouvrir.

De la même façon, nous avons choisi de laisser l'Oracle attribuer leur valeur aux pions passés. Plus un pion passé est sur une colonne éloignée du roi adverse, plus grande est sa valeur. Compte tenu de la mobilité réduite du roi, il n'est pas déraisonnable de ne tenir compte que de la position du roi à la racine, ce qui simplifie grandement l'évaluation de la structure de pions (cette approche n'est pas possible lorsqu'il s'agit d'appliquer la règle du carré, dans laquelle la position exacte du roi est importante : l'évaluation doit alors être entièrement dynamique).

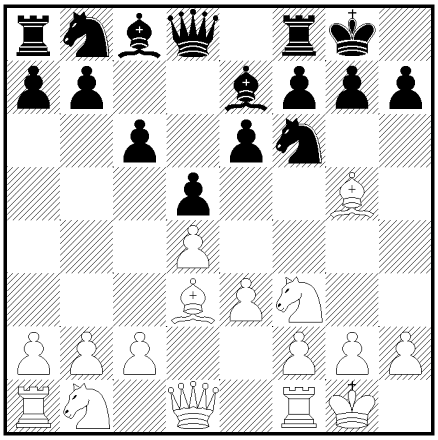


Diagramme 5.2: *Évaluation du second ordre non triviale*

Donnons enfin un dernier exemple de critère du second ordre qui, s'il paraît simple à ramener à du premier ordre, a nécessité l'utilisation d'une évaluation du second ordre pour donner des résultats satisfaisants dans le programme ECUME : dans le position du diagramme 5.2, il semble facile de dire au programme de pousser le pion c et de mettre le cavalier en c3 à l'aide d'un tableau positionnel, mais les pondérations correctes sont délicates à trouver. Dans la pratique, le bonus attribué au cavalier en c3 doit être suffisant pour éviter le mouvement passif ♘d2, mais pas trop grand, de manière à ce que le programme joue d'abord la poussée c2c4. Si la poussée n'est pas possible immédiatement, il faut la préparer en décourageant les mouvements ♘c3 (qui rend la poussée irréalisable) et ♘d2 (car le cavalier sera mieux placé en c3, une fois la poussée c4 effectuée) : on est conduit à attribuer un malus aux mouvements ♘c3 et ♘d2. L'interaction avec des évaluations dynamiques (principalement la mobilité) rend très difficile le jeu sur les coefficients positionnels, et fréquemment ECUME avait tendance à jouer c2c3 et ♘d2, au lieu de préparer c2c4 puis de jouer ♘c3, transformant une position active (bonne pour un programme) en position solide mais passive. Le problème fut résolu en encourageant à la fois la poussée c2c4 et le développement ♘c3 par le truchement du tableau positionnel, et en reconnaissant dynamiquement la position ♘c3 c2 (second ordre) comme très mauvaise, et la position ♘c3 c4 comme bonne (les valeurs correspondantes sont déterminées par l'Oracle, en fonction de la position racine).

Le choix des évaluations que l'on peut assimiler à du premier ordre et de celles qu'il faut calculer dynamiquement dépendra entre autres choses de la rapidité de la machine sur laquelle tournera le programme.

5.1.3.2 Planification introduite par l'Oracle

Un joueur humain établit un plan en examinant la position sur l'échiquier. Il analyse des variantes pour rechercher d'éventuelles combinaisons, et surtout pour vérifier l'applicabilité de la stratégie qu'il vient de choisir. Il s'efforce d'évaluer les positions finales des variantes qu'il analyse (une des grandes difficultés) du mieux possible, avec pour critère principal la progression dans la voie de la réalisation de son plan. L'approche par Oracle procède d'une démarche similaire. L'oracle peut, à l'aide de considérations stratégiques, choisir un plan. Par exemple, si le programme a l'avantage d'espace à l'aile roi et au centre, une avance de développement et que le centre est fermé, un plan possible est l'attaque à l'aile roi. L'oracle, qui a tout le temps de se livrer à une analyse stratégique fine de la position, va ainsi choisir un des nombreux plans qu'il a en bibliothèque. Il va jouer sur les valeurs de la table positionnelle pour encourager les pièces à se masser à l'aile roi et les pions à y avancer, et le programme aura un comportement conforme aux exigences stratégiques de la position. Dans les positions où l'oracle ne sait pas appliquer de plan tout fait, sa connaissance des structures de pions lui permettra d'adopter un placement des pièces judicieux.

Donnons un exemple, simplifié à l'extrême, où l'application d'un plan grâce à l'approche Oracle combiné avec une table positionnelle est possible.

L'idée de la table positionnelle devient naturelle dès que l'on s'efforce d'obtenir d'un programme un comportement décent dans les fins de partie. Dans la plupart des fins de parties, les joueurs humains savent identifier des objectifs à court ou moyen terme, et établir des plans ou des stratégies relativement directs. C'est le domaine de la technique.

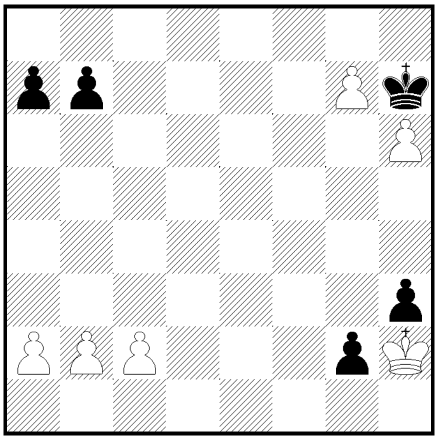


Diagramme 5.3: *Une position très simple pour un humain et pour un Oracle.*

Dans la position du diagramme 5.3, la création d'un pion passé – l'un des principaux objectifs en finale – ne pose aucune difficulté à un joueur humain : 1. c4 a5 (sinon 2. b4 et le gain blanc est encore plus rapide) 2. c5 ♔g8 3. b3 (3. a3? a4! =) ♔h7 4. a3 ♔g8 5. b4 ab 6. ab ♔h7 7. b5 ♔g8 8. c6 bc 9. bc. Dans toutes les positions de ce type, un joueur humain trouvera le bon plan et saura le mener à bien en un clin d'oeil. Or il a fallu 17 demi-coups pour voir apparaître le pion passé dans cette position, qui est pourtant simplifiée à l'extrême. La position demande à être jouée avec précision, car le malheureux 1. b4?? b5! = gâche toute chance de gain. C'est une position typique où ce qui semble être un calcul tactique trivial (en fait ce calcul est la vérification analytique des connaissances techniques du joueur) demande une trop grande profondeur d'analyse pour la plupart des programmes en temps de tournoi. Les connaissances que mettront vraisemblablement en oeuvre les joueurs face à cette position sont données par Kmoch[86] et avant lui par Nimzowitch[121]:

- Candidat en avant !
- Les acolytes doivent venir à l'aide du candidat quand celui ci ne peut plus avancer davantage.

Le candidat est le pion qui donnera naissance au pion passé (c2), les acolytes sont les pions qui vont l'aider dans sa progression (b2). Un candidat se trouve nécessairement sur une colonne semi ouverte, mais cela n'est pas suffisant pour être un vrai candidat : il faut que ses acolytes soient en mesure de lui faire franchir le barrage des sentinelles adverses (b7). Il est aisé d'obtenir ce comportement à l'aide d'un tableau positionnel, en jouant sur les malus et les bonus des pions. Dans la position du diagramme, on donnera un malus à un pion blanc b4 et un bonus à un pion blanc c4, c5, c6, c7. Dans les positions suivantes, lorsque le candidat aura avancé, un pion blanc b4 recevra un bonus. L'écriture d'une fonction d'évaluation dynamique est également possible, mais outre qu'elle est plus complexe et plus coûteuse en temps de calcul, elle ne permet pas de « forcer » le comportement du programme comme le permet l'approche par tableau positionnel. En effet, dans la position du diagramme, le bonus attribué à la poussée c2-c4 pourra être très élevé, le reste du tableau étant à 0, sans grand inconvénient : on a la certitude qu'une fois le mouvement c4 joué le tableau sera à nouveau rempli judicieusement par une analyse poussée de la position (Oracle). Cela permet de ne pas mélanger les plans, d'être sûr qu'ils sont exécutés et qu'ils le sont dans le bon ordre. Même si l'évaluation retournée est peu précise, une telle approche permet de jouer le bon mouvement. Une fonction d'évaluation dynamique est plus délicate à pondérer ; la gradation des notes à donner pour encourager l'avance du pion c est délicate, car la même fonction va devoir comparer la succession des positions jusqu'à la création du pion passé, tout en retournant des évaluations précises (elles doivent être précises à cause du blemish effect).

Dans la position du diagramme, les choses étaient simples, mais la présence d'acolytes et de sentinelles doublés rend le travail de reconnaissance des vrais candidats très coûteux à faire dynamiquement. Généralement, on se trouvera en présence de plusieurs pions qui ont des allures de pion candidat de part et d'autre. Il sera essentiel de savoir discriminer les vrais candidats pour :

- savoir quels pions adverses le roi doit surveiller (ce qui va peut-être l'immobiliser) ;
- savoir quels pions pousser, sachant que la perte d'un temps peut être fatale.

Un Oracle qui n'analysera que la position racine aura tout son temps et saura exactement quels pions sont les vrais candidats. Il lui sera aisé de dicter au roi sa zone de manoeuvre, et d'indiquer aux pions, à l'aide d'un tableau positionel, lesquels d'entre-eux doivent avancer.

5.1.3.3 Avantages et Inconvénients de l'approche Oracle

Avantages Outre le gain en vitesse, cette approche présente plusieurs avantages importants : le nombre de connaissances que l'on peut mettre dans un oracle est pratiquement illimité, et leur représentation n'est plus contrainte par des soucis d'efficacité d'exécution car un oracle est plus aisé à mettre au point qu'une évaluation standard dynamique. Un expert du domaine pourra facilement intervenir dans l'écriture d'un Oracle et enrichir sa bibliothèque de structures de pions et de plans standard associés. Quand il ne trouve pas de plan tout fait, l'Oracle sait produire un jeu positionnel décent car il connaît les manœuvres standard pour exploiter les colonnes semi-ouvertes, les chaînes de pions, les roques opposés, etc.

Il sait également déterminer les rapports de forces dans les diverses zones de l'échiquier : avantage d'espace, de pression, et en déduire des objectifs[4].

La classe à laquelle vont appartenir les positions rencontrées pendant la recherche est déterminée par l'Oracle, et ne va pas varier au cours de la recherche, donc l'approche n'est pas sujette au « blemish effect ».

Inconvénients L'incapacité du tableau positionnel à gérer les évaluations du second ordre impose de recourir en partie à des évaluations dynamiques. Ces évaluations pourront être choisies et paramètrées par l'oracle avant la recherche, ce qui permettra souvent de limiter leur coût.

Par construction, l'Oracle est incapable de détecter les changements de phase au cours

4. L'usage d'un Oracle peut aussi se justifier par la littérature échiquéenne : Aaron Nimzowitch, dans ses théories de la sur-protection et de la prophylaxie[120, 122], explique que

> «*le contact établi entre le point fort et la pièce qui le sur-protège augmente la valeur de l'un et de l'autre.[...] Bénéfice pour le "sur-protecteur" car le point est pour lui une source d'énergie d'où il peut sans cesse puiser de nouvelles forces*» [120]

A propos du mouvement 14. c3 de sa partie contre Capablanca, New York 1927, Nimzovitch fait le commentaire suivant :

> «*With this move begins the over protection of the point Q4, which fully deserves this concentrated attention, for this is a centrally situated blockade square and, be it noted, a pawn stood there once, which formed the basis of a proud chain of pawns. And even if the pawn at Q4 has disappeared long ago, and furthermore, if the pawn chain is now based on KB4, nevertheless the glamour and the immense importance of Q4 are practically unchanged.*»[122]

Ainsi, pour Nimzovitch, les modifications positionnelles à court et même moyen terme ne changent pas la valeur des points stratégiques fondamentaux de la position, ni l'importance qu'il y a à les contrôler. L'analyse par un oracle stratégique (en ne perdant pas de vue les théories de Nimzovitch) et le placement des pièces par un tableau positionnel s'en trouvent justifiés : les points stratégiques reconnus à la racine par l'oracle resteront le plus souvent des points stratégiques dans les variantes explorées par la recherche tactique, même en cas d'altération de la structure de pions ou de changement de phase, et les notes du tableau positionnel continueront d'induire un placement judicieux des pièces.

de la recherche. Il évite le « blemish effect », mais peut se tromper de classe de position. S'il considère la position comme une position de milieu de partie, mais qu'un échange de dames est possible immédiatement, faisant passer en finale, alors toutes les positions issues de cette branche de la recherche seront évaluées à tort comme des positions de milieu de partie. Les transitions de phase constituent la grosse faiblesse de l'approche par Oracle. Cet effet se manifeste de façon très visible au cours d'une partie. Après chaque transition de phase ou simplement après des échanges ou altérations importantes de la structure de pions, l'évaluation de la position à la racine (et donc le score résultant de la recherche) va connaître une variation importante (généralement une chute, le programme s'étant efforcé de maximiser son score pour l'évaluation utilisée lors des mouvements précédents et non pour celle qui est utilisée après la transition).

5.1.4 CHEVAL : un langage pour l'écriture de fonctions d'évaluation

La fonction d'évaluation d'un programme d'échecs est longue à écrire et sa mise au point requiert un travail perpétuel. L'utilisation des langages de programmation classiques est peu adaptée à l'écriture et à la maintenance d'une telle fonction; elle rend de plus leur compréhension difficile et est un obstacle à l'intervention efficace d'un expert du jeu d'échecs. Dans le cadre du projet FRENCHESS, nous avons défini et réalisé une première version d'un langage d'écriture de fonctions d'évaluation du jeu d'échecs : le langage CHEVAL (CHess EVAluation Language). Il est fondé sur l'expérience acquise lors de l'écriture de nos programmes.

Cette première version du langage CHEVAL permet au joueur d'échecs d'exprimer son savoir d'une manière proche du vocabulaire échiquéen standard, et à la machine d'utiliser efficacement ce savoir.

5.1.4.1 Finalité du langage CHEVAL

Le langage de description de fonctions d'évaluation CHEVAL doit faciliter l'écriture, la lecture, l'évolution et la maintenance des fonctions d'évaluation, et surtout leur compréhension par des gens qui n'ont pas participé à leur élaboration de a à z, facilitant ainsi le travail en équipe. Il devra aussi fournir des aides au test et au déverminage.

Le langage CHEVAL doit être indépendant des choix d'implantation de la fonction d'évaluation : il saura aussi bien se mettre au service de l'approche standard dynamique que de l'approche par Oracle et tableau positionnel.

5.1.4.2 Ce que l'on peut exprimer en CHEVAL

CHEVAL connaît un certain nombre de définitions et caractéristiques échiquéennes de base, telles les notions de case, de pièces, de couleurs de pièces et de cases, de centre, de colonne, de rangée, de diagonale, d'aile dame, de bélier[5], de levier[6] etc. Ces définitions peuvent être facilement enrichies. Pour les manipuler, CHEVAL fournit des variables de différents types (entier, booléen, pièce ou encore région de l'échiquier), des opérations arithmétiques et ensemblistes, ainsi que les principales structures de contrôle classiques des langages de programmation. CHEVAL permet également de définir des fonctions et des structures – Pattern : le langage ayant été conçu dans le souci de la plus grande diffusion possible – ce sont des condition à vérifier dans une région de l'échiquier.

Enfin, CHEVAL introduit la notion d'action qui découle logiquement de l'approche par Oracle et constitue un premier niveau de planification. Notons toutefois que la notion d'action n'est pas uniquement destinée à l'écriture d'un Oracle, et qu'une implantation de CHEVAL ne la traduira pas nécessairement de la même manière qu'une autre. Pour fixer les idées, donnons quelques exemples d'actions CHEVAL :

- `Play piece region` : incite le programme à placer la pièce `piece` sur les cases de `region` ;
- `Avoid piece region` : comme `play` mais avec une note négative : pour éviter d'éventuelles erreurs de signes dans la fonction d'évaluation, `play` n'acceptera que des notes positives, et `avoid` que des notes négatives.
- `Controle region` : incite le programme à contrôler des cases avec ses pièces ;
- `Exchange piece` : incite le programme à échanger une pièce particulière ;
- `Keep piece` : incite le programme à conserver une pièce particulière ;
- `Reach pattern` : permet de noter (ou d'inciter à atteindre) une structure ;
- `Open region` : incite le programme à ouvrir une région, c'est à dire à y altérer la structure de pions de manière à y créer des lignes ouvertes pour les figures à marche linéaire.
- `Close region` : le contraire de `Open`.
- `Attack region` : incite le programme à contrôler et investir une région, et à y ouvrir des lignes.
- `Defend region` : incite le programme à contrôler une région et à y fermer des lignes.

Finalement, la valeur de l'évaluation doit rester cachée au niveau le plus haut, et toute manipulation de valeurs arithmétiques précises sera évitée dans la très grande majorité des cas à l'expert du domaine. Il sera toutefois possible d'avoir recours à de telles valeurs dans certaines situations exceptionnelles (Il est clair que la généralité des fonctions d'évaluation qui y auront recours, sera amoindrie). Des valeurs relatives à la valeur des pièces seront manipulables, mais leur utilisation ne sera pas encouragée non plus. Nous postulons en effet que,

5. Un bélier est constitué par un pion blanc et un pion noir face à face : chacun des deux pions empêche l'autre d'avancer

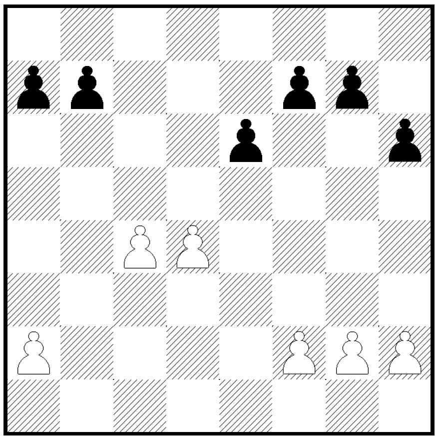
6. Un levier est constitué par un pion blanc et un pion noir en prise mutuelle

dans la très grande majorité des cas, un système d'évaluation calqué sur les conventions échiquéennes d'annotation de parties (!, !!, ? et ??, et peut être aussi !? et ?!) sera suffisant. Dans ce symbolisme, une action gratifiée d'un !! sera très fortement encouragée (ou notée, selon les choix d'implantation), elle le sera moins si elle ne se voit attribuer qu'un !, et elle sera encouragée/notée de manière "normale" si elle n'en compte aucun. Le ? signifiera que l'action est à déconseiller, et le ?? qu'elle est à déconseiller fortement. Un !?, qui signifie généralement qu'un mouvement est intéressant, ou un ?!, qui qualifie un mouvement vraisemblablement douteux, pourront infléchir le comportement du programme dans certaines situations particulières, qui font intervenir la personnalité du programme, de son adversaire, et la situation dans le tournoi. Ils indiqueront les mouvements ou les plans propres à déséquilibrer radicalement la position, avec une prise de risque «raisonnable» (!?) ou importante (?!). A titre d'exemple, un programme devant affronter des humains en parties éclairs aura intérêt à prendre des risques, et donc à favoriser les mouvements ou les plans !? et ?!.

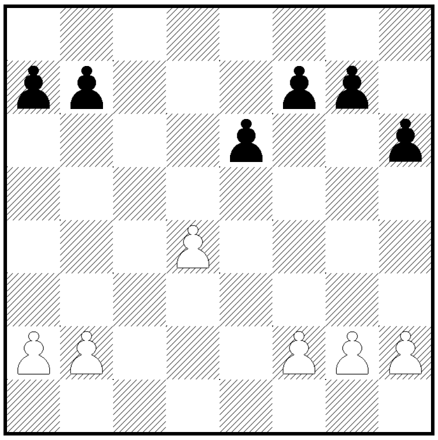
Il est bien évident que CHEVAL, qui connaît la notion de couleur de joueur, sait appliquer pour les noirs ce qui a été écrit pour les blancs, sauf interdiction explicite à l'aide de l'instruction `nosymetry`.

Une description plus complète du langage CHEVAL pourra être trouvée dans [158].

5.1.4.3 Exemple de plan exprimé dans le langage CHEVAL



La paire de pions isolés : les pions pendants



Le pion dame isolé

Diagramme 5.4: *Deux structures de pions fondamentales.*

Certaines structures de pions sont d'une importance fondamentale pour la compréhension d'une famille d'ouvertures. Le pion dame isolé et ses proches parents les pions pendants (Kmoch alla jusqu'à parler de pion pendant au singulier, appellation qui convient particulièrement bien au pion dame isolé), en sont l'exemple le plus connu (figure 5.4). Certains tenteront d'éviter à tout prix de se voir infliger de telles faiblesses, d'autres accepteront volontiers de jouer des deux côtés de ces positions : leur compréhension n'est pas seulement affaire de finesse de jugement, mais aussi de style. Le pion dame isolé est une cible facile, et les cases qui l'entourent sont affaiblies. Il devient presque toujours une faiblesse en fin de partie. Mais en milieu de partie, le pion dame isolé allié à un jeu dynamique peut être une arme redoutable. Tarrasch alla jusqu'à considérer qu'il constituait principalement un avantage, et Botvinnik ne craignait pas l'isolation de son pion dame.

Le pion dame isolé se transforme facilement en pions pendants (par exemple après un échange en c3). Comme le pion dame isolé peut être qualifié de pion pendant, les pions pendants seront parfois vus comme un duo de pions isolés. Ils ont aussi leurs avantages et leurs inconvénients, leurs forces et leurs faiblesses, qui vont dépendre principalement du dynamisme de la position et de leur mobilité. Le programme GIDEON, par exemple, les considère comme un avantage. Notre programme ECUME se contente de les noter comme une structure de pions quelconque (ECUME reconnaît dynamiquement les pions arriérés, les ilôts, les duos, les cases fortes et faibles, les pions isolés, et donne généralement une note convenable aux pions pendants), mais son Oracle connaît les principes du jeu avec –ou contre– les pions pendants.

Dans l'exemple que nous donnons, nous sommes conscients du fait que notre bibliothèque d'ouvertures peut amener le programme dans une position à pion dame isolé, et, après avoir défini ces structures de pions (cf figure 5.2), nous considérons qu'il est favorable de transformer le pion dame isolé en pions pendants (instruction Reach). Puis nous donnons des indications sur le placement des pièces, et enfin nous regardons si la position est suffisamment favorable au camp qui possède le pion dame isolé pour qu'il monte une attaque sur le petit roque adverse ; nous donnons alors la marche à suivre (cf. figure 5.3).

5.1.5 Effets induits par la précision des fonctions d'évaluation

Il est clair que si une fonction d'évaluation est réduite à sa forme la plus simple, c'est à dire, aux échecs, au critère matériel, son calcul est plus rapide. De plus, les variations entre les évaluations des différents mouvements deviennent quasi-inexistantes, donc la probabilité que ces mouvements soient bien triés augmente et les recherches $\alpha\beta$ évaluent moins de nœuds. Ceci a motivé une expérience que nous avons menée sur l'effet de la précision des fonctions d'évaluation : nous avons pris notre meilleur programme d'échecs, ECUME et nous avons arrondi la valeur retournée par la fonction d'évaluation au multiple de 2 le plus proche, au multiple de 4, de 8 et de 16 (Un pion étant représenté par 100 points). Nous avons fait jouer ces 4 versions contre la version de référence où l'évaluation n'est pas arrondie et contre une version du programme ZARKOV.

La table 5.1 montre que dans toutes les cadences les plus élevées, la réduction de la précision de la fonction d'évaluation nuit à la force du programme, si elle est faite de manière trop brutale. Par contre, il est net que la version où la fonction d'évaluation est arrondie au multiple le plus proche de 2 donne pour notre programme d'assez bons résultats.

Ceci est dû au fait que le choix des mouvements stratégiques n'est pas trop touché par la

```
    (* Pions pendants : *)
    Pattern  P_PionsPendants
         IsOpenWhite (bFile, eFile)
         IsOpenBlack (cFile, dFile)
    EndPattern

    (* Pion dame isole  : *)
    Pattern P_PionDameIsole
         IsOpenWhite (cFile) Or IsOpen (cFile);
         IsOpenWhite (eFile) Or IsOpen (eFile);
         IsOpenBlack (dFile)
    EndPattern

    (* la définition des régions ouvertes et semi ouvertes *)
    (* pouvant être : *)
Func IsOpenWhite (Region R_Region)
    if (white pawn In R_Region Or Not black pawn In R_Region)
    Then IsOpenWhite := false;
    else IsOpenWhite := true;
    Endif
End

Func IsOpenBlack (Region R_Region)
    if (black pawn In R_Region Or Not white pawn In R_Region)
      Then IsOpenBlack := false;
      else IsOpenBlack := true;
    Endif
End


Func IsOpen (Region R_Region)
    if (white pawn In R_Region Or black pawn in R_Region)
       Then IsOpen := false;
       else IsOpen := true;
    Endif
End

Func IsClosed (Region  R_Region)
    if (white pawn In R_Region And black pawn in R_Region)
       Then IsClosed := true;
       else IsClosed := False;
    Endif
End
```

Fig. 5.2 - *Structures de pions simples exprimées en* CHEVAL

```
(* conduite à tenir avec le pion dame isolé (PDI) *)

if (P_PionDameIsole) Then
  White Actions
         (* tenter de transformer le PDI en pions pendants *)
         (* qui donnent plus de chances à un programme *)
       reach P_PionsPendants!!;
         (* éviter les échanges, car la faiblesse du PDI
          * augmente en finale *)
       Keep White Pieces!;
       if (white pawn In d4) Then
           Controle d5 !;
           Play d5!
           Attack Kingside;
       Endif
    EndActions
   Black Actions
        Controle d5 !;
        Controle dFile;
        Play Pieces d5;
        Play Knight d5!;
        Play Rook c8, Rook d8;
   EndActions

(* reconnaissance et préparation d'une attaque à l'aile roi,
 *  si la position s'y prête
 *)
if ( IsClosed (center) and WhiteSpace(KingSide) > BlackSpace(KingSide)
     and WhiteControl(KingSide) > BlackControl(KingSide) Then
      White Actions
         Attack Kingside;
         Close Center;
          Play Knight e5;
         Open KingSide ?!;
      EndActions
      Black Actions
         Attack Center!;
         Defend KingSide;
         Open Center!!;
      EndActions
Endif
Endif
```

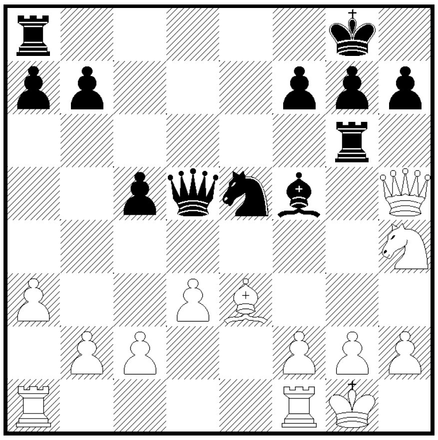
Fig. 5.3 - *Plan simple pour un Oracle exprimé en* CHEVAL

TAB. 5.1 - *Résultats des matchs avec des versions à évaluations tronquées.*

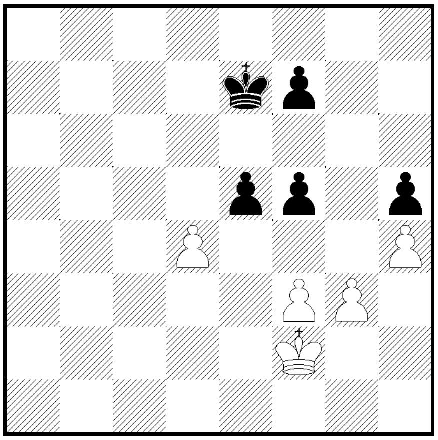
Chaque résultat est obtenu pour 102 parties.

	Versions avec fonction d'évaluation arrondies à			
	2	4	8	16
Parties avec 5 secondes de moyenne par mouvements				
ECUME	50,5 %	44,1 %	47,6 %	45,1 %
ZARKOV 3.0	38,8 %	34,8 %	40,2 %	38,3 %
Parties avec 10 secondes de moyenne par mouvements				
ZARKOV 3.0	38,7 %	32,5 %	38,8 %	45 %
Parties avec 30 secondes de moyenne par mouvements				
ECUME	43,7 %	50 %	53,4 %	54,5 %
ZARKOV 3.0	34,3 %	36,27 %	41,18 %	45,1 %

précision plus faible de la fonction d'évaluation, et par le fait que les versions avec arrondis sont plus rapides à profondeur égale, comme le montrent les résultats que nous obtenons avec ces versions sur les positions du diagramme 5.5 : dans la première de ces positions les temps pour trouver le bon mouvement sont de 45 secondes et 44 (resp. 32, 38, 30 secondes) pour le programme ECUME avec une fonction d'évaluation pas arrondie et une fonction d'évaluation arrondie au multiple de 2 (resp. 4, 8, 16) le plus proche. Et pour la position de droite, les temps respectifs sont de 8 minutes, 6 minutes 45 secondes, 7 minutes 24 secondes, 6 minutes 31 secondes et 6 minutes 58 secondes.



Combinaison n° 5[124]



Finale n° 9[125]

Diagramme 5.5: *Deux positions issues des positions tests de Pierre Nolot*

Ainsi, nous voyons que le fait de rendre une fonction d'évaluation moins précise influe sur le temps de réponse en induisant un « meilleur » ordre des mouvements dans l'arbre de recherche[7] et en rendant donc la recherche plus efficace. De plus, cette réduction de précision peut être effectuée modérément sans avoir trop d'influence sur le comportement stratégique du programme[8].

5.1.6 Évaluation des fins de parties

La fin de partie aux échecs est un domaine particulier du fait que :

- de nombreuses analyses existent et permettent de définir des techniques (humaines) de résolution;
- dans le cas où le matériel est très restreint (moins de 7 pièces en comptant les 2 rois), des **bases de données de finales** ont été construites par analyse rétrograde[174, 169].

Dans le cas où le matériel est encore important, l'Oracle est spécialisé par l'introduction de règles générales. Par exemple pour les finales de tours, il détermine quels sont les pions passés dangereux, les faiblesses à exploiter, etc.

Quand le matériel est plus réduit, les bases de données de finales sont utilisées pour produire un jeu parfait. Si les bases de données ne sont pas utilisées, nous distinguons 4 types fondamentaux de finales :

1° Les finales simples (par exemple, ♔♕ contre ♔);

2° Les finales où il est difficile de discriminer les positions gagnantes mais où si l'on reste dans ces positions gagnantes, le gain est facile : ♔♕ contre ♔♕, ♔♖ contre ♔♘, ♔♕ contre ♔P;

3° Les finales où il est difficile de trouver le plan qui nous permettra de gagner : ♔♘♗ contre ♔, ♔♕ contre ♔♖, ♔♗♗ contre ♔♘;

4° les finales où les deux difficultés apparaissent.

Le premier cas est traité par l'adjonction de fonctions d'évaluation spécialisées dont les valeurs de retour sont judicieusement choisies de manière à éviter le « blemish effect ». Le deuxième type est plus complexe et relève de la synthèse automatique de fonctions de classification que nous traitons dans le § 5.2.3.

Pour le troisième cas, s'il existe une méthode humaine pour forcer le gain et si cette méthode est valide et bien définie, nous pouvons introduire ces connaissances. La méthode est décomposée en étapes, une étape étant une classe de position. A chaque étape est associée une fonction d'évaluation telle que si nous notons f_i, la fonction d'évaluation associée à

7. Pour avoir une idée de pourquoi statistiquement l'ordre est de mieux en mieux, on peut considérer que plus la fonction d'évaluation est arrondie, plus les mouvements sont égaux entre eux et donc le premier mouvement à un nœud a plus de chances d'avoir une évaluation égale à celle du meilleur mouvement.

8. Sans avoir pu le mesurer précisément, nous avons remarqué que dans le programme ECUME, les principaux défauts stratégiques induits par la réduction de la précision sont visibles dans les phases de fin de partie.

l'étape i et si l'ensemble des positions de l'étape i ($\mathcal{E}_i$) sont plus proches du gain (au sens de la méthode) que l'ensemble des positions de l'étape j, alors $\forall P, Q \in (\mathcal{E}_i \times \mathcal{E}_j)$ nous avons $f_i(P) > f_j(Q)$ pour éviter le « blemish effect » et pour que la méthode converge vers le gain. De telles descriptions sont possibles pour des finales simples, comme par exemple ♔♘♗ contre ♔ ou ♔♘ contre ♔[76, 75], mais le cas général est plus complexe et des règles de bases seront définies directement dans l'Oracle.

5.2 Techniques d'apprentissage dans la programmation des jeux de réflexion.

Il existe cinq catégories principales pour les méthodes d'apprentissage dans les programmes de jeux de réflexion :

1° l'apprentissage par cœur;

2° des apprentissages statistiques;

3° l'apprentissage de la classification;

4° la reconnaissance de structure;

5° l'auto-formation de critères.

5.2.1 L'apprentissage par cœur

Les méthodes d'apprentissage par cœur consistent en un stockage de l'expérience passée de telle sorte qu'elle soit facilement accessible et utilisable dans les recherches futures. Dans le programme de dames anglaises (Checker) de Samuel[144, 145], les positions rencontrées dans les parties étaient stockées avec leur évaluation. Ainsi, dans les parties suivantes, le programme, dès lors qu'il examinait dans son arbre une position stockée, utilisait l'évaluation qui s'y reportait dans les tables si les précédentes recherches étaient d'un niveau au moins égal au niveau demandé.

Ce genre de technique est encore utilisé de nos jours, par exemple par le programme DEEP THOUGHT[78] qui stocke les différentes positions rencontrées pendant ses parties avec leur évaluation dans un fichier en les indexant à l'aide du matériel restant et de la configuration des pions. Lors du début d'une nouvelle recherche, les positions stockées dont le matériel et la structure de pions permettent de penser qu'elles pourront apparaître dans la recherche sont entrées dans la table de transposition. Il s'agit d'une variante de la méthode proposée par Slate[163] et modifiée ensuite par l'équipe du programme BEBE[153][9].

Le programme d'Othello LOGISTELLO[37] utilise une technique similaire pour modifier sa bibliothèque d'ouverture après la perte d'une partie. Sa bibliothèque d'ouverture est constituée de deux arbres de jeux représentant respectivement l'ensemble de ses parties perdues avec les blancs (resp. avec les noirs) plus de l'ensemble des mouvements alternatifs dans les positions

9. Cette même méthode fut utilisée dans une machine commerciale de la firme FIDELITY.

perdues. Chaque feuille de l'arbre étant évaluée soit à la valeur réelle de la partie, dans le cas de la fin d'une partie nulle ou perdue, soit par l'évaluation des mouvements alternatifs à une profondeur fixée. Mais une bonne ligne d'ouverture, ainsi apprise, peut être suivie d'un mauvais mouvement qui entraîne la fin de la partie.

5.2.2 Les méthodes d'apprentissage statistiques

L'autre type de méthode très utilisée dans les programmes de jeux de réflexion est la construction de fonctions d'évaluation par des méthodes statistiques. Ces méthodes statistiques utilisent une base de données de parties et en déduisent un ajustement des coefficients des différents facteurs de la fonction d'évaluation. Là encore, le pionnier est Samuel[144, 145]. En 1961, Max Euwe, ancien champion du monde d'échecs, a monté un groupe de travail pour analyser la faisabilité de la méthode de Samuel dans le cadre du jeu d'Échecs[74] ; il a conclu que dans le cadre des échecs, la méthode de Samuel « would have to deal not with 31 components numbers but with 31 million or even more»[50]. En d'autres termes cette méthode leur semblait inadéquate aux échecs.

Nitsche[123] décrit un programme qui ajuste automatiquement les «importances» de certains heuristiques en utilisant la méthode des moindres carrés pour que la valeur de la fonction d'évaluation des meilleurs mouvements (c'est à dire ceux qu'il faut jouer dans les positions d'apprentissage) soit le plus proche possible de $+1.0$ et la valeur des autres mouvements proches de 0. Cette méthode fut employée avec succès dans les finales ♔♕ contre ♔ et ♔♖ contre ♔.

Dans le cas du jeu d'Othello, les techniques d'apprentissage de coefficients de fonctions d'évaluation font l'objet de nombre de recherches. La base commune des méthodes proposées est la connaissance d'un nombre N de positions P_i de référence avec leur évaluation exacte s_i (positions de fin de partie) ou supposée telle. On définit alors un nombre k de critères positionnels v_j pondérés par des coefficients c_j, la fonction d'évaluation $f(P_i)$ étant définie par $\sum_{j=1}^{j=k} c_j v_j(P_i)$. Mitchell[60] détermine les c_j de manière à minimiser l'erreur $\sum_{i=1}^{i=n}(s_i - \sum_{j=1}^{j=k} c_j v_j(P_i))^2$. Lee et Mahajan[94], quant à eux, scindent l'ensemble des positions P_i en deux ensembles, les positions gagnées et les positions perdues, et, en fonction des matrices de covariance définies par les critères positionnels et de leur moyenne, définissent une fonction d'évaluation représentant la probabilité de gain. Malheureusement, si chaque critère est « antisymétrique » (la valeur du critère est l'opposé de la valeur du critère pour l'adversaire) ou «symétrique» (les valeurs des critères sont les mêmes pour les deux joueurs), les termes quadratiques dans la fonction d'évaluation obtenue s'effacent et la non-linéarité restante n'est qu'une expression d'un bruit statistique[14]. Buro propose, pour remédier à ce problème tout en gardant une approche stochastique, de former une fonction d'évaluation mesurant la probabilité de gain par une régression «logistique»[38]. Dans sa thèse il montre, pour les critères définis pour son programme d'Othello LOGISTELLO, que la régression logistique est meilleure que la régression quadratique proposée par Lee et Mahajan, elle même meilleure que la régression linéaire si l'on permet uniquement l'apprentissage de coefficients positifs[38].

Dans les programmes d'échecs, il est difficile de déterminer la valeur exacte d'une position. Par contre, on peut construire un ensemble de positions dans lesquelles on connaît un mouvement optimal (c'est à dire joué par un joueur supposé fort, comme un grand-maître ou un autre programme). Marsland[100] puis Meulen[108] définissent des algorithmes d'escalades pour ajuster la fonction d'évaluation de telle sorte que le meilleur mouvement soit déterminé par le principe negamax par application de la fonction d'évaluation sur un arbre de jeu de profondeur réduite à un demi-coup. Hsu, Anantharaman, Campbell et Nowatzyk[78] améliorent ces schémas en considérant des arbres de profondeur supérieure. A chaque pas du schéma itératif, et pour un ensemble de coefficients donnés, sont calculées, pour chaque position P de l'ensemble d'apprentissage et chaque mouvement M_i, les positions *dominantes* P_i, positions obtenues en suivant les sous-variantes principales des mouvements M_i jusqu'aux positions feuilles où est appliquée la fonction d'évaluation en considérant une recherche de la tranquillité minimale ne tenant compte que des captures. Après la comparaison de la position dominante obtenue par le mouvement de référence de la base d'apprentissage et de la position dominante obtenue par l'application de la fonction d'évaluation, on ajuste les coefficients de telle manière que la position dominante du mouvement de référence (qui elle-même est dépendante de la fonction d'évaluation) soit supérieure (du point de vue du joueur au trait dans la position P) à la position dominante du mouvement choisi par l'application de la fonction d'évaluation. Ce processus fut appliqué avec succès pour améliorer l'évaluation du programme DEEP-THOUGHT[66].

5.2.3 Apprentissage de la classification

L'apprentissage de la classification consiste en la construction de règles[10] permettant de classer un large ensemble de positions dans un petit ensemble de classes. Le problème type est la classification d'un ensemble de positions de finales en positions perdues, nulles ou gagnées[134, 161, 188].

Le but est la construction d'un arbre de décision. Un *arbre de décision* est une représentation d'une procédure de choix pour déterminer la classe d'un objet. Chaque nœud d'un arbre de décision représente soit un nom de classe soit un prédicat (fonction qui prend comme argument un objet et qui a pour image l'ensemble (Vrai, Faux)) qui partitionne l'ensemble des instances possibles des objets en fonction du résultat du prédicat en plusieurs sous-ensembles. A chaque sous-ensemble de la partition correspond un sous-problème de classification, résolu par un sous-arbre de décision. Formellement, un arbre de décision est défini par ses feuilles, qui contiennent le nom d'une classe, et des nœuds internes, qui contiennent un prédicat et dont chaque branche aboutit à un sous-arbre de décision en fonction de la valeur du prédicat.

Les algorithmes utilisés pour construire des arbres de décision sont dérivés de l'algorithme ID3 de Quinlan[134, 135] donné dans la figure 5.4. Les constructions des arbres $ARBRE$ dans les étapes 3 et 4c sont montrées par la figure 5.5.

L'utilisateur interagit avec l'algorithme ID3 en lui fournissant une base de données de positions à classer et un ensemble de prédicats $TEST$. L'algorithme ID3 essaye alors de construire un arbre de décision qui classe correctement la base de données. S'il n'y parvient pas, c'est

10. Ainsi que l'a noté Gondran[68], dans les problèmes de classification où il n'existe pas de théorie bien établie, on prendra soin de différencier les règles affirmatives et négatives.

1. L'utilisateur fournit *TEST*, un ensemble de prédicats qui retournent une valeur booléene selon la position donnée en argument;
2. Soit *FENETRE* un ensemble de positions choisies au hasard.
3. Soit *ARBRE* l'arbre de décision classifiant les positions de l'ensemble *FENETRE* à l'aide des prédicats de l'ensemble *TEST*.
4. Tant qu'il existe une position de l'ensemble mal classée par l'arbre *ARBRE*
   (a) Soit *MAUVAIS*, un ensemble non vide de positions mal classées par l'arbre *ARBRE* choisies aléatoirement.
   (b) $FENETRE \leftarrow FENETRE \cup MAUVAIS$
   (c) Soit *ARBRE*, l'arbre de décision classifiant les positions de l'ensemble *FENETRE* à l'aide des prédicats de l'ensemble *TEST*.
5. L'arbre de décision *ARBRE* classe correctement l'ensemble des positions.

FIG. 5.4 - *L'algorithme* ID3

1. Si toutes les positions sont dans la même classe, alors l'arbre de décision est la feuille qui contient le nom de la classe.
2. Sinon,
   (a) Soit p_0, le prédicat qui minimise $E(p) = I(v_{C0}, v_{C1}) + I(f_{C0}, f_{C1})$ où v_{C0} (resp. f_{C0}) est le nombre de positions A de la classe 0 telle que $p(A)$ soit vrai (resp. faux) et $I(x, y) = -\frac{x}{x+y}\log\frac{x}{x+y} - \frac{y}{x+y}\log\frac{y}{x+y}$, une mesure entropique basée sur la théorie de l'information[159]
   (b) Donner au nœud le nom du prédicat p_0 et créer deux nouvelles branches issues de ce nœud correspondant aux deux valeurs possibles du prédicat.

FIG. 5.5 - *Construction de l'arbre* ARBRE *par l'algorithme* ID3

à dire si il existe deux positions A_1 et A_2 telles que $\forall p \in TEST, p(A_1) = p(A_2)$ avec A_1 et A_2 dans des classes différentes, l'utilisateur devra fournir à l'algorithme un nouvel ensemble $TEST$ et ainsi de suite jusqu'à ce que l'arbre de décision construit par ID3 permette de classer totalement la base de données[161, 188].

```
Func BK_cannot_move_up

    if WQ controls b2 and WK controls b2 and not (BQ controls b2)
    then
        return true
    endif

    if WK controls (b2, b1) and WQ controls a4
      and not (BQ controls a4 or BQ controls a3)
    then
        return true
    endif

    for square in Afile do
        if WQ controls square and not (BQ controls square)
          and WK on b3 and not (WQ controls c2)
        then
            return true
        endif
    endfor

    return false

end
```

FIG. 5.6 - *Prédicat intervenant dans la discrimination des positions perdues Dame et Roi contre Dame et Roi.*

Le roi noir est ramené par symétries dans le triangle ayant pour coins les cases a1 a4 et c4. Ce prédicat permet de savoir si le roi noir en première rangée peut aller en deuxième rangée sans se faire mater ou perdre sa dame au prochain mouvement.

Comme nous l'avons montré dans [188], les prédicats intervenant peuvent devenir de plus en plus complexes même pour des finales considérées comme simples par les joueurs d'échecs. La figure 5.6 nous montre un des prédicats que nous avons définis en langage CHEVAL lors de la construction d'un arbre de décision pour la finale ♕ et ♔ contre ♕ et ♔. Il nous a fallu définir 46 prédicats différents pour classer les 1165815 entrées de la base de données. A l'aide de ces prédicats et de l'algorithme ID3, nous avons obtenu un arbre de décision de 258 feuilles.

Soit P, une position à ajouter à l'arbre de décision $ARBRE$.

1. Si $ARBRE$ est vide, le réduire à une feuille étiquetée par la classe de la position P, et définir l'ensemble des positions tributaires de cette feuille comme étant réduit à P.

2. Sinon, Si l'arbre $ARBRE$ est réduit à une feuille étiquetée par la même classe que celle de la position P, ajouter P à l'ensemble des positions tributaires de cette feuille.

3. Sinon,

   (a) si l'arbre est réduit à une feuille, l'étendre d'un niveau en choisissant le prédicat de test arbitrairement;

   (b) Pour tous les prédicats de l'ensemble $TEST$, mettre à jour (ou calculer) le nombre d'instances positives et négatives pour ce prédicat d'après l'ensemble des positions tributaires.

   (c) Si le prédicat associé au nœud courant ne minimise par la fonction E, alors

      i. l'arbre est restructuré de manière à associer un prédicat qui minimise la fonction E à la racine.

      ii. De manière récursive, les sous-arbres positifs et négatifs sont restructurés à l'exception du sous-arbre mis à jour dans l'étape 3d.

   (d) Mettre à jour récursivement le sous-arbre qui correspond à la valeur du prédicat associé à la racine selon la position P.

La restructuration de l'arbre à l'étape 3(c)i étant mis en œuvre par l'algorithme suivant ;

1. Si le prédicat $a_{nouveau}$ à associer à la racine de l'arbre y est déjà, alors arrêt.

2. Sinon,

   (a) Associer récursivement le prédicat $a_{nouveau}$ aux sous arbres immédiats de la racine. (Si un des sous-arbres est une feuille, l'étendre d'un niveau)

   (b) Grouper les sous-arbres entre eux en associant les sous arbres en fonction des résultats du prédicats et en réalisant l'union des ensembles de position tributaires respectifs.

FIG. 5.7 - *L'Algorithme ID5R[181]*

D'autres algorithmes dérivés d'ID3 on été définis pour construire des arbres de décisions de taille inférieures. Les algorithmes ID4[154] et ID5R[180, 181] sont bâtis autour de ID3, mais au lieu de reconstruire un nouvel arbre depuis le début dès qu'une nouvelle position est incluse dans l'ensemble $FENETRE$, ces algorithmes modifient l'arbre de décision précédent pour y inclure la nouvelle position. Pour l'algorithme ID5R cela est permis en incluant dans l'arbre de décision pour chaque nœud et chaque feuille l'ensemble des positions de l'ensemble $FENETRE$ qui sont *tributaires* de cette partie de l'arbre. La figure 5.7 détaille l'algorithme ID5R. Cet algorithme génère en moyenne des arbres de décisions plus concis que l'algorithme ID3 mais est plus lent du fait de la gestion plus complexe de l'arbre de décision et de la complexité des réorganisation. Nous avons comparé l'algorithme ID5R et ID3 sur la finale ♔♕ contre ♔♕. Pour accélérer les calculs, nous avons défini, à l'aide des 1165815 positions et des 46 prédicats, l'ensemble des vecteurs de la base de données : chaque position de la base de données définit un vecteur de dimension 47, les 46 premières coordonnées de ce vecteur sont les valeurs respectives des 46 prédicats appliqués à la position et la 47$^{\text{e}}$ coordonnée à pour valeur la classe de la position. Les 1165815 positions de cette base de données se résument en 21464 vecteurs différents, donc 21464 positions de références sont nécessaires et suffisantes pour classer la base entière. Nous avons alors utilisé les algorithmes ID3 et ID5R sur ces 21464 positions. Sur 10 exécutions, ID3 construit en moyenne des arbres de 262 feuilles en 4 minutes (temps moyen), le plus petit arbre généré étant de 248 feuilles. L'algorithme ID5R, quant à lui, construit en moyenne des arbres de 253 feuilles pour un temps moyen de 6 minutes, le plus petit arbre de décision généré étant de 232 feuilles. Nous avons aussi testé l'algorithme IDL tel que défini par van de Velde[182], qui, pour minimiser la taille des arbres générés, utilise un critère de topologie des arbres ; malheureusement, notre implantation de cet algorithme ne converge qu'après 2 jours de calculs et nous fournit des arbres de plus de 500 feuilles.

5.2.4 La reconnaissance de structures

Les programmes qui utilisent la reconnaissance de structures utilisent des structures *abstraites* (d'après la terminologie de F. GOBET : par opposition aux structures *perceptibles*). De plus, dans la plupart des approches (à l'exception de MORPH et CHUMP) leurs applications sont réservées à des sous-domaines du jeu d'échecs.

Le premier programme utilisant la reconnaissance de structures est le programme de Zobrist et Carlson[195]. Les structures étant définies par un expert à l'aide d'un langage particulier, des conseils étaient définis avec ces structures. À l'aide de ces conseils, le programme générait et évaluait les mouvements. Une extension possible de ce modèle est le programme Mach de George et Schaeffer[62] et, bien sûr, l'emploi que nous faisons de l'Oracle inclus dans ECUME.

Les programmes CHUNKER de Berliner et Campbell[23, 39] avaient comme domaine une sous-partie des finales de pions. Une bibliothèque de l'ensemble des configurations de pions est générée, et chaque configuration est associée à une liste de propriétés (nombre de mouvements avant la promotion, valeur théorique de la position). Cette bibliothèque permet l'extension de la fonction d'évaluation utilisée dans une recherche de type $\alpha\beta$ classique. Pour diminuer la croissance de la complexité de la fonction d'évaluation, et pour rendre la recherche plus efficace, CHUNKER II[39] découpe les configurations de pions de la position en plusieurs *mor-*

ceaux indépendants et ne pouvant interagir entre eux. Avec un mécanisme de recherche de valeurs supplémentaire, ce programme est capable de jouer l'ensemble des finales de pions. Mais, du fait que les *morceaux* doivent être indépendants, le processus de recherche utilisé par le programme n'est pas généralisable à une partie entière.

Le programme MORPH[96] apprend en raffinant ses structures. À chaque structure est associé un poids, qui est une estimation de l'évaluation des positions comprenant cette structure. Ces différents poids sont combinés tous ensemble pour former la fonction d'évaluation. Le choix d'un mouvement est fait en utilisant ce principe sur les positions terminales de l'arbre de jeu d'une profondeur de un demi-coup à l'aide du processus minimax.

Une autre approche consiste à considérer l'ensemble des structures à nombre de pièces fixé et d'associer à chaque structure un mouvement. Un mouvement sera choisi alors s'il est associé à un maximum de structures de la position. Ces structures peuvent être extraites par un module simulant le mouvement de l'oeil. Gobet et Jansen[66] ont analysé cette méthode dans le programme CHUMP sur l'ensemble des parties de Mikhail Tal et sur la finale ♕ et ♔ contre ♖ et ♔. Leur méthode est à mi-chemin entre l'apprentissage par cœur (CHUMP ne peut pas suggérer un mouvement qu'il n'a jamais vu avec les structures de la position) et l'apprentissage statistique (un des paramètres entrant dans le choix des mouvements est l'activation : c'est le nombre de fois, pour une structure S, qu'un mouvement est associé aux positions de la base d'apprentissage contenants S.).

5.2.5 L'auto-formation de critères

Jusqu'à présent, nous avons vu des méthodes d'apprentissage qui reposaient sur des critères pré-définis et donc qui contiennent potentiellement la connaissance. Ainsi, aux méthodes d'apprentissage citées ci-dessus, sont fournis non seulement l'information disponible sur le plan de jeux, mais aussi des pré-critères tels que le matériel. Dans les méthodes elles-mêmes, des pré-critères sont implicitement définis dans les relations permises entre l'information et sa représentation interne dans le processus d'apprentissage. Il est en effet difficile de ne pas biaiser l'apprentissage et l'interprétation des résultats par l'usage d'une représentation interne qui reflète déjà la forme des critères attendus. Les approches connexionistes ayant comme paramètres d'entrée uniquement ceux de la position sans regroupement a priori, peuvent être considérées comme une auto-formation de critères; ceci étant, on admettra bien que le biais existe par la topologie interne des connexions. Cette approche fut utilisée avec succès dans le programme de Backgammon de Tesauro et Sejnowski[171].

Le programme SAL simplifie le problème en considérant des critères basés sur les règles du jeu auquel doit se rapporter l'apprentissage[63].

5.3 Conclusion

Dans ce chapitre, nous avons montré comment introduire la notion de plan stratégique dans un programme d'échecs à l'aide de l'approche d'Oracle couplée avec un tableau positionnel. Cette approche est l'un des constituants fondamentaux de la réussite de nos programmes d'échecs.

L'écriture des fonctions d'évaluation pour les programmes d'échecs peut être grandement facilitée par l'utilisation du langage CHEVAL qui garantit une écriture indépendante des choix ultérieurs d'implantation des structures de données internes du programme.

Les algorithmes de synthèse de fonctions d'évaluation se sont montrés fort efficaces dans les jeux tels que le Backgammon ou Othello, mais restent d'une utilisation marginale dans les programmes d'échecs, sans doute par la plus grande importance des notions stratégiques.

Chapitre 6

Résultats

Dans ce chapitre, après avoir présenté nos programmes d'échecs, nous donnons leurs parties importantes de compétition.

6.1 Présentation de nos programmes

Echec : Il s'agit du premier des programmes issus des travaux concernés par ce mémoire. Les toutes premières versions de ce programme étaient basées sur le programme TECH de Gillogly[64]. Une rapide analyse de la position de la racine ordonnait les mouvements, et le mouvement était choisi par une recherche *Negascout* où la fonction d'évaluation était réduite au critère matériel. Cette première version nous a permis d'explorer le domaine des heuristiques modifiant l'enveloppe de la recherche. Rapidement, il nous est apparu que des notions stratégiques simples ne pouvaient être comprises par ce schéma de choix de coup, et nous avons opté pour une évaluation basée sur une approche de type Oracle. Dans le programme ECHEC 1.5, qui a participé aux olympiades de Londres 1989, seuls des critères de premier ordre étaient considérés. Nous avons incorporé par la suite des critères d'évaluations de plus en plus fins, et surtout rajouté des critères de second ordre pour évaluer la structure de pions et la mobilité. En même temps, nous avons modifié les extensions statiques du programme pour arriver à la version ECHEC 1.9, qui fut « champion du monde 1990 des programmes sur ordinateur personnel », et « champion de France des ordinateurs 1992 ».

Cumulus 1.x : Contemporain du programme ECHEC, il apparut au championnat du monde 1990. Notre but était d'incorporer dans la recherche plus de compréhension des problèmes tactiques. Cela nous a amené à définir une recherche de la tranquillité basée sur la gestion des pièces en prise. De plus, de par sa structure, le programme CUMULUS nous a permis d'utiliser l'heuristique du mouvement nul[1]. Malheureusement, la gestion des pièces en prise, si elle permettait d'avoir de bons résultats dans certaines positions (cf. § 3.3.4), ralentissait trop la recherche. Ce qui rendait le programme CUMULUS faible

1. Cette heuristique n'était pas compatible avec le programme ECHEC car elle interférait avec le critère calculant la mobilité qui était calculé incrémentalement pendant la recherche.

tactiquement. Le programme CUMULUS finit sa carrière sur le titre de « Vice-Champion de France des ordinateurs 1992 »

Ecume : ECUME est la fusion d'une réécriture de la fonction d'évaluation d'ECHEC pour augmenter le rôle des plans stratégiques et d'un moteur de génération de coups, gestion de l'algorithme *Negascout*, des extensions de recherche [2] et des critères du second ordre en assembleur [3]. Dès le deuxième mois suivant le début de la programmation du programme ECUME, il devenait « Vice Champion du monde des ordinateurs 1992, toutes catégories ». Sa force de jeu provient du moteur très rapide permettant d'éviter la plupart des bévues tactiques, lié à l'intelligence de l'Oracle qui lui permet de jouer dans certains types de positions à l'aide de plans. ECUME est aussi « Vice-Champion de France des ordinateurs 1993 » et « Vice-Champion du monde des micro-ordinateurs en Blitz 1993 ». ECUME incorpore dans sa fonction d'évaluation différents critères du second ordre dont les pondérations sont gérées par l'Oracle. Ces critères incluent une évaluation extensive de la structure de pions, un critère mesurant l'activité des pièces, la paire de fous, les tours connectées et, dans les finales de pions, un critère sur les carrés des Rois. Des évaluations spéciales sont incorporées pour les fins de parties : Roi dépouillé, Roi et pion contre Roi.

Joker : JOKER est le frère cadet d'ECUME dont il a hérité une partie du moteur en assembleur. La principale différence provient de l'utilisation d'extensions entropiques [4]. Dans tous les tests comparatifs que nous avons faits entre JOKER et ECUME, il apparaît que JOKER est plus faible qu'ECUME, du fait que les extensions entropiques étendent trop de branches non utiles, ce qui rend le programme JOKER plus faible tactiquement. Nonobstant ce handicap, le programme JOKER a toujours eu de meilleurs résultats que le programme ECUME dans les compétitions [5]. Il est ainsi « Champion de France des ordinateurs 1993 ».

2. cf 3.2.4

3. Les programmes ECHEC et CUMULUS étant écrits en langage C.

4. voir le § 3.2.5.

5. Ce résultat, quoi qu'en grande partie absurde, peut s'expliquer par le fait que le programme JOKER incorpore dans sa fonction d'évaluation le critère d'entropie calculé pour les définitions des extensions de recherche. Ce critère rend le programme plus « prudent », il minimise son entropie, et plus « agressif », il maximise l'entropie de l'adversaire.

6.2 Parties de championnats

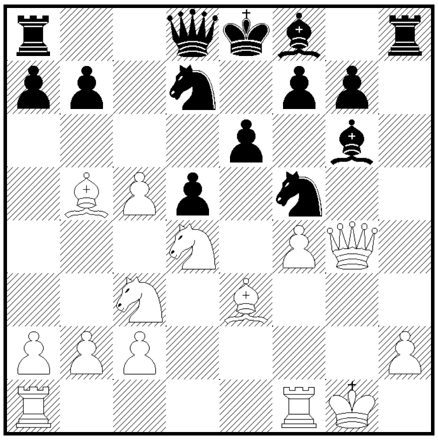
6.2.1 Computers Olympiads

Les Olympiades Informatiques organisées par David Levy furent les premières sorties de notre programme ECHEC en tournoi. Les échecs sur micro-ordinateurs étaient alors largement dominés par les machines dédiées (des marques MEPHISTO, FIDELITY) et aucun programme sur PC ne semblait pouvoir les inquiéter. Notre programme écrit en C sur un 386 16Mhz allait pourtant faire trembler les favoris, réussissant même à contraindre l'incontesté numéro 1 d'alors, le MEPHISTO de l'anglais Richard Lang, à forcer la nulle par échec perpétuel. Mais notre programme était encore à un stade bien expérimental et faisait montre d'un niveau de jeu très irrégulier.

Echec 1.5 – Rebell

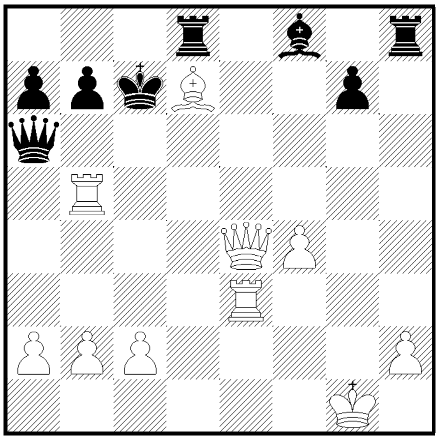
LONDRES, 1990

1 e4 c6 2 d4 d5 3 e5 ♗f5 4 ♘c3 e6 5 g4 ♗g6 6 ♘e2 c5 7 ♗e3 ♘c6 8 d×c5 ♘×e5 9 ♘d4 ♘e7 10 f4 ♘d7 11 ♗b5 h5 !? Donne une porte de sortie au Fg6 et questionne les pions blancs de l'aile roi. Cependant les noirs avaient une priorité absolue : le développement et le contrôle du centre, et ce coup n'y contribue en rien. **12 O–O !** N'ayant pas peur des fantômes, les blancs roquent du côté le plus ouvert de la position. Ils gardent ainsi le contrôle du centre et une grande avance de développement qui oblige les noirs à la plus grande prudence. Moins convaincant était 12.f5. [12 f5 e×f5 13 c6 b×c6 14 ♘×c6 ♘×c6 15 ♗×c6 ♖c8 16 ♕×d5 ♕c7 17 g×f5 ♕×c6] Et les noirs se libèrent alors que les blancs ont perdu leur avantage. **12 ... h×g4 !?** Suite logique du coup précédent, mais donne aux blancs une avance de développement écrasante, et un contrôle du centre total. La punition ne devrait pas tarder. **13 ♕×g4** La position blanche a fière allure! **13 ... ♘f5**



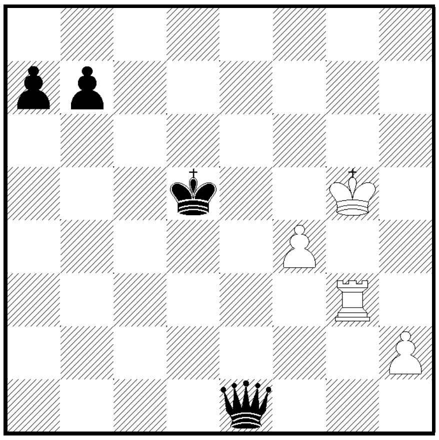
[13 ... a6 14 ♗a4 ♖c8 15 ♖ae1 ♗h5 16 ♕g5 ♘c6 17 ♘×c6 ♕×g5 + 18 f×g5 b×c6] permettait peut être de tenir **14 ♘×e6!!** Un humain dirait que la position est « mûre ». La combinaison qui suit est beaucoup trop profonde pour les machines de l'époque en temps de tournoi : il s'agit en fait d'un sacrifice positionnel. **14 ... ♘×e3?** Si Rebell avait été capable de voir ce qui va suivre, nul doute qu'il aurait joué Dh4, qui laisse malgré tout un bon avantage aux blancs. [14 ... ♕h4 15 ♕×h4 ♖×h4 16 ♘×f8 ♔×f8 17 ♗f2 ♖g4 + 18 ♔h1 ♖d8 ±] **15 ♖ae1!** La position est gagnante pour les blancs, mais extrêmement compliquée tactiquement. Aucun des deux adversaires n'était capable d'analyser correctement les positions qui vont suivre dans le temps imparti, et le hasard va jouer un rôle crucial, avec un biais en faveur du matériel le plus puissant : le Rebell avait donc une chance de s'en sortir. **15 ... ♗e4** [15 ... ♘×g4 16 ♘×g7#] [15 ... ♗e7 16 ♖×e3 f×e6 17 ♕×g6+ ♔f8 18 f5 e5 19 ♖g3 ♗f6 20 ♘×d5 ♖h6 21 ♕g4 ♘×c5 22 ♖d1+–] Les deux Dames sont en prise, mais la Dame blanche est imprenable. **16 ♖×e3**[16 ♘×e4 d×e4 (16 ... ♘×g4 17 ♘c7+ ♔e7 (17 ... ♕×c7 18 ♘d6+ ♔d8 19 ♖e8#) 18 ♘×d5+ ♔e6

19 ♘f6+ ♘e3 20 ♖×e3+ ♘e5 21 ♖×e5#) 17 ♖×e3 f×e6 18 ♕×e6+ ♕e7 19 ♗×d7+ ♔d8 20 ♖d1 ♕×e6 21 ♗×e6+ ♔e8+−] était également possible **16 ... f×e6 17 ♕×e6+ ♕e7** [17 ... ♗e7 18 ♘×e4 d×e4 19 ♖d1 a6 20 ♖×d7 a×b5 21 ♖×d8+ +−] **18 ♗×d7+ ♔d8 19 ♘×e4 d×e4** [19 ... ♕×d7 20 ♘d6 ♗×d6 21 c×d6 ♖e8 22 ♕×e8+ ♕×e8 23 ♖×e8+ ♔×e8 24 ♖e1+ ♔d7 25 ♖e7+ ♔×d6 26 ♖×b7 g6 27 ♖g7 +−] **20 ♖d1 ♔c7** [20 ... ♕×e6 21 ♗×e6+ ♔c7 22 ♖d7+ ♔c6 23 b4 +−] **21 ♕d5 ♕×c5** [21 ... ♖d8 22 c6 ♕c5 23 ♕×e4 ♖h3 24 ♖e1 ♖×e3 25 ♖×e3 b×c6 26 ♗e6 +−] **22 ♕×e4 ♖d8** Dh5 était vraisemblablement un peu plus résistant. **23 ♖d5 ♕b6** [23 ... ♖×d7 24 ♕e5+ ♕d6 25 ♖d3 +−] **24 ♖b5?** Au tour d'ECHEC de se fourvoyer dans la jungle des variantes. L'écrasement s'obtenait par De5+. La position n'en reste pas moins gagnante pour les blancs. [24 ♕e5+! ♗d6 25 ♖×d6 ♕×e3 + (25 ... ♕×d6 26 ♖c3+ ♔×d7 27 ♕×g7+ ♔e6 28 ♖e3+ ♔d5 29 ♕×b7+ ♔c5 30 ♖c3+ ♔d4 31 ♖d3+ ♔c5 32 b4+ ♔c4 33 ♕e4+ ♕d4 + 34 ♖×d4+ ♖×d4 35 ♕c6+ ♔×b4 36 c3+ +−) 26 ♕×e3 ♔×d6 27 ♗f5 +−] **24 ... ♕a6**



[24 ... ♖×d7 25 ♖×b6 a×b6 26 b4 +−] **25 ♖c3+?** Une nouvelle imprécision. A nouveau, les blancs pouvaient achever les noirs, mais aucun programme sur PC de l'époque n'était capable de trouver comment et pourquoi, même avec une heure de réflexion. [25 ♖b3!! ♕b6 + (25 ... b6 26 ♕c6+ ♔b8 27 ♖×b6+ a×b6 28 ♖×b6+ ♕×b6 + 29 ♕×b6+ ♔a8 30 ♗c6#) (25 ... ♖×d7 26 ♖×b7+ ♔d8 27 ♖b8+ ♔c7 28 ♖3b7+ ♕×b7 29 ♕×b7+) (25 ... ♗c5 + 26 ♖×c5+ ♔b8 27 ♖a5) 26 ♖×b6 ♗c5 + 27 ♔h1 a×b6 28 ♖d3 ♖×d7 29 ♕e5+ ♗d6 30 ♖×d6 ♖×d6 31 ♕×g7+] **25 ... ♔b8 26 ♖d3** Dans cette position d'une grande mobilité (62 coups légaux) le programme doté du matériel le plus puissant jouit d'un avantage considérable. A nouveau, les blancs ne trouvent pas le coup le plus fort. [26 ♕d5 ♗e7 27 ♖b3 ♕g6 + 28 ♖g3 ♕a6 29 ♖×g7 ♗f6 30 ♖g6 ♗d4 + 31 ♔h1] **26 ... ♖h6 27 ♗c6?** Dilapide une partie de l'avantage. Les blancs pouvaient maintenir la pression et gagner facilement par : [27 ♗g4 ♖d6 28 a4 ♖×d3 29 c×d3 ♗d6 30 ♗f3 b6+−] **27 ... ♖×d3 28 ♕e8+ ♔c7 29 c×d3 ?**Après ce coup la position blanche est toujours supérieure, mais le gain n'est plus aussi certain. [29 ♖×b7+ ♕×b7 30 ♗×b7 ♖d1 + 31 ♔g2 ♔×b7 32 ♕×f8 ♖d7 33 c4 ±] **29 ... ♖×c6** [29 ... ♕×c6 30 ♕f7+ ♗e7 31 ♕×e7+ ♔b8 32 ♕f8+ ♕c8 33 ♖×b7+ ♔×b7 34 ♕×g7+ ♕c7 35 ♕×h6+−] **30 ♕f7+ ♔b8 31 ♕×f8+ ♖c8 32 ♕b4??** Cette fois-ci les blancs perdent tout avantage. **♕g6 + 33 ♔f2 ♕×d3 34 ♖e5?**[34 ♖c5 ♖h8 35 ♖c3 ♕d2 + 36 ♔f3 ♕d1 + 37 ♔g3 ♕g1 + 38 ♔f3=] **34 ... ♖c2! +** Évident et fort. **35 ♔g1 ♕g6 + 36 ♖g5 □♖c1 + 37 ♔f2??** Finalement, les blancs jouent un coup carrément perdant dans une position d'équilibre dynamique. [37 ♔g2 ♕d3 38 ♖e5 ♖c2 + 39 ♔g1 ♕d1 + 40 ♕e1 ♕d4 + 41 ♕e3 ♕×e3 + 42 ♖×e3 ♖×b2 =] **37 ... ♕d3 38 ♕f8+ ♖c8 39 ♖f5** Les blancs sont menacés de mat et leur dame est en prise. Ils ne peuvent pas parer les deux menaces à la fois. **♖×f8 40 ♖×f8+ ♔c7 41 ♖f7+ ♔c6** La position est clairement gagnée pour les noirs, mais après une telle partie il est difficile de ne pas croire aux miracles! **42 b3 ♕c2 + 43 ♔f3 ♕×a2 44 ♖×g7 ♕×b3 + 45 ♔g4 ♕d1 + 46 ♔h4 ♕e1 + 47 ♖g3**

♔d5 48 ♔g5

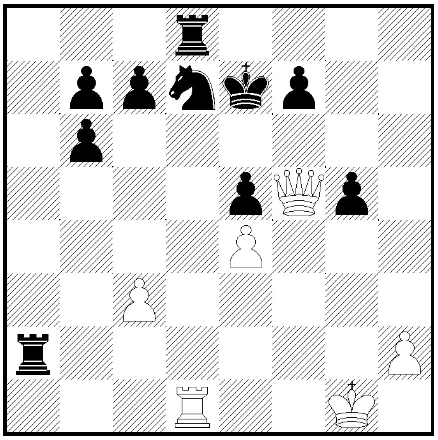


48 . . . ♕×g3?? + Et le miracle s'accomplit! Les noirs appliquent bêtement la règle du carré : leur Roi est dans le carré des pions blancs alors que le Roi blanc n'est pas dans celui des pions noirs. Mais la règle du carré n'a jamais été applicable avec plus d'un Roi et un pion. Croyant simplifier dans une finale facilement gagnante, les noirs rentrent dans une finale de pions nulle. **49 h×g3 b5 50 f5 b4 51 f6 ♔e6 52 ♔g6 b3 53 f7 b2 54 f8=♕ b1=♕ + 55 ♔g5 ♕c1 + 56 ♕f4**[56 ♔h5=] ne changeait rien **56 . . . ♕×f4 + 57 ♔×f4** La position est une finale de pions nulle, mais difficile à calculer pour des programmes qui n'ont pas de table de transposition. **♔d5 58 ♔f5 ??**A peu près le seul coup perdant dans la position, joué sous l'oeil du grand expert britannique des finales Jon Speelman. Re3 annulait sans peine, et g4 assurait également le partage du point. **a5 59 g4 a4 60 g5 ♔d6 61 g6 ♔e7 62 ♔g5 ♔f8 63 ♔h6 ♔g8 0–1**

Mephisto – Echec 1.5

Londres 1989

1 e4 MEPHISTO, l'incontesté numéro 1 des programmes sur micro, écrit en assembleur sur un 68030 36 MHz, était bien sûr grand favori face à un programme amateur inconnu, écrit en C et tournant sur un 386 à 16 MHz. **e5 2 f4** Une surprise de la part de MEPHISTO, qui avait jusque là une prédilection pour les débuts fermés et les parties positionnelles. **2 . . . ♗c5 3 ♘f3 d6 4 ♘c3 ♘f6 5 ♗c4 ♘c6 6 d3 ♗g4 7 ♘a4 ♗b6 8 ♘×b6 a×b6 9 O–O ♗×f3? 10 ♖×f3 ±O–O 11 c3 ♘a5 12 f×e5 d×e5 13 ♗g5 ♘×c4 14 ♗×f6 ♘×b2 15 ♕d2 g×f6 16 ♖h3 ♖e8 17 ♕h6 ♕×d3 □18 ♕×h7+ ♔f8 19 ♖×d3 ♘×d3** Les noirs ont Tour Cavalier et pion pour la Dame : cela peut être une compensation suffisante. Les pions blancs sont disloqués, mais le Roi noir n'est pas en sécurité : c'est une position difficile à juger pour une machine. « Optiquement », les blancs sont cependant nettement mieux. **20 ♖d1 ♘c5 21 ♖f1 ♔e7 22 ♕f5 ♘d7** Les noirs ont réussi à construire une position solide. S'ils ne succombent pas à une combinaison, l'issue de la partie dépendra de la faiblesse des pions blancs et de la vélocité du pion h. **23 ♖d1 ♖ed8 24 g4 ♖×a2 25 g5** Un coup naturel, mais MEPHISTO a une idée précise *en tête*. Estimant sa position inférieure –et ses pions sont effectivement faibles– il a vu une manière forcée d'obtenir la partie nulle. **f×g5**

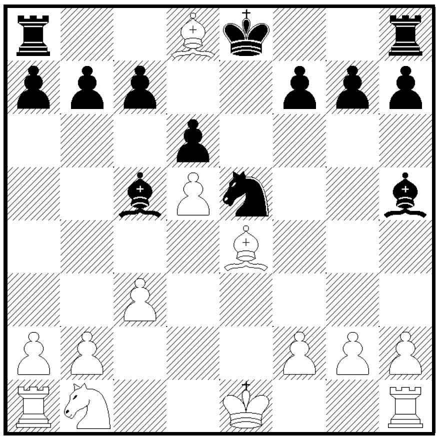


26 ♖×d7+ ?! [26 ♕×g5+ f6 27 ♕g7+ ♔e6 ±] **26 … ♖×d7 27 ♕×e5+ ♔f8 28 ♕h8+ ♔e7 29 ♕e5+ ♔d8 30 ♕h8+ ♔e7 31 ♕e5+ $\frac{1}{2}$ — $\frac{1}{2}$**

Echec 1.8 – Zugzwang

LONDRES 1990

1 e4 e5 2 ♘f3 ♘c6 3 c3 Le début Ponziani, une arme un peu antique mais intéressante qui peut conduire à des variantes très tranchantes. **♘f6 4 d4 ♘×e4 5 d5 ♘e7 6 ♘×e5 ♘g6 7 ♗d3 ♘×e5 8 ♗×e4 ♗c5 9 ♕h5!?**[9 O–O d6 10 b4 ♗b6 11 a4 a5 12 ♕c2 ±] **9 … d6 10 ♗g5 ♗g4 11 ♗×d8 ♗×h5**



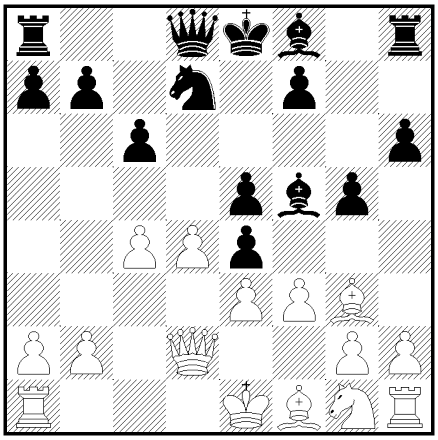
12 ♗×c7 ?![12 ♗h4 ♗g6 =] **12 … ♖c8 13 ♗a5 b6 14 b4 ♗×f2 + 15 ♔×f2 b×a5 16 b×a5 ♖c5 17 ♖f1 ♗g6 18 ♗×g6 h×g6 19 ♔g1 ♖×d5 20 ♖e1 ♖×a5 21 ♘d2 ♔d7 22 ♘b3 ♖d5 23 ♖e4 ♘c6 24 c4 ♖f5 25 ♖d1 ♔c7 26 ♘d4 ♘×d4 27 ♖×d4 ♖f6 28 ♖4d3 ♖h5 29 ♖a3 a5 30 ♖d5 ♖f5 31 h3 ♖×d5 32 c×d5 ♔b6 33 ♖c3 ♔b5 34 ♖c6 ♔b4 35 ♖a6 a4 36 ♖c6 a3 37 h4 ♔b5 38 g3 ♖f3 39 ♔g2 ♖d3 40 ♖c7 ♖d2 + 41 ♔f3 ♖×a2 42 ♖×f7 ♔b6 43 ♖d7 ♖b2 44 ♖×d6+ ♔a7 45 ♖×g6 ♖b3 + 46 ♔e4 ♖b7 47 ♖c6 a2 48 ♖c1 ♖b1 49 ♖c7+ ♔b8 50 ♖×g7 a1=♕ 51 ♖g8+ ♔c7 52 ♖g5 ♕f6 53 ♔e3 ♖b3 + 54 ♔e4 ♖b4 + 55 ♔e3 ♕c3 + 56 ♔f2 ♖b2 + 57 ♔f1 ♕c1 #**

6.2.2 Championnats du Monde des Micros

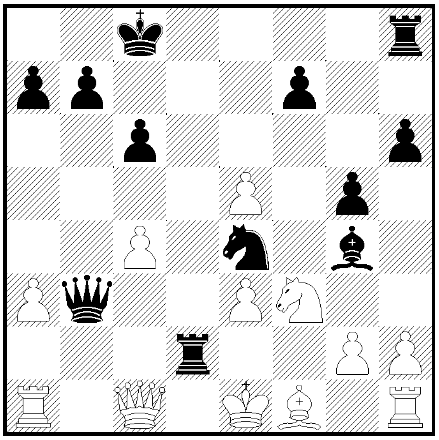
The King – Echec 1.9

LYON, 1990

1 d4 d5 2 c4 c6 3 ♘c3 ♘f6 4 ♗g5 ♘e4 5 ♘×e4 d×e4 6 ♕d2[6 e3?? ♕a5 + 7 ♕d2 ♕×g5 –+] **6 … ♗f5 7 e3 ♘d7 8 f3 h6 9 ♗h4 g5 10 ♗g3 e5 !?**



Une variante à double tranchant, qui donne un excellent jeu de pièces aux noirs tandis que les blancs ont un léger retard de développement. Le fort centre blanc leur donne l'avantage, mais nous avions jugé, à l'époque, que cette variante était bonne pour les noirs entre machines. Les blancs, quant à eux, étaient hors bibliothèque depuis longtemps déjà. **11 d×e5?!** Prendre le pion est bien trop dangereux. Les blancs atteindraient une position jugée comme favorable par la théorie après o-o-o. Nous ne sommes pas certains que cela soit le cas entre deux programmes. **♕b6 12 ♔e2?!** Les blancs « paniquent » dans une position délicate. [12 a3 ♗c5 13 ♕c3 O–O–O ∓] **12 ... ♗b4 13 ♕c1 ♘c5 14 f×e4 ♘×e4** [14 ... ♗g4 + 15 ♘f3 O–O–O] **15 ♘f3 O–O–O 16 ♗e1 ♗×e1 17 ♔×e1**[17 ♕×e1 ♕×b2 + 18 ♕d2 ♖×d2 + 19 ♘×d2 ♕×d2 + 20 ♔f3 ♕f2 #] **17 ... ♕a5! + 18 ♔e2 ♗g4 19 b4 ♕×b4 20 a3 ♖d2 + 21 ♔e1 ♕b3**



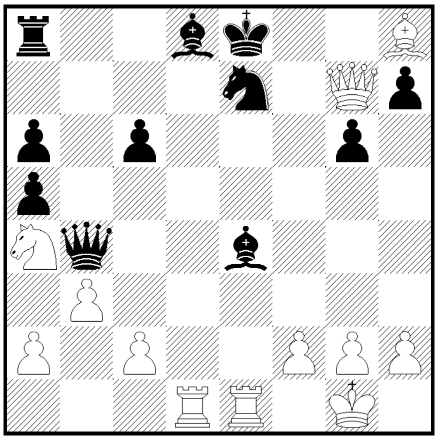
et 0-1 car si [22. Be2 Q×e3 23. Q×d2 N×d2 24. ♔a2 ♔d8 −+]

La partie CUMULUS-BB sera le tournant du championnat du monde sur micro-ordinateurs 1990. En effet, elle démontre la présence d'un bogue dans le programme CUMULUS, bogue que nous ne parviendrons pas à isoler pendant les rondes suivantes et dont nous ne savions pas s'il était dans les parties communes avec le programme ECHEC. Il s'avérat lorsque le problème fut corrigé qu'il n'affectait que CUMULUS, mais pendant le tournoi, nous n'arrivâmes pas à le démontrer.

Cumulus-BB

Lyon, 1990

1 e4 e5 2 ♘f3 ♕f6? Une partie étrange qui démontre qu'il n'est pas aussi facile de mettre au point un programme d'échecs que le laisseraient penser les parties généralement publiées. Les noirs jouent très faiblement, et les blancs vont, dans un premier temps, exploiter ces faiblesses magistralement. **3 ♘c3 ♘a6?**
4 ♗×a6 b×a6 5 ♘d5 ♕d6 6 d4 f6 7 O–O c6 8 ♘c3 ♕b8 9 ♘h4 g6 10 ♘f3 6a5 11 ♕d3 e×d4? 12 ♘×d4 ♗d6 13 ♘f3 a6 14 b3 ♗c5 15 ♘a4 ♗d6 16 ♗e3 ♗c7 17 ♗d4 ♗d8? 18 e5! d6 19 ♖fe1 f×e5 20 ♘×e5 d×e5 21 ♗×e5 ♕b7 22 ♗×h8+ ♘e7 23 ♕c4 ♔f8 24 ♕h4 ♗f5 25 ♕h6+ ♔e8 26 ♕g7 ♕b4 27 ♖ad1 ♗e4



Les Blancs sont nettement mieux. Ils peuvent capturer la Dame noire par [28 c3!] CUMULUS a bien vu ce coup et son score est supérieur à 10 pions quand ... **28 ♖×e4??** CUMULUS annonce un mat imaginaire en 5 coups. Le bogue est évident et causera la perte de cette partie. CUMULUS pour obtenir ce mat imaginaire donnera la plupart de ses pièces.**28 ... ♕×e4 29 ♘b6?? ♗×b6 30 ♖f1 ♕×c2 31 ♕b2 ♕×b2 32 ♗×b2 ♖d8 33 ♗c1 ♗d4 34 ♗a3 ♗f6 35 ♗c1 ♖d3 36 h3 ♔d7 37 ♔h2 ♗e5 + 38 ♔g1 ♘d5 39 ♖e1 ♗c3 40 ♖f1 ♗d4 41 ♖e1 ♘c3 42 ♗e3 ♗×e3 43 f×e3 ♘×a2 44 ♖a1 ♘b4 45 ♖×a5 ♖×e3**

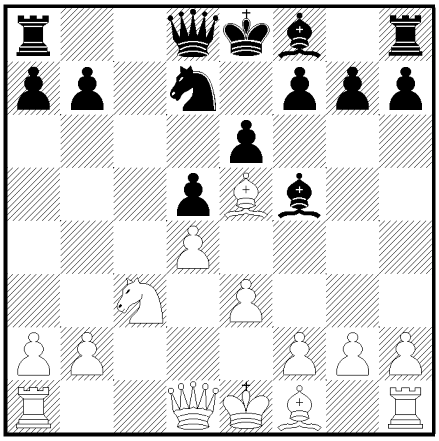
46 ♖a3 c5 47 ♔f2 ♖c3 48 ♖a1 ♖×b3 49 ♖d1+ ♔c7 50 g3 ♘d3 + 51 ♔g2 ♖b2 + 52 ♔f3 ♘f2 53 ♖a1 ♔b6 54 h4 ♘d3 55 ♔e4 ♘b4 56 ♖d1 ♔c7 57 ♖c1 ♔c6 58 h5 g×h5 59 ♖h1 ♘d5 60 ♖a1 ♖e2 + 61 ♔f5 ♔b5

La partie suivante fut jouée à la dernière ronde, après la partie CUMULUS - BB. Craignant que le bogue qui s'y était manifesté fût également présent dans Echec, nous avions décidé de mettre un « contempt factor » élevé : le programme rechercherait la partie nulle tant que son avantage ne dépasserait pas quatre pions.

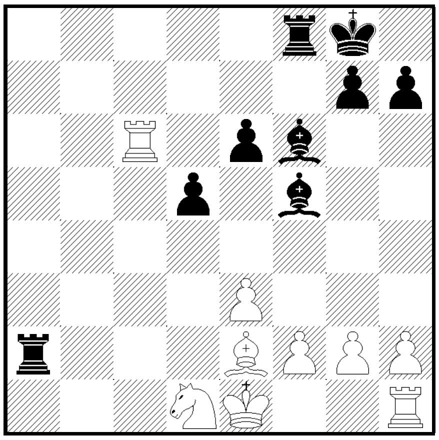
Mephisto – Echec 1.9

LYON, 1990

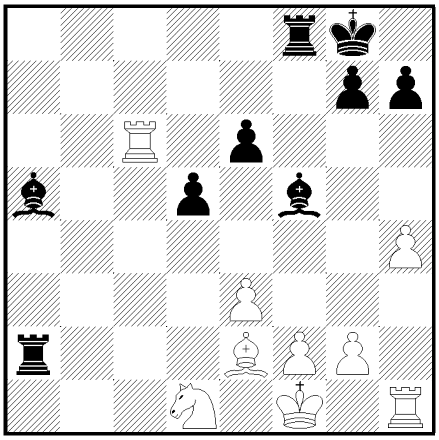
1 d4 d5 2 c4 c6 3 c×d5 c×d5 4 ♘f3 ♘f6 5 ♘c3 ♘c6 6 ♗f4 ♗f5 7 e3 e6 8 ♘e5 ♘×e5 9 ♗×e5 ♘d7!?



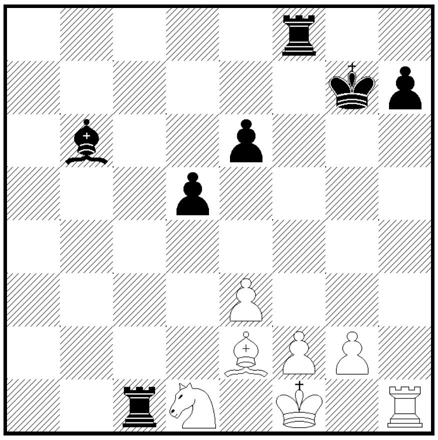
Un bon coup, mais douteux pour une machine, qui n'est jamais à l'aise une fois déroquée dans l'ouverture. Mieux valait 9...a6! pour empêcher définitivement Fb5. Un bon exemple de variante où le meilleur coup selon la théorie des ouvertures n'est pas le plus indiqué pour un programme. **10 ♕b3 ♘×e5 11 d×e5 ♗e7 12 ♕×b7?!** Ce coup n'est même pas envisagé par la théorie, mais le pion est trop tentant pour une machine : pour construire une bonne bibliothèque d'ouvertures, on ne peut pas se contenter de recopier les livres. A la décharge de Mephisto, il faut dire que son évaluation à ce moment de la partie était proche de l'égalité malgré le gain d'un pion. Sur Fb5+ la partie est incertaine, mais les noirs sont déroqués : [12 ♗b5+ ♔f8 13 O–O (13 ♗e2 ♕b6∞) 13 ... ♕b6 14 ♘a4 ♕c7 ∞] (Youssoupov) **12 ... O–O** Plus de risque d'être déroqués pour les noirs qui vont maintenant gagner beaucoup de temps en chassant la dame blanche. **13 ♖c1 ♖b8 14 ♕×a7 ♖×b2 15 a3 f6!** En avance de développement et en possession de la paire de Fous, les noirs ont tout intérêt à ouvrir le centre, surtout si c'est en échangeant leur pion f contre le pion e5. Le dédoublage des pions blancs est sans importance. **16 e×f6 ♗×f6 17 ♘d1** La position blanche est déjà stratégiquement perdue. Après l'ouverture de la colonne f, toutes les pièces noires jouent alors que les blancs n'ont pas encore développé leur aile roi et que leur dame est exposée. Rien d'étonnant à ce que ce repli soit forcé. [17 ♗e2 d4 18 ♘d1 (18 e×d4 ♗×d4 19 ♕a6 ♗×c3 + 20 ♖×c3 ♖b1 + 21 ♗d1 ♕×d1 #) 18 ... d3 19 ♘×b2 d2 + 20 ♔d1 d×c1=♕ + 21 ♔×c1 ♗×b2 + 22 ♔×b2 ♕d2 + 23 ♔a1 ♕×e2 –+] [17 ♗a6 ♗×c3 + 18 ♖×c3 ♖b1 + 19 ♔e2 ♖×h1 –+] **17 ... ♖a2 18 ♗b5 ♕d6 19 ♕c5** Roquer de suite était peut-être plus prudent. [19 O–O ♖×a3 20 ♕d7 ♕×d7 21 ♗×d7 ♖a7 ∓] **19 ... ♕×c5 20 ♖×c5 ♖×a3 ∓21 ♖c6 ♖a2 22 ♗e2**



Le Fou n'a pas de bonne case. Ici, il n'est protégé que par le Roi : les blancs ne peuvent pas roquer. Une situation qui n'est pas sans rappeler la partie CHESS GENIUS - Anand qui eu lieu au tournoi PCA de Londres, en août 1994 (CHESS GENIUS est une version du programme MEPHISTO sur PC). **22 ... ♗d8** Les noirs arrivent encore à améliorer leur position : le Fou noir sera plus utile à l'aile dame d'où il pourra viser le monarque blanc, incapable de s'échapper. **23 h4** Les blancs tentent de se libérer comme ils peuvent. Leur position est perdue. [23 ♘c3? ♗a5 −+ 24 f4 ♖c2 25 O–O ♖×c3] **23 ... ♗a5 ! + 24 ♔f1**



24 ... ♗c2!! −+ 25 ♖×c2 Pour parer la menace Tb8-b2, les blancs doivent donner la qualité. **♖×c2 26 h5 ♖c1 27 h6 ♗b6 28 h×g7 ♔×g7 29 ♔e1 ♗a5 + 30 ♔f1 ♗b6 31 ♔e1 ♗a5 + 32 ♔f1 ♗b6 ??**



Il suffisait de continuer par Tb8... L'évaluation d'ECHEC étant inférieure à quatre pions, il choisit de répéter la positon. Sur le moment nous étions soulagés : le bogue n'aurait plus l'occasion de venir gâcher une partie gagnante. Ce n'est qu'après le tournoi que nous vîmes qu'ECHEC n'était pas affecté par le même problème que CUMULUS. **33 ♔e1** $\frac{1}{2} - \frac{1}{2}$

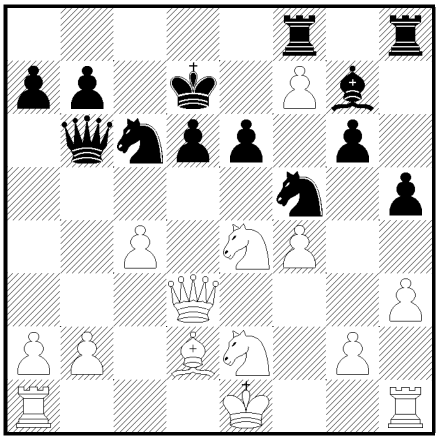
6.2.3 Championnats du Monde des Ordinateurs

Cumulus 2.0 – Echec 1.95b

MADRID 1992

1 e4 g6 2 d4 ♗g7 3 ♘c3 d6 4 f4 ♘f6 5 ♘f3 c5 6 ♗b5+ Une variante aiguë de la défense Pirc. **♗d7 7 e5 ♘g4 8 e6 ♗×b5 9 e×f7+ ♔d7 10 ♘×b5 ♕a5 + 11 ♘c3 c×d4 12 ♘×d4 h5 13 h3 ♘c6 14 ♘e2 ♘h6 15 ♗e3**

♖af8 16 ♕d3 ♘f5 17 ♗d2 e6 18 ♘e4 ♕b6 19 c4

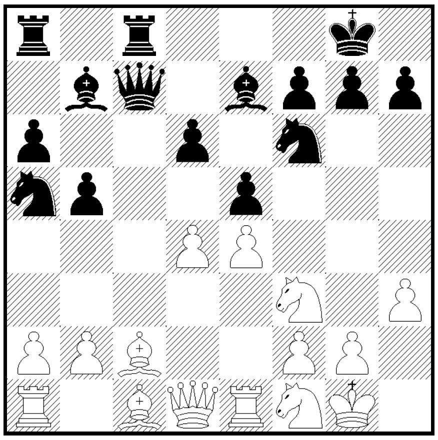


Les blancs se sont mieux sortis de l'ouverture, et ils s'emploient maintenant à fixer la faiblesse d6, défendue seulement par le Roi noir. L'attaque d'un pion faible sur une colonne semi-ouverte fait partie des stratagèmes de base dans l'arsenal de l'Oracle dont nous avons dôté CUMULUS 2.0. **19 … ♘d8 20 c5 ♕×b2 21 ♖b1 ♕×a2 22 ♘×d6**[22 c×d6 ♕d5 23 ♕c2 ♕c6 24 ♘c5+ ♔c8 25 d7+ ♔b8 (25 … ♔c7 26 O–O ♘d4 27 ♕d3 ♘×e2 + 28 ♕×e2 ♗d4 + 29 ♗e3 ♗×c5 30 ♖fc1 b6 −+) 26 ♘a6+ ♔a8 =(26 … ♕×a6?? 27 ♕c8#)] **22 … ♘×d6 23 c×d6 ♖×f7 24 ♕×g6 ♕d5 25 ♗c3 ♖g8 26 ♗b4 ♖f8 27 ♕h7 ♖e8 28 O–O ♘f7 29 ♔h1 ♖ef8 30 ♖f3 ♕c4 31 ♗c3 b6 32 ♗×g7 ♕×e2 33 ♖c3 ♖c8 34 ♖×c8 ♖×c8 35 ♗e5 ♖f8 36 ♖c1 ♕d2 37 ♖c7+ ♔e8 38 ♔h2 h4 39 ♕g7 ♕×d6 40 ♗×d6 ♖h8 41 ♕×f7+ ♔d8 42 ♕e7#**

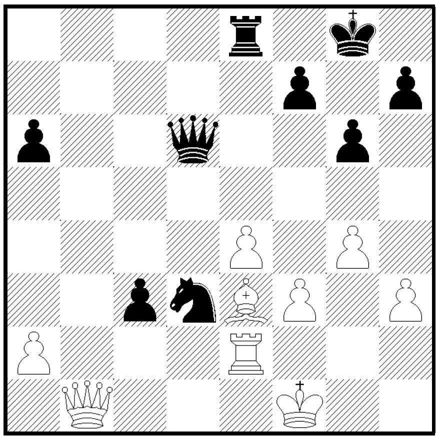
Zugzwang – Cumulus 2.0

MADRID, 1992

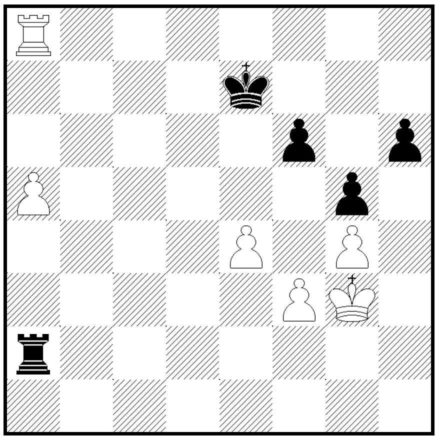
1 e4 e5 2 ♘f3 ♘c6 3 ♗b5 a6 4 ♗a4 ♘f6 5 O–O ♗e7 6 ♖e1 b5 7 ♗b3 O–O 8 c3 d6 9 h3 ♘a5 La défense Tchigorine de la partie espagnole : un système très exploré, mais dont les subtilités ne se laissent pas maîtriser aisément. Les machines n'y sont vraiment pas à l'aise. **10 ♗c2 c5 11 d4 ♕c7 12 ♘d2 c×d4 13 c×d4 ♗b7 14 ♘f1 ♖fc8!?**



Jusqu'ici tout était bien connu et répertorié, mais pour son premier coup hors bibliothèque CUMULUS se trompe de Tour. La Tour f sera sûrement utile sur la colonne d (ou e). Quant à l'autre Tour, elle sera bien placée si la colonne a s'ouvre par a4xb5, mais pour l'instant elle n'est pas très active. La théorie ne connaît que Tac8 dans cette position (contre attaque Panov). **15 ♗d3 e×d4?** Cette altération de la structure de pions n'est pas heureuse. L'idée des noirs devrait être de jouer d5, et c'est effectivement le coup théorique lorsque c'est l'autre Tour qui est venue en c8. **16 ♘×d4 ♘c4 17 ♘f5 g6 18 ♘×e7+ ♕×e7 19 ♗×c4 ♖×c4** Malgré leurs faiblesses (affaiblissement de l'aile roi en l'absence du Fou des cases noires et pion d6), les noirs ont une position dynamique, et surtout une avance de développement. Ils doivent maintenant essayer de se débarrasser de d6 en poussant d6-d5. **20 f3 ♕e5 21 ♗e3 ♕×b2?!** Risqué, et il n'est jamais conseillé d'échanger un pion central contre une pion de l'aile. Cependant, CUMULUS voit qu'il se débarrasse d'une faiblesse (d6) pour se créer une majorité à l'aile dame : cette aventure lui semble correcte d'un point de vue positionnel. **22 ♕×d6 ♖e8 23 ♖ac1 ♖a4 24 ♖cd1 ♕e5 25 ♕b6 ♗c8 26 ♖d2 ♗e6 27 ♖c1 ♗c4 28 ♗d4** Les noirs arrivent à garder une position très active. **♘d7 29 ♕b7 ♕e6 30 ♘e3 ♘f6 31 ♖c2 ♘h5** Vise f4, après quoi la pression noire sur les cases blanches de l'aile roi noire sera très désagréable. **32 g4? ♘f4!** ∓Maintenant il est clair que les noirs sont beaucoup mieux. **33 ♘×c4 ♖×c4 34 ♖×c4 b×c4 35 ♗e3 ♘d3 36 ♖e2** Les noirs sont pratiquement gagnants, mais la réalisation technique n'est pas commode, surtout contre un adversaire aussi rapide. **c3 37 ♕b1 ♕d6 38 ♔f1**



♔g7?? Considérant l'issue de la partie comme normale compte tenu du rapport de forces (1024 tranputers contre un PC), la plupart des commentateurs sont passés à côté de Td8!!, et tout leur jugement de la position et de la partie s'en est trouvé erroné. Il n'y a pourtant rien d'étonnant à ce que les noirs aient un coup gagnant dans cette position. Le coup de la partie transforme une position gagnante en position difficile à défendre. Après correction des autres bogues révélés pendant ce tournoi, CUMULUS trouve rapidement Td8. [38 … ♖d8!! −+ 39 f4 ♘×f4 40 ♖e1 ♘×h3 41 ♕c2 ♕f6 + 42 ♔g2 ♘f4 + 43 ♔g3 ♘d3 −+] **39 ♕b6 ♕×b6 40 ♗×b6 ♖b8 41 ♗d4+ f6** Le pion c perdu, les noirs sont condamnés à une difficile défense. **42 ♖c2 ♖b1 + 43 ♔e2 ♘f4 + 44 ♔e3 g5 45 ♗×c3 ♘×h3 46 ♗d4 ♖e1 + 47 ♔d2 ♖f1 48 ♖c7+ ♔g8 49 ♖c3 ♔f7 50 a3 ♘g1 51 ♖c7+ ♔g6 52 ♗×g1 ♖×g1 53 ♖a7 ♖b1 54 ♖×a6 ♔f7** Les finales avec ce type de structure de pions (pions des deux côtés à l'aile roi et pion blanc a passé) sont bien connues, mais difficiles à programmer. Cumulus n'avait pas les connaissances nécessaires pour la défendre correctement. **55 a4 ♖b3 56 ♔e2 ♖a3 57 ♔f2 ♖a2 + 58 ♔g3 ♔e7 59 a5 h6 60 ♖a8**



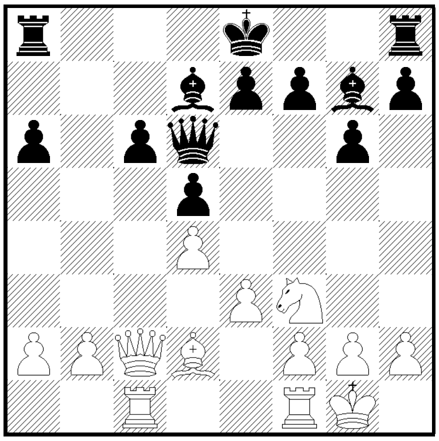
♔d7? Le Roi noir devait rester en g7 et h7 pour protéger les pions de l'aile roi, afin d'empêcher, une fois la position avec a7 Ta8 atteinte, Th8 Txa7 h7+- gagnant la Tour noire. Sur une défense convenable, la partie était vraisemblablement nulle. **61 a6 ♔c7 62 a7 ♖a6 63 ♖h8 ♔b7 64 a8=♗ + ♖×a8 65 ♖h7+ ♔c8 66 ♖×h6 ♖a6** Le Roi noir est trop éloigné de l'action pour que la partie soit défendable. **67 e5 ♔d7 68 e×f6 ♖a5 69 f4 ♖a3 + 70 ♔f2 g×f4 71 ♖h5 ♔e8 72 ♖f5 ♔f7 73 ♖×f4 ♔f8 74 g5 ♖a5 75 g6 ♖g5 76 g7+ ♔f7 77 ♔f3 ♖g6 78 ♔e4 ♖g1 79 ♔e5 ♖e1 + 80 ♖e4 ♖g1 81 ♖c4 ♖e1 + 82 ♔f5 ♖f1 + 83 ♖f4 ♖c1 84 ♖d4 ♖f1 + 85 ♔e5 ♖e1 + 86 ♔f4 ♖f1 + 87 ♔g5**

♖g1 + 88 ♖g4 ♖f1 89 ♖g3 ♖f2 90 ♖g1 ♖b2 91 ♖c1 ♖g2 + 92 ♔f5 ♖f2 + 93 ♔e5 ♖e2 + 94 ♔d4 ♖d2 + 95 ♔e3 ♖d8 96 ♖h1 ♖e8 + 97 ♔f4 ♖e6 98 ♖h6 ♖e1 99 ♖h8 ♖f1 + 100 ♔g5 ♖g1 +

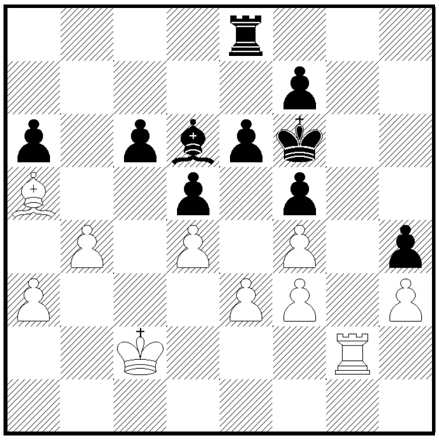
Hiarcs – Cumulus 2

MADRID, 1992

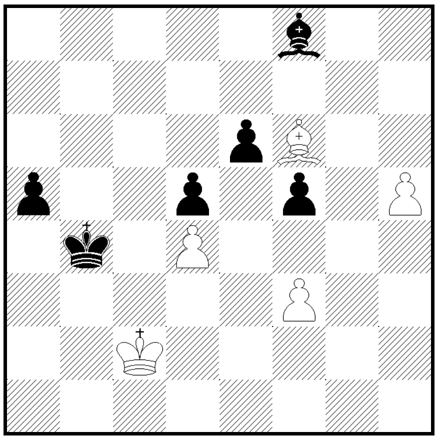
1 d4 Une partie décisive à la dernière ronde. HIARCS, sur Sparc 2, contre ECUME, chacun ayant besoin du point pour finir le tournoi en beauté. **d5 2 c4 c6 3 ♘c3** La défense slave, très en vogue depuis quelques années, offre aux noirs une position solide et des chances d'obtenir plus de jeu que dans le gambit dame classique. **♘f6 4 e3** Une variante tranquille, qui indique l'intention des blancs d'amener la partie dans des chemins positionnels. **g6 5 c×d5 ♘×d5 6 ♘×d5 c×d5 7 ♗b5+ ♘c6 8 ♗d2 a6 9 ♗×c6+ b×c6** Les blancs cèdent la paire Fous pour créer un pion arriéré aux noirs sur la colonne c semi-ouverte. C'est logique dans la mesure où le Ff8, une fois développé en g7, ne pourra pas soutenir la poussée égalisatrice c6-c5 ; rien n'indique toutefois que cette finesse ait compté dans l'évaluation de HIARCS, qui devait simplement estimer que la paire de Fous n'avait pas d'importance dans la position. **10 ♖c1 ♕d6 11 ♕c2 ♗d7 12 ♘f3 ♗g7 13 O–O**



13 … ♖b8?! Il était urgent de roquer et de lier les Tours. Mais comme dans la partie ECUME – KALLISTO, ECUME considère le petit roque noir comme une faute. Il va donc se débrouiller pour achever son développement sans roquer. Malgré l'ingéniosité de la manoeuvre, il va perdre beaucoup de temps. **14 ♗a5 ♗g4!?** A nouveau, o-o était indiqué. **15 b4** Les blancs fixent la faiblesse c6, protègent ainsi le pion b et menacent vraiment la prise en c6. A nouveau, le thème du pion faible sur une colonne semi-ouverte est exploité de manière convaincante par une machine. Inconvénient : le Fou a5 est immobilisé, mais c'est difficile à voir pour un programme. La notion d'immobilisation durable d'une pièce n'est, à notre connaissance, encore maîtrisée par aucun programme. **♗×f3 16 g×f3 ♖c8 17 ♕e2 ♖a8 18 ♖c5 ♔d7 19 ♖c1 ♖hc8 20 ♕f1 ♕e6 21 ♕g2 ♕f5 22 ♕g4 e6 23 a3 ♗f8 24 ♕×f5 g×f5 25 ♖5c2 ♖ab8 26 f4** D'un côté comme de l'autre, il est difficile de progresser. La position est égale. **h5 27 ♔f1 ♗e7 28 ♔e2 h4 29 h3 ♗f6 30 ♖c5 ♖b5 31 ♖5c3 ♖b8 32 f3 ♗e7 33 ♖3c2 ♗d6 34 ♔d3 ♗e7 35 ♖c3 ♗d6 36 ♖1c2 ♗e7 37 ♔d2 ♖b5 38 ♖c1 ♗f6 39 ♔e2 ♖b8 40 ♔d3 ♗e7 41 ♖3c2 ♗d6 42 ♖g2 ♖g8 43 ♖g1 ♖×g2 44 ♖×g2 ♔e7 45 ♔c3 ♔f6 46 ♖g5 ♖c8 47 ♔d3 ♖h8 48 ♔d2 ♖e8 49 ♖g1 ♖c8 50 ♔d3 ♖b8 51 ♔e2 ♖c8 52 ♔d2 ♖a8 53 ♔d1 ♖e8 54 ♖g2 ♖h8 55 ♔c2 ♖e8**



56 a4 ?! Voulant éviter la nulle, les blancs modifient leur structure de pions, et prennent le risque d'affaiblir b4. **♖b8 57 ♔c3 ♖e8 58 ♖g1 ♖b8 59 ♖g5 ♗e7 60 ♖g2 ♗d6 61 ♖g1 ♗f8 62 ♖c1 ♔g6 63 ♖b1 ♗d6 64 ♖f1 ♔f6 65 ♖c1 ♔e7 66 ♔d3 ♔d7 67 ♖g1 ♔e7 68 ♖g2 ♔f6 69 ♖c2 ♗×b4 70 ♖b2 ♗d6 71 ♖c2 ♗b4** Ces coups ne deviennent clairs que lorsque l'on sait qu'ECUME estimait avoir intérêt à chercher la répétition et HIARCS à la fuir. **72 ♗c7 ♖b7 73 ♖×c6 ♔e7 74 ♗e5 f6 75 ♖c7+ ♖×c7 76 ♗×c7 ♔e8 77 ♗b6 ♔d7 78 ♔c2 ♔e7 79 ♔d1 ♔d7 80 ♔e2 ♔c8 81 ♔f2 ♔d7 82 ♔f1 ♔e8 83 ♔e2 ♔e7 84 a5 ♔d7 85 ♔d3 ♗e7 86 ♔c3 ♔c6 87 ♔c2 ♔b5 88 ♔d3 ♗b4 89 ♗d8 ♗×a5 90 ♗×f6** A force de refuser les propositions implicites de nulle d'ECUME, HIARCS se retrouve dans une position compromise. **♗e1 91 ♔e2 ♗g3 92 ♗e7 a5 93 ♔d3 ♔a4 94 ♔c2 ♗f2 95 ♔c3 ♗×e3 96 ♗×h4 ♗×f4 97 ♗f6 ♗d6 98 h4 ♗b4 + 99 ♔c2 ♗f8 100 h5 ♔b4**



Le règlement prévoyait que la partie soit arbitrée au 100e coup. L'arbitre, le Maître International Mike Valvo, démontra le gain pour les noirs.

Cumulus 2 – Tasc R40

BLITZ, MUNICH 1993

1 e4 e6 2 d4 d5 3 e×d5 e×d5 4 ♘f3 ♘f6 5 ♗d3 c5 6 O–O c4 !? Les programmes ont du mal à savoir quand il est bon de maintenir la tension au centre : les deux adversaires ont généralement tous les deux intérêt à la maintenir, et les positions qui résultent d'une rupture de cette tension sont le plus souvent difficiles à comparer avec la position racine. Mais un coup comme c5-c4 est toujours suspect si les noirs n'ont pas de raison tactique ou d'avantage important à mettre en valeur. Si Fe2 était forcé, créant un petit embouteillage dans la position blanche, ce coup ne serait pas totalement dénué de justification, mais... **7 ♖e1+ ♗e7 8 ♗f1 O–O 9 ♗g5 ♗g4 10 h3 ♗×f3?! **L'abandon de la paire de Fous nous paraît peu judicieux dans cette position. **11 ♕×f3 ♘c6 12 c3 ♕c7 ?!**Non, ce coup n'est pas la bévue d'un joueur humain! Tasc, qui jouait en mode « agressif », donne fréquemment des

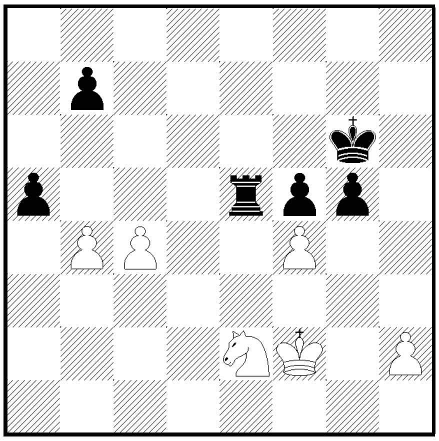
pions pour obtenir de l'activité. C'est souvent payant, surtout contre d'autres programmes, qui se précipitent invariablement sur le cadeau : quand cela fonctionne, on jurerait que c'est un joueur humain qui est en train de démontrer les lacunes des programmes d'échecs. Mais le cadeau n'est pas toujours empoisonné, et ici TASC donne un pion central très important alors que sa position était déjà inférieure. **13 ♗×f6** Ici, au moins, l'abandon de la paire de Fous est justifié par le gain du pion central d5.**13 ... ♗×f6 14 ♕×d5 ♖ae8 !?** Le placement des Tours de TASC laisse à désirer, de la même manière que CUMULUS avait joué la mauvaise Tour en c8 contre le programme ZUGZWANG**15 ♘d2 ♘e7 16 ♕×c4** Les Blancs empochent un deuxième pion, et les compensations noires sont toujours inexistantes **16 ... ♘c6 17 d5! +− ♕d8 18 ♘e4 ♘e5 19 ♕b3 ♗h4 20 ♕×b7?!** Un sacrifice de pion plus prometteur : la Dame blanche va avoir du mal à revenir de l'aile dame alors que les noirs commencent à avoir un semblant d'attaque à l'aile roi. Sur trois sacrifices de pion, un seul nous paraît justifié ; mais en partie rapide, où la tactique compte davantage, l'activité est très importante et peut valoir de tels investissements. Tasc dispose de cinq modes de jeu : son concepteur ne le fait pas jouer en mode agressif sans raison.**20 ... f5 21 ♘c5 ♕g5 22 ♘e6 ♘f3 + 23 ♔h1 ♖×e6 24 ♖×e6 ♗×f2**Et les noirs n'obtiendront rien de plus. Peut-être est-ce dû au hasard qui veut que le placement de leurs pièces n'ait pas permis de combinaisons décisives. Nous pensons que la position d'ECUME était suffisament supérieure pour que son vingtième coup soit
jouable. **25 d6 +− ♘d2 26 ♕d5 ♔h8 27 d7 ♗b6 28 ♖e8 ♕f6 29 ♕×d2 g5 30 ♖e1 ♖g8 31 ♗c4 ♖f8 32 ♖×f8+ ♕×f8 33 ♖e8 1:0**

Fritz 2 – Cumulus 2

BLITZ, MUNICH 1993

1 d4 ♘f6 2 ♘f3 d5 3 ♗f4 c5 4 d×c5 ♘a6 5 e3 ♘×c5 6 ♘c3 !? FRITZ ne se soucie pas de la poussée c2-c4, pourtant thématique dans les positions de ce type. C'est un programme dont la réputation est d'être bon tactiquement mais faible positionnellement. **6 ... a6 7 ♗e5!?** FRITZ viole maintenant sans raison le principe qui veut qu'on ne joue pas plusieurs coups avec une même pièce avant d'avoir terminé son développement.

7 ... e6 8 ♗d3 ♗d6 9 O–O O–O 10 ♘e2 ♗×e5 11 ♘×e5 ♕d6 En cas de retrait du Cavalier blanc, les noirs jouent e6-e5 avec une position très supérieure. FRITZ l'a compris, mais pour protéger son Cavalier il va se créer une faiblesse que son évaluation ne prend vraisemblablement pas en compte. **12 f4 ♘d7 13 ♘×d7 ♗×d7 14 ♕d2 ♘g4** CUMULUS a repéré la faiblesse e3 et cherche à l'exploiter.**15 c3 ♖fd8 16 ♖ad1 ♖ac8 17 ♘d4 e5!** ∓Cette poussée libératrice transforme le mauvais Fou d7 en bon Fou, gagne la bataille pour le contrôle du centre et ouvre le jeu dans une position supérieure : les bons programmes actuels ne laissent pas passer de telles occasions.**18 ♘f5 ♕f6 19 f×e5 ♕×e5 20 ♖f4 h5 21 ♗c2 g6** Si le Cavalier bouge, le pion e3 tombe. Et sur la prise immédiate en d5, les noirs prennent tout simplement le Cavalier g4.**22 ♖×g4 h×g4 23 ♕×d5 ♕×d5 24 ♘e7+ ♔g7 25 ♘×d5 ♗e6** Leur petite combinaison n'a rapporté aux blancs qu'un pion pour la qualité, et il leur reste la faiblesse en e3 : les noirs sont gagnants. **26 e4 ♖c5 27 ♘f4 ♖×d1 + 28 ♗×d1 ♗×a2 29 ♗×g4 ♖b5 30 b4 ♖e5 31 ♗f3 ♗b1 32 ♔f2 ♗×e4 33 c4 g5 34 ♘h5+ ♔g6 35 ♘g3 ♗×f3 36 g×f3 f5 37 ♘e2 a5 38 f4** Les noirs vont conclure de manière élégante, alors que FRITZ n'avait rien vu venir.



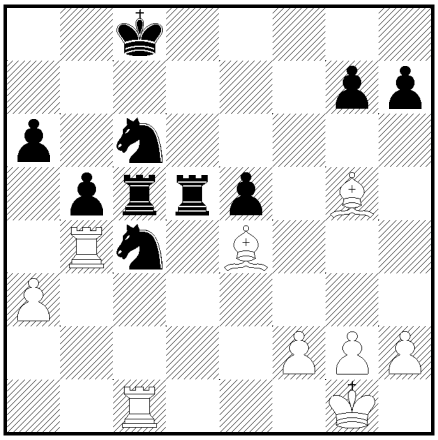
38 ... ♖×e2!! + Transpose dans une finale de pions gagnante. CUMULUS, grâce à ses connaissances dynamiques sur les finales de Rois et pions, sait qu'il atteint ainsi une finale de pions gagnante car le Roi adverse ne pourra pas arrêter tous les pions. Notons qu'un Oracle seul aurait des difficultés dans cette position, car des deux pions passés que les noirs vont obtenir aucun n'est hors du carré du Roi blanc, et de plus les blancs vont obtenir un pion h passé et peut-être même un pion c passé (ils n'en auront en réalité pas le temps). Ici la vitesse de FRITZ ne suffit pas, et de loin, à compenser son manque de connaissances échiquéennes. **39 ♔×e2 a4 40 ♔d3 a3 41 ♔c2 g×f4 42 ♔b3 f3 43 ♔×a3 f2 44 b5 f1=♕ 45 ♔b4 f4 46 ♔c5 f3 47 h4 ♕g1 + 48 ♔b4 f2 49 c5 ♕b1 + 50 ♔a5 f1=♕ 51 h5+ ♔×h5 52 ♔b6 ♕b×b5 + 53 ♔a7 ♕f4 54 ♔a8 ♕a6 #**

6.2.4 Tournois Hommes-Machines

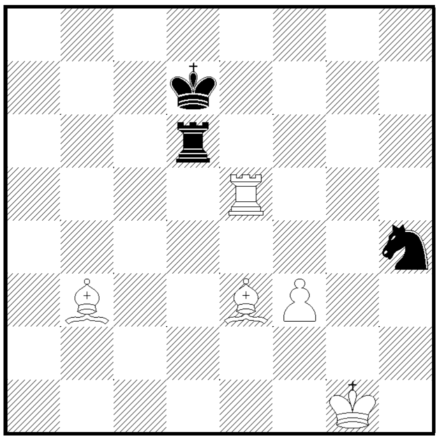
Ecume – Vaganian

AEGON, 1994

1 e4 e6 Vaganian est le plus grand spécialiste de la défense française au monde. **2 d4 d5 3 e×d5!** Le meilleur coup contre un fort humain qui comprend tous les schémas habituels de la défense française. **e×d5 4 ♘f3 ♗d6 5 c4** C'est la position que nous souhaitions obtenir contre Vaganian : assez ouverte pour que le programme puisse s'exprimer pleinement. **♗b4?! +** joué après seulement quelques secondes de réflexion ; Vaganian sort ECUME de sa bibliothèque. Les débutants savent qu'ils ne faut pas jouer la même pièce deux fois de suite sans raison, mais les Grand-Maîtres savent enfreindre les règles. **6 ♘c3 ♘e7 7 ♗d3 ♗e6 8 ♘g5** ECUME cherche à s'emparer de la paire de Fous dans cette position qui peut devenir très ouverte. La perte de temps est tolérable, à cause du coup Fb4 des noirs. **♕d7 9 O–O ♘c6 10 ♘×e6 f×e6 11 a3 ♗×c3 12 b×c3 O–O–O 13 ♕e2** Le position est en faveur des blancs. **d×c4 14 ♗×c4 ♘d5 15 ♕b2 e5 16 ♖b1 ♘b6 17 d5 ♘a5 18 ♗a2 ♕a4 19 ♕b4 ♕×b4 20 ♖×b4 ♖d7 21 c4 c5 22 ♖b5 ♘×c4 23 ♖×c5+ ♖c7 24 ♖b5 a6 25 ♖b4 ♖d8 26 ♖d1 ♖c5 27 ♗g5 ♖d7 28 ♖c1 b5 29 ♗b1 ♖×d5 30 ♗e4 ♘c6 ±**



Une position très difficile à jouer contre une machine. La paire de Fous des blancs peut faire des ravages, et les Tours et le Roi noirs sont très exposés, même si les noirs ont quelques compensations. **31 ♖b1 ♖d6 32 ♗×h7** Vaganian espère enfermer le Fh7. Ecume connaît le danger de ce genre de prises : avec un pion noir en f7, il n'aurait jamais capturé en h7 puisque g6 aurait irrémédiablement enfermé le Fh7. **g6 33 a4 ♔d7 34 a×b5 a×b5 35 h4 ♘d4 36 ♖e1 ♖c8 37 ♖a1 ♘e6 38 ♗e3 ♘c7 39 ♗c5 ♖c6 40 ♖ed1+ ♔e6 41 ♗a7** Une position des plus originales. Vaganian est un très fort joueur tactique et semblait à l'aise dans les complications. **♔f6 42 ♖d3 ♖a6 43 ♖×a6+ ♘×a6 44 ♖g3 ♔g7 45 ♗×g6** +– Mais il devient clair que Vaganian a fait fausse route, et la position est maintenant gagnante pour les blancs. **♔f8 46 ♗h7 ♖c7 47 ♖g8+ ♔e7 48 ♗d3 ♔f7 49 ♖h8 ♖c6 50 ♗e4 ♖d6 51 g3 b4 52 ♖h7+ ♔e6 53 ♗f3 b3 54 ♗e2 ♔d5 55 ♖b7 ♘d2 56 ♗e3 ♘c5 57 ♖b5** Les noirs sont complètement perdus, mais contre une machine, on ne sait jamais... **♘e4 58 f3 ♘×g3 59 ♖×c5+ ♔e6 60 ♗c4+ ♔d7 61 ♗×b3** Vaganian pourrait aussi bien abandonner mais... **♘f5 62 ♖×e5 ♘×h4**



63 ♖d5?! ECUME sait que 2F contre C gagne toujours, mais n'avait pas de base de données à l'AEGON. Il valait mieux gagner tranquillement avec le pion et la pièce en plus. **♘×f3 +** Vaganian ne laisse pas passer l'occasion. **64 ♔f1 ♖×d5 65 ♗×d5** La position est gagnante pour les blancs en 54 coups sur une défense parfaite ; nul doute qu'avec une base de données les choses auraient duré moins longtemps. **♘e5** Et nulle au 114e coup, après une bonne défense en blitz de Vaganian.

Larry Christiansen est un joueur doté d'un grand sens pratique : dès sa première partie dans le tournoi Aegon, il a compris que, contrairement à l'idée communément répandue parmi ses collègues, l'accélération de la cadence de jeu au quarantième coup est favorable au joueur humain pourvu qu'il ait géré son temps intelligemment. Dans les fins de partie, les programmes ont en effet besoin de calculer profondément pour comprendre ce qu'un Grand-Maître humain voit rapidement. Contre JOKER, Christiansen va s'efforcer d'atteindre le quarantième coup en conservant le plus de temps possible à sa pendule et en gardant assez de vie à la position. JOKER, lui, était peut être le programme le plus affaibli par le changement de cadence. La stratégie de Christiansen va porter ses fruits, mais non sans lui avoir demandé un gros effort : il dira par la suite que ce fut sa partie la plus difficile du tournoi.

Joker – Christiansen

AEGON, 1994

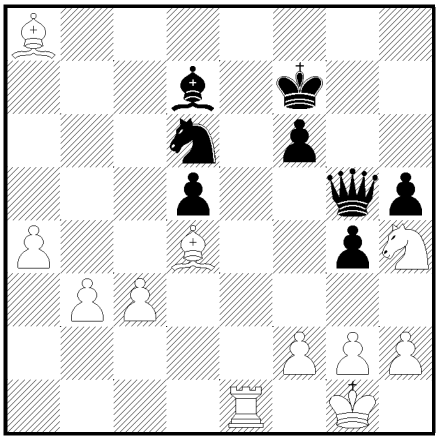
1 e4 g6 2 d4 ♗g7 3 c3 d6 4 f4 ♘f6 5 ♗d3 e5 Toujours théorique, mais les noirs ont déjà un jeu facile. C'est le genre de système que Christiansen affectionne. **6 ♘f3 ♗g4 7 f×e5 d×e5 8 ♗g5 h6 9 ♗h4 ♘d7 10 ♘d2 O–O 11 h3 ♗×f3 12 ♕×f3 ♕e8 13 O–O ♘h5 14 ♘b3 a5 15 ♗f2 a4 16 ♘d2 ♕e7 17 ♘c4 ♘f4 18 ♗c2 ♘e6 19 d5 ♘g5 20 ♕g4 ♘f6 21 ♕e2 ♘h5 22 h4 ♘f4 23 ♕e3 ♘h7 24 ♗g3 ♘h5 25 ♗e1 ♘5f6 26 ♕e2 h5 27 ♖d1 ♖a6 28 ♗g3 ♘g4 29 ♔h1 ♗h6 30 ♖d3 ♘f6 31 ♖f3 ♘e8 32 ♘a3 ♖a8 33 ♗d3 ♕c5 34 ♘c4 ♕e7 35 ♘a3 ♘d6 36 c4 c5 37 d×c6 b×c6 38 c5 ♘e8 39 ♗c4 ♘f6 40 ♗d3 ♖a5 41 ♗e1 ♖a7** Le

40^{e} coup est passé et JOKER va jouer beaucoup plus faiblement à cause de l'accélération de la cadence : le rythme de jeu était de 40 coups en 1h45 puis 15 minutes pour finir la partie. **42 ♗g3 ♗g7 43 ♕c2 ♖d8 44 ♗e1 ♖d7 45 ♗c4 ♔h7 46 ♕×a4 ♕×c5 47 ♕a5 ♕d4 48 ♗c3 ♕×e4 49 ♗×f7 ♖d1 50 ♗e1 ♘e3 51 ♖g1 ♘×g2 52 ♔×g2 ♖8d3 53 ♖f1 ♖×f3 54 ♖×f3 ♖×e1 55 ♗e6 ♘g4 56 ♗g8+ ♔×g8 57 ♕a8+ ♔h7 58 ♕a7 ♘e3 +** et 0-1

Joker - Lupu

AUBERVILLIERS, 1994

1 e4 e6 2 d4 d5 3 ♘d2 c5 4 e×d5 e×d5 5 ♗b5+ ♗d7 6 ♕e2+ ♘e7 7 d×c5 ♕a5 8 ♗d3 ♘c6 9 c3 ♕×c5 10 ♘b3 ♕d6 11 ♘f3 g6 12 ♗e3 ♗g7 13 O–O O–O 14 ♗c5 ♕f6 15 ♘a5 ♖fe8 16 ♘×b7 ♘f5 17 ♕d2 ♗h6 18 ♕d1 ♗f4 19 ♕b3 ♖ab8 ∓Le pion b7 s'annonce difficile à digérer... **20 ♖fe1 ♘a5 21 ♖×e8+ ♗×e8 22 ♘×a5 ♖×b3 23 ♘×b3 a6 24 ♘bd4 ♘d6 25 ♖e1 ♗d7 26 b3 h5 27 ♘e2 ♘e4 28 ♗d4 ♕d6 29 ♘×f4 ♕×f4 30 ♗e5 ♕f5 31 ♗×a6 g5 32 ♗d4 f6 33 ♗b7 ♘d6 34 ♗a8 ♔f7 35 a4 g4 36 ♘h4 ♕g5**



37 f4 Cette fois-ci il valait mieux faire confiance à la machine! **♕×h4 38 ♗×d5+ ♔g6 39 g3 ♕h3 40 ♗g2 ♔f5 41 ♗×h3 g×h3** et 1-0 au 73e coup

Velikov – Joker

AUBERVILLIERS, 1994

1 ♘f3 d5 2 d4 ♘f6 3 c4 e6 4 ♘c3 d×c4 5 e3 c5 6 ♗×c4 ♘c6 7 O–O c×d4 8 e×d4 ♗d6 9 a3 ♘a5 10 ♗a2 ♗d7 11 ♗g5 h6 12 ♗h4 O–O 13 ♕d3 ♕c7 ? Un mouvement curieux, qui va permettre Fxf6 pendant plusieurs coups. Le joueur humain fit-il alors confiance à la machine? toujours est-il qu'après cela, JOKER ne se sentait plus très bien... **14 ♖ac1 ♖fc8 15 ♗b1 ♔f8 16 ♖fe1 ♗f4 17 ♖cd1 ♘c4 18 ♕e2 ♘×b2 19 ♕×b2 ♕×c3 20 ♕e2 g5** Maintenant, les blancs ne pourront plus jouer Fxf6! Après ces échanges surréalistes le Roi noir est toujours mal à l'aise. **21 ♗g3 ♗×g3 22 h×g3 ♕×a3 23 ♕e5 ♘g4 24 ♕h8+ ♔e7 25 ♕h7 ♕a5 26 d5 ♕c5??** Une horrible gaffe : suite à un petit problème de réglage, le programme a joué ce coup instantanément, sans aucune réflexion préalable. **27 d6+ ♔f6 28 ♖d4??** une non moins horrible gaffe! Ch2 gagnait tout de suite. Sans doute Velikov a-t-il fait confiance au programme. En tout cas, il prenait le temps de réfléchir. **♖h8 29 ♕d3 e5 30 ♖d5 ♕×f2 + 31 ♔h1 ♕×e1 + 32 ♘×e1 ♘f2 + 33 ♔h2 ♘×d3 34 ♘×d3 ♗f5 35 d7 ♖ad8**

6.3 Évolution de la force de nos programmes

Le calcul de la force réelle d'un programme n'est pas une chose aisée et ne peut se calculer que sur un nombre important de parties (plus de 40) contre des joueurs humains classés. En deça de ce nombre les évaluations de force de programme ne sont pas très précises. Les différentes versions de nos programmes n'ayant pas joué assez de parties, nous ne pouvons donner que des approximations.

Nous avons donc choisi de donner les résultats de nos programmes d'échecs lors des tournois de parties semi-rapides d'Aubervilliers des années 1990 à 1994.

TAB. 6.1 - *Performances de nos programmes d'échecs aux tournois de parties semi-rapides d'Aubervilliers*

Année	Programme n° 1			Programme n° 2		
	Nom	Perf. Elo	Ordinateur	Nom	Perf. Elo	Ordinateur
1990	Echec 1.6	1871	386 33Mhz	Echec Y	1797	386 25Mhz
1991	Echec 1.9	2172	486 33Mhz	Cumulus 1.2	1907	386 33Mhz
1992	Echec 1.95	2073	Sparc ELC	Cumulus 1.5	2150	Sparc SLC
1993	Cumulus 2.0	2272	486 50Mhz	Cumulus Y	2292	486 50Mhz
1994	Joker	2380	486 66Mhz	Ecume	1930	486 66Mhz

ECHEC Y était un programme basé sur ECHEC 1.6 avec une recherche de la tranquilité dérivée de celle définie par Schrüfer pour le programme BOBBY[156].
CUMULUS Y était le programme CUMULUS 2.0 sans les extensions de recherche.

Le tableau 6.1 résume les résultats que nos programmes ont obtenus à Aubervilliers. Nous y notons certaines incohérences, comme la performance ELO des programmes ECHEC 1.95 en 1992 et ECUME en 1994, mais dans l'ensemble ces résultats reflètent une situation que nous pensons proche de la réalité. A ordinateurs de performance égale, on note une augmentation sensible du niveau de jeu.

Globalement, nous pouvons y voir que la force de nos programmes s'est élevée d'un niveau de bon joueur de club (1871) à un niveau de fort joueur international (> 2300).

Chapitre 7

Conclusions et Perspectives

Synthèse

Ce document est relatif aux techniques mises en œuvre dans la programmation des échecs de championnats sur ordinateurs séquentiels et parallèles. Notre travail se décompose en trois parties.

– Une première partie est consacrée à l'étude des algorithmes séquentiels. Cette partie est décomposée en une analyse des processus de type minimax suivie de la présentation des méthodes et heuristiques utilisées pour les accélérer.

  Nous y présentons l'algorithme NegaC* basé sur l'utilisation de recherche $\alpha\beta$ avec des fenêtres minimales. Comparé aux autres algorithmes minimax, il s'avère efficace et ceci aussi bien sur des arbres à évaluation aléatoire où pour un encombrement mémoire identique[1] il est plus économe que l'algorithme SSS* en nombre de nœuds, que sur des arbres fortement ordonnés. Jusqu'ici en effet les fenêtres minimales étaient surtout utilisées via le *NegaScout* ou ses dérivés pour des arbres fortement ordonnés.

  Certains algorithmes minimax ont des usages bien particuliers. Ainsi l'algorithme de recherche du nombre de preuve ne peut pas être utilisé pour le calcul de la valeur minimax aux échecs mais il démontre ses capacités dans les recherches sur des arbres non uniformes dans la démonstration de mats forcés en temps limité face à notre programme ECUME et un programme utilisant un *Negascout* sans extensions.

  Ensuite nous montrons l'apport des heuristiques de tri des mouvements et de coupures a priori pour l'efficacité des programmes séquentiels. Nous exposons aussi les méthodes d'extension utilisées pour nos différents programmes d'échecs : extensions statiques non entières pour ECUME, extensions entropiques pour JOKER et extensions de la tranquillité pour CUMULUS.

  Durant cette première partie, nous montrons comment accélérer les programmes d'échecs. Partant d'un programme simple P_0, nous montrons dans le chapitre 3 que le fait de bien

1. L'encombrement mémoire peut être nettement plus réduit par l'utilisation d'une table de transposition qui ne gardera que le sommet de l'arbre de recherche. Nous utilisons avec succès ce schéma dans notre programme d'Othello pour les recherches de fins de parties.

ordonner les coups permet d'aller jusqu'à 5 fois plus vite. Le choix de l'algorithme *Negascout* permettant d'aller 1,5 fois plus vite, nous pouvons sans sélectivité obtenir un programme 7.5 fois plus rapide que le programme P_0. De plus, l'utilisation de l'heuristique du mouvement nul, heuristique raisonnable dans le sens qu'elle ne perturbe pas trop la qualité des coups, permet une accélération de l'ordre de 1,89. Donc, nous obtenons un programme 14 fois plus rapide que le programme P_0. Cette accélération est sans aucun doute un des facteurs qui permettent de mieux expliquer la croissance de la force des programmes d'échecs, ces vingt dernières années. De plus, les fonctions d'évaluation se sont raffinées et comprennent de plus en plus de critères spécifiques pour analyser le jeu positionnel. Et pour certaines fins de parties, les programmes ont continuellement augmenté leur niveau de jeu du fait de l'intégration de connaissances supplémentaires dans leurs fonctions d'évaluation. Un autre facteur est bien sûr l'augmentation de la puissance des ordinateurs : ainsi en 1977, le meilleur programme, CHESS 4.7, développait 503 positions par seconde sur un CDC 6400[189] et, sans parler de DEEP BLUE [2], un programme actuel sur micro-ordinateur développe sur un processeur PENTIUM à 90 Mhz aux alentours de 50000 positions par seconde [3].

- La seconde partie de ce document est consacrée à l'étude des schémas de parallélisation du principe minimax. Après avoir passé en revue les algorithmes proposés dans la littérature et notamment les algorithmes YBWC et $\alpha\beta^*$ qui ont obtenu de très bons résultats sur des machines à base de transputers, nous analysons ces algorithmes sur la parallélisation de deux programmes de jeu, un programme d'Othello et un programme d'échecs. Les résultats d'accélération relative que nous obtenons sur une Connection Machine 5 sont bien inférieurs à ceux précédemment publiés, ceci sans doute en raison de la différence de granularité des machines mais aussi de la différence des problèmes traités. Nos programmes séquentiels sont déjà des programmes très compétitifs.

  Nous avons alors défini un algorithme hybride prenant le meilleur de ces deux méthodes ce que nous appelons l'algorithme $\alpha\beta^*$ couplé avec le contrôle spéculatif du droit d'aînesse. Ce nouvel algorithme s'avère le plus efficace sur le programme d'échecs mais n'égale pas les performances du YBWC pour le programme d'Othello. La différence de performances de cet algorithme pour ces deux applications s'expliquant par les différences des types d'arbres traités : étroits, profonds et ordonnés pour le programme d'Othello et larges, moins profonds et fortement ordonnés pour le programme d'échecs.

  Nous montrons donc que le choix entre les deux modèles de parallélisation l'algorithme $\alpha\beta^*$ et YBWC se résume par un choix entre surcoût de recherche et surcoût de synchronisation. Le bon choix dépend alors du problème traité et de la machine cible.

  Notre expérience montre que s'il est possible d'implanter efficacement la table de transposition partagée, l'algorithme $\alpha\beta^*$ est toujours très simple et rapide à implanter et donnera des résultats satisfaisant pour tous les types de problèmes. Pour certains problèmes, comme nous l'avons vu, l'algorithme YBWC sera plus efficace mais il nécessite un temps d'implantation et de mise au point bien plus important.

- La dernière partie est consacrée aux problèmes d'évaluation et plus particulièrement à l'approche de type Oracle. Nous y montrons qu'un oracle couplé avec des tables

2. DEEP BLUE est un programme basé sur la mise en parallèle de circuits VLSI spécialisés qui développent chacun plusieurs centaines de millier de positions par seconde.

3. Le programme FRITZ développant 100 000 positions par seconde.

positionnelles permet d'augmenter la vitesse de l'évaluation sans trop dégrader la qualité des mouvements choisis. Et d'une manière encore plus importante, l'Oracle, après une analyse stratégique de la position, va permettre de choisir la fonction d'évaluation (c'est-à-dire pas uniquement les tables positionnelles mais aussi d'autres paramètres) qui sera appliquée pour la détermination du mouvement. Cette fonction d'évaluation va être dirigée vers la réalisation d'un plan. Ainsi, l'Oracle est en partie constitué d'une bibliothèque de plans, chaque plan étant associé à un type de position stratégique. Le langage CHEVAL permet de donner de façon déclarative de telles connaissances de telle sorte qu'elles soient aisément codées et modifiées pour refléter une partie du savoir d'un expert aux échecs.

Pour clore la partie consacrée à l'évaluation, nous montrons les différentes techniques mises en œuvre pour la synthèse automatique de fonctions d'évaluation.

Motivations et évolutions de nos programmes

Quand nous avons commencé la compétition dans les programmes d'échecs, les meilleurs programmes se répartissaient en deux catégories :

1º Les programmes d'échecs universitaires et industriels, qui tournaient sur les meilleurs ordinateurs ou sur des puces spécialisées. Ces programmes tiraient surtout leur force de la puissance de calcul de leurs supports.

2º Les programmes d'échecs commerciaux écrits par des programmeurs de talent sur des micro-ordinateurs, capables avec un peu de chance de battre un programme universitaire. Les algorithmes utilisés dans ces programmes étaient censés être plus intelligents et la fonction d'évaluation plus fine que celle des programmes universitaires.

Certains pensaient même que les programmes de la seconde catégorie seraient imbattables s'ils tournaient sur les mêmes ordinateurs que les programmes universitaires. Et, qu'au contraire, les programmes universitaires perdraient beaucoup de leur valeur s'ils tournaient sur des micro-ordinateurs.

Notre motivation première fut de vérifier ces affirmations. Pour ce faire, nous avions décidé de construire un programme en utilisant les techniques connues car sources de publications, utilisées par les programmes universitaires, et de comparer ces programmes avec les programmes commerciaux sur des matériels équivalents.

La première étape de nos recherches aboutit lors du championnat du monde des micro-ordinateurs qui se tenait à Lyon en 1990, où nous avons montré que, finalement, un programme universitaire sur micro-ordinateur pouvait faire jeu égal avec les meilleurs programmes commerciaux. Ce fut, en effet, la première fois qu'un programme universitaire montait dans ces championnats sur la deuxième marche du podium. Nous avions montré que les techniques utilisées dans les programmes universitaires étaient suffisamment efficaces pour ce type de championnat. Nous avions, du même coup, mis en valeur l'approche de type Oracle combinée à une table positionnelle.

La deuxième étape était l'introduction de l'intelligence dans nos programmes. Pour ECHEC, nous avons remodelé la fonction d'évaluation afin qu'elle incorpore plus de plans, plus de connaissances théoriques, et pour CUMULUS, nous avons raffiné ses connaissances dans la recherche de la tranquillité.

Il nous est alors apparu que les recherches effectuées sur CUMULUS ne permettaient d'avoir un plus fin tacticien qu'ECHEC que sur un ensemble restreint de positions. Donc, nous avons défini les caractéristiques du moteur d'ECUME que nous avons réalisé en assembleur pour qu'il soit compétitif avec les meilleurs programmes commerciaux. Une fois le premier prototype de ce moteur réalisé, nous avons pu y adjoindre la dernière version de notre Oracle plus positionnel et un ensemble plus important de critères d'évaluations du second ordre dont l'importance était réglée par l'Oracle. ECUME démontra à Madrid sa capacité à être dans les meilleurs programmes, toutes catégories confondues, ratant même un gain face au programme ZUGZWANG qu'il avait dominé dans la compréhension positionnelle de la partie.

La troisième étape, que nous poursuivons actuellement, est la construction d'un programme parallèle qui utilise les techniques que nous avons développées dans nos programmes séquentiels.

Perspectives

Actuellement, notre recherche s'oriente vers plusieurs axes. Dans un premier temps, nous continuons le développement du schéma $\alpha\beta$* avec droit d'aînesse dans le but de réduire le surcoût de recherche. Ceci peut être obtenu par un meilleur partage des informations des processus analysant une même position.

Nous pensons porter nos programmes parallèles vers plusieurs machines cibles comme des réseaux de stations de travail, un CRAY/SUN 6400 ou un CRAY T3D. Ce portage est possible du fait de la redéfinition de nos programmes autour du noyau de passage de messages PVM. Ensuite, nous optimiserons les messages pour obtenir une table de transposition distribuée suffisamment rapide pour pouvoir supporter le schéma $\alpha\beta$*.

Une autre méthode pour utiliser le parallélisme est la coopération entre programmes : différents programmes tournent en parallèle sur la même position et un superviseur est chargé de choisir en fonction des analyses des différents programmes le coup à jouer. Les programmes utilisés peuvent avoir des caractéristiques variées : des programmes forts tactiquement qui analysent l'arbre de jeu très rapidement, du fait de fonctions d'évaluation très simplifiées, et des programmes forts positionnellement. Bien évidemment, le problème du choix final laissé au superviseur est un problème d'analyse multi-critère difficile.

Nos développements continuent aussi autour de la redéfinition de la fonction d'évaluation dans le langage CHEVAL.

Une autre voie de prolongement de nos travaux est constituée par l'étude et l'implantation des algorithmes B* et BPIP. En effet, le nombre d'implantation de ces algorithme est très faible et les résultats qu'ils ont obtenus ne reflètent donc pas totalement leurs potentiels. Une autre question qui peut être soulevée par ces algorithmes, est aussi leur parallélisation.

Bibliographie

[1] S.G. Akl, D.T. Barnard, et R.J Doran. « Design, Analysis and Implementation of a Parallel Tree Search Algorithm ». *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 4, n° 2, 1982, p. 192–203.

[2] S.G. Akl et M.M. Newborn. « The Principle Continuation and the Killer Heuristic ». *In : ACM Annual Conference*, p. 466–473. 1977.

[3] J.D. Allen. « A Note on the Solution of Connect-Four ». *In : Heuristic Programming in Artificial Intelligence*, éd. par D.N.L. Levy et D.F. Beal. p. 134–135. Ellis Horwood, 1989.

[4] L.V. Allis. « *A Knowledge-Based Approach of Connect-Four. The Game is Solved: White wins* ». Amsterdam, Master's thesis, Faculty of Mathematics and Computer Science, Vrije Universiteit, 1988.

[5] L.V. Allis. « *Searching for Solutions in Games and Artificial Intelligence*». Maastricht, The Netherlands, Thèse de PhD, University of Limburg, 1994.

[6] L.V. Allis et D. Breuker. « How to Mate : Applying Proof-Number Search to Chess ». *In : Advances in Computer Chess 7*, éd. par H.J. van den Herik, I.S. Herschberg, et J.W.H.M. Uiterwijk. University of Limburg, Maastricht, 1994.

[7] L.V. Allis, M. van der Meulen, et H.J. van der Herik. « Proof Number Search ». *Artificial Intelligence*, vol. 66, n° 1, 1994, p. 91–124.

[8] I. Althöfer. « A Parallel Game Tree Search Algorithm with a Linear Speedup ». *Journal of Algorithms*, 1993.

[9] T.S. Anantharaman. « Confidently Selecting a Search Heuristic. ». *ICCA Journal*, vol. 14, n° 1, 1991, p. 3–16.

[10] T.S. Anantharaman. « Extension Heuristics. ». *ICCA Journal*, vol. 14, n° 2, 1991, p. 47–65.

[11] T.S. Anantharaman, M.S. Campbell, et F. Hsu. « Singular Extensions: Adding Selectivity to Brute-Force Searching. ». *ICCA Journal*, vol. 11, n° 4, 1988, p. 135–143.

[12] G. Baudet. « On the Branching Factor of the Alpha-Beta Pruning algorithm ». *Artificial Intelligence*, vol. 10, n° 2, 1978, p. 173–199.

[13] M.-F. Baudot, J.-L. Seret, et J.-C. Weill. « Echecs : où en sont vraiment les programmes ? ». *Bulletin de l'AFIA*, octobre 1994, p. 53–61.

[14] E.B. Baum et W.D. Smith. « *Best Play for Imperfect Players and Game Tree Search* ». Rapport technique, Princeton, NJ, Nec Research Institute, 1993.

[15] D.F. Beal. « An Analysis of Minimax ». *In : Advances in Computer Chess 2*, éd. par M.R.B. Clarke. p. 103–109. Edinburgh University Press, 1980.

[16] D.F. Beal. « Benefits of Minimax Search ». *In : Advances in Computer Chess 3*, éd. par M.R.B. Clarke. p. 17–24. Pergamon Press, 1982.

[17] D.F. Beal. « Experiments with the Null Move Search ». *In : Advances in Computer Chess 5*, éd. par D.F. Beal. p. 65–79. Elsevier Science Publishers, Amsterdam, 1989.

[18] H.J. Berliner. « *Chess as Problem Solving: The Development of a Tactics Analyzer* ». Technical Report n° AD-784 881, Carnegie Mellon University, mai 1974.

[19] H.J. Berliner. « The B* Tree Search Algorithm ». *Artificial Intelligence*, vol. 12, n° 1, 1979, p. 23–40.

[20] H.J. Berliner. « Backgammon Computer Program Beats World Champion ». *Artificial Intelligence*, vol. 14, 1979, p. 205–220.

[21] H.J. Berliner. « On the Construction of Evaluation Functions for Large Domains ». *In : 6Th International Joint Conference on Artificial Intelligence*, p. 53–55. 1979.

[22] H.J. Berliner. « Playing Computer Chess in the Human Style ». *ICCA Journal*, vol. 16, n° 3, septembre 1993, p. 176–182.

[23] H.J. Berliner et M.S. Campbell. « Using Chunking to Solve Chess Pawn Endgames ». *Artificial Intelligence*, vol. 23, 1984, p. 97–120.

[24] H.J. Berliner et C. Ebeling. « Pattern Knowledge and Search: The SUPREM Architecture ». *Artificial Intelligence*, vol. 38, n° 2, mars 1989, p. 161–198.

[25] H.J. Berliner et C. McConnell. « *B* Probability Based Search* ». Rapport technique n° CMU-CS-94-168, School of Computer Science, Carnegie Mellon University, mai 1994.

[26] A. Bernstein, N. Dev. Roberts, T. Arbuckle, et M.A. Belsky. « A Chess Playing Program for the IBM 704 ». *In : Proceedings Western Joint Computer Conference*, p. 157–159. Los Angeles, 1958.

[27] P. Bettadapur. « Influence of Ordering on Capture Search. ». *ICCA Journal*, vol. 9, n° 4, 1986, p. 180–188.

[28] S. Bhattacharya et A. Bagchi. « *Unified Recursive Scheme for Search in Game Trees* ». Working Paper Series n° 144, Calcutta, Indian Inst. of Management, 1990.

[29] J.A. Birmingham et P. Kent. « Tree-searching and Tree-Pruning Techniques ». *In : Advances in Computer Chess*, éd. par M.R.B. Clarke, p. 89–107. 1977.

[30] M.M. Botvinnik. *Computers, Chess and Long-Range Planning*. Springer Verlag, 1970.

[31] M.M. Botvinnik. *Computers in Chess: Solving Inexact Search Problems*. Springer Verlag, 1984.

[32] M.M. Botvinnik. « Three Positions. ». *ICCA Journal*, vol. 16, nº 2, 1993, p. 71–75.

[33] M.M. Botvinnik, D. Cherevik, V. Vladimirov, et V. Vygodsky. « Solving Shannon's Problem: Ways and Means ». *In : Advances in Computer Chess 7*, éd. par H.J. van Den Herik, I.S. Herschberg, et J.W.H.M. Uiterwijk. p. 9–18. University of Limburg, Maastricht, 1994.

[34] I. Bratko et M. Gams. « Error Analysis of the Minimax Principle ». *In : Advances in Computer Chess 3*, éd. par M.R.B. Clarke. p. 1–16. Pergamon Press, Oxford, 1982.

[35] I. Bratko et D. Kopec. « A Test for Comparison of Human and Computer Performance ». *In : Advances in Computer Chess III*, éd. par M.R.B. Clarke. p. 31–56. Pergamon Press, 1982.

[36] M. Brockington. « *CMPUT 535 Project – An Implementation of the Young Brothers Wait Concept* ». Rapport technique, University of Alberta, 1994.

[37] M. Buro. «Communications personnelles», 1993–1994.

[38] M. Buro. « *Techniken für die Bewertung von Spielsituationen anhand von Beispielen* ». Thèse de PhD, Fachbereich Mathematik / Informatik – Universität GH Paderborn, 1994.

[39] M.S. Campbell. « *Chunking as an Abstraction Mecanism* ». Pittsburgh, Thèse de PhD, School of Computer Science, Carnegie-Mellon University, 1988.

[40] M.S. Campbell et T.A. Marsland. « A comparison of Minimax Tree Search Algorithms ». *Artificial Intelligence*, vol. 20, nº 4, 1983, p. 347–367.

[41] J.H. Condon et K. Thompson. « Belle Chess Hardware ». *In : Advances in Computer Chess 3*, éd. par M.R.B. Clarke. p. 45–54. Pergamon Press, 1982.

[42] K. Coplan. « A Special-Purpose Machine For An Improved Search Algorithm For Deep Chess Combinations ». *In : Advances In Computer Chess 3*, éd. par D. F. Beal. p. 25–43. Pergamon Press, 1982.

[43] V.-D. Cung. « *Contribution à l'Algorithmique Non Numérique Parallèle : Exploration d'Espaces de Recherche* ». Thèse de PhD, Université Paris VI, avril 1994.

[44] V.-D. Cung et C. Roucairol. « *Parcours parallèle d'arbres minimax* ». Rapport technique nº 1549, INRIA, novembre 1991.

[45] V. David. « *Algorithmique parallèle sur les arbres de décision et raisonnement en temps contraint. Étude et application au Minimax* ». Thèse de PhD, ENSAE, 1993.

[46] V. David. «Communications personnelles», 1993–1994.

[47] A.D. De Groot. « Chess Playing Programs ». *In : Proc. Kononkl Nederlands Akod. Wentensch.* 1964.

[48] C. Donninger. « Null Move and Deep Search: Selective- Search Heuristics for Obtuse Chess Programs. ». *ICCA Journal*, vol. 16, nº 3, 1993, p. 137–143.

[49] C. Ebeling. « *All the Right Moves* ». Thèse de PhD, Department of Computer Science, Carnegie Mellon University, 1986.

[50] M. Euwe. « Computers and Chess ». *In : The Encyclopaedia of Chess*, éd. par A. Sunnucks. St Martin's Press, New York, 1970.

[51] R. Feldmann. « *Spielbaumsuche mit massiv parallelen Systemen* ». Thèse de PhD, Fachbereich Mathematik / Informatik – Universität GH Paderborn, 1993.

[52] R. Feldmann, B. Monien, et P. Mysliwietz. « A Fully Distributed Chess Program ». *In : Advances in Computer Chess 6*, éd. par D.F. Beal. p. 1–27. Ellis Horwood, Chichester, England, 1991.

[53] R. Feldmann, B. Monien, P. Mysliwietz, et O. Vornberger. « Distributed Game-Tree Search. ». *ICCA Journal*, vol. 12, n° 2, 1989, p. 65–73.

[54] R. Feldmann, B. Monien, P. Mysliwietz, et O. Vornberger. « Distributed Game Tree Search ». *In : Parallel Algorithms for Machine Intelligence and Vision*, éd. par V. Kumar, Kanal L.N., et P.S. Fopalkrishman. p. 66–101. Springer Verlag, 1990.

[55] R. Feldmann, P. Mysliwietz, et O. Vornberger. « *A Local Area Network used as as Parallel Architecture* ». Rapport technique, University of Paderborn, 1986.

[56] C. Ferguson et R.E. Korf. « Distributed Tree Search and its Application to Alpha-Beta pruning ». *In : Proceedings AAAI-88, Seventh National Conference on Artificial Intelligence*, p. 128–132. 1988.

[57] J.P. Fishburn. « *Analysis of Speedup in Distributed Algorithms* ». Thèse de PhD, Computer Sciences Department University of Wisconsin, 1981. Voir aussi le livre de même titre édité en 1984 par UMI research press.

[58] J.P. Fishburn. « Another Optimization of Alpha-Beta Search. ». *SIGART Newsletters*, vol. 84, 1983, p. 37–38.

[59] P.W. Frey. « Machine Othello ». *Personal Computing*, vol. 4, n° 7, 1980, p. 89–90.

[60] P.W. Frey. « An Empirical Technique for Developing Evaluation Functions. Some thoughts with a review of Mitchell's thesis. ». *ICCA Journal*, vol. 8, n° 1, 1985, p. 17–22.

[61] A. Geist, A. Berguelin, J. Dongarra, W. Jiang, R. Manchek, et V. Sunderam. « *PVM 3.0 User's Guide and Reference Manual* ». Oak Ridge National Laboratory, 1993.

[62] M. George et J.H. Schaeffer. « Chunking for Experience. ». *ICCA Journal*, vol. 13, n° 3, 1990, p. 123–132.

[63] M. Gherrity. « *A Game-Learning Machine* ». San Diego, Thèse de PhD, University of California, 1993.

[64] J.J. Gillogly. « The Technology Chess Program ». *Artificial Intelligence*, vol. 3, 1972, p. 145–163.

[65] J.J. Gillogly. « *Performance Analysis of the Technology Chess program* ». Rapport technique n° CMU-CS-78-189, Computer Science Department, Carnegie-Mellon University, 1978.

[66] F. Gobet et P. Jansen. « Towards a Chess Program Based on a Model of Human Memory ». *In : Advances in Computer Chess 7*, éd. par H.J. van den Herik, I.S. Herschberg, et J.W.H.M. Uiterwijk. p. 35–60. University of Limburg, Maastricht, 1994.

[67] G. Goetsch et M.S. Campbell. « Experiments with the Null-Move Heuristic ». *In : Computers, Chess, and Cognition*, éd. par T.A. Marsland et J.H. Schaeffer. p. 159–168. Springer-Verlag, 1990.

[68] M. Gondran. « *Les règles négatives en apprentissage* ». Rapport technique n° HI-00/93–004, EDF - DER - TIEM - IMA, 1993.

[69] M. Gondran et M. Minoux. *Graphes et Algorithmes.* Eyrolles, 1979.

[70] R.D. Greenblatt, D.E. Eastlake, et S. D. Crocker. « The Greenblatt Chess Program ». *In : Proceedings of the Fall Joint Computer Conference*, p. 801–810. 1967.

[71] D. Hartmann. « Butterfly Boards. ». *ICCA Journal*, vol. 11, n° 4, 1988, p. 163.

[72] D. Hartmann et P. Kouwenhoven. « The 9th Dutch Computer-Chess Championship ». *ICCA journal*, vol. 12, n° 4, 1989, p. 249–251.

[73] D. Hartmann et P. Kouwenhoven. « Sundry Computer Chess Topics ». *In : Advances in Computer Chess 6*, éd. par D.F. Beal. p. 61–72. Ellis Horwood, 1990.

[74] E. Hearst. « Man and Machine : Chess Achievements and Chess Thinking ». *In : Chess Skill in Man and Machine*, éd. par P.W. Frey. p. 167–200. Springer Verlag, 1977.

[75] H.J. van den Herik. « *Computerschaak, Schaakwerelden Kunstmatige Intelligentie* ». Thèse de PhD, Delft University of Technology, 1983.

[76] H.J. van den Herik. « Strategy in Chess Endgames ». *In : Computer Game-Playing*, éd. par M.A. Bramer. p. 87–105. Ellis Horwood Limited, 1983.

[77] F.-H. Hsu. « *Large Scale Parallelization of Alpha-Beta Search ; An alorithmic and Architectural Study with Computer Chess* ». Thèse de PhD, Department of Computer Science, Carnegie Mellon University, 1990.

[78] F.-H. Hsu, T. Anantharaman, M. Campbell, et A. Nowatzyk. « Deep Thought ». *In : Computer, Chess and Cognition*, éd. par T.A. Marsland et J.H. Schaeffer. p. 55–78. Springer-Verlag, 1990.

[79] B.J. Huberman. « *A Program to Play Chess End Games* ». Thèse de PhD, Computer Science Department, Stanford University, 1968.

[80] M. Hyatt, B.W. Suter, et H.L. Nelson. « A parallel alpha/beta searching algorithm ». *Parallel Computing*, vol. 10, n° 3, 1989, p. 299–308.

[81] H. Kaindl. « Quiescence search in computer chess ». *SIGART Newsletter*, n° 80, 1982, p. 124–131.

[82] H. Kaindl. « Tree Searching Algorithms ». *In : Computers, Chess, and Cognition*, éd. par T.A. Marsland et J.H. Schaeffer. p. 133–158. Springer-Verlag, 1990.

[83] R.M. Karp et Y. Zhang. « On Parallel Evaluation of Game Trees ». *In : Proceedings of SPAA '89*, p. 409–420. 1989.

[84] Y.-F. Ke et T.-M. Parng. « The Guard Heuristic: Legal Move Ordering with Forward Game-Tree Pruning. ». *ICCA Journal*, vol. 16, n° 2, 1993, p. 76–85.

[85] A. Kierulf. « Brand — an Othello program ». *In : Computer Game-Playing*, éd. par M.A. Bramer. p. 197–208. Ellis Horwood, 1983.

[86] H. Kmoch. *L'art de Jouer les Pions*. Payot/Diffec, 1980.

[87] D.E. Knuth. *Sorting and Searching*. The Art of Computer Programming, Addison Wesley, 1973.

[88] D.E. Knuth et R.W. Moore. « An Analysis of Alpha-Beta Pruning ». *Artificial Intelligence*, vol. 6, n° 4, 1975, p. 293–326.

[89] R.E. Korf. « Depth-first iteraive-deepening : An optimal admissible tree search. ». *Artificial Intelligence*, vol. 27, n° 1, 1985, p. 97–109.

[90] A. Kotok. « *A Chess Playing Program for the IBM 7090* ». Master's thesis, MIT, 1962.

[91] H.-J. Kraas. « *Zur Parallelisierung des SSS*-Algorithmus* ». Thèse de PhD, Universität Braunschweig, 1990.

[92] B.C. Kuszmaul. « *Synchronized MIMD Computing* ». Thèse de PhD, Department of Computer Science, Massachusetts Institute of Technology, mai 1994.

[93] K.-F. Lee et S. Mahajan. « A Pattern Classification Approach to Evaluation Function Learning ». *Artificial Intelligence*, vol. 36, n° 1, août 1988, p. 1–25.

[94] K.-F. Lee et S. Mahajan. « The development of a world class Othello program ». *Artificial Intelligence*, vol. 43, n° 1, avril 1990, p. 21–36.

[95] C.E. Leiserson, Z.S. Abuhamdeh, D.C. Douglas, C.R. Feynman, M.N. Ganmukhi, J.V. Hill, W.D. Hillis, B.C. Kuszmaul, M.A. St. Pierre, D.S. Wells, M.C. Wong, S.-W. Yang, et R. Zak. « The Network Architecture of the Connection machine CM-5 ». juillet 1992.

[96] R. Levinson et R. Snyder. « Adaptative Pattern-oriented Chess ». *In : Proceedings of the AAAI*, p. 601–606. 1991.

[97] D.N.L. Levy et M.M. Newborn. *How Computers Play Chess*. W.H. Freeman and Company, 1991.

[98] F.J. MacWilliams et N.J.A. Sloane. *The Theory of Error-Correcting Codes*. North Holland, 1988, 8 édition.

[99] T.A. MARSLAND. « Relative Efficiency of Alpha-Beta Implementations ». *In : 8th Internation International Joint Conference on Artificial Intelligence*, p. 763–766. 1983.

[100] T.A. MARSLAND. « Evaluation-Function Factors. ». *ICCA Journal*, vol. 8, n° 2, 1985, p. 47–57.

[101] T.A. MARSLAND. «communications personelles, Munich», 1993.

[102] T.A. MARSLAND et M.S. CAMPBELL. « Parallel Search of Strongly Ordered Game Trees ». *Computing Surveys*, vol. 14, n° 4, 1982, p. 553–562.

[103] T.A. MARSLAND, M. OLAFSSON, et Schaeffer J.H.. « Multiprocessor Tree-Search Experiments ». *In : Advances in Computer Chess IV*, éd. par D.F. BEAL. p. 37–51. Pergamon Press, 1986.

[104] T.A. MARSLAND et F. POPOWICH. « Parallel Game Tree Search ». *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 7, n° 4, 1985, p. 442–452.

[105] T.A. MARSLAND, A. REINEFELD, et J.H. SCHAEFFER. « Low Overhead Alternatives to SSS* ». *Artificial Intelligence*, vol. 31, n° 2, février 1987, p. 185–199.

[106] D.A. MCALLESTER. « Conspiracy Numbers for Min-Max Search ». *Artificial Intelligence*, vol. 35, n° 3, juillet 1988, p. 287–310.

[107] J. MCCARTHY. « Chess as the Drosophila of AI ». *In : Computers, Chess, and Cognition*, éd. par T.A. MARSLAND et J. SCHAEFFER. p. 228–238. Springer-Verlag, 1990.

[108] M. van der MEULEN. « Weight Assessment in Evaluation Functions ». *In : Advances in Computer Chess 5*, éd. par D.F. BEAL. p. 81–89. Elsevier Science Publishers, 1989.

[109] D. MICHIE. « King and Rook Against King ». *In : Advances in Computer Chess I*, éd. par M.R.B. CLARKE. p. 30–59. Edinburgh University Press, 1977.

[110] F. MORSCH. «Communications personnelles, Munich», 1993.

[111] W. NAGL. « Best-Move-Proving: A Fast Game-Tree Searching Algorithm ». *In : Heuristic Programming in Artificial Intelligence*, éd. par D. N. L. LEVY et D. F. BEAL. p. 255–272. Ellis Horwood, 1989.

[112] D.S. NAU. « Pathology in Game Trees : A Summary of Results ». *In : AAAI Conference Proceedings*, p. 102–104. 1980.

[113] D.S. NAU. « The Last Player theorem ». *Artificial Intelligence*, vol. 18, 1982, p. 53–65.

[114] D.S. NAU. « Decision Quality as a function of Search Depth on Game trees ». *Journal of the ACm*, vol. 30, n° 4, 1983, p. 687–708.

[115] D.S. NAU. « On Game Graph Structure and its Influence on Pathology ». *International Journal of Computer and Information Sciences*, vol. 12, n° 6, 1983, p. 367–383.

[116] D.S. NAU. « Pathology on Game Trees Revisited and an Alternative to Minimaxing ». *Artificial Intelligence*, vol. 21, n° 1-2, 1983, p. 221–244.

[117] R. Nemes. « The Chess-Playing Machine ». *In : Acta Technica*. p. 215–239. Academy of Sciences, 1951.

[118] M.M. Newborn. *Computer Chess*. Academic Press, 1975.

[119] M.M. Newborn. « Unsynchronized Iteratively Deepening Parallel Alpha-Beta Search ». *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. PAMI-10, n° 5, septembre 1988, p. 687–694.

[120] A. Nimzovitch. *Mon système*. Hatier, 1979.

[121] A. Nimzovitch. *Blockade*. ChessEnterprises Inc., 1980.

[122] A. Nimzovitch. *Chess Praxis*. Prinring Craft Ltd., 1986.

[123] T. Nitsche. « A Learning Chess Program ». *In : Advances in Computer Chess 3*, éd. par M.R.B. Clarke. p. 113–120. Pergamon Press, 1982.

[124] P.A. Nolot. « Des tests qui résistent au progrès ». *Europe Echecs*, n° 326, février 86, p. 15–18.

[125] P.A. Nolot. « Une nouvelle grille de tests pour des ordinateurs très en progrès ». *Europe Echecs*, n° 360, décembre 88, p. 19–22.

[126] S.W. Otto et E.W. Felten. « Chess on a Hypercube ». *In : The Third Conference on Hypercube Concurrent Computers and Applications*, p. 1329–1341. 1988.

[127] A.J. Palay. « The B* Tree Search Algorithm – New Results ». *Artificial Intelligence*, vol. 19, n° 2, 1982, p. 145–163.

[128] A.J. Palay. « *Searching with Probabilities* ». Thèse de PhD, Department of Computer Science, Carnegie Mellon University, 1983.

[129] J. Pearl. « Asymptotic Properties of Minimax Trees and Game-Searching Procedures ». *Artificial Intelligence*, vol. 14, 1980, p. 113–138.

[130] J. Pearl. « The solution for the Branching Factor of the Alpha-Beta Pruning Algorithm and Its Optimality ». *CACM*, vol. 25, n° 8, août 1982, p. 559–564.

[131] J. Pearl. *Heuristics : Intelligent Search Strategies for Computer Problem Solving*. Addison-Wesley Publishing Company, 1984.

[132] J. Pitrat. « A Chess Combination program which uses plans ». *Artificial Intelligence*, vol. 8, n° 3, juin 1977, p. 275–321.

[133] W. Poundstone. *Prisoner's dilemma*. Oxford University Press, 1993.

[134] J.R. Quinlan. « *Induction over large data bases* ». Heuristic Programming Project n° HPP-79-14, Stanford University, 1979.

[135] J.R. Quinlan. « Learning Efficient Classification Procedures and Their Application to Chess End Games ». *In : Machine Learning*, éd. par R.S. Michalski, J.G. Carbonel, et T.M. Mitchell. p. 463–482. Springer Verlag, 1984.

[136] A. Reinefeld. « An Improvement to the Scout Tree-Search Algorithm. ». *ICCA Journal*, vol. 6, n° 4, 1983, p. 4–14.

[137] A. Reinefeld. « A Minimax Algorithm Faster than Alpha-Beta ». *In : Advances in Computer Chess 7*, éd. par H.J. van den Herik, I.S. Herschberg, et J.W.H.M. Uiterwijk. p. 237–250. University of Limburg, Maastricht, 1994.

[138] A. Reinefeld et T.A. Marsland. « *Memory Functions in Iterative-Deepening Search* ». Rapport technique n° FBI-HH-M-198/91, Univ. Hamburg, 1991.

[139] A. Reinefeld et T.A. Marsland. « Enhanced Iterative-Deepening Search ». *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 16, n° 7, juillet 1994, p. 701–710.

[140] A. Reinefeld, J.H. Schaeffer, et T.A. Marsland. « Information Acquisition in Minimal Window Search ». *In : Proceeding of The 9th IJCAI*, p. 1040–1043. 1985.

[141] F. Reinfeld. *Win At Chess.* Dover, 1958.

[142] J. Rezaie et R. Finkel. « *A Comparison of Some Parallel Game-Tree Search Algorithms (Revised Version)* ». Rapport technique n° 214, Department of Computer-Science, Univesity of Kentucky, septembre 1992.

[143] R.L. Rivest. « Game Tree Searching by Min/Max Approximation ». *Artificial Intelligence*, vol. 34, n° 1, janvier 1988, p. 77–96.

[144] A.L. Samuel. « Some Studies in Machine Learning using the Game of Checker ». *IBM Journal of Research and Development*, vol. 3, n° 3, 1959, p. 210–229.

[145] A.L. Samuel. « Some Studies in Machine Learning using the Game of Checker II – Recent Progress ». *IBM Journal of Research and Development*, vol. 11, n° 6, 1967, p. 601–617.

[146] J.H. Schaeffer. « The History Heuristic. ». *ICCA Journal*, vol. 6, n° 3, 1983, p. 16–19.

[147] J.H. Schaeffer. « *Experiments in Search and Knowledge* ». Thèse de PhD, University of Waterloo, 1986.

[148] J.H. Schaeffer. « Comment on 'Distributed Game-Tree Search'. ». *ICCA Journal*, vol. 12, n° 4, 1989, p. 216–217.

[149] J.H. Schaeffer. « Conspiracy Numbers ». *In : Advances in Computer Chess 5*, éd. par D.F. Beal. p. 199–217. North-Holland, 1989.

[150] J.H. Schaeffer. « Distributed Game-Tree Search ». *Journal of Parallel and Distributed Computing*, vol. 6, n° 2, 1989, p. 90–114.

[151] J.H. Schaeffer. « A Rejoinder to a Comment on 'Distributed Game-Tree Search'. ». *ICCA Journal*, vol. 13, n° 1, 1990, p. 21.

[152] J.H. Schaeffer. « Comment on 'The Guard Heuristic' ». *ICCA Journal*, vol. 16, n° 3, 1993, p. 185–186.

[153] T. SCHERZER, L. SCHERZER, et D. TJADEN.. « Learning in Bebe. ». *ICCA Journal*, vol. 14, n° 4, 1991, p. 183–191.

[154] J.C. SCHLIMMER et D. FISHER. « A Case Study of Incremental Concept Induction ». *In : Proceedings of the Fifth National Conference on Artificial Intelligence*. p. 135–141. Morgan Kaufman, Philadelphia, 1986.

[155] G. SCHRÜFER. « Presence and Absence of Pathology on Game Trees. ». *In : Advances in Computer Chess 4*, éd. par Don F. BEAL. p. 101–112. Pergamon Press, 1986.

[156] G. SCHRÜFER. « A Strategic Quiescence Search. ». *ICCA Journal*, vol. 12, n° 1, 1989, p. 3–9.

[157] J.J. SCOTT. « A Chess-Playing Program ». *In : Machine Intelligence 4*, éd. par B. MELTZER et D. MICHIE. p. 255–265. Edinburgh University Press, 1969.

[158] J.-L. SERET, M.-F. BAUDOT, et J.-C. WEILL. « *CHEVAL : Un langage d'écriture de fonctions d'évaluation pour programmes d'échecs* ». Rapport technique n° HI-23/94-001, EDF - DER - TIEM - IMA, mars 1994.

[159] C. E. SHANNON. « A mathematical theory of communication ». *Bell System Technical Journal*, vol. 27, 1948, p. 369–423.

[160] C. E. SHANNON. « Programming a Computer for Playing Chess ». *Philosophical Magazine*, vol. 41, n° 7, 1950, p. 256–275.

[161] A. SHAPIRO et T. NIBLETT. « Automatic Induction of Classification Rules For A Chess Endgame ». *In : Advances in Computer Chess*, éd. par M.R.B. CLARKE. p. 73–92. Pergamon Press, 1983.

[162] J.R. SLAGLE et J.K. DIXON. « Experiments with some Programs that Search Games Trees ». *Journal of the ACM*, vol. 16, 1969, p. 189–207.

[163] D.J. SLATE. « A Chess Program that Uses its Transposition Table to Learn from Experience. ». *ICCA Journal*, vol. 10, n° 2, 1987, p. 59–71.

[164] D.J. SLATE et L.R. ATKIN. « Chess 4.5 – The Northwestern University Chess Program ». *In : Chess Skill in Man an Machine*, éd. par P. FREY. p. 82–118. Springer-Verlag, 1977.

[165] W.D. SMITH. «Communications Personnelles», 1993–1994.

[166] I. STEINBERG et M. SOLOMON. « *Searching Game Trees in Parallel* ». Rapport technique n° 877, Computer Science Department, University of Wisconsin-Madison, 1989.

[167] I. STEINBERG et M. SOLOMON. « Searching Game Trees in Parallel ». *In : International Conference on Parallel Processing*, p. 9–17. 1990.

[168] G.C. STOCKMAN. « A Minimax Algorithm Better than Alpha-Beta? ». *Artificial Intelligence*, vol. 12, n° 2, 1979, p. 179–196.

[169] T. STRÖHLEIN. « *Untersuchungen über kombinatorische Spiele* ». Thèse de PhD, Faculty of General Science, Technical University of Munich, 1970.

[170] M. Tarsi. « Optimal Search on Some Game Trees ». *Journal of the ACM*, vol. 30, n° 3, 1983, p. 389–396.

[171] G. Tesauro et T.J. Sejnowski. « A parallel network that learns to play backgammon ». *Artificial Intelligence*, vol. 39, 1989, p. 357–390.

[172] Thinking Machines Corporation, Cambridge, Massachusetts. « *Connection Machine CM-5 Technical Summary* », janvier 1992.

[173] Thinking Machines Corporation, Cambridge, Massachusetts. « *CMMD Reference Manual V3.0* », mai 1993.

[174] K. Thompson. « Computer Chess Strength ». *In : Advances in Computer Chess 3*, éd. par M.R.B. Clarke. p. 55–56. Pergamon Press, 1982.

[175] A. M. Turing, C. Strachey, M.A. Bates, et B.V. Bowden. « Digital Computers Applied to Games ». *In : Faster Than Thought*, éd. par B.V. Bowden. p. 286–310. Pitman, 1953.

[176] A.M. Turing. « Computing Machinery and Intelligence ». *Mind*, vol. 59, n° 2236, octobre 1950, p. 433–460.

[177] J.W.H.M. Uiterwijk. « Hermann Kaindl: Problemloesen durch heuristische Suche in der artificial Intelligence. ». *ICCA Journal*, vol. 12, n° 3, 1989, p. 170–171.

[178] J.W.H.M. Uiterwijk. « The Countermove Heuristic. ». *ICCA Journal*, vol. 15, n° 1, 1992, p. 8–15.

[179] M.J. Uniacke. « *Investigation of Tactical and Positional Evaluation in Computer Chess* ». Project report, The Hatfield Polytechnic, mai 1990.

[180] P.E. Utgoff. « ID5: An Incremental ID3 ». *In : Proc. of the 5th Int'l Workshop on Machine Learning*, p. 107–120. 1988.

[181] P.E. Utgoff. « Incremental Induction of Decision Trees ». *Machine Learning*, vol. 4, n° 2, novembre 1989, p. 161–186.

[182] W. van de Velde. « Incremental Induction of Topologically Minimal Trees ». *In : Proc. of the 7th Int'l Workshop on Machine Learning*, p. 66–74. 1990.

[183] J. von Neuman et O. Morgenstern. *Theory of Games and Economic Behaviour*. Princeton University Press, 1947.

[184] T. Warnock et B. Wendroff. « Search Tables in Computer Chess. ». *ICCA Journal*, vol. 11, n° 1, 1988, p. 10–13.

[185] J.-C. Weill. « *Contribution à la programmation des jeux de réflexion.* ». Mémoire de D.E.A., Université Paris 8 – Vincennes, octobre 1989.

[186] J.-C. Weill. « Experiments With the NegaC* Search — An Alternative for Othello Endgame Search ». *In : Heuristic Programming in Artificial Intelligence 2*, éd. par D.N.L. Levy et Beal D.F.. p. 174–188. Ellis Horwood, 1991.

[187] J.-C. Weill. « The NegaC* Search. ». *ICCA Journal*, vol. 15, n° 1, 1992, p. 3–7.

[188] J.-C. Weill. « How Hard is The Correct Coding of an Easy Endgame ». *In : Advances in Computer Chess 7*, éd. par H.J. van Den Herik, I.S. Herschberg, et J.W.H.M. Uiterwijk. University of Limburg, Maastricht, 1994.

[189] H. Wertz. « *Programmation de jeux : les échecs* ». Rapport technique n° 11-77, Université Paris 8, 1977.

[190] N. Wiener. *Cybernetics : Or control and communication in the animal and the machine.* John Wiley and Sons, Inc, 1948.

[191] C. Ye et T.A. Marsland. « Experiments in Forward Pruning with Limited Extensions. ». *ICCA Journal*, vol. 15, n° 2, 1992, p. 55–66.

[192] E. Zermelo. « Über eine Anwendung der Mengenlehre auf die Theorie des Schachspiels ». *In : Internationaler Mathematikerkongress*, p. 510–504. Cambridge, 1912.

[193] A.L. Zobrist. « *A New Hashing Method with Applications for game Playing* ». Rapport technique n° 88, Computer Science Department, University of Wisconsin, 1970. see also [194].

[194] A.L. Zobrist. « A New Hashing Method with Applications for Game Playing. ». *ICCA Journal*, vol. 13, n° 2, 1990, p. 69–73.

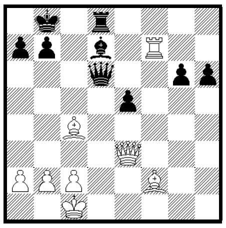
[195] A.L. Zobrist et F.R. Carlson. « An Advice-Taking Chess Computer ». *Scientific American*, vol. 228, n° 6, juin 73, p. 93–105.

[196] K. Zuse. *Der Plankalkül.* 1945. reimprimé dans [197].

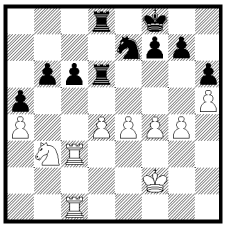
[197] K. Zuse. « *Der Plankalkül* ». Gesellschaft für mathematik und datenverarbeitung n° 106, Bonn university, 1976.

Annexe A

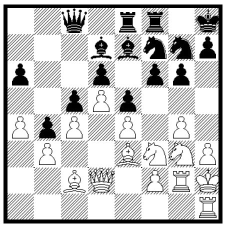
Les positions tests de Bratko et Kopec.



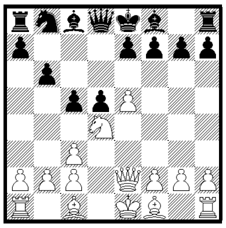
Andersson-Knuttson
(... ♕d1+)



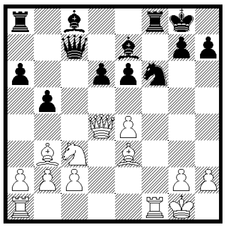
Bogolyubow-Spielman
(d5)



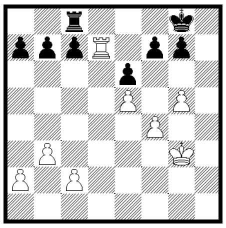
Evans-Rossolimo
(... f5)



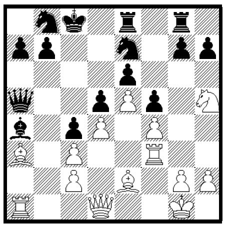
Spielmann-Walter
(e6)



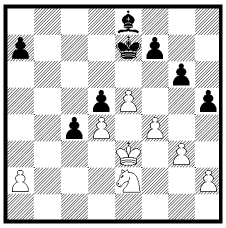
Rogolewicz-Jarecz
(♘d5 ou a4)



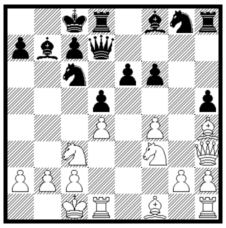
(g6)



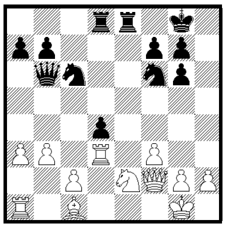
Golyak-Gaiduk
(♘f6)



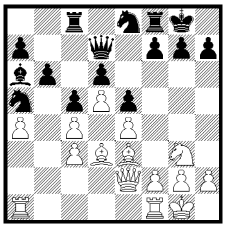
Alekhine-Yates
(f5)



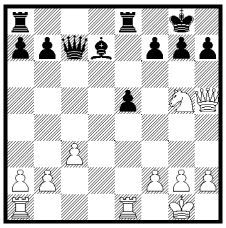
Jansa-Ornstein
(f5)



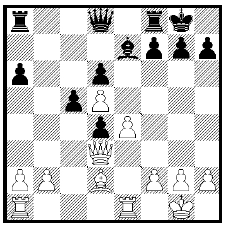
Kabadzjan-Cibelasvili
(... ♘e5)



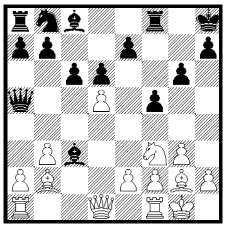
Byrne-Kotov
(f4)



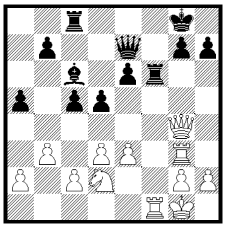
(... ♗f5)



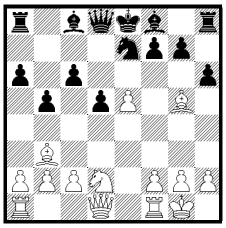
Pfeiffer-Trifunovich
(b4)



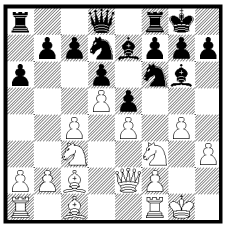
Robatsch-Jansa
(♕d2)



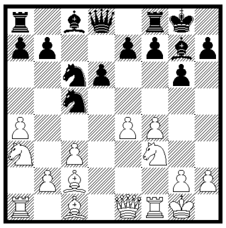
Fischer-Mecking
(♕×g7+)



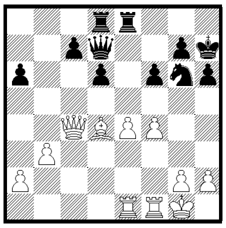
Vasilcuk-Bobolovitch
(♘e4)



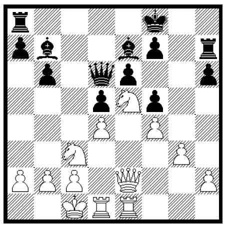
Bosch-Kmoch
(... h5)



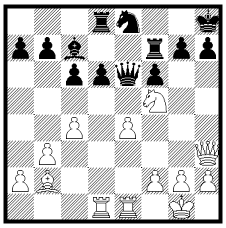
Maric-Fischer
(... ♘b3)



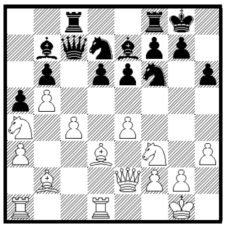
Euwe-Keres
(... ♖×e4)



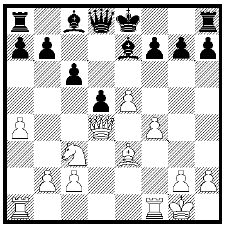
Euwe-Flohr
(g4)



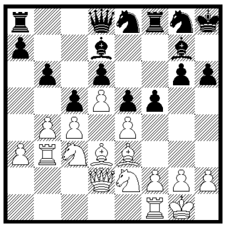
Tarrasch-Blackburne
(♘h6)



Najdorf-Reshevsky
(... ♗× e4)



Jansa-Kavalek
(... f6)



Szabo-Ivkov
(f4)

Glossaire