Parcours parallele d'arbres Minimax

Van-Dat Cung

UNITÉ DE RECHERCHE
INRIA-ROCQUENCOURT

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P. 105
78153 Le Chesnay Cedex
France
Tél.:(1) 39 63 55 11

Rapports de Recherche

N° 1549

Programme 1
Architectures parallèles, Bases de données,
Réseaux et Systèmes distribués

PARCOURS PARALLELE D'ARBRES MINIMAX

Van-Dat CUNG
Catherine ROUCAIROL

Novembre 1991

RR_1549

Parcours Parallèle d'Arbres Minimax

Parallel Minimax tree searching

Van-Dat CUNG et Catherine ROUCAIROL

Octobre 1991

Laboratoire MASI
Université P. et M. Curie - PARIS 6
4, place Jussieu
75252 PARIS Cedex 05, FRANCE

INRIA - Action Paradis
Domaine de Voluceau - BP 105 Rocquencourt
78153 Le Chesnay

e-mail: Cung@seti.inria.fr

Résumé

L'algorithme $\alpha - \beta$ est utilisé fréquemment dans le parcours d'arbres de jeux de type Minimax. Ces arbres sont généralement d'une taille trop grande pour être parcourus dans des temps raisonnables. Le parallélisme est l'un des moyens employés pour diminuer les temps de parcours. Après avoir présenté et classifié les principaux résultats obtenus antérieurement, nous proposons un nouvel algorithme parallèle (CABP) pour les machines multi-processeurs à mémoire partagée. Cet algorithme est fondé sur l'exploration d'une arborescence critique spécifique à l'algorithme $\alpha - \beta$ séquentiel. Nous avons introduit un degré d'élagages variable pour éliminer de façon concurrente certaines branches de l'arborescence. La répartition en deux sous-ensembles des nœuds de l'arbre permet la résolution des problèmes de famine classiques dans la distribution de travail en parallèle. Les résultats expérimentaux obtenus sur une SEQUENT BALANCE 8000 avec 10 processeurs, dont un est alloué au système, montrent l'intérêt de tels choix.

Abstract

The Alpha-Beta pruning is commonly used in Minimax tree searching. However, the size of such trees is usually too large, it is therefore impossible to explore them in a suitable time. Techniques of parallelization are then used to reduce the searching times. After classifying the main studies of previous work, we propose a new parallel algorithm (CABP) for MIMD machines with shared memory. This algorithm is based on the exploration of the critical tree which is specific to the sequential Alpha-Beta pruning. Also, we introduce a variable pruning degree in order to cut-off concurrently some vertices of a Minimax tree. The splitting of the set of vertices into two disjoint subsets makes it possible to solve classical problems of starvation in the distribution of tasks between processors. Experimental results obtained on a SEQUENT BALANCE 8000 with 10 processors - one is allocated to the system - show the relevance of these choices.

Centres d'intérêt.

Algorithme $\alpha - \beta$, parcours d'arbres, parallélisme, machines multi-processeurs à mémoire partagée.

Table des matières

Introduction

Un problème important que nous pouvons trouver en Intelligence Artificielle et en Recherche Opérationnelle est celui du parcours des arbres de solutions dans la résolution des problèmes non numériques. Comme la taille de ces arbres de recherche augmente de façon exponentielle d'après la complexité des problèmes, il est presque impossible d'examiner un arbre en entier dans un temps acceptable.

La classe des algorithmes d'évaluation et séparation (General Branch and Bound GBB [23]) regroupe des méthodes de parcours d'arbres qui ont une caractéristique principale commune : elles évitent toutes l'examen complet d'un arbre pour trouver une ou la solution, par acquisition de connaissances leur permettant d'effectuer des élagages tout au long des parcours. Elles présentent par conséquent un gain en temps et en espace mémoire par rapport aux algorithmes d'énumération explicite. Or nous savons qu'un progrès en temps et/ou en espace permet souvent de résoudre des problèmes plus grands, plus complexes, ou trouver des solutions meilleures. D'où, l'idée d'appliquer sur ces algorithmes la technique de la programmation parallèle offerte par les machines parallèles, pour essayer d'obtenir un gain supplémentaire en temps.

La parallélisation de ces algorithmes n'est pas facile, car ils sont souvent intrinsèquement séquentiels. En conséquence, une parallélisation brutale et réalisée sans précaution peut engendrer un résultat inverse de celui qu'on attend : la perte des propriétés de l'optimisation séquentielle et donc une perte de temps.

Dans la même classe que GBB, nous pouvons citer A^* pour le parcours de graphe d'états, AO^* pour celui des graphes ET/OU, dans des méthodes très utilisés en Recherche Opérationnelle.

Nous nous intéresserons plus particulièrement aux algorithmes alpha-beta ($\alpha - \beta$) et SSS^* qui sont largement exploités en Intelligence Artificielle pour la recherche d'un meilleur coup à jouer dans un arbre de jeux. Leurs parallélisations posent un certain nombre de questions générales communes

à toute recherche arborescente en parallèle :

- Comment éviter que certains processeurs deviennent inactifs en attendant que d'autres finissent leurs travaux (problème de famine) ?
- Comment assurer un bon équilibrage des tâches ?
- Comment gérer la communication inter-processus ?
- Comment éviter l'interblocage entre processus ?
- Comment reconnaître la terminaison des processus ?
- Quelles performances peut-on espérer de la parallélisation ? Quelle accélération peut-on obtenir ?

Et des questions plus spécifiques aux algorithmes comme $\alpha-\beta$ et SSS^* :

- Comment effectuer la remontée de la valeur du meilleur coup ?
- Comment élaguer les nœuds durant les parcours d'arbres ?
- Quelle est l'arborescence critique de ces parcours ?

Les programmes de jeux de stratégie à deux joueurs comme les échecs, Othello, ou le Go, reposent essentiellement sur des techniques d'élagages de branches des arbres développés (algorithmes $\alpha-\beta$, SSS^* et leurs variantes) pour diminuer la croissance exponentielle de leurs tailles. Notre intérêt pour l'algorithme $\alpha-\beta$ [18] provient du fait qu'à ce jour, avec SSS^* [28], c'est l'un des meilleurs algorithmes pour explorer des arbres de jeux qui ont une propriété supplémentaire (fortement ordonnés [6]). Puisque leurs parallélisations ont fait l'objet de nombreuses recherches ces dernières années (de quelques processeurs [2, 16, 22, 27] à une centaine [8, 14, 15], voir un millier [15]), il nous a paru intéressant d'évaluer ces propositions et de concevoir de nouveaux algorithmes parallèles dédiés à des machines commerciales multi-processeurs à mémoire partagée.

Outre les programmes de jeux, l'algorithme $\alpha-\beta$ trouve aussi une importante application dans les systèmes experts déductifs à chaînage arrière où les arbres sont composés de nœuds OU (MAX) et de nœuds ET (MIN). Les valeurs de ces nœuds sont 0 ou 1. Une valeur nulle à un nœud ET suffit pour élaguer les autres fils de ce nœud ET.

Nous allons dans les deux premiers chapitres rappeler respectivement les principes fondamentaux des algorithmes $\alpha - \beta$ et SSS^*, et les travaux antérieurs de leurs parallélisations. Puis dans le troisième chapitre, nous ferons une analyse approfondie de l'algorithme $\alpha - \beta$ pour établir les fondements d'un nouvel algorithme parallèle. Le quatrième chapitre sera consacré à la validation de l'algorithme décrit dans le chapitre précédent. Elle se fera par une implémentation sur machine réelle avec des structures de données choisies. Nous ferons finalement une analyse des résultats obtenus.

Chapitre 1

Les algorithmes séquentiels de parcours d'arbres Minimax

Parmi les algorithmes de parcours d'arbres de jeux, il existe principalement deux grandes familles, issus de l'algorithme $\alpha - \beta$ ou de SSS^*. Nous allons dans les paragraphes suivants voir progressivement les propriétés de chacun.

1.1 Définitions des arbres de jeux et des arbres de recherche

Les jeux qui nous concernent ont des règles de jeux et des buts précis. Ce sont des jeux de stratégie qui présentent les caractéristiques suivantes :

1. Ces jeux sont pratiqués sur une planche de jeu où les pièces sont placées et déplacées.
2. Ces jeux sont pratiqués par exactement deux joueurs.
3. Ces jeux sont de somme nulle, dans le sens où les gains d'un joueur sont exactement les pertes de l'autre joueur. L'issue pour un joueur est une victoire, une défaite ou un match nul.
4. Aucune notion de chance n'est impliquée.
5. A tout instant de la partie, chaque joueur connaît la situation globale, il n'y a pas de détails cachés.

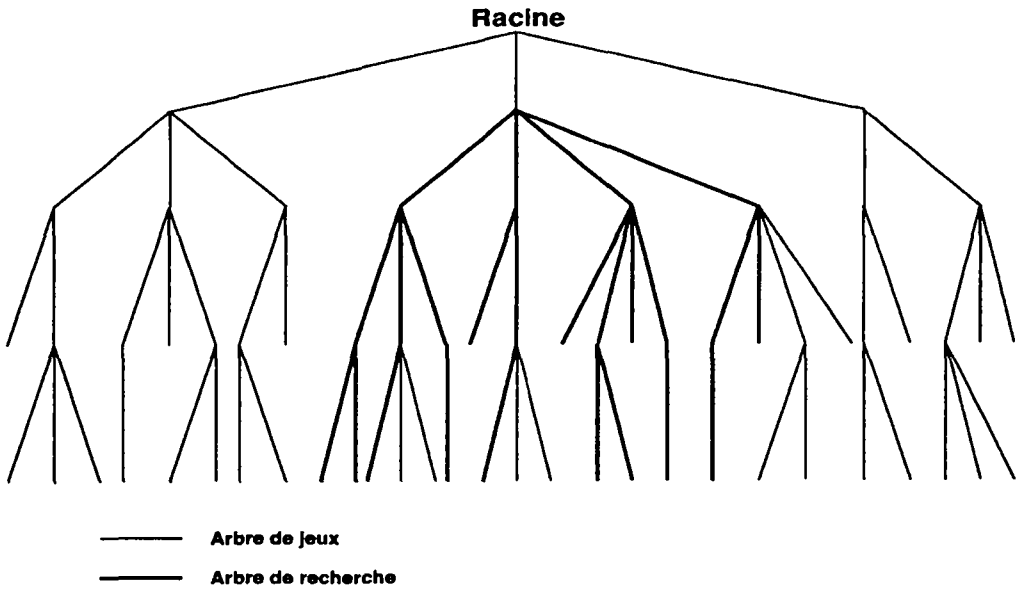

Figure 1.1 : Arbre de jeux et arbre de recherche.

On peut ainsi donner la définition de l'arbre de jeux d'une partie (Figure 1.1).

Définition 1

Un arbre de jeux A^J (Figure 1.1) est défini par le couple (S, *succ*) avec :

- S l'ensemble de toutes les situations possibles du jeu,
- $succ : S \longrightarrow Reg(S)$ fonction donnant toutes les situations filles s_j d'une situation s_i si et seulement s'il existe un coup régulier faisant passer de la situation s_i à la situation s_j.

Chacun des joueurs peut, à tour de rôle, choisir son meilleur coup parmi tous les coups réguliers qui lui sont ouverts par la situation de jeu. Dans la littérature, un coup d'un joueur s'appelle un demi-coup, alors qu'un coup entier est un demi-coup d'un joueur suivi d'un demi-coup de son adversaire. Pour qu'un joueur puisse décider si un demi-coup est meilleur qu'un autre, il doit évaluer dans le cas le plus simple (le joueur prévoit uniquement son demi-coup) la situation du jeu sans se soucier de la riposte de l'adversaire. Cette évaluation dépend essentiellement de la connaissance du jeu pratiqué. Une machine peut effectuer le même genre d'évaluation, toute comparaison gardée, avec la définition d'une fonction heuristique.

Définition 2 *Fonction heuristique d'évaluation f.*

$f : S \longrightarrow \mathcal{Z}$ où $\mathcal{Z}$ est l'ensemble des entiers relatifs. f donne la valeur intrinsèque d'une situation statique du jeu.

On peut espérer, qu'en général, cette fonction f donnerait des valeurs de plus en plus précises au fur et à mesure qu'une partie avance, car le nombre de pièces diminuant les situations deviennent généralement moins complexes, notamment en fin de partie.

Cependant, un bon joueur, ne se satisfait pas d'évaluer uniquement son demi-coup, mais si possible un coup entier ou même plus, afin de prévoir la riposte de son adversaire pour mieux contre-attaquer et ainsi de suite. C'est dans cette phase d'évaluation que la machine prend sa revanche sur l'homme grâce à sa vitesse de calcul. En effet, elle peut examiner des milliers de situations à l'avance et compenser ainsi l'imprécision relative de la fonction d'évaluation.

Toutefois, une machine n'a pas le temps d'examiner toutes les branches d'un arbre de jeux dans les limites imposées par les réglements des jeux. Une profondeur de recherche P fixée à l'avance est en général nécessaire. Cette profondeur est souvent appelée niveau de jeu. Plus P est grande, plus la durée de recherche est longue et meilleur est le coup joué.

Enfin pour être complet sur les arbres, nous donnons ci-dessous trois autres définitions.

Définition 3 *Arbres de recherche*

Les arbres de recherche sont des sous-arbres des arbres de jeux, de profondeur P donnée.

Définition 4 *Arbres ordonnés*

Un arbre de jeux A^J est dit ordonné :

- *meilleur d'abord* ou *dans le meilleur des cas*, si le premier fils gauche de chaque nœud de l'arbre a toujours le meilleur coup,
- *fortement*, si le meilleur coup est à 70% du temps le premier fils gauche et 90% du temps le premier quart des fils gauches de chaque nœud,

- *dans le pire des cas*, si le meilleur coup se trouve sur le premier fils droit de chaque nœud.

Définition 5

Les arbres aléatoires et uniformes de jeux de degré N et de profondeur P ont :

- tous les nœuds non-terminaux ont exactement N fils,
- tous les nœuds terminaux sont à la profondeur P,
- les valeurs des nœuds suivent une loi uniforme et sont toutes indépendantes.

1.2 L'algorithme Minimax et les arbres Minimax

Pour parcourir les arbres de jeux, deux algorithmes similaires existent : celui de Fuller, Gaschnig et Gillogy, 1973 [12] appelé Minimax, et celui de Knuth et Moore, 1975 [18] appelé Negamax.

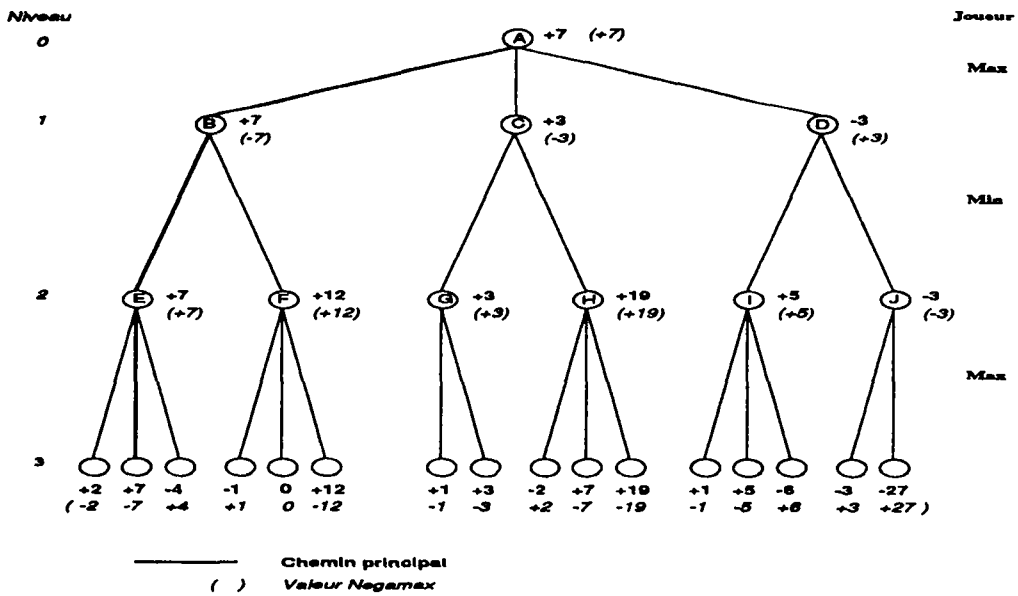

Figure 1.2 : Un arbre Minimax (ou Negamax).

Dans l'approche Minimax, les joueurs sont appelés respectivement Max et Min. Chaque joueur est représenté par une procédure. Lorsque Max joue,

il choisit parmi tous les coups réguliers le coup qui a la valeur maximale pour lui. Cette évaluation se fait récursivement en supposant que l'adversaire Min choisira une riposte qui minimisera le gain de Max, et vice-versa. La racine de l'arbre représente la situation du jeu actuelle et les fils sont les situations éventuellement atteintes après un coup régulier du joueur qui a le trait. L'évaluation avec la fonction heuristique se fait à la profondeur P. Cette évaluation est plus précise que si elle avait été faite à la racine, car elle tient compte de la prévision des coups de chaque joueur jusqu'à la profondeur P. Et l'algorithme Minimax doit simplement remonter la meilleure valeur jusqu'à la racine pour que le joueur qui a le trait puisse choisir son meilleur coup (Figure 1.2).

Définition 6 *Chemin principal (Principal variation en anglais).*

On appelle chemin principal la séquence de coups prédite par l'algorithme Minimax ou Negamax comme étant optimale pour les deux joueurs (Figure 1.2).

L'approche Negamax est une variante de Minimax. La procédure Negamax (Figure 1.3) est plus compacte et plus facile à manipuler. Au lieu de distinguer les joueurs Max et Min, il suffit de considérer que le joueur Min prend la valeur maximale des valeurs opposées des situations filles. A l'aide de cette remarque, les deux procédures Max et Min sont ramenées à une seule.

```
Negamax (s : situation)

        m,t : entier;
        SI s est un nœud terminal
           ALORS retourner f(s);
        m = -∞;
        TANTQUE succ(s) FAIRE { explorer les fils de s }
                    t = -Negamax( succ(s) );
                    SI t > m ALORS m = t;
                    { retenir la meilleure valeur Negamax }
        FTQ;
        retourner m;

Fin Negamax.
```

Figure 1.3 : Procédure Negamax.

1.3 L'algorithme $\alpha - \beta$, ses propriétés et ses variantes

1.3.1 L'intérêt de l'algorithme $\alpha - \beta$

Les deux algorithmes présentés précédemment effectuent à l'évidence un parcours brutal de l'arbre de jeux. A partir de la racine, Negamax (ou Minimax) examine récursivement tous les nœuds de niveaux inférieurs de l'arbre pour obtenir le coup optimal (Figure 1.2).

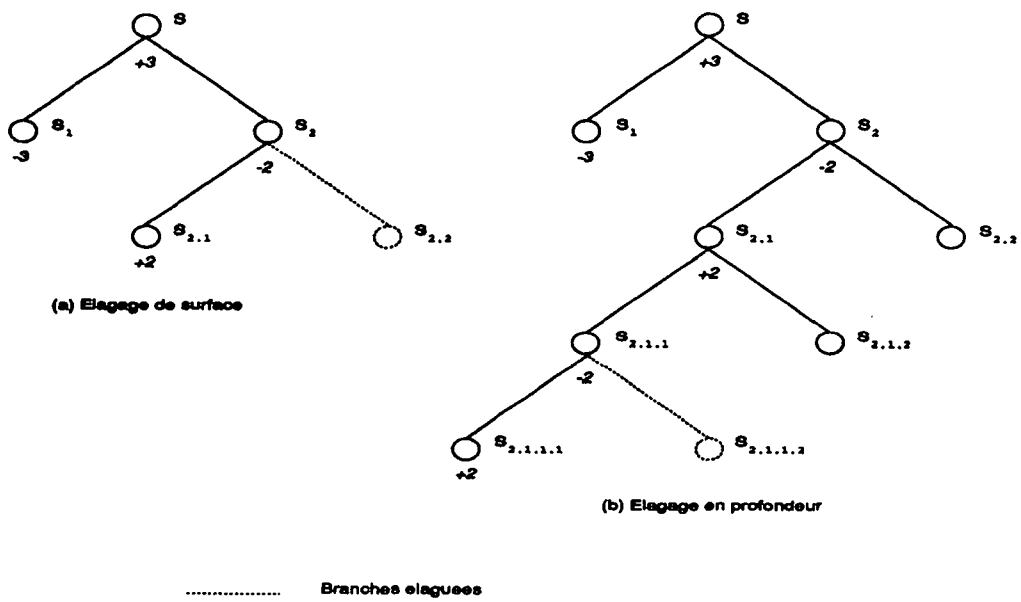

Figure 1.4 : Exemple des deux types d'élagages.

Propriété 1

Nous savons qu'à un nœud s donné de l'arbre de jeux :

$$Negamax(s) \geq -Negamax(s_j) \quad \text{où} \quad s_j \text{ est un fils de } s.$$

Considérons les deux types d'élagages (Figure 1.4). Et prenons d'abord le cas d'*élagage de surface.* A la racine, l'algorithme $\alpha - \beta$ va examiner le fils gauche et trouve :

$$f(s_1) = -3 \quad \text{et} \quad Negamax(s) = -f(s_1) = 3.$$

Puis, s_2 est examiné à son tour. Mais pour obtenir la valeur $Negamax(s_2)$, on évalue $f(s_{2.1})$:

$$f(s_{2.1}) = 2 \quad \text{et} \quad Negamax(s_2) = -2.$$

Avec la propriété 1, on sait que :

$$Negamax(s_2) \geq -Negamax(s_{2.j}) \quad \text{où} \quad s_{2.j} \text{ est un fils de } s_2.$$

Donc on déduit que :

$$Negamax(s_2) \geq -2.$$

Et récursivement :

$$-Negamax(s_2) \leq 2 \leq Negamax(s) = 3.$$

Il est par conséquent inutile d'examiner les autres fils de s_2 sous peine de faire du travail inutile en plus.

L'élagage en profondeur est illustré par la Figure 1.4(b). Il fait intervenir des élagages de plus de deux niveaux de profondeur. Comme dans le cas d'élagage de surface, après l'examen de s_1, $Negamax(s) = 3$. Or l'examen de $s_{2.1.1.1}$ nous donne :

$$Negamax(s_{2.1.1.1}) = 2 \quad \text{et} \quad Negamax(s_{2.1.1}) = -Negamax(s_{2.1.1.1}) = -2.$$

On distingue deux cas :

- si $Negamax(s_{2.1}) > -Negamax(s_{2.1.1}) = -2$ alors la valeur de $s_{2.1}$ ne dépend que des fils $s_{2.1.j}$ avec $j \neq 1$, donc les autres fils $s_{2.1.1.j}$ avec $j \neq 1$ n'ont pas besoin d'être examinés;
- si $Negamax(s_{2.1}) = -Negamax(s_{2.1.1}) = -2$ alors

  $$Negamax(s_2) \geq -Negamax(s_{2.1}) = -2$$

  et

  $$-Negamax(s_2) \leq Negamax(s) = 3.$$

  C'est inutile d'examiner les autres fils de $s_{2.1.1}$.

1.3.2 L'algorithme $\alpha - \beta$ et ses propriétés

Pour la réalisation des deux types d'élagages, deux paramètres et deux tests ont été introduits dans l'algorithme Negamax. L'algorithme $\alpha - \beta$ a été

```
Alphabeta (s : situation; α,β : entier)

        m,t : entier;
        SI s est un nœud terminal
           ALORS retourner f(s);
        m = α;
        TANTQUE succ(s) FAIRE { explorer les fils de s }
                    t = -Alphabeta( succ(s),-β,-m );
                    SI t > m ALORS m = t;
                    { retenir la meilleure valeur Negamax }
                    SI m ≥ β ALORS retourner m;
                    { on élague les autres fils }
        FTQ;
        retourner m;

Fin Alphabeta.
```

Figure 1.5 : Procédure $\alpha - \beta$ classique.

étudié en détail par Knuth et Moore [18]. Une version adaptée est citée ici (Figure 1.5).

Les paramètres α et β jouent le rôle de bornes de recherche pour les niveaux pairs ou impairs de l'arbre de jeux. Aux niveaux pairs, on maximise; et aux niveaux impairs, on minimise. La racine est au niveau 0.

Dans le cas des échecs par exemple, la borne α représente la meilleure valeur des coups explorés à tout instant du parcours pour le joueur qui a le trait, alors que la borne β est celle de son adversaire. Ainsi les coups d'un joueur qui ont des valeurs inférieures ou égales à la borne α ne seront pas retenus, car ils ne sont pas le meilleur coup. De même, les coups qui ont des valeurs supérieures ou égales à la borne β seront élagués, puisqu'ils représenteraient des coups de valeurs moindres et que l'adversaire ne les choisirait certainement pas.

Définition 7 *Fenêtre de recherche.*

Nous appelons fenêtre de recherche, l'intervalle $[\alpha, \beta]$ où α et β sont des entiers relatifs. Et nous appellerons *fenêtre de recherche pleine* :

$$[\alpha, \beta] =]-\infty, +\infty[,$$

et *fenêtre minimale* l'intervalle :

$$[\alpha, \alpha + 1].$$

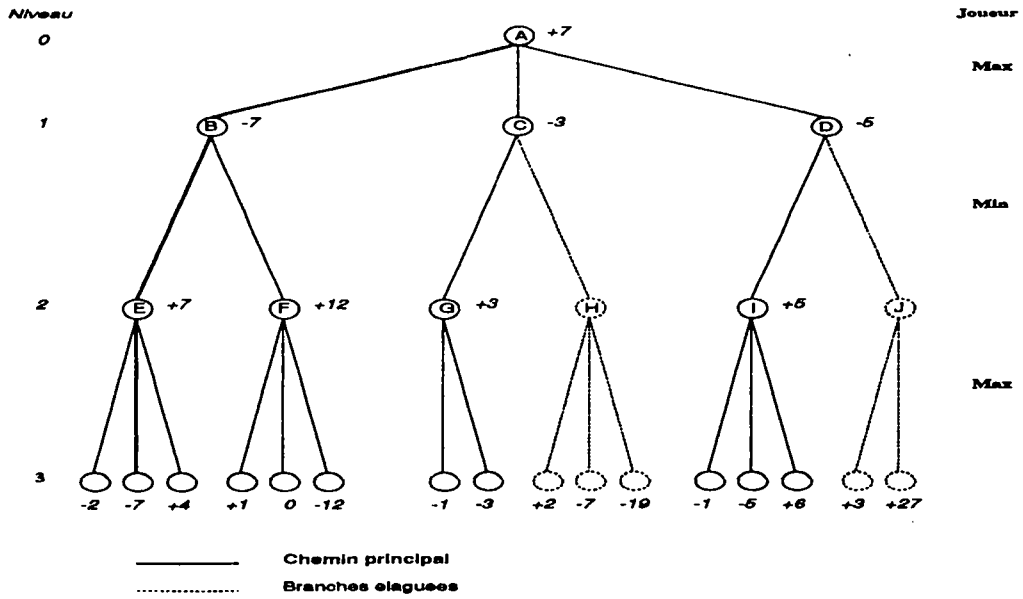

Figure 1.6 : Arbre de recherche de l'algorithme $\alpha - \beta$.

Knuth et Moore ont montré les propriétés suivantes :

Propriété 2

A la racine s d'un arbre de jeux, si la procédure Alphabeta est appelée par $Alphabeta(s, -\infty, +\infty)$ avec une fenêtre de recherche plcine, alors on a :

$$Alphabeta(s, \alpha, \beta) \leq \alpha \quad \text{si } Negamax(s) \leq \alpha, \quad (1.1)$$
$$Alphabeta(s, \alpha, \beta) \geq \beta \quad \text{si } Negamax(s) \geq \beta, \quad (1.2)$$
$$Alphabeta(s, \alpha, \beta) = Negamax(s) \quad \text{si } \alpha < Negamax(s) < \beta. \quad (1.3)$$

Ces propriétés traduisent les faits ci-dessous :

- (1.1), un autre coup de valeur α a été trouvé, meilleur que le coup en cours d'examen. La borne α ne peut être améliorée, on continue à évaluer les fils restants. Ce cas est appelé *échouer bas* (fail-low).
- (1.2), le coup en cours d'examen a une valeur suffisamment grande pour provoquer l'abandon de l'étude des autres fils, on élague. Ce cas est appelé *échouer haut* (fail-high).
- (1.3), le coup en cours d'examen a une valeur qui permet d'améliorer la borne α de la fenêtre de recherche.

L'algorithme $\alpha - \beta$ est appliqué sur l'arbre de jeux de la Figure 1.2. On peut remarquer que le nombre de nœuds examinés a nettement diminué par rapport à ceux examinés par Negamax (Figure 1.6). L'algorithme $\alpha - \beta$ domine donc Negamax.

Définition 8

Un algorithme A est dit dominé par un algorithme B, si A explore plus de nœuds que B quelque soit l'arbre de jeux donné.

Une version affaiblie de l'algorithme $\alpha - \beta$ a aussi été présentée par Knuth et Moore [18]. Dans cette version, la borne α est systématiquement mise à -∞, et β est remplacé par *borne*. Cette version n'effectue pas les élagages profonds. Knuth et Moore l'ont dénommé $F1$.

Propriété 3

L'algorithme $\alpha - \beta$ affaibli $F1$ a conservé seulement deux des trois propriétés de l'algorithme classique :

$$F1(s, borne) \geq borne \quad \text{si} \quad Negamax(s) \geq borne, (3.1)$$
$$F1(s, borne) = Negamax(s) \quad \text{si} \quad Negamax(s) < borne. (3.2)$$

C'est à dire respectivement (1.2) et (1.3) des propriétés de l'algorithme classique.

1.3.3 Condition Nécessaire et Suffisante d'examen d'un nœud

Dans ce paragraphe, nous introduirons une condition nécessaire et suffisante pour qu'un nœud d'un arbre de jeux soit examiné par l'algorithme $\alpha - \beta$ classique et affaibli. Puis nous verrons le total des nœuds examinés.

Notations et définitions

Comme Knuth et Moore [18], nous utiliserons la notation décimale de Dewey pour représenter un nœud d'un arbre de jeux. Une séquence d'entiers naturels $a_1.a_2.....a_l$ désigne une position au niveau l d'un arbre. La racine d'un arbre a la séquence vide ε. Si s est un nœud non-terminal, alors $s.j$

sera le $j^{ième}$ fils du nœud s, avec $j = 1, \ldots, N$. Dans la Figure 1.7, le nœud 4.1.3.4.3 est le $3^{ième}$ nœud de la profondeur 5.

La valeur Negamax associée à s sera notée $\eta(s)$. Si s est un nœud terminal, alors $\eta(s)$ est la valeur de la situation statique de jeu. Et si s est un nœud non-terminal avec N fils, alors nous avons :

$$\eta(s) = \max\{-\eta(s.j) | 1 \leq j \leq N\}.$$

Pour tous les nœuds $s.j$ de profondeur $P \geq 1$, nous définissons :

$$c(s.j) = \max\{-\eta(s.i) | 1 \leq i \leq j-1\}.$$

$c(s.j)$ est la plus grande valeur Negamax de profondeur $P \geq 1$ de l'ensemble des nœuds examinés avant le nœud $s.j$. $c(s.j)$ prend la valeur $-\infty$ quand $j = 1$ (i.e. l'ensemble des nœuds est vide). Dans l'arbre de la Figure 1.7, on a des exemples de valeurs de $c(s.j)$ à chaque niveau.

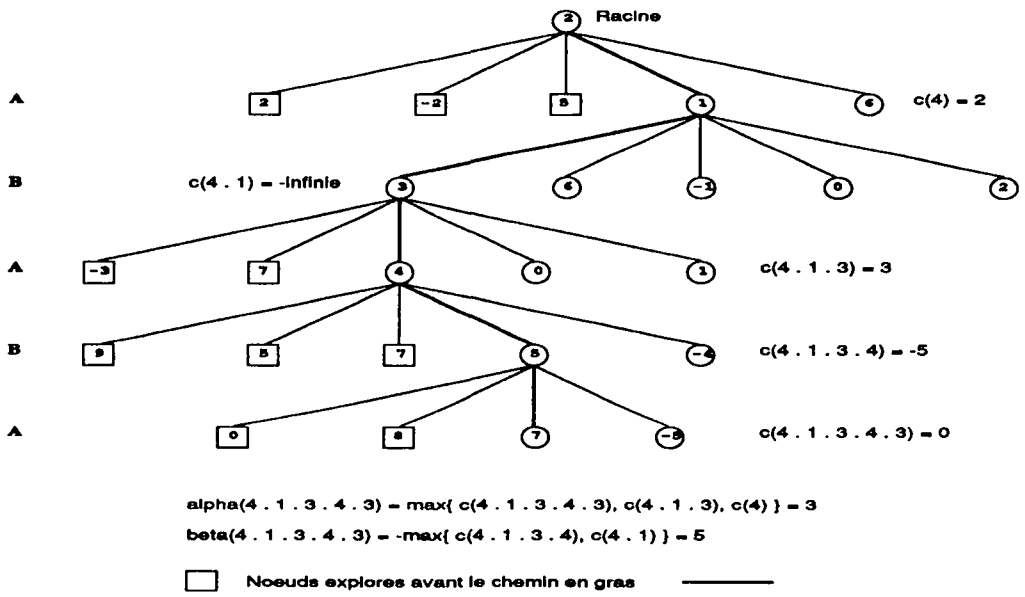

Figure 1.7 : Les valeurs de $A(s)$ et de $B(s)$.

Si un nœud à la profondeur $P \geq 1$ est noté $s = a_1.a_2.\ldots.a_P$, nous définissons $s_i = a_1.a_2.\ldots.a_{P-i}$ pour $0 \leq i < P$. Ainsi s_0 est le nœud s lui-même et s_1 est le père de s.

Théorème d'examen d'un nœud

Théorème 1 *Condition Nécessaire et Suffisante d'examen d'un nœud.*

Avec une fenêtre de recherche pleine, l'algorithme $\alpha - \beta$ explorera un nœud s de profondeur P si et seulement si :

$$A(s) + B(s) < 0$$

où

$$\begin{aligned} A(s) &= \max\{c(s_i) | i \text{ est pair}, 0 \leq i < P\} \\ B(s) &= \max\{c(s_i) | i \text{ est impair}, 0 \leq i < P\}. \end{aligned}$$

Ce théorème a été établi par Fuller, Gaschnig et Gillogy [12]. Les valeurs $A(s)$ et $B(s)$ sont respectivement α et $-\beta$. On retrouve bien la condition $\alpha < \beta$.

Pour l'algorithme $\alpha - \beta$ affaibli, une condition nécessaire et suffisante similaire existe.

Théorème 2 *Condition Nécessaire et Suffisante d'examen pour l'algorithme affaibli.*

Avec une fenêtre de recherche pleine, l'algorithme $\alpha - \beta$ affaibli explorera un nœud s de profondeur P si et seulement si :

$$c(s_i) + c(s_{i-1}) < 0 \quad \text{pour} \quad 0 < i < P.$$

A partir de ces deux théorèmes ci-dessus, il est clair que la version $\alpha - \beta$ classique domine la version affaiblie dans le sens que l'algorithme classique n'examine jamais un nœud de plus que l'algorithme affaibli. Quand la profondeur de l'arbre de jeux dépasse 3, les élagages profonds commencent pour l'algorithme classique alors que les deux versions effectuent les mêmes élagages de surface.

Cependant, Knuth et Moore [18] prétendent que les élagages en profondeur n'ont qu'un effet secondaire sur l'efficacité de la recherche. En théorie, on a vu que la différence du nombre de nœuds explorés par les deux algorithmes augmente lorsque la profondeur des arbres croît. Mais en pratique, cette dominance n'est pas absolue, à cause notamment de l'utilisation des tables de transposition (cf. paragraphe 1.3.5) qui est une table de hash-coding permettant la sauvegarde des situations de jeu examinées. On évite avec cette table de réexplorer des situations identiques à deux instants différents de la partie.

1.3.4 Notion d'arbre minimal et nombre minimal de nœuds explorés

L'algorithme $\alpha - \beta$ est directif, c'est à dire que l'examen des nœuds va de gauche à droite. Son efficacité dépend donc fortement de l'ordonnancement des nœuds de l'arbre. De nombreux élagages peuvent être réalisés si l'arbre est ordonné meilleur d'abord. Mais si un arbre est ainsi parfaitement ordonné, on n'aurait plus besoin de le parcourir pour trouver la meilleure solution !

Cependant, dans les programmes modernes de jeux, la fonction heuristique d'évaluation statique f donne des valeurs suffisamment précises pour que l'arbre de jeux soit presque totalement ordonné.

La connaissance du nombre minimal de nœuds explorés par les algorithmes $\alpha - \beta$ est donc un des facteurs essentiels pour vérifier leur efficacité.

Définition 9 *Nœuds critiques.*

Les nœuds d'un arbre ordonné meilleur d'abord qui sont explorés par les algorithmes $\alpha - \beta$, sont appelés les nœuds critiques.

Définition 10 *Règles de détermination des nœuds critiques.*

Il existe trois types de nœuds critiques pour l'algorithme $\alpha - \beta$ classique :

1. La racine de l'arbre de jeux est un nœud de type 1.
2. Le premier fils d'un nœud de type 1 est de type 1, les autres fils sont de type 2.
3. Le premier fils d'un nœud de type 2 est de type 3.
4. Tout fils d'un nœud de type 3 est de type 2.

Avec la notation décimale de Dewey, les règles s'écrivent comme suit :

1. Un nœud $a_1.a_2.\ldots.a_l$ est critique si $a_i = 1$ pour tout i (type 1), ou pour i pair (type 2), ou pour i impair (type 3).
2. Si $a_i = 1$ pour tout i, alors le nœud est de type 1.
3. Si $a_{l-2n-1} = 1$ pour toutes les valeurs entières possibles de n, alors le nœud est de type 2.
4. Si $a_{l-2n} = 1$ pour toutes les valeurs entières possibles de n, alors le nœud est de type 3.

Pour la version affaiblie, les nœuds de type 3 sont remplacés par les nœuds de type 1.

Définition 11 *Arbre critique (Figure 1.8).*

Les sous-arbres de jeux composés uniquement de nœuds critiques sont appelés arbres critiques.

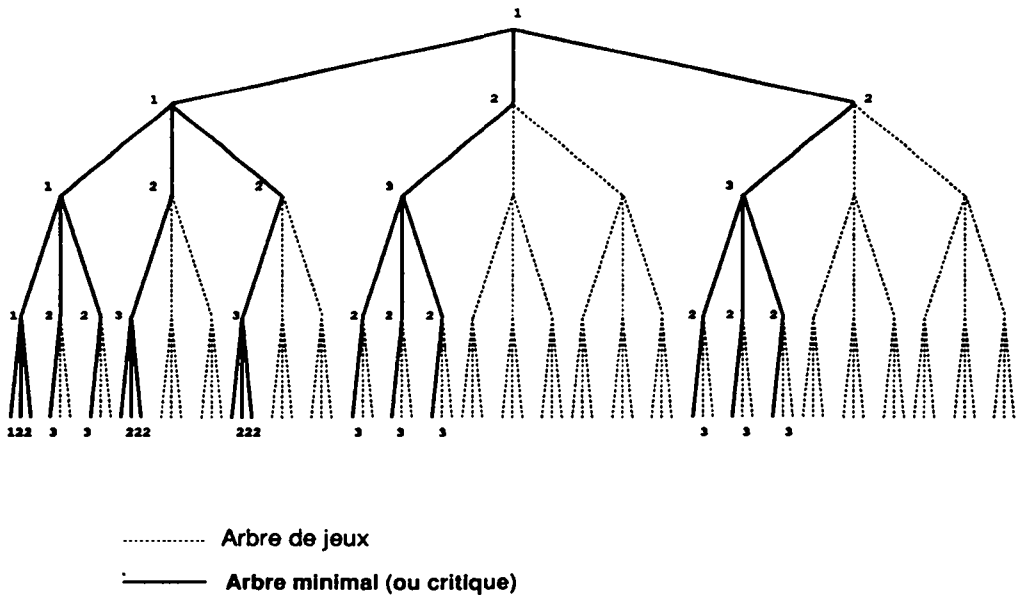

Figure 1.8 : Arbre minimal (ou critique).

Knuth et Moore [18] ont montré que l'arbre critique est systématiquement exploré par l'algorithme $\alpha - \beta$ classique avec une fenêtre de recherche pleine $]-\infty, +\infty[$ et respectivement par la version affaiblie avec *borne* mise à $+\infty$. Et ceci est fait quelque soit la fonction heuristique d'évaluation f.

Théorème 3

Avec une fenêtre de recherche pleine à la racine, l'algorithme $\alpha - \beta$ classique explorera toujours les nœuds critiques.

Preuve :

1. Un nœud $s = a_1.a_2.\ldots.a_P$ est de type 1 si et seulement si $a_i = 1$ pour $1 \leq i \leq P$. Or de la définition de $A(s)$ et de $B(s)$ du Théorème 1, nous avons $A(s) = B(s) = -\infty$. Donc nous avons toujours $A(s) + B(s) < 0$ vérifiée. Par conséquent, les nœuds de type 1 sont toujours explorés par une fenêtre de recherche pleine à la racine.

2. Pour les nœuds $s = a_1.a_2.\ldots.a_P$ de type 2, nous avons par définition $a_{P-i} = 1$ pour $0 \leq i < P$, et i impair. Or par la définition de $B(s)$, nous savons que $B(s) = -\infty$. Donc $A(s) + B(s) < 0$ est encore vérifiée et les nœuds de type 2 sont explorés par une fenêtre de recherche pleine à la racine.
3. De même pour les nœuds s de type 3, nous avons $A(s) = -\infty$. Donc $A(s) + B(s) < 0$ est vérifiée. Les nœuds de type 3 sont aussi explorés par une fenêtre de recherche pleine à la racine.

□.

Propriété 4

Le nombre de nœuds terminaux d'un arbre critique de degré N et de profondeur P est de :

$$N^{\lceil P/2 \rceil} + N^{\lfloor P/2 \rfloor} - 1 \approx 2M^{\frac{1}{2}}.$$

M est le nombre de nœuds terminaux de l'arbre.

A part le total de nœuds explorés, un autre indicateur de performance [18] appelé *facteur de branchement* est utilisé dans les études théoriques. En pratique, une grande profondeur P des arbres nécessiterait une mémoire importante, le calcul du facteur n'est donc pas réalisable.

Définition 12 *Facteur de branchement.*

Si $T_{N,P}$ est le nombre de nœuds terminaux examinés dans un arbre de jeux de degré N et de profondeur P, on appelle facteur de branchement :

$$\lim_{P \to +\infty} (T_{N,P})^{\frac{1}{P}}.$$

1.3.5 Les améliorations et les variantes de l'algorithme $\alpha - \beta$

Dans les programmes récents de jeux, des techniques de programmation et des modifications ont été apportées pour tenter d'améliorer les performances de l'algorithme $\alpha - \beta$ initial. Parmi elles, on peut en citer principalement quatre :

1. la *levée de* α (α raising),

2. la *recherche probabiliste* (aspiration search),

3. l'*approfondissement itéré* (iterative deepening),

4. l'utilisation des *tables de transposition.*

Les deux premières modifications sont liées fortement aux propriétés de l'algorithme $\alpha - \beta$ (Propriété 2).

La levée de α consiste à explorer le dernier fils (ou coup) de la racine d'un arbre de jeux non pas avec la fenêtre $[\alpha, +\infty[$ mais avec une fenêtre minimale $[\alpha, \alpha + 1]$. Etant données que les valeurs des nœuds sont entières, le cas où il faut affiner les valeurs de la fenêtre de recherche ne sera jamais réalisé. Donc si on est dans le cas d'échouer bas (1.1) alors on est sûr que ce dernier fils n'a pas la meilleure valeur. Par contre, si on échoue haut (1.2) alors il faut retenir ce dernier fils comme étant le meilleur. Cette méthode à l'avantage d'explorer moins de nœuds et donc de gagner du temps sur le dernier fils de la racine, puisque la fenêtre de recherche est nettement moins grande et de nombreux élagages peuvent être faits. Mais l'inconvénient est qu'on ne peut connaître la valeur exacte du dernier fils. Cette modification a donné naissance à l'algorithme Lalphabeta.

La levée de α est aussi utilisée dans d'autres variantes comme Palphabeta, Scout et PVSearch (Principle Variation Search [22]). Ces trois algorithmes privilégient l'examen du premier fils gauche d'un nœud, puisque les arbres de jeux sont supposés ordonnés fortement. Palphabeta, Scout et PVSearch examinent le premier fils gauche avec une fenêtre de recherche identique à celle de l'algorithme $\alpha - \beta$, mais ils utilisent la fenêtre $[-\alpha - 1, -\alpha]$ plutôt que $[-\beta, -\alpha]$ pour essayer de réfuter rapidement les autres fils avec la propriété d'échouer bas. Cependant, si le test de réfutation échoue haut alors il faut réexaminer le fils concerné avec la fenêtre $[-\beta, -\alpha]$. Ce réexamen peut provoquer la réexploration de certains nœuds lors du test, et donc augmente par la même occasion le nombre final de nœuds examinés.

La recherche probabiliste consiste à fixer une fenêtre de recherche plus serrée dès le début de la recherche. Au lieu d'appeler Alphabeta avec la fenêtre habituelle de $]-\infty, +\infty[$, on va utiliser plutôt celle de $[V - e, V + e]$ où V est une estimation sur la valeur Negamax, et e l'erreur attendue. On peut prendre par exemple comme estimation $V = f(s)$, où $f(s)$ est la valeur donnée par la fonction heuristique sur la situation de jeu s. On fait appel de nouveau à la propriété 2 de l'algorithme $\alpha - \beta$. Si l'estimation de la valeur Negamax V est correcte alors on aura gagné du temps grâce à une fenêtre

de départ plus petite. Si on échoue bas alors on rappelle Alphabeta avec la fenêtre $]-\infty, V-e]$ et on est sûr d'obtenir la solution. De même si on échoue haut, on rappelle Alphabeta avec la fenêtre $[V+e, +\infty[$. Dans le pire des cas où il faut rappeler Alphabeta avec une fenêtre adéquate, on n'aura examiné qu'une fenêtre de $]-\infty, V+e]$ ou $[V-e, +\infty[$ qui est plus serrée que la fenêtre pleine. Cependant, des nœuds peuvent être réexplorés. La recherche probabiliste a donné l'algorithme Falphabeta (fail-soft-alphabeta).

L'approfondissement itéré n'est pas à proprement parler une modification de l'algorithme $\alpha - \beta$. Il consiste à réordonner les nœuds de chaque niveau après leurs explorations. Ceci dans le but d'obtenir un arbre mieux ordonné pour des futurs examens. Par exemple pour un arbre de profondeur P, on évalue les fils de la racine avec la fonction heuristique f et on les ordonne suivant leurs valeurs, puis on les explore jusqu'à la profondeur 2 pour avoir des valeurs Negamax plus fines et on les réordonne de nouveau, et ainsi de suite jusqu'à la profondeur P. Il est évident que cette technique de recherche est très coûteuse si l'arbre de jeux n'est pas fortement ordonné. Car il faut trier après l'exploration de chaque niveau supplémentaire.

La table de transposition est plus liée aux jeux qu'aux parcours d'arbres. En effet, elle a pour rôle de sauvegarder les situations de jeu déjà examinées avec leurs valeurs Negamax. Ainsi, si à un autre moment du jeu, on parvient à une situation existant dans la table, il suffit d'aller chercher la valeur Negamax plutôt que de réexaminer la situation. On accède à cette table à l'aide d'une fonction hash. Cette table ne fait pas partie de l'objet de recherche de cette étude, mais il faut savoir que dans une implémentation réelle d'un programme de jeu, la vitesse de parcours d'un arbre de jeux peut dépendre fortement de la gestion de la table.

1.4 Les algorithmes SSS^*

L'algorithme SSS^* [28], contrairement à l'algorithme $\alpha - \beta$, est non directif. Il ne parcourt pas les fils d'un nœud non-terminal de gauche à droite comme le fait l'algorithme $\alpha - \beta$. Nous verrons que malgré une approche complètement différente, les deux algorithmes sont assez similaires [14].

L'idée de SSS^* est de considérer l'arbre de jeux comme un arbre ET/OU. La racine de l'arbre est de type ET (respectivement OU), et tous les fils immédiats des nœuds ET (resp. OU) sont de type OU (resp. ET).

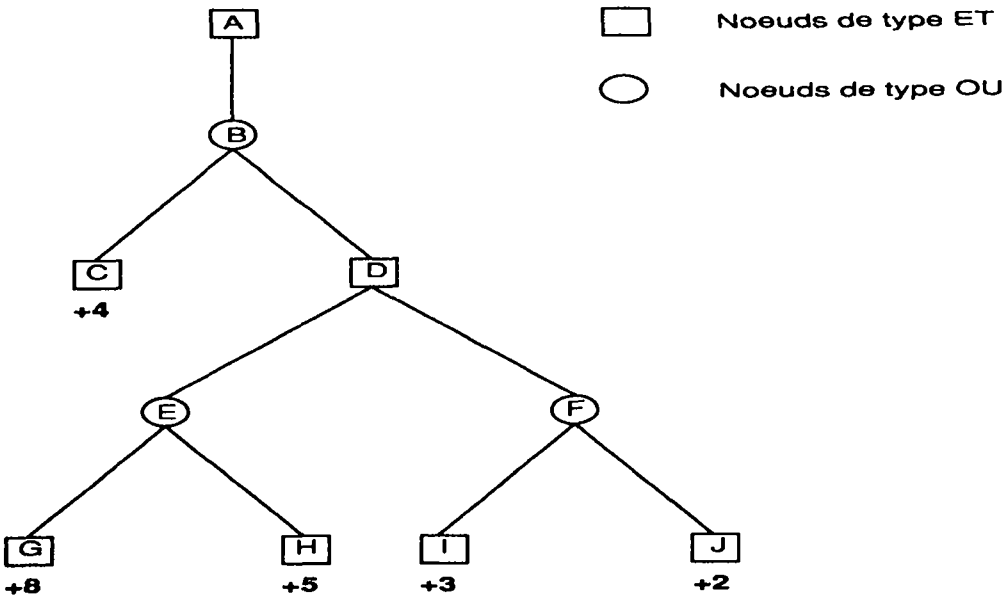

Figure 1.9 : Arbre de jeux et arbre de recherche.

Définition 13

Un *arbre solution* A^S d'un arbre de jeux A^J (Définition 1) est un arbre de recherche avec les propriétés suivantes :

- Le nœud racine de A^J est aussi nœud racine de A^S,
- Si un nœud non-terminal de A^J est dans A^S, alors tous ses fils sont dans A^S s'ils sont du type ET, et seulement un des fils est dans A^S s'ils sont du type OU.

La valeur de l'arbre de solution A^S est notée $f_{A^S}(r)$ où r est la racine de l'arbre de jeux. Cette valeur est définie comme étant la valeur minimale de tous les nœuds terminaux de A^S.

Propriété 5

En général, pour un arbre solution A^S d'un arbre de jeux A^J avec la racine r, on a $Negamax(r) \geq f_{A^S}(r)$. Et pour certains arbres de solution, on a l'égalité.

La propriété précédente nous garantit la validité de l'algorithme SSS^*.

L'arbre de solution A^S est constitué de nœuds d'état. Chaque nœud d'état est un triplet (s, e, v) où s est un nœud de A^S, e est l'état du nœud s (RÉSOLU ou NON), et v est la valeur de l'état.

1. >(A,N,+∞)# { A : nœud de type ET, non-terminal }
2. >(B,N,+∞)# { B : OU, non-terminal }
3. >(C,N,+∞)# { C : ET, non-terminal }
4. >(C,R,+4)#
5. >(D,N,+4)# { D : ET, non-terminal }
6. >(F,N,+4) (E,N,+4)# { E,F : OU, non-terminaux }
7. >(I,N,+4) (E,N,+4)# { I : ET, terminal }
8. >(E,N,+4) (I,R,+3)#
9. >(G,N,+4) (I,R,+3)# { G : ET, terminal }
10. >(G,R,+4) (I,R,+3)#
11. >(I,R,+3) (H,N,+4)# { H : ET, terminal }
12. >(H,N,+4) (J,N,+3)# { J : ET, terminal }
13. >(H,R,+4) (J,N,+3)#
14. >(J,N,+3) (E,R,+4)#
15. >(E,R,+4) (J,R,+2)#
16. >(D,R,+4)# { On élague J et donc F }
17. >(B,R,+4)#
18. >(A,R,+4)#

> : Tête de la liste OUVERT, # : Queue de la liste OUVERT

N : NON, R : RÉSOLU

Figure 1.10 : Liste OUVERT pour l'arbre ET/OU de la Figure 1.9.

Voici l'algorithme SSS^* :

1. Placer l'état de départ (RACINE, NON, $+\infty$) dans une liste appelée OUVERT.
2. Prendre un état $n=(s,e,v)$ à la tête de OUVERT avec la plus grande valeur v. Comme la liste OUVERT est classée dans l'ordre décroissant de v, n est premier dans la liste.
3. Si s=RACINE et e=RÉSOLU, finir l'exploration de l'arbre de jeux et renvoyer v comme étant la valeur de l'arbre.
4. Développer l'état n en appliquant l'opérateur Γ décrit dans le Tableau 1.1.
5. Aller en 2.

Un exemple de déroulement de SSS^* est donné dans les Figures 1.9 et 1.10.

L'algorithme SSS^* domine celui de $\alpha - \beta$, car il explore un sous ensemble des nœuds explorés par $\alpha - \beta$. Cependant, sa dominance est moindre pour les arbres fortement ordonnés, et pour un arbre ordonné meilleur d'abord, il explore le même nombre de nœuds terminaux que $\alpha - \beta$. De plus, pour les arbres de jeux de degré N et de profondeur P, la liste OUVERT est de l'ordre de $N^{\frac{P}{2}}$ éléments !

Des variantes de SSS^* pour diminuer la taille de la liste OUVERT existent. L'une d'elles est le SSS^* *par étape* qui consiste à exécuter SSS^* jusqu'à une certaine profondeur p, puis en appliquant de nouveau SSS^* sur les états obtenus sur une profondeur de p, et ceci récursivement jusqu'à la profondeur P fixée au départ. La deuxième variante serait le SSS^*/Alphabeta, on applique aux premiers niveaux de l'arbre de jeux l'algorithme SSS^*, et $\alpha - \beta$ aux derniers niveaux. Mais ces deux variantes ont des performances moindre que $\alpha - \beta$ ou SSS^* [6]. La dernière variante s'appelle Alphabeta/SSS^*, on applique $\alpha - \beta$ sur le haut de l'arbre de jeux et SSS^* sur le bas de l'arbre. Cette variante domine $\alpha - \beta$ et elle est dominée par SSS^*. Mais elle a l'avantage d'utiliser moins de place mémoire.

Hiromoto et al. [14] ont montré que l'algorithme $\alpha - \beta$ peut être formulé avec une liste OUVERT comme SSS^*; excepté que $\alpha - \beta$ extraira toujours le nœud d'état le plus à gauche dans l'arbre de jeux de la liste OUVERT, tandis que SSS^* extraira le nœud d'état qui a la plus grande valeur v.

Bien qu'il ne soit pas nettement plus performant que $\alpha - \beta$, il est intéressant d'étudier la formulation de SSS^* avec la liste OUVERT qui est une file de priorité. En effet, la plupart des algorithmes en Intelligence Artificielle et en Recherche Opérationnelle sont formulés comme un problème de liste OUVERT. Aussi, les algorithmes parallèles récents que nous verrons dans le chapitre 2 travaillent justement avec des listes OUVERT dont des processus parallèles ont des accès concurrents. Les idées nouvelles sur la gestion des listes OUVERT dans l'algorithme $\alpha - \beta$ peuvent donc être également appliquées pour d'autres algorithmes comme A^* ou AO^* [23] pour ne citer qu'eux.

Cas de l'opérateur Γ	Conditions satisfaites par le nœud d'état (s, e, v)	Actions de l'opérateur Γ
1.	$e =$ RÉSOLU $s \neq$ RACINE $type(s) =$ OU	Mettre en fin de la liste OUVERT $(m = parent(s), e, v)$ et éliminer de OUVERT tous les nœuds d'état (k, e, v) où m est un ancêtre de k.
2.	$e =$ RÉSOLU $s \neq$ RACINE $type(s) =$ ET $prochain(s) \neq$ Nil	Mettre en fin de la liste OUVERT $(prochain(s), NON, v)$.
3.	$e =$ RÉSOLU $s \neq$ RACINE $type(s) =$ ET $prochain(s) =$ Nil	Mettre en fin de la liste OUVERT $(parent(s), e, v)$.
4.	$e =$ NON $s =$ terminal	Insérer dans OUVERT $(s, RÉSOLU, \min(v, value(s)))$ derrière tous les nœuds d'état ayant une plus grande valeur v et devant tous ceux de même valeur v dont s est à gauche dans l'arbre de jeux.
5.	$e =$ NON $s \neq$ terminal $type(premier(s)) =$ ET	Mettre en fin de la liste OUVERT $(premier(s), e, v)$.
6.	$e =$ NON $s \neq$ terminal $type(premier(s)) =$ OU	$s = dernier(s)$; Tantque $s \neq$ Nil Faire Mettre en fin de la liste OUVERT (s, e, v); $s = précèdent(s)$; Ftq.

type() : fonction donnant le type du nœud s,
premier() : fonction donnant le premier fils d'un nœud s,
dernier() : comme *premier*(), mais donne le dernier fils,
prochain() : donne le nœud frère de droit d'un nœud s,
précèdent() : donne le nœud frère de gauche d'un nœud s,
parent() : donne le nœud père d'un nœud s,
ancêtre() : donne tous les nœuds ancêtre d'un nœud s.

Tableau 1.1 : L'opérateur Γ.

Chapitre 2

Les algorithmes parallèles de parcours d'arbres Minimax

Dans ce chapitre, nous allons faire un état de tous les travaux antérieurs de parallélisation des divers algorithmes de parcours d'arbres du chapitre 1.

Nous avons vu que les algorithmes séquentiels utilisent des optimisations intrinsèquement séquentielles, notamment le reserrement progressif de la fenêtre de recherche pour l'algorithme $\alpha - \beta$. On peut alors se demander pourquoi vouloir effectuer des parallélisations qui risquent de faire perdre les effets des élagages ?

Nous savons qu'aux échecs par exemple, la possibilité d'analyser un demi-coup supplémentaire permet au programme de gagner 200 à 250 points. Et chaque fois que la vitesse de recherche est multipliée par un facteur deux, le gain est d'environ 100 points [13, 29]. Par les techniques de parallélisation, un gain de temps éventuel permettrait de trouver de meilleures solutions dans les temps imputés à chaque joueur.

Cependant, les problèmes suivants restent posés :

- Comment paralléliser ces algorithmes séquentiels de parcours d'arborescence ?
- Quelle est la taille des tâches (problème de granularité) ?
- Quel type d'architecture utiliser ?
- Comment éviter la famine en optimisant l'équilibrage des charges ?
- A quelle accélération et à quelle efficacité ?

2.1 Les travaux antérieurs de parallélisation

De nombreux auteurs [2, 4, 11, 19, 21] se sont intéressés à la parallélisation de l'algorithme $\alpha - \beta$ et ses diverses variantes; d'autres [14] ont travaillé sur l'algorithme SSS^* et certains proposent de créer un nouvel algorithme séquentiel plus apte à être parallélisé que $\alpha - \beta$ [27].

D'après les propriétés d'élagages de ces algorithmes, il est clair que les ordinateurs de type SIMD à forte synchronisation de tâches ne conviennent pas. Notamment l'amélioration de la fenêtre de recherche ne pourra être faite que de façon totalement asynchrone par les processus en parallèle. Sinon les propriétés de l'optimisation seront entièrement perdues. La plupart des recherches ont été faites sur des machines MIMD avec ou sans mémoire partagée. Nous distinguons deux types de configuration des processus pour les implémentations de ces algorithmes :

1. la *configuration hiérarchique* où tous les processus sont disposés en arbre, appelé *arbre de processus* [11, 21]. Souvent, un processus est affecté à un processeur et nous avons alors un *arbre de processeurs*. Les processus de niveaux supérieurs sont des *maîtres* de ceux de niveaux inférieurs appelés *esclaves*. Les maîtres donnent du travail aux esclaves et maintiennent à jour des variables avec les résultats que les esclaves leur auront retournés.
   Pour éviter les ambiguïtés, les termes de *fils* et *père* seront employés pour les nœuds de l'arbre de jeux et les termes *maître* et *esclave* pour les processus hiérarchisés.

2. la *configuration en pool de processus*. Il n'y a pas de relation hiérarchique entre les processus. Chaque processeur va chercher du travail dans une ou des files de priorité. Tous participent au maintien de la ou des files de priorité et à la mise à jour des variables.

Cependant, des mixages de ces deux types de configuration existent [15].

Aussi, une attention particulière doit être portée sur les mesures de performance des différents auteurs. Certains présentent des résultats obtenus uniquement avec des arbres de jeux simulés. Alors que d'autres ont des résultats obtenus en situation de jeu réelle avec notamment la gestion de la table de transposition qui pose des difficultés dues aux accès concurrents et qui peut diminuer les performances lors de l'implémentation des algorithmes.

2.1.1 Les algorithmes avec un arbre de processus

Découpage de l'arbre de jeux : Tree-Splitting

L'algorithme *Tree-Splitting* a été introduit par Fishburn et Finkel [9, 11]. Un arbre binaire de processeurs de hauteur 2 est superposé sur l'arbre de jeux (Figure 2.1). Ici, un processeur exécute un seul processus à la fois.

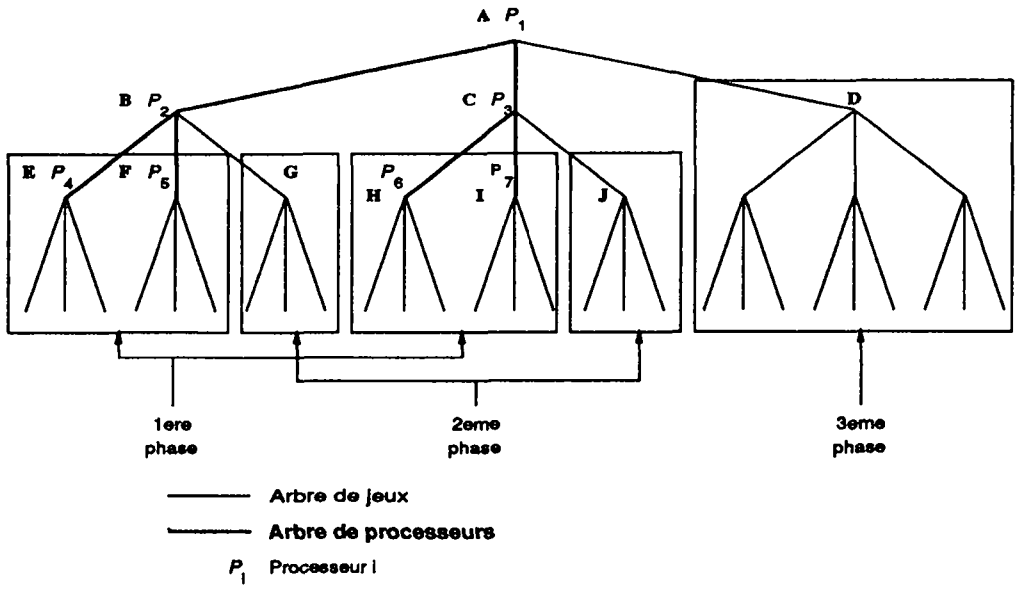

Figure 2.1 : Parcours issu de l'algorithme Tree-Splitting.

Cet algorithme est fondé sur $\alpha - \beta$. Le travail d'un processeur maître est d'engendrer les fils de son nœud et les faire explorer par ses esclaves. S'il y a plus de fils que d'esclaves, alors les fils sont mis dans une file en attendant qu'un des esclaves devienne inactif. A l'exception du processeur de la racine (P_1 de la Figure 2.1), tous les processeurs non-terminaux de l'arbre de processeurs sont à la fois maître et esclaves (P_2 et P_3 de la Figure 2.1), les processeurs terminaux sont uniquement des esclaves (P_4, P_5, P_6 et P_7 de la Figure 2.1). Les processeurs terminaux exécutent l'algorithme $\alpha - \beta$ séquentiel sur les sous-arbres des nœuds qui leur sont confiés. Une fois la meilleure valeur Negamax du sous-arbre trouvée, les processeurs terminaux remontent cette valeur aux processeurs pères. Cette valeur pourrait servir éventuellement à resserrer les fenêtres de recherche des autres processeurs esclaves et ainsi provoquer des élagages.

Prenons la Figure 2.1. Le processeur P_2 donne deux des trois nœuds à explorer à ses deux esclaves et met le troisième en attente (1ère phase).

Lorsque P_4 ou P_5 (respectivement pour P_6 et P_7) a fini son exploration, la valeur de son sous-arbre peut servir à rétrécir la fenêtre de recherche pour le nœud G, et respectivement J (2ème phase). Remarquer que le nœud D n'est pas examiné tant que les parcours issus de B et C ne sont pas finis (3ème phase).

Une étude théorique a montré que cet algorithme n'aura qu'une accélération en $O(k^{\frac{1}{2}})$ pour les arbres de jeux ordonnés meilleur d'abord, où k est le nombre de processeurs; et une accélération égale à k pour les arbres de jeux ordonnés dans le pire des cas. Cet algorithme a été implémenté sur un réseau de machines de l'Université de Wisconsin-Madison et les résultats mesurés avec des arbres de jeux simulés confirment les prédictions. Une accélération de 5,31 est mesurée pour un arbre de 27 processeurs avec trois esclaves par maître.

Deux principaux défauts expliquent ces résultats relativement moyens de cet algorithme :

- le mauvais équilibrage des charges entre les processeurs terminaux, surtout quand l'arbre a une grande profondeur,
- la sous-utilisation des processeurs maîtres qui attendent les résultats de leurs esclaves.

Une amélioration serait de permettre aux processeurs maîtres de participer aux explorations des nœuds.

Découpage du chemin principal : Principal Variation Splitting (PVS)

C'est de loin l'algorithme [21, 22] le plus implémenté dans les jeux d'échecs actuellement [15]. L'idée de base est l'algorithme séquentiel PVSearch où on recherche tout d'abord la valeur du chemin principal (Figure 2.2, première phase), et ceci dans le but d'obtenir une borne pour resserrer la fenêtre de recherche $[\alpha, \beta]$. Le chemin principal est celui prédit par l'algorithme comme étant le meilleur pour les deux adversaires et a priori celui formé par tous les nœuds les plus à gauche de l'arbre de jeux, car on est sous l'hypothèse que les arbres de jeux sont fortement ordonnés. Puis dans une deuxième phase, on applique l'algorithme séquentiel PVSearch aux autres fils pour essayer d'abord de les réfuter en parallèle. Et on recommence les mêmes opérations aux niveaux supérieurs de l'arbre de jeux (3ème et 4ème phase).

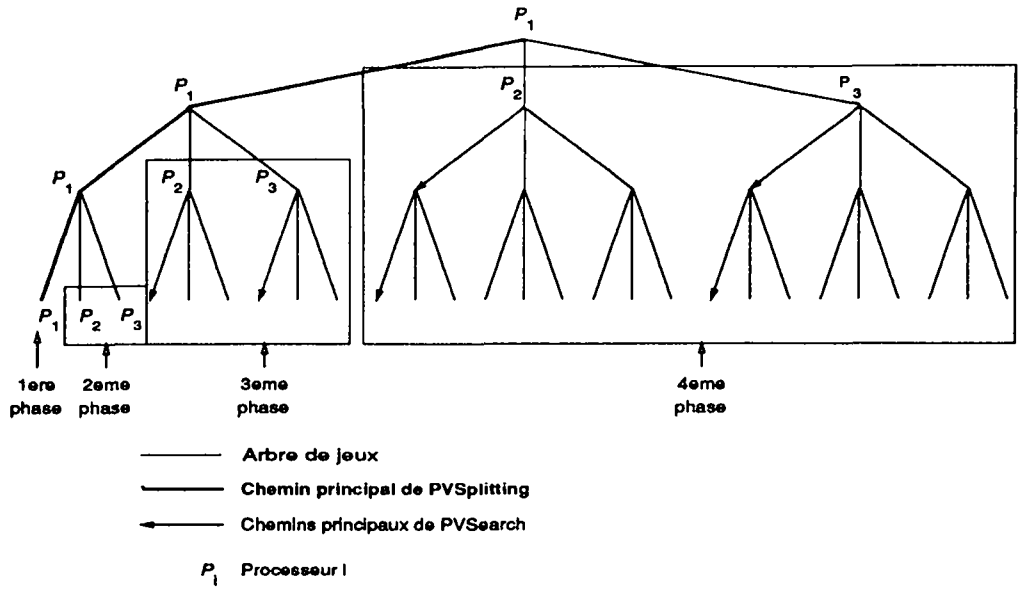

Figure 2.2 : Parcours issu de l'algorithme PVS.

Cet algorithme a été implémenté sur un réseau de stations SUN configuré en arbre par Marsland et Popowich, 1985 [22]. Un processeur exécute un processus à la fois. L'accélération, avec les arbres de jeux réels du jeu d'échecs sans table de transposition, est de 3,06 sur un arbre de 4 processeurs. Avec tables de tranposition locales, l'accélération est de 3,10.

Les difficultés posées sont les suivantes :

- l'équilibrage des charges de travail entre processeurs, notamment dans les phases 2 et 3 de la Figure 2.2. En effet, certains sous-arbres peuvent être explorés plus rapidement que d'autres grâce aux élagages de l'algorithme séquentiel. Certains processeurs sont donc bloqués à un niveau de l'arbre de jeux et restent inactifs.
- le coût de communication est élevé avec un réseau distribué de processeurs.

Des corrections des défauts précédents ont été essayées par Hyatt et Suter [16], leur algorithme est appelé Enhanced Principal Variation Splitting (EPVS) :

- dès qu'un des processeurs devient inactif et qu'il en existe d'autres en train d'explorer des sous-arbres ayant moins d'élagages, on arrête alors

tous les processeurs et on descend de deux niveaux dans un des sous-arbres difficiles et ses nœuds sont de nouveau partagés entre tous les processeurs. Les processeurs restent donc toujours actifs. Le seul risque est que l'on réexamine certains nœuds.
Cette amélioration implique en fait un démantèlement momentané de la hiérarchie des processeurs.

- le problème de communication est résolu avec un ordinateur de type MIMD à mémoire partagée où le coût de communication est moindre.

Les résultats sur les arbres de jeux réels montrent des accélérations plus importantes par rapport à PVS. Avec 16 processeurs, EPVS atteint une accélération de 6 alors que celle de PVS est de 4,59. Une limite supérieure semble cependant exister pour ce type d'algorithme.

2.1.2 Les algorithmes avec un pool de processus

Les algorithmes avec un pool de processus sont implémentés généralement sur les machines de type MIMD à mémoire partagée.

La recherche probabiliste : Aspiration Search

La parallélisation de la recherche probabiliste (Aspiration Search), proposée par Baudet [4], consiste à découper la fenêtre de recherche pleine $]-\infty, +\infty[$ en sous-fenêtres de recherche :

$$\alpha = a_0 = -\infty < a_1 < \ldots < a_{n-1} < \beta = a_n = +\infty.$$

Puis d'appliquer l'algorithme $\alpha - \beta$ à la racine s de l'arbre de jeux sur chaque processeur avec comme fenêtre de recherche une des sous-fenêtres $[a_i, a_{i+1}]$ et de trouver la sous-fenêtre i tel que :

$$0 \leq i < n, a_i < \mathit{Alphabeta}(s, a_i, a_{i+1}) = \mathit{Alphabeta}(s, -\infty, +\infty) < a_{i+1}.$$

L'implémentation de cet algorithme est donc très simple. Il n'y a pas d'échanges d'informations entre les processeurs. Pour les arbres de jeux ordonnés dans le pire des cas, l'accélération est égale à k pour k processeurs, puisque dans ce cas l'algorithme $\alpha - \beta$ séquentiel doit parcourir tout l'arbre de jeux et aucun élagage est possible. L'accélération est bornée par 6 quel que soit le nombre de processeurs, pour les arbres ordonnés aléatoirement.

Surtout, pour les arbres ordonnés meilleur d'abord, il est clair que l'accélération serait minime. En effet, pour l'intervalle solution i, un seul processeur explorerait tous les nœuds examinés par l'algorithme séquentiel avec une fenêtre $[a_i, a_{i+1}]$, le peu de gain en temps par rapport à la version séquentielle mono-processeur serait dû à une fenêtre de recherche plus petite au départ.

Le travail obligatoire d'abord : Mandatory Work First (MWF)

Le principe de cet algorithme proposé par Akl et al. [2, 3] en 80, a suscité par la suite énormément de recherches. L'hypothèse de travail ne prend pas en compte les élagages en profondeur de l'algorithme $\alpha - \beta$ classique, mais uniquement les élagages de surface comme la version affaiblie de $\alpha - \beta$.

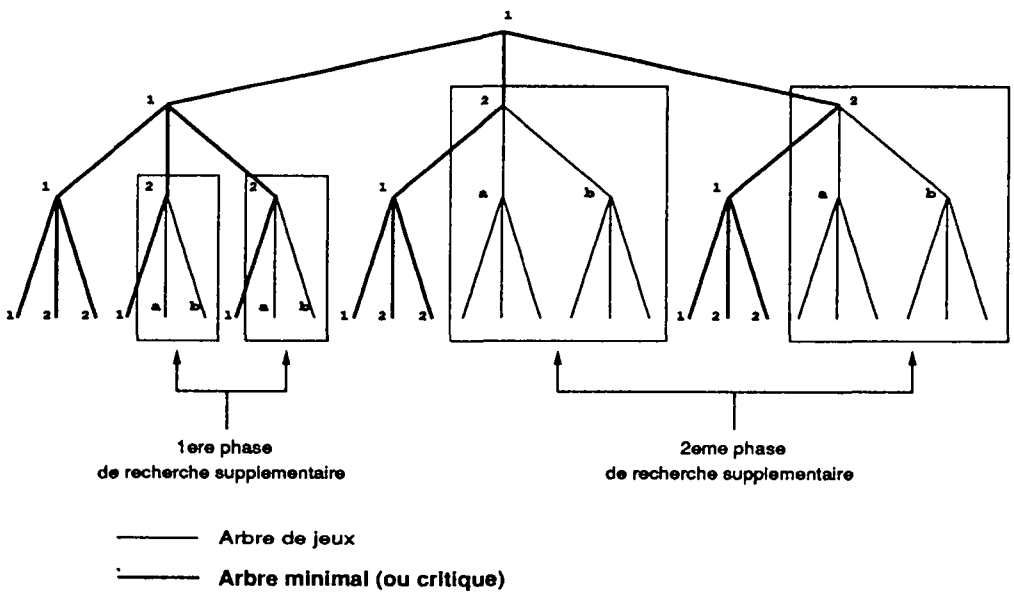

Figure 2.3 : Parcours issu de l'algorithme MWF.

L'idée est inspirée de l'arbre minimal (ou critique). Puisque l'algorithme $\alpha - \beta$ séquentiel parcourt bien les arbres de jeux fortement ordonnés et qu'il examine toujours l'arbre minimal, un programme parallèle doit explorer au moins les nœuds critiques. Ci-dessous, nous présentons les principales étapes de l'algorithme (Figure 2.3) :

Étape 1. L'arbre de jeux est exploré :

1. en parcourant récursivement tous les fils (type 1 et 2) du premier fils (type 1) à gauche de la racine (type 1),
2. en parcourant uniquement les premiers fils gauches (type 1) des fils droits (type 2) de la racine.

Cette étape effectue le parcours de l'arbre minimal et elle permet de donner une valeur définitive à tous les fils gauches (type 1) de l'arbre; et une valeur provisoire, la même que leur frère de gauche, au fils droits (type 2).

Étape 2. Si les valeurs temporaires ne permettent pas d'élaguer les autres fils droits qui ne font pas partie de l'arbre minimal (ni de type 1, ni de type 2), alors on les explore séquentiellement pour essayer d'élaguer les autres fils de droite par une meilleure valeur trouvée. Dans le cas de la Figure 2.3, les nœuds **a** sont examinés avant les nœuds **b**, mais les nœuds **a** (respectivement **b**) peuvent être examinés en parallèle.

A l'implémentation, chaque nœud exploré de l'arbre est associé à un processus qui est mis dans une file de priorité. La priorité est donnée aux nœuds les plus à gauche et en cas d'égalité celui qui a la plus grande profondeur dans l'arbre de jeux. L'algorithme se termine quand la file de priorité est vide.

Les performances mesurées sur une machine parallèle simulée, et avec des arbres de jeux simulés, montrent une accélération limitée à 6, quelque soit le nombre de processeurs. Cela est dû, d'une part, à la famine créée par les conflits d'accès à la file de priorité. D'autre part, l'exploration en parallèle des nœuds **a** (respectivement **b**) sans avoir les informations nécessaires pour les élagages peuvent provoquer des examens supplémentaires de nœuds par rapport à l'algorithme séquentiel. Une difficulté supplémentaire serait la taille des arbres de jeux qui impose une mémoire importante pour stocker la file de priorité.

Une analyse faite par Fishburn et Finkel, 1983 [10] sur cet algorithme avec un arbre binaire de processeurs prédit une accélération comprise entre $k^{0,78}$ et $k^{0,82}$ où k est le nombre de processeurs, pour les arbres de jeux de degré 38 ordonnés meilleur d'abord; et une accélération comprise entre $k^{0,93}$ et $k^{0,96}$ pour les arbres de degré 38 ordonnés dans le pire des cas.

L'algorithme proposé par Lindström, 1985 [19], nommé KEY NODE METHOD, est plus ou moins le même que celui d'Akl et al. Cependant, certains auteurs [15] jugent que les accélérations de 10 à 13 avec 20 processeurs sont

obtenues avec des arbres de jeux simulés trop irréalistes. Mais les résultats n'ont pu être analysés.

SSS^* parallèle

Hiromoto et al. [14] énoncent leur formulation du problème comme un problème de liste OUVERT identique à celle de SSS^*, excepté qu'au lieu d'extraire un seul nœud de la liste OUVERT, ils proposent d'en extraire plusieurs à la fois.

Des accélérations de l'ordre de 45 pour 128 processeurs sont mesurées sur des arbres de jeux simulés avec diverses versions de son algorithme (une version simulant $\alpha - \beta$, une autre SSS^* et une dernière SSS^* dual). Bien que les arbres simulés soient peu réalistes : degré 4 et profondeur 10, alors que les arbres de jeux réels sont de degré 38 et de profondeur 10; on peut retenir principalement les résultats suivants :

- l'algorithme $\alpha - \beta$ parallèle est aussi efficace que SSS^* parallèle,
- un surcoût de recherche important dû à l'absence d'informations pour des élagages est provoqué par les explorations parallèles. Ce surcoût croît avec l'augmentation du nombre de processeurs,
- l'existence de famine au début des explorations, dû au nombre insuffisant de nœuds à explorer pour l'ensemble des processeurs,
- la longueur de la liste OUVERT est très grande. Pour un arbre de degré N et de profondeur P avec k processeurs ($k < N^P$), et une stratégie d'exploration en profondeur d'abord, la longueur de OUVERT est égale à :

$$N^L + k * (N - 1) * (P - L) \text{ où } L = \lceil \log_N k \rceil.$$

L'algorithme d'Évaluation-Réfutation (ER)

Proposé par Steinberg et Solomon, 1989 [27], cet algorithme parallèle n'est pas une parallélisation de l'algorithme $\alpha - \beta$, ni de SSS^*. Cependant le principe de l'algorithme séquentiel ER ressemble beaucoup à Palphabeta ou PVSearch : on évalue un nœud fils et on essaie de réfuter les autres. Une différence doit tout de même être signalée : au lieu de considérer d'office que le premier fils gauche est à évaluer comme dans Palphabeta ou PVSearch, l'algorithme ER va d'abord évaluer les premiers petits fils gauches pour décider le fils à évaluer. ER anticipe en quelque sorte sur un demi-coup puisqu'il

examine à deux niveaux plus bas au lieu d'un. Les élagages profonds ne sont pas pris en compte dans les hypothèses de travail.

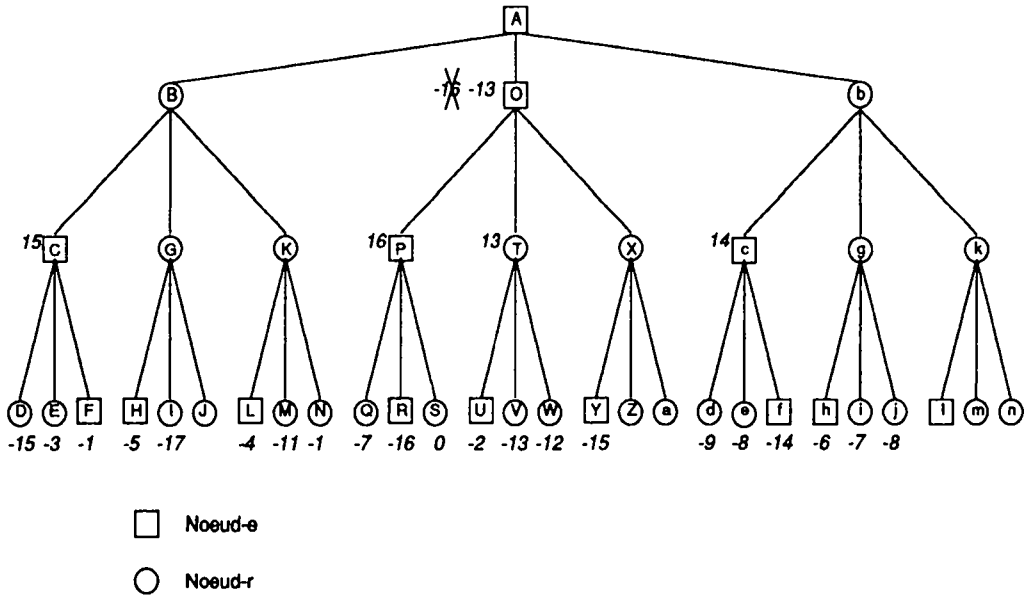

Figure 2.4 : Arbre issu de l'algorithme ER.

Prenons comme exemple l'arbre de la Figure 2.4. Les nœuds de l'arbre de jeux sont de deux types : nœuds-e (pour évaluation) et nœuds-r (pour réfutation). La racine A de l'arbre est un nœud-e, pour choisir un nœud-e fils de A, les nœuds C,P et c sont évalués. Quand le nœud O devient un nœud-e avec la valeur -16, on va essayer de réfuter les autres nœuds-r. Mais la réfutation échoue avec le nœud-r T qui vaut 13, la valeur de O est mise à jour avec -13. Et les nœuds-r B et b peuvent être réfutés. B est à gauche de b et il est réfuté en premier. Pour réfuter B, il faut réfuter G qui le sera après l'exploration du nœud I. Le nœud-r K a la valeur 11 et n'est pas réfuté. La valeur de B est alors mise à -11 and B est réfuté. Pour réfuter le nœud-r b, on doit réfuter g. Mais la valeur de g a la valeur 8 après l'exploration de j et g n'est pas réfuté. Cependant la valeur -8 acquise par b est suffisante pour le réfuter.

L'algorithme parallèle est présenté comme un problème de files de priorité. Ici, on utilise deux files de priorité : l'une appelée primaire et l'autre spéculative. La file primaire est ordonnée par profondeur décroissante. Tandis que la file spéculative est ordonnée par nombre croissant de nœuds-e fils et, en cas d'égalité de nombre de nœuds-e fils, elle est ordonnée par profondeur croissante. La file primaire a aussi la priorité de traitement sur la file spéculative. Un nœud est extrait de la file spéculative uniquement si la file primaire est vide. La file spéculative sert surtout à éviter la famine en début

de recherche.

L'accélération est de 11 pour 16 processeurs aussi bien avec les arbres simulés qu'avec les arbres du jeux d'Othello. La dégradation des performances est due aux conflits d'accès à la structure partagée des arbres de jeux. Une réduction des profondeurs des arbres réduirait cette dégradation, mais augmenterait en contrepartie la famine. Aussi les explorations en parallèle sans informations pour effectuer des élagages contribuent à un surcoût de recherche de l'ordre de 50%.

Il semblerait que l'algorithme ER séquentiel soit meilleur que $\alpha - \beta$ sur certains arbres de jeux d'après les résultats rapportés. Une étude théorique plus poussée doit cependant être faite, car les mesures réalisées sont trop insuffisantes (3 arbres de jeux simulés et 3 arbres de jeux d'Othello).

2.1.3 Les algorithmes hybrides

L'hybridité concerne aussi bien les algorithmes que la configuration des processus.

Récemment, Hsu [15] a implémenté un algorithme appelé *Delayed Branching Tree Expansion* (DBTE). Il consiste à employer un machine hôte (mono ou multi-processeurs) qui explore le partie d'arbre au dessus de l'*horizon de parallélisation* (un certain niveau de l'arbre de jeux qui sert à partager l'arbre en deux parties : haute et basse) définie à l'avance. Cette exploration se fait en suivant les règles du travail obligatoire d'abord (MWF). Tous les nœuds engendrés jusqu'au niveau de parallélisation sont mis dans deux files de priorité auxquelles des processeurs de tâches viennent chercher leurs nœuds afin d'explorer en parallèle les niveaux en dessous de l'horizon de parallélisation. Pour préserver les élagages, les fils issus d'un même nœud qui se situe en dessous de l'horizon sont explorés séquentiellement par l'algorithme $\alpha - \beta$ classique. Mais deux fils issus de deux nœuds différents peuvent être explorés en parallèle avec l'algorithme $\alpha - \beta$ classique.

En fait, cela revient à utiliser $\alpha - \beta$ classique en dessous de l'horizon de parallélisation et à employer la version affaiblie au dessus. La machine hôte équivaut au maître, et les processeurs de tâches aux esclaves.

Des accélérations de l'ordre de 43 pour 100 processeurs simulés ont été obtenues avec des arbres générés, mais des accélérations de l'ordre de 60 pour 100 processeurs simulés sont attendues avec des arbres de jeu d'échecs réels.

Il semblerait, d'après les résultats, que le problème de famine est presque inexistant ici. Puisque dans la configuration de 1000 processeurs simulés,

les processeurs restent actifs à près de 90% sur des arbres de jeux d'échecs. La dégradation des accélérations est due au surcoût de recherche qui atteint 250% dans cette même configuration.

2.2 Récapitulatif et conclusions des travaux antérieurs

Les Tableaux 2.1 et 2.2 font un bref récapitulatif de tous les algorithmes parallèles de parcours d'arbres de jeux.

Nom de l'algorithme parallele	Tree–Splitting	Principal variation Splitting (PVS)	Enhanced PVS (EPVS)	Delayed Branching Tree Expansion (DBTE)
Auteur(s)	Fishburn et Finkel 1980	Marsland 1982	Hyatt et Suter 1989	Hsu 1990
Nom de l'algorithme sequentiel parallelise	alpha–beta classique	Principal Variation Search	Principal Variation Search	alpha–beta affaibli et classique
Type d'architecture de machine	Reseau experimental	Reseau de stations	Sequent Balance 21000 MIMD a memoire partagee	Multi–processeurs a memoire partagee (machine dediee)
Configuration des processeurs	Arbre de processeurs	Arbre de processeurs	Arbre de processeurs	Hybride
Avantages de la methode	Chaque processeur terminal s'occupe d'un sous–arbre Elagages profonds	Elagages profonds Tres implemente	Par rapport a PVS la famine est reduite grace a un equi–librage des charges mieux repart.	Peu de probleme de famine
Inconvenients de la methode	Famine Surcout de recherche Communication inter–processus Equilibrage des charges	Famine Equilibrage des charges Profondeur de l'arbre de jeux plus grande que la profndeur de l'arbre de processeurs Surcout de recherche		Surcout important
Acceleration (k cpu) arbre ordonne –meilleur d'abord	Avec le modele Palphabeta $O(k^{1/2})$ 4,56 avec 6 processeurs			
–dans le pire des cas	Avec le modele Palpha–beta en O(k)			
–autres		limitee a 8 3,04 pour 4 processeurs 4,59 pour 16 processeurs	6 pour 16 processeurs (arbres d'echecs)	60 pour 100 processeurs (fort. ordonnes) 350 pour 1000 proc. (arbres d'echecs)

Tableau 2.1 : Récapitulatif des algorithmes parallèles avec arbre de processus.

Diverses études ont donc été menées sur la parallélisation de l'algorithme $\alpha - \beta$ séquentiel et autres. Parmi elles, nous pouvons principalement distinguer trois méthodes de parallélisation :

1. la *parallélisation verticale* [4, 11, 21] qui associe un sous-arbre de l'arbre parcouru à un processus; ce processus a la charge d'explorer tout le sous-arbre qui lui est attribué; cette méthode est implémentée avec une *configuration hiérarchique* des processus;

Nom de l'algorithme parallele	Parallel aspiration search	Mandatory Work First (MWF)	SSS* parallele	ER	Mandatory Work First (MWF) (etude theorique)
Auteur(s)	Baudet 1978	Akl et al. 1980	Hiromoto et al. 1987	Steinberg et Solomon	Finkel et al. 1983
Nom de l'algorithme sequentiel parallelise	Aspiration search Alpha–beta	alpha–beta affaibli	SSS*	ER	alpha–beta affaibli
Type d'architecture de machine		MIMD a memoire partagee	MIMD a memoire partagee	MIMD a memoire partagee	
Configuration des processeurs	pool de processeurs	pool de processeurs	pool de processeurs	pool de processeurs	Arbre de processeurs
Avantages de la methode	Facile a implementer Elagages profonds		Elagages profonds	Mieux adaptee a la parallelisation	
Inconvenients de la methode	Famine et equilibrage des charges Pas d'acceleration sur les arbres ordonnes meilleur d'abord Surcout de recherche important	Pas d'elagages profonds Taille memoire importante Surcout de recherche important Conflit d'acces	Famine en debut d'exploration Taille memoire importante Surcout de recherche important	Pas d'elagages profonds Surcout de recherche important	
Acceleration (k cpu) arbre ordonne –meilleur d'abord	Pas ou peu d'acceleration				arbre binaire degre 38 $[k^{0,78}\ k^{0,82}]$
–dans le pire des cas	limitee a 5–6 meme pour les arbres ordonnes aleatoirement				arbre binaire degre 38 $[k^{0,93}\ k^{0,96}]$
–autres		limitee a 6	45 pour 128 processeurs	11 pour 16 processeurs	

Tableau 2.2 : Récapitulatif des algorithmes parallèles avec pool de processus.

2. la *parallélisation horizontale* [2, 14, 19, 27] qui consiste à associer non pas un sous-arbre, mais un nœud de l'arbre parcouru à un processus; ce type de parallélisation est implémenté avec une *configuration en pool* des processus;

3. la *parallélisation hybride* [15] qui combine les deux types de parallélisation précédents en découpant, à un niveau fixé d'avance, l'arbre en deux parties : haute et basse. Le haut de l'arbre est exploré par la méthode horizontale et le bas de l'arbre par la méthode verticale.

Les résultats expérimentaux récents [14, 15, 27] mettent en évidence les points suivants :

1. si des accélérations presque linéaires sont atteintes pour une dizaine de processeurs, la tendance actuelle consiste à s'intéresser au parallélisme massif asynchrone,

2. les techniques qui utilisent un pool de processus semblent très adaptées aux machines multi-processeurs à mémoire partagée,

3. éviter tout problème de famine sans provoquer un surcoût de recherche trop important est un problème clé.

D'autres résultats théoriques [5, 17] confirment le premier point dans le sens qu'une accélération linéaire est toujours possible si $P \leq h + 1$ où P est le nombre de processeurs utilisés et h la hauteur de l'arbre exploré.

Chapitre 3

Un algorithme $\alpha - \beta$ parallèle utilisant l'arbre critique

Nous allons donner ici notre démarche théorique menée pour la parallélisation de l'algorithme $\alpha - \beta$. Nous appellerons notre algorithme *Concurrent Alpha-Beta Pruning* (CABP).

Akl et al. [2, 3] ont expérimenté les premiers un algorithme parallèle fondé sur la notion d'arbres critiques. L'idée est de partager l'exploration de l'arborescence critique entre un pool de processus parallèles. Cependant leur algorithme ne parallélise que la version affaiblie de l'algorithme $\alpha - \beta$ [18] et nous savons que cette version examine a priori plus de nœuds par comparaison à la version classique.

Afin d'étayer nos choix quant à un algorithme à paralléliser, nous ferons dans les paragraphes suivants une étude approfondie des arbres critiques des deux versions de $\alpha - \beta$, classique et affaiblie.

3.1 Notion et description mathématique des arbres critiques

Dans tout parcours d'arbres, il existe un arbre critique dont l'exploration est nécessaire pour obtenir la solution du problème. Dans les algorithmes Branch and Bound, les arbres critiques dépendent des instances de données.

Lorsque la fenêtre de recherche (cf. définition 7) $[\alpha, \beta]$ est égale à $]-\infty, +\infty[$ (ou $-\infty$ pour la borne de la version affaiblie de $\alpha - \beta$), nous

pouvons connaître exactement la taille des arborescences critiques parcourues par les deux versions de l'algorithme $\alpha - \beta$, pour les arbres de degré N et de profondeur P.

Par la suite, nous allons analyser la taille de ces arborescences critiques en termes de nœuds critiques terminaux et du total de nœuds critiques explorés. Pour cela, nous utiliserons uniquement les arbres de degré N et de profondeur P.

3.1.1 Arbre critique de la version affaiblie

La version affaiblie de l'algorithme $\alpha - \beta$ n'effectue pas les élagages profonds (cf. Figure 1.4).

Les nœuds critiques de cette version sont de type 1 et 2. Nous rappelons ici leurs règles de détermination :

- la racine de l'arbre de jeux est un nœud de type 1;
- le premier fils d'un nœud de type 1 est un nœud de type 1, les autres fils sont des nœuds de type 2;
- le premier fils d'un nœud de type 2 est un nœud type 1.

A partir des règles précédentes, Akl [1] en a déduit les propriétés suivantes :

Propriété 6

Le *nombre de nœuds critiques terminaux de type 1 et 2* à la profondeur p $(0 \leq p \leq P)$ d'un arbre de degré N pour la version affaiblie de l'algorithme $\alpha - \beta$ est :

$$T_1{}^p = \frac{(1+(1+4.(N-1))^{\frac{1}{2}})^{p+1} - (1-(1+4.(N-1))^{\frac{1}{2}})^{p+1}}{2^{p+1}.(1+4.(N-1))^{\frac{1}{2}}},$$

$$T_2{}^p = (N-1).\frac{(1+(1+4.(N-1))^{\frac{1}{2}})^{p} - (1-(1+4.(N-1))^{\frac{1}{2}})^{p}}{2^{p}.(1+4.(N-1))^{\frac{1}{2}}}.$$

Où, $T_j{}^i$ représente le nombre de nœuds critiques terminaux de type j à la profondeur i de l'arbre avec $0 \leq i \leq P$.

Preuve :

Avec les règles de détermination des nœuds critiques, nous obtenons les formules de récurrence suivantes pour les nœuds de type 1 :

$$\begin{aligned} {T_1}^0 &= 1, \\ {T_1}^1 &= 1, \\ {T_1}^2 &= {T_1}^1 + {T_2}^1, \\ &\vdots \\ {T_1}^p &= {T_1}^{p-1} + {T_2}^{p-1}; \end{aligned}$$

et pour les nœuds critiques de type 2 :

$$\begin{aligned} {T_2}^0 &= 0, \\ {T_2}^1 &= (N-1).{T_1}^0, \\ &\vdots \\ {T_2}^p &= (N-1).{T_1}^{p-1}. \end{aligned}$$

Nous déduisons alors :

$$ {T_1}^p = {T_1}^{p-1} + (N-1).{T_1}^{p-2} \quad \text{avec } 2 \leq p. \qquad (3.1) $$

Soit (a^p) la suite géométrique satisfaisant l'équation 3.1. Les racines de l'équation $a^2 = a + (N-1)$ sont :

$$\begin{aligned} a_1 &= \frac{1 + (1 + 4.(N-1))^{\frac{1}{2}}}{2}, \\ a_2 &= \frac{1 - (1 + 4.(N-1))^{\frac{1}{2}}}{2}. \end{aligned}$$

Alors nous savons que la suite $(\lambda.{a_1}^p + \mu.{a_2}^p)$ satisfait aussi l'équation 3.1. En plus :

$$\begin{aligned} &\text{si } \; p = 0, \; {T_1}^0 = 1 = \lambda.{a_1}^0 + \mu.{a_2}^0 = \lambda + \mu, \\ &\text{si } \; p = 1, \; {T_1}^1 = 1 = \lambda.a_1 + \mu.a_2, \end{aligned}$$

d'où

$$\begin{aligned} \lambda &= \frac{1 + (1 + 4.(N-1))^{\frac{1}{2}}}{2.(1 + (1 + 4.(N-1))^{\frac{1}{2}})}, \\ \mu &= \frac{-1 + (1 + 4.(N-1))^{\frac{1}{2}}}{2.(1 + (1 + 4.(N-1))^{\frac{1}{2}})}. \end{aligned}$$

□.

Corollaire 4

Le *nombre total de nœuds critiques de type 1 et 2* à la profondeur p d'un arbre de degré N pour la version affaiblie de l'algorithme $\alpha - \beta$ est égal à :

$$S^p_{aff} = \sum_{i=0}^{p}(T_1^i + T_2^i) = \sum_{i=1}^{p+1} T_1^i.$$

Cette propriété est très utile pour connaître la taille mémoire nécessaire pour stocker les nœuds des arbres.

Exemple numérique 1

Prenons un exemple numérique pour des comparaisons futures avec la version classique de l'algorithme $\alpha - \beta$. Soit $p = 5$ et $N = 20$, nous avons alors :

$$\begin{array}{lll} T_1^0 = 1, & & T_2^0 = 0, \\ T_1^1 = 1, & & T_2^1 = 19, \\ T_1^2 = 20, & \text{et} & T_2^2 = 19, \\ T_1^3 = 39, & & T_2^3 = 380, \\ T_1^4 = 419, & & T_2^4 = 741, \\ T_1^5 = 1160, & & T_2^5 = 7961; \end{array}$$

soit

$$S^5_{aff} = 10760.$$

3.1.2 Arbre critique de la version classique

Nous allons faire dans ce paragraphe la même étude que précédemment pour la version classique de l'algorithme $\alpha - \beta$.

Les nœuds critiques pour la version classique de $\alpha - \beta$ sont de trois types : 1, 2 et 3. Les règles de leur détermination sont :

- La racine de l'arbre de jeux est un nœud de type 1;
- Le premier fils d'un nœud de type 1 est de type 1, les autres fils sont de type 2;
- Le premier fils d'un nœud de type 2 est de type 3;

- Tout fils d'un nœud de type 3 est de type 2.

Nous avons calculé le nombre de nœuds critiques de la version classique pour comparer avec la version affaiblie.

Propriété 7

Le *nombre de nœuds critiques terminaux de type 1, 2 et 3* à la profondeur p ($0 \leq p \leq P$) d'un arbre de degré N pour la version classique de l'algorithme $\alpha - \beta$ est :

$$T_1^p = 1, \tag{3.2}$$

$$T_2^p = \begin{cases} N^{\frac{p}{2}} - 1 & \text{si } p \text{ est paire,} \\ N^{\frac{p+1}{2}} - 1 & \text{si } p \text{ est impaire,} \end{cases} \tag{3.3}$$

$$T_3^p = \begin{cases} N^{\frac{p}{2}} - 1 & \text{si } p \text{ est paire,} \\ N^{\frac{p-1}{2}} - 1 & \text{si } p \text{ est impaire.} \end{cases} \tag{3.4}$$

Où, $T_j{}^i$ représente le nombre de nœuds critiques terminaux de type j à la profondeur i de l'arbre avec $0 \leq i \leq P$.

Preuve :

D'après les règles de détermination des nœuds critiques, nous avons les formules de récurrence suivantes pour les nœuds critiques de type 1 :

$$\begin{aligned} T_1{}^0 &= 1, \\ T_1{}^1 &= 1, \\ &\vdots \\ T_1{}^p &= 1; \end{aligned}$$

pour les nœuds critiques de type 2 :

$$\begin{aligned} T_2{}^0 &= 0, \\ T_2{}^1 &= (N-1).T_1{}^0 + N.T_3^0, \\ &\vdots \\ T_2{}^p &= (N-1).T_1{}^{p-1} + N.T_3^{p-1}; \end{aligned}$$

et pour ceux de type 3 :

$$\begin{aligned} T_3^0 &= 0, \\ T_3^1 &= T_2^0, \\ &\vdots \\ T_3^p &= T_2^{p-1}. \end{aligned}$$

La démonstration pour T_1^p est évidente. Tandis que celle pour T_3^p dépend de celle de T_2^p. Or nous avons :

$$\begin{aligned} T_2^p &= (N-1).T_1^{p-1} + N.T_3^{p-1} \\ \Leftrightarrow\ T_2^p &= (N-1) + N.T_2^{p-2} \quad \text{car} \quad T_3^p = T_2^{p-1}. \end{aligned}$$

Soit la suite (U_p) telle que $U_p = T_2^p + c$, alors U_p satisfait l'équation suivante :

$$\begin{aligned} U_p - c &= (N-1) + N.(U_{p-2} - c) \\ \Leftrightarrow \quad U_p &= (N-1) + N.U_p - N.c + c \\ \Leftrightarrow \quad U_p &= (N-1) + N.U_p + c.(1-N). \end{aligned}$$

La suite (U_p) est une suite géométrique si et seulement si :

$$U_p = N.U_{p-2} \quad \text{i.e.} \quad (c-1).(1-N) = 0 \Rightarrow c = 1 \text{ si } N \neq 1.$$

Le cas $N = 1$ étant facile à déterminer, nous n'effectuons pas ici le calcul. Ainsi pour $N \neq 1$, nous obtenons :

$$\begin{aligned} &\text{si } p \text{ est paire} \quad U_p = N^{\frac{p}{2}}.U_0 \\ &\qquad\qquad\qquad\quad U_p = N^{\frac{p}{2}}.(T_2^0 + 1) = N^{\frac{p}{2}}, \\ &\text{si } p \text{ est impaire} \quad U_p = N^{\frac{p-1}{2}}.U_1 \\ &\qquad\qquad\qquad\quad U_p = N^{\frac{p-1}{2}}.(T_2^1 + 1) = N^{\frac{p+1}{2}}; \end{aligned}$$

soient

$$\begin{aligned} &\text{si } p \text{ est paire} \quad T_2^p = N^{\frac{p}{2}} - 1, \\ &\text{si } p \text{ est impaire} \quad T_2^p = N^{\frac{p+1}{2}} - 1. \end{aligned}$$

Comme nous avons la relation $T_3^p = T_2^{p-1}$, la démonstration pour T_3^p se déduit immédiatement.

□.

Cette propriété nous permet de retrouver un corollaire connu [18].

Corollaire 5

Tout algorithme évaluant un arbre de degré N et de profondeur P, doit explorer au moins :

$$N^{\lceil P/2 \rceil} + N^{\lfloor P/2 \rfloor} - 1$$

nœuds terminaux.

Preuve :

Il suffit de sommer les trois types de nœuds critiques terminaux. Nous trouvons alors :

si P est paire, cette somme est $N^{\frac{P}{2}} + N^{\frac{P}{2}} - 1$,
si P est impaire, cette somme est $N^{\frac{P+1}{2}} + N^{\frac{P-1}{2}} - 1$,

d'où le résultat en regroupant les deux formules en une seule.
□.

Pour connaître la *taille mémoire nécessaire* dans le stockage des nœuds des arbres, nous avons calculé la somme de chaque type de nœuds critiques pour en déduire le total.

Corollaire 6

Les sommes respectives des nœuds critiques de type 1, 2 et 3 pour un arbre de degré N et de profondeur P sont :

$$\sum_{i=0}^{P} T_1^i = P + 1; \tag{3.5}$$

$$\sum_{i=0}^{P} T_2^i = \begin{cases} \frac{2.N.(N^{\frac{P}{2}}-1)}{N-1} - P & \text{si } P \text{ est paire,} \\ \frac{N^{\frac{P+1}{2}}.(N+1)-2.N}{N-1} - P & \text{si } P \text{ est impaire;} \end{cases} \tag{3.6}$$

$$\sum_{i=0}^{P} T_3^i = \begin{cases} \frac{N^{\frac{P}{2}}.(N+1)-N-1}{N-1} - P & \text{si } P \text{ est paire,} \\ \frac{2.N.(N^{\frac{P-1}{2}}-1)+N-1}{N-1} - P & \text{si } P \text{ est impaire;} \end{cases} \tag{3.7}$$

et

$$S_{cla}^P = \sum_{i=0}^{P} (T_1^i + T_2^i + T_3^i).$$

Preuve :

Avec les relations de récurrence vues pour la propriété précédente, l'équation 3.5 se déduit immédiatement.

Pour les équations 3.6 et 3.7, cela nécessite quelques calculs. Nous avons :

$$\begin{aligned}
& \sum_{i=0}^{P} T_2^i = \sum_{i=1}^{P} T_2^i \quad \text{car } T_2^0 = 0 \\
\Leftrightarrow \quad & \sum_{i=0}^{P} T_2^i = \sum_{i=1}^{P} ((N-1).T_1^{i-1} + N.T_3^{i-1}) \\
\Leftrightarrow \quad & \sum_{i=0}^{P} T_2^i = (N-1).P + N.\sum_{i=0}^{P-1} T_3^i \\
\Leftrightarrow \quad & \sum_{i=0}^{P} T_2^i = (N-1).P + N.\sum_{i=2}^{P-1} T_3^i \quad \text{car } T_3^0 = T_3^1 = 0 \\
\Leftrightarrow \quad & \sum_{i=0}^{P} T_2^i = (N-1).P + N.\sum_{i=2}^{P-1} T_2^{i-1} \quad \text{car } T_3^i = T_2^{i-1} \\
\Leftrightarrow \quad & \sum_{i=0}^{P} T_2^i = (N-1).P + N.\sum_{i=1}^{P-2} T_2^i.
\end{aligned}$$

Cette relation est du type :

$$U_P = (N-1).P + N.U_{P-2} \quad \text{avec } U_P = \sum_{i=1}^{P} T_2^i.$$

Elle s'écrit aussi :

$$U_P + P = N.(U_{P-2} + (P-2)) + 2.N. \tag{3.8}$$

Si nous posons la suite (V_P) telle que $V_P = U_P + P$ avec $V_0 = 0$ et $U_0 = 0$, l'équation 3.8 devient :

$$V_P = N.V_{P-2} + 2.N. \tag{3.9}$$

En posant de nouveau une suite (W_P) telle que $W_P = V_P + c$, la relation 3.9 s'écrit :

$$\begin{aligned}
W_P - c &= N.(W_{P-2} - c) + 2.N \\
\Leftrightarrow \quad W_P &= N.W_{P-2} + c.(1-N) + 2.N.
\end{aligned}$$

La suite (W_P) est une suite géométrique si et seulement si :

$$c.(1-N) + 2.N = 0 \quad \text{i.e. } c = \frac{2.N}{N-1} \text{ si } N \neq 1.$$

Pour $N \neq 1$, nous obtenons alors :

- si P est paire :

$$\begin{aligned} & W_P = N^{\frac{P}{2}}.W_0 \\ \Leftrightarrow \quad & W_P = N^{\frac{P}{2}}.(V_0 + \frac{2.N}{N-1}) \\ \Leftrightarrow \quad & W_P = \frac{2.N.N^{\frac{P}{2}}}{N-1} \quad \text{car } V_0 = 0, \end{aligned}$$

d'où

$$\begin{aligned} & V_P = W_P - \frac{2.N}{N-1} \\ \Leftrightarrow \quad & V_P = \frac{2.N.N^{\frac{P}{2}}}{N-1} - \frac{2.N}{N-1} \\ \Leftrightarrow \quad & V_P = \frac{2.N.(N^{\frac{P}{2}}-1)}{N-1}, \end{aligned}$$

et finalement

$$\begin{aligned} & U_P = V_P - P \\ \Leftrightarrow \quad & U_P = \frac{2.N.(N^{\frac{P}{2}}-1)}{N-1} - P; \end{aligned}$$

- si P est impaire :

$$\begin{aligned} & W_P = N^{\frac{P-1}{2}}.W_1 \\ \Leftrightarrow \quad & W_P = N^{\frac{P}{2}}.(V_1 + \frac{2.N}{N-1}) \\ \Leftrightarrow \quad & W_P = \frac{N^{\frac{P+1}{2}}.(N+1)}{N-1} \quad \text{car } V_1 = U_1 + 1 = N, \end{aligned}$$

d'où

$$\begin{aligned} & V_P = W_P - \frac{2.N}{N-1} \\ \Leftrightarrow \quad & V_P = \frac{N^{\frac{P+1}{2}}.(N+1)}{N-1} - \frac{2.N}{N-1} \\ \Leftrightarrow \quad & V_P = \frac{N^{\frac{P+1}{2}}.(N+1) - 2.N}{N-1}, \end{aligned}$$

et finalement

$$\begin{aligned} & U_P = V_P - P \\ \Leftrightarrow \quad & U_P = \frac{N^{\frac{P+1}{2}}.(N+1) - 2.N}{N-1} - P. \end{aligned}$$

Les résultats pour les nœuds de type 3 se déduisent facilement de ceux des nœuds de type 2.

□.

Exemple numérique 2

Reprenons l'exemple numérique de la version affaiblie avec $P = 5$ et $N = 20$. Nous trouvons cette fois :

$$\begin{array}{lll} T_1^0 = 1, & T_2^0 = 0, & T_3^0 = 0, \\ T_1^1 = 1, & T_2^1 = 19, & T_3^1 = 0, \\ T_1^2 = 1, & T_2^2 = 19, & T_3^2 = 19, \\ T_1^3 = 1, \quad \text{et} & T_2^3 = 399, \quad \text{et} & T_3^3 = 19, \\ T_1^4 = 1, & T_2^4 = 399, & T_3^4 = 399, \\ T_1^5 = 1, & T_2^5 = 7999, & T_3^5 = 399, \end{array}$$

soit

$$S_{cla}^5 = 9677.$$

3.1.3 Conclusions

Nous pouvons déjà constater que le total des nœuds critiques examinés par la version classique de $\alpha - \beta$ est plus petit que celui de la version affaiblie. Cette différence croît avec P et N. Il est donc plus intéressant de paralléliser la version classique.

Akl et al. [1] n'ont parallélisé que la version affaiblie pour une raison de structure de données dans la remontée des valeurs Negamax. Les tables de scores distribuées, proposées par ces auteurs, empêchent la réalisation des élagages profonds. Nous verrons dans le chapitre suivant, consacré aux structures de données, comment nous avons surmonté cette difficulté à l'aide notamment d'un arbre de score partagé.

En parallélisant la version classique de $\alpha - \beta$, nous pouvons espérer que pour les arbres ordonnés meilleur d'abord, notre algorithme parallèle explorerait uniquement les nœuds critiques. Désormais, lorque le terme *arbre critique* sera employé, il fera référence à l'arbre critique de la version classique.

Nous verrons dans le chapitre 4, quelles sont les structures de données que nous avons adoptées, pour partager les explorations des nœuds critiques et les autres (nœuds non-critiques) entre les processus parallèles afin de résoudre les problèmes tels que la famine et les conflits d'accès aux données.

3.2 Les spécificités des élagages de $\alpha - \beta$

L'algorithme $\alpha - \beta$ séquentiel effectue deux types d'élagages (de surface et en profondeur) afin d'éviter d'explorer entièrement l'arbre de jeux. La dominance de $\alpha - \beta$ sur l'algorithme Negamax réside justement dans ces coupures des arbres de jeux.

Dans les algorithmes de Branch and Bound, une borne de la fonction d'évaluation est calculée à chaque nœud et comparée à la meilleure valeur trouvée. Un nœud est élagué si la borne associée au nœud est inférieure ou supérieure (maximisation ou minimisation) à la meilleure valeur trouvée [26].

L'élagage d'un nœud par l'algorithme $\alpha - \beta$ n'est pas effectué suivant la borne d'une fonction d'évaluation, mais les valeurs des niveaux ancêtres du nœud en question. L'ensemble des valeurs des niveaux ancêtres d'un nœud forme les bornes α et β (cf. paragraphe 1.3.3 dans le chapitre 1). Un nœud s est élagué si $\alpha(s)$ est supérieure ou égale à $\beta(s)$. De plus, si un nœud est élagué, alors l'ensemble des nœuds frères non explorés issus d'un même nœud père peuvent eux aussi être élagués (cf. Figure 3.1). Remarquez que seuls les *nœuds non-critiques* peuvent être élagués, ces nœuds sont des nœuds frères de ceux de type 3. Cette propriété d'élagages aura son importance dans le choix des structures de données.

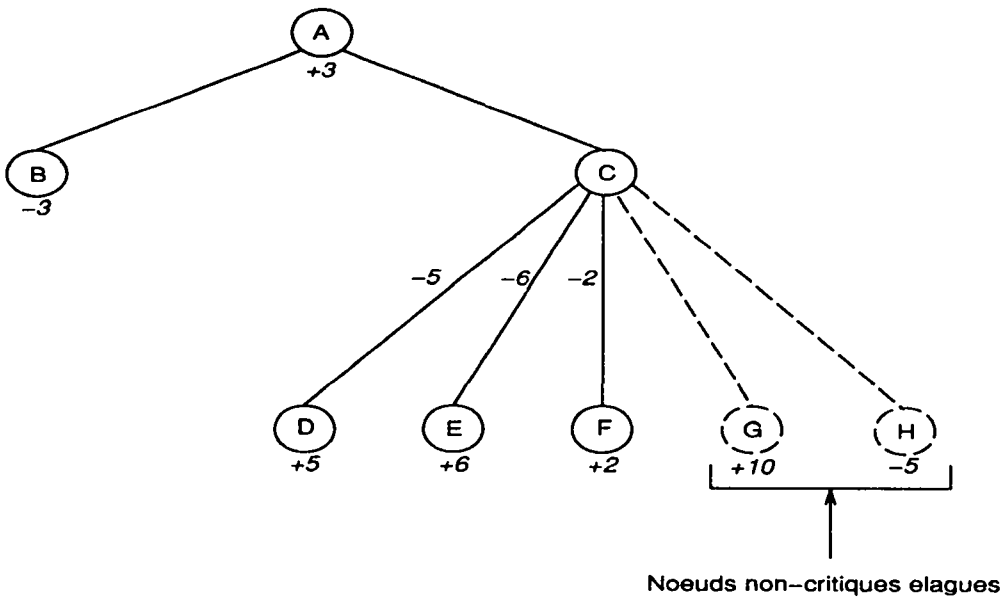

Figure 3.1 : Élagage des nœuds non-critiques.

Cependant, comme les versions parallèles des algorithmes Branch and Bound, certains élagages effectués par les algorithmes séquentiels ne le sont

plus en parallèles [26]. Il en est de même pour $\alpha - \beta$, par contre d'autres élagages impossibles en séquentiel sont rendus possibles grâce au parallélisme [1, 3].

3.2.1 Introduction d'un degré d'élagages parallèles : k

Considérons la Figure 3.2. L'algorithme $\alpha - \beta$ explorerait séquentiellement les nœuds de gauche à droite jusqu'au moment où le nœud F, qui permettrait d'élaguer les autres, aurait été atteint.

Définition 14

Nous définissons le *degré d'élagages parallèles* k, avec $1 \leq k$, comme étant le nombre de nœuds non-critiques issus d'un même nœud père, et examinés en parallèle par k processus.

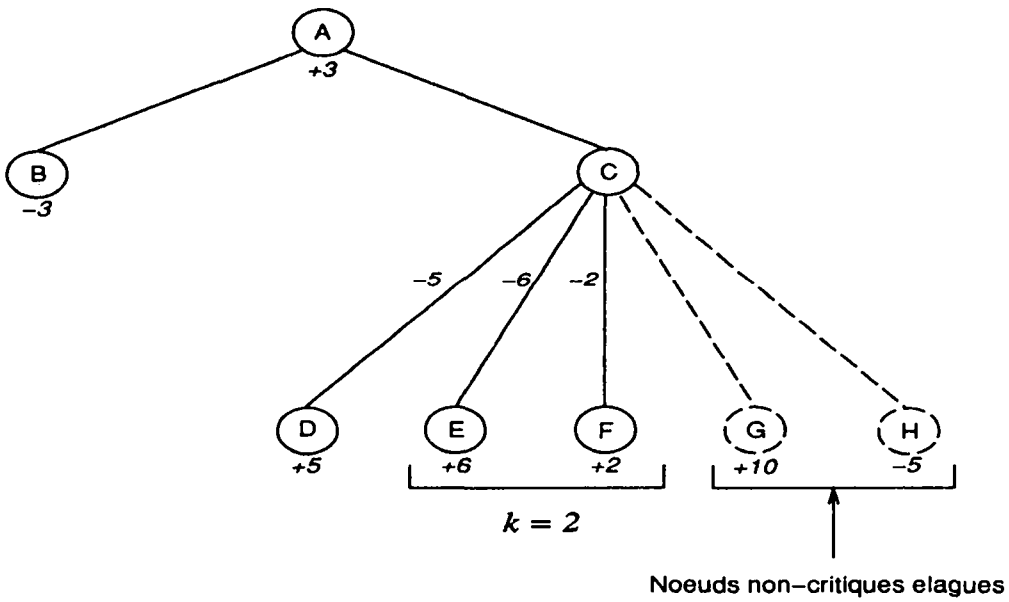

Figure 3.2 : Le degré d'élagages parallèles k.

Comme il importe d'arriver au plus tôt à l'exploration du nœud F, l'idée serait de faire *explorer en parallèle les nœuds non-critiques k par k* (k est le degré d'élagages parallèles). Cette façon d'explorer les nœuds convient

parfaitement aux arbres fortement ordonnés (cf. définition 4), puisqu'en définisant correctement k le k ième processus pourrait explorer dès la première itération le nœud qui permettrait l'élagage des autres. Nous gagnerions alors du temps sur l'exploration séquentielle qui aurait mis k itérations avant d'arriver au nœud en question.

Dans l'exemple de la Figure 3.2, en prenant $k = 2$, les élagages sont trouvés en une seule itération en explorant E et F en même temps. Alors qu'en séquentiel il aurait fallu explorer d'abord E puis F.

Cependant, cette méthode pose un inconvénient dans le cas où k est mal paramètré. Par exemple lorsque k est supérieur au nombre de nœuds qu'il faudrait explorer pour pouvoir élaguer les autres. Dans ce cas, des nœuds supplémentaires sont explorés par rapport à l'algorithme $\alpha - \beta$ séquentiel. Ceci sera un des surcoûts du parallélisme.

3.3 Description de l'algorithme parallèle utilisant l'arbre critique

L'idée de base de notre algorithme parallèle sur une machine parallèle à mémoire partagée est de séparer les nœuds critiques et non-critiques dans deux files d'attente, la première qualifiée de *file critique* et la seconde de *file non-critique*.

Les buts de cette séparation sont :

1. la résolution du problème des contentions d'accès à une file d'attente unique par plusieurs processus,
2. l'élagage facile des nœuds non-critiques de la seconde file.

L'algorithme parallèle que nous proposons est un algorithme à granularité fine. Chaque processus n'explore qu'un nœud de l'arbre à la fois.

Les bases fondamentales de notre algorithme parallèle sont les suivantes :

1. Tout l'arbre critique est engendré dans la file critique et exploré de gauche à droite. Les processus parallèles partagent l'exploration des nœuds de la file critique tant qu'ils ne l'ont pas épuisée.
   Les nouveaux nœuds critiques de type 1, 2 et 3 engendrés sont mis dans

la file critique, tandis que les nœuds non-critiques pas encore élagués et fils des nœuds critiques de type 2 sont insérés dans la file non-critique.

2. La file non-critique est explorée uniquement lorsque la file critique est épuisée et que les nœuds critiques de type 3 n'ont pas causé l'élagage de leurs nœuds frères non-critiques.
   L'examen de cette file se fait comme suit :

   - les nœuds non-critiques issus d'un même nœud père de type 2 sont explorés en parallèle k par k avec k processus ($1 \leq k$).
   - l'exploration de ces nœuds non-critiques s'effectue dans un ordre de priorité précis : les nœuds les plus à gauche dans l'arbre de jeux sont explorés les premiers.
   - si la condition d'élagage est satifaite pour l'un des nœuds non-critiques, tous les autres nœuds frères non-critiques seraient aussi élagués.

Un processus initial explore en séquentiel au début de l'algorithme la racine de l'arbre, et insère les premiers nœuds fils critiques dans la file critique afin de lancer les processus parallèles pour le reste de l'exploration.

L'algorithme de haut niveau des processus parallèles fondé sur le principe des algorithmes de Branch and Bound cités dans [26] est donné dans le paragraphe suivant.

3.4 L'algorithme *Concurrent Alpha-Beta Pruning* (CABP)

Variables partagées

```
File_C : File critique contenant des nœuds critiques de l'arbre de
         jeux, un nœud de la file contient un nœud critique de l'arbre
         de jeux.
NB_C : Nombre de nœuds dans File_C.
File_NC : File non-critique contenant des listes de nœuds non-critiques
          issus d'un même père. Un noeud de la file contient un pointeur
          sur une liste.
NB_NC : Nombre de nœuds dans File_NC.
Nb_pus : Nombre de tâches actives.
```

Score_arbre : Arbre de score contenant les valeurs negamax des nœuds pères. A chaque nœud de cet arbre, on a le nombre de fils du nœud restant à explorer pour avoir la valeur negamax définitive du nœud. Si le nombre de fils est nul, cette valeur peut servir à mettre à jour la valeur du père du nœud, puis celle du grand-père, ainsi de suite.

Verrou et événement

MUTEX : Verrou d'exclusion mutuelle sur Nb_pus.
MUTEX_C : Verrou d'exclusion mutuelle sur File_C et NB_C.
MUTEX_NC : Verrou d'exclusion mutuelle sur File_NC et NB_NC.

INSER : Événement envoyé aux tâches en attente lorsqu'une tâche insère des nœuds dans les files alors que celles-ci étaient vides, ou lorsque l'algorithme est fini et qu'il faut réveiller toutes les tâches en attente afin d'avoir une terminaison correcte.

PROCESSUS INITIAL :

liste_C : Liste de nœuds critiques issus d'un même père.
Racine : racine de l'arbre à explorer.

Début

```
ajout_SCORE_ARBRE(Racine);
/*
* Ajouter un nœud dans SCORE_ARBRE et le relier à son
* père dans SCORE_ARBRE, mettre le nombre de fils à zéro.
*/

prepare_liste_C(Racine, generer_fils(Racine), liste_C);
/*
* générer les fils de Nœud, les mettre dans une liste liste_C
* avec le premier fils de type 1 et les autres de type 2.
* mettre à jour le nombre de fils du nœud dans SCORE_ARBRE.
*/

ajouter_File_C(liste_C, NB_C);
/*
* ajouter la liste des nœuds fils non-critiques dans
* la file File_C.
* mettre à jour NB_C = NB_C + 1
*/
```

Fin

PROCESSUS PARALLÈLE :

```
liste_NC : Liste de nœuds non-critiques issus d'un même père.
liste_C : Liste de nœuds critiques issus d'un même père.
Nœud : Un nœud sélectionné dans File_C ou File_NC.
Fin : Fin de la tâche.
```

Début

```
FIN := FAUX;
TANT QUE (FIN = FAUX) FAIRE

     lockon(MUTEX_C);
     SI File_C est vide ALORS /* file critique est vide */
        lockoff(MUTEX_C); /* libère la file critique */

        lockon(MUTEX_NC);
        SI File_NC est vide ALORS /* file non-critique est vide */
           lockoff(MUTEX_NC); /* libère la file non-critique */

           lockon(MUTEX);
           SI Nb_pus = 0 ALORS
              /* il n'y a plus de tâches actives */

              Fin := VRAI;
              lockoff(MUTEX);
              envoyer_evenement(INSER);

           SINON /* il reste encore des tâches actives */

              lockoff(MUTEX);
              attendre_evenement(INSER);

           FSI

        SINON /* la File_NC est non vide */

           Selectionner_File_NC(Nœud);
           /* prendre le nœud le plus prioritaire dans
           * la file File_NC */
           NB_NC := NB_NC - 1;

           lockoff(MUTEX_NC);

           /* incrémenter le nombre de tâches actives */
           lockon(MUTEX);
           Nb_pus := Nb_pus + 1;
           lockoff(MUTEX);

           SI (elaguer(Nœud) = VRAI) ALORS

              elaguer_freres(Nœud, NB_NC);
```

```
   /* élaguer les frères de Nœud et
   * NB_NC = NB_NC - nombre de frères élagués.
   */

   remonter_SCORE_ARBRE(Nœud);
   /* Cette procédure remonte la valeur Negamax du père
   * de ce nœud dans SCORE_ARBRE pour mettre à jour les
   * valeurs Negamax des ancêtres de ce nœud.
   */

SINON /* la condition d'élagage n'est pas vérifiée */

   SI Nœud est terminal ALORS

      mettre_a_jour_SCORE_ARBRE(Nœud);
      /* Cette procédure évalue la valeur Negamax de
      * Nœud et met à jour la valeur Negamax
      * du père de ce nœud, puis décrémente
      * d'un le nombre de fils restant à explorer du père.
      * Si le nombre de fils du père est nul, on remonte
      * la valeur Negamax du père vers le grand-père,
      * et on décrémente d'un le nombre de fils du
      * grand-père, ainsi de suite.
      */

   SINON /* le Nœud n'est pas terminal */

      ajout_SCORE_ARBRE(Nœud);

      prepare_liste_NC(Nœud, generer_fils(Nœud), liste_NC);
      /* générer les fils de Nœud, les mettre dans
      * une liste liste_NC mettre à jour le nombre de
      * fils du nœud dans SCORE_ARBRE.
      */

      lockon(MUTEX_NC);

      ajouter_File_NC(liste_NC, NB_NC);
      /* ajouter la liste des nœuds fils non-critiques dans
      * la file File_NC.
      * mettre à jour NB_NC = NB_NC + nombre de nœuds
      * fils générés
      */

      lockoff(MUTEX_NC);

      SI File_C et File_NC sont vides ALORS
         envoyer_evenement(INSER);

   FSI
FSI
```

```
      /* mise à jour du nombre de tâches actives */
      lockon(MUTEX);
      NB_pus := NB_pus - 1;
      lockoff(MUTEX);

   FSI

SINON
/* File_C n'est pas vide, on va explorer ses nœuds en premier */

   premier_File_C(Nœud);
   /* prendre le premier nœud quelconque dans la file File_C */
   NB_C := NB_C - 1;

   lockoff(MUTEX_C);

   /* incrémenter le nombre de tâches actives */
   lockon(MUTEX);
   Nb_pus := Nb_pus + 1;
   lockoff(MUTEX);

   SI Nœud est terminal ALORS

      mettre_a_jour_SCORE_ARBRE(Nœud);

   SINON /* le Nœud n'est pas terminal */

      ajout_SCORE_ARBRE(Nœud);

      CAS type(Nœud) DE

          1 : /* nœud critique de type 1 */

              prepare_liste_C(Nœud, generer_fils(Nœud), liste_C);
              /* générer les fils de Nœud, les mettre dans
              * une liste liste_C avec le premier fils de type 1
              * et les autres de type 2.
              * Mettre à jour le nombre de fils du nœud dans
              * SCORE_ARBRE.
              */

              lockon(MUTEX_C);
              ajouter_File_C(liste_C, NB_C);
              lockoff(MUTEX_C);

              SI File_C et File_NC sont vides ALORS
                 envoyer_evenement(INSER);

          2 : /* nœud critique de type 2 */
```

```
                    prepare_liste_C(Nœud, generer_premier_fils(Nœud),
                                    liste_C);
                    /* générer le 1er fils de Nœud, le mettre dans
                    * une liste liste_C avec le premier fils de type 3,
                    * mettre à jour le nombre de fils du nœud dans
                    * SCORE_ARBRE.
                    */

                    prepare_liste_NC(Nœud, generer_autres_fils(Nœud),
                                     liste_NC);
                    /* les autres fils de Nœud sont de type 0 */

                    lockon(MUTEX_C);
                    ajouter_File_C(liste_C, NB_C);
                    lockoff(MUTEX_C);

                    lockon(MUTEX_NC);
                    ajouter_File_NC(liste_NC, NB_NC);
                    lockoff(MUTEX_NC);

                    SI File_C et File_NC sont vides ALORS
                       envoyer_evenement(INSER);

                autres : /* nœud critique de type 3 */

                    prepare_liste_C(Nœud, generer_fils(Nœud), liste_C);
                    /* les nœuds fils sont de type 2 */

                    lockon(MUTEX_C);
                    ajouter_File_C(liste_C, NB_C);
                    lockoff(MUTEX_C);

                    SI File_C et File_NC sont vides ALORS
                       envoyer_evenement(INSER);

              FCAS
           FSI

           /* mise à jour du nombre de tâches actives */
           lockon(MUTEX);
           NB_pus := NB_pus - 1;
           lockoff(MUTEX);

        FSI
     FTQ
```

Fin

Chapitre 4

Les implémentations de l'algorithme parallèle CABP

Nous allons discuter du choix des structures de données pour l'implémentation des deux files d'attente et de l'arbre de score.

L'accès à ces files peut être fait de façon exclusive ou concurrente. L'accès exclusif est généralement plus facile à gérer et il convient bien aux structures des listes chaînées. Cependant, lorsque le nombre de processus travaillant en parallèle croît, l'accès exclusif pose des problèmes de contention d'accès et l'accès concurrent s'impose dans ce cas.

Une première implémentation des files d'attente avec des listes à accès exclusif a été effectuée dans le but de vérifier la validité de l'algorithme parallèle. Aussi les résultats de cette implémentation nous ont permis d'effectuer un compromis dans le choix des structures de données.

Dans la seconde implémentation, nous avons opté pour la file non-critique : une structure de tas (skew-heap concurrent [20]) à accès concurrent; alors que nous avons gardé pour la file critique la structure de liste chaînée à accès exclusif. Le changement de structure de la file non-critique apporte une nette amélioration des résultats. Les résultats de cette implémentation seront analysés au chapitre 5.

4.1 Description des structures de données

Les données nécessaires à l'algorithme sont stockées essentiellement dans trois types de structures.

La première structure est appelée VABT, elle est la structure de base de l'arbre de score. A chaque nœud non terminal de l'arbre à explorer correspond une instance de cette structure. Elle permet de toujours garder la meilleure valeur Negamax des nœuds fils du nœud non terminal. Une fois la meilleure valeur Negamax des nœuds fils trouvée, sa valeur sera remontée vers une autre instance VABT mère. Nous y reviendrons plus en détail dans le paragraphe 4.2.

Voici sa définition :

VABT = article

LOCK	:	verrou d'exclusion mutuelle pour un accès concurrent à l'arbre de score.
VALEUR	:	meilleure valeur Negamax trouvée parmi les nœuds fils.
NB_FILS	:	indique le nombre de nœuds fils restant à explorer avant de remonter la valeur Negamax.
PÈRE_VABT	:	pointeur sur l'instance destinataire dans la remontée de la valeur Negamax; pointeur de type VABT.
P_LNC	:	pointeur sur un élément de type TETE_LNC de la liste non-critique contenant les nœuds non-critiques à explorer. Ce pointeur est nul si les nœuds fils sont critiques.
MAXMIN	:	indique si c'est un niveau MAX ou MIN pour la remontée de la valeur Negamax.

Fin de l'article VABT.

La deuxième structure est appelée NŒUD, elle contient toutes les informations sur un nœud de l'arbre à explorer, elle est l'élément de base des files d'attente :

NŒUD = article

TYPE	:	indique le type de nœud, 0 pour non-critique, 1, 2 et 3 pour les nœuds critiques;
CHEMIN	:	pointeur sur une liste d'octets indiquant le chemin dans l'arbre à explorer qui mène au nœud;
VALEUR	:	pointeur sur la valeur Negamax du nœud;
P_VABT	:	pointeur sur une instance de VABT contenant la meilleure valeur Negamax trouvée parmi les autres nœuds frères.
SUIVANT	:	pointeur sur un nœud suivant dans la file critique, et sur un nœud frère dans la file non-critique. Pointeur de type NŒUD.

Fin de l'article NŒUD.

La dernière structure TETE_LNC, avec la structure NŒUD, est l'élément constituant la file non-critique. Chaque instance de cette structure est une tête de liste de nœuds non-critiques issus d'un même nœud père dans

l'arbre à explorer. Elle est la même pour les deux structures (liste ou tas) et elle est définie comme suit :

TETE_LNC = article

LOCK	:	verrou d'exclusion mutuelle, surtout utile pour la seconde implémentation avec accès concurrent à la file non-critique.
L_NDNC	:	pointeur de type NŒUD sur une liste de nœuds non-critiques issus du même nœud père dans l'arbre à explorer.
NB_ND	:	nombre de nœuds non-critiques dans la liste pointée par **L_NDNC**.
k	:	degré d'élagages parallèles k.
CUT	:	booléen indiquant si les nœuds non-critiques issus d'un même nœud père sont élagués ou non.
PNC ou GNC	:	pointeur de type TETE_LNC. Dans l'implémentation avec une liste doublement chaînée, **PNC** pointe sur le prédécesseur. Dans l'implémentation avec un tas, **GNC** pointe sur le fils gauche.
SNC ou DNC	:	pointeur de type TETE_LNC. Dans l'implémentation avec une liste doublement chaînée, **SNC** pointe sur le successeur. Dans l'implémentation aves un tas, **DNC** pointe sur le fils droit.

Fin de l'article TETE_LNC.

4.2 L'arbre de score partagé

A chaque nœud non terminal de l'arbre à explorer correspond une instance de VABT. Cette instance est allouée dynamiquement en mémoire partagée et elle ne sera détruite que lorsque tous les nœuds fils du nœud associé à cette instance auront été explorés et/ou élagués. La valeur Negamax est remontée vers une autre instance VABT mère, et ainsi de suite jusqu'à la racine de l'arbre de score.

Une instance de VABT est associée à une instance de TETE_LNC contenant les nœuds fils non-critiques d'un même nœud père. La Figure 4.1 donne le synopsis des relation entre les trois structures sur l'arbre de la Figure 3.1.

Après l'exploration du nœud A, la racine de l'arbre de score est créée et elle contiendra la solution finale du parcours. L'exploration du nœud B modifie la valeur de la racine en prenant le maximum de $-\infty$ et $+3$; c'est la règle de remontée des valeurs Negamax. Puis l'exploration du nœud C

engendre la création d'un autre sommet de l'arbre de score avec la valeur $-\infty$; un nœud critique D de type 3 et quatre nœuds frères de D non-critiques sont également créés.

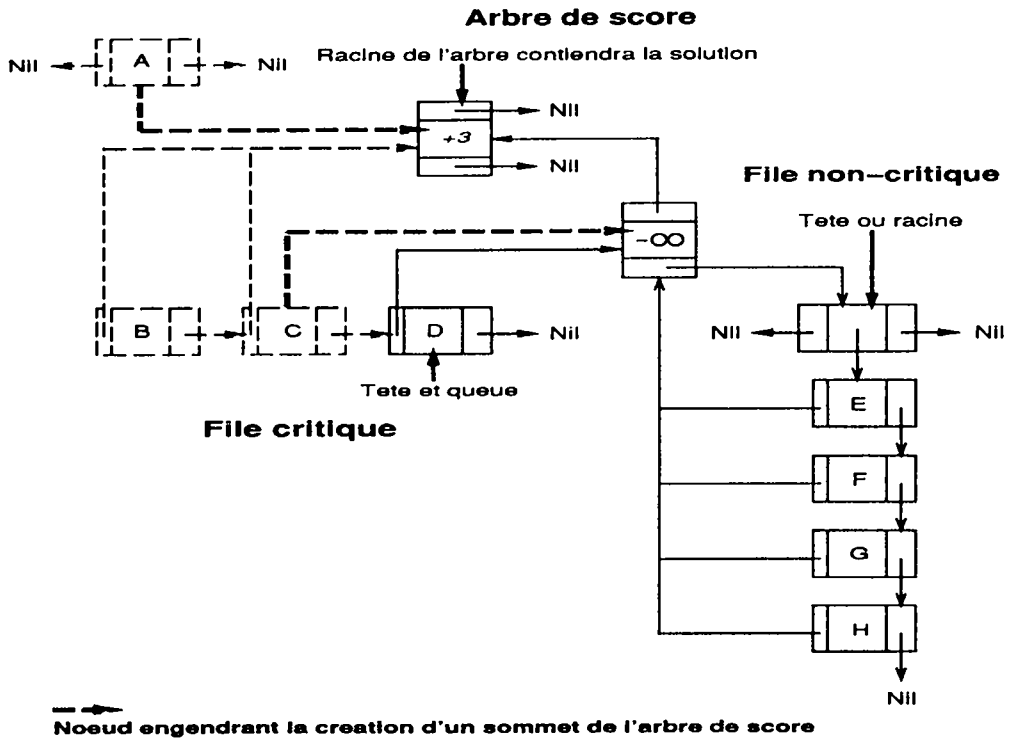

Figure 4.1 : Synopsis de l'arbre de score et les deux files d'attente.

L'accès à l'arbre de score se fait de façon concurrente à l'aide d'un verrou sur chaque instance de VABT. De plus, comme la remontée des valeurs Negamax se fait toujours du bas vers le haut de l'arbre, les verrous mortels ne se produisent pas.

Cet arbre de score autorise les élagages profonds qui auraient été impossibles à réaliser avec les tables de score distribuées. Alors qu'une table de score globale aurait donné des résultats erronés dans la remontée des valeurs Negamax [1].

Le seul inconvénient dans l'utilisation d'un tel arbre aurait été sa taille. En effet, on aurait pu avoir un arbre de score de N^{P-1} sommets pour un arbre à explorer de profondeur P et de degré N (i.e. N^P nœuds, car tous les nœuds non terminaux sont associés à un sommet VABT) ! Mais cela ne se produit pas grâce à une gestion dynamique de l'arbre de score. Les instances de VABT sont libérées dès que les nœuds fils associés sont explorés et/ou élagués.

4.3 La file critique

Cette file d'attente est implémentée sous forme de liste chaînée. L'accès est exclusif, mais il est en $O(1)$. Une implémentation en tas est possible avec un accès concurrent, mais l'accès aurait été en $O(\log n)$ où n est le nombre de nœuds existant dans la file.

Les nœuds critiques sont extraits de la tête de liste, et les nouveaux nœuds critiques engendrés sont mis en queue de liste. La Figure 4.2 donne un schéma de cette file.

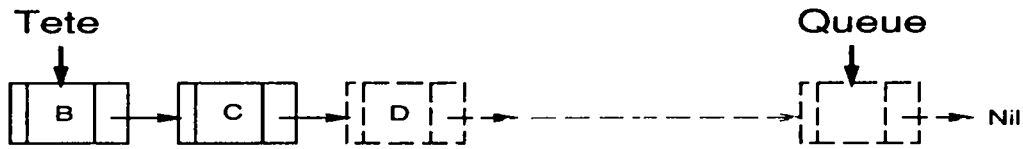

Figure 4.2 : La file critique.

4.4 La file non-critique

Pour la première implémentation, cette file est une double liste doublement chaînée. C'est une double liste car chaque élément TETE_LNC de la file contient aussi une liste de nœuds non-critiques issus d'un même nœud père. Elle est doublement chaînée, car chaque élément TETE_LNC de cette liste a un pointeur sur son prédécesseur et son successeur.

Les nœuds non-critiques sont extraits de la tête de liste k par k. Tandis que les nouveaux nœuds non-critiques sont insérés liste par liste suivant les valeurs des chemins de leurs nœuds pères. Les chemins les plus à gauche de l'arbre à explorer sont prioritaires. La Figure 4.3 schématise cette file.

L'accès est exclusif, l'extraction des nœuds se fait en $O(1)$. Mais malheureusement, l'insertion se fait en $O(n)$ où n est le nombre d'éléments de TETE_LNC dans la file. Ceci est dû à la priorité imposée sur les valeurs des chemins.

Pour la seconde implémentation, cette file est un tas à accès concurrent (skew-heap concurrent [20]). Les insertions et les extractions se font alors en $O(\log n)$, et plusieurs processus peuvent accéder à cette structure de manière concurrente. Les résultats obtenus sont nettement meilleurs. La Figure 4.4 donne une vue sur cette structure.

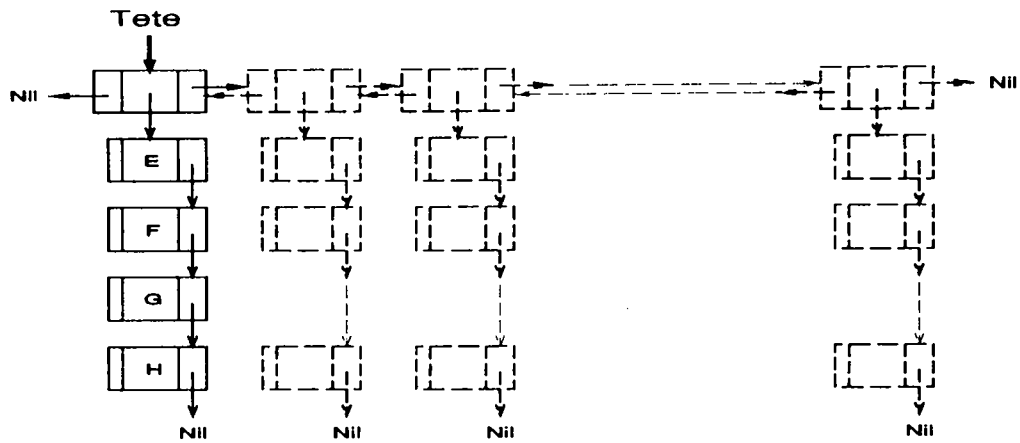

Figure 4.3 : La file non-critique en liste.

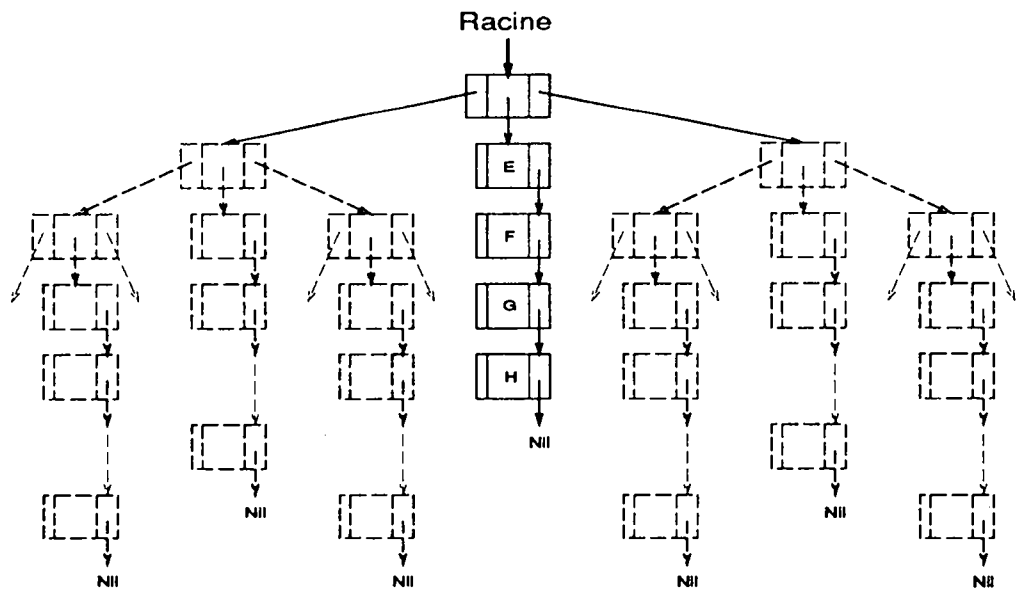

Figure 4.4 : La file non-critique en tas.

Chapitre 5

Quelques résultats expérimentaux

Les implémentations sont faites sur une machine MIMD à mémoire partagée : SEQUENT BALANCE 8000. Cette machine possède 10 processeurs parallèles dont un est réservé au système. Les processeurs fonctionnent en asynchrone.

La procédure de génération des arbres à explorer et les détails sur les diverses méthodes sont exposés dans l'annexe A. Nous avons choisi d'intégrer le générateur d'arbres dans nos programmes afin de pouvoir simuler des arbres de tailles importantes. Les arbres générés pour les simulations sont de trois types :

1. les arbres *parfaitement ordonnés* (cf. définition 4),
2. les arbres *fortement ordonnés*,
3. les arbres *aléatoirement ordonnés*.

Ces arbres présentent des caractéristiques intéressantes. D'abord ils sont communément utilisés dans la plupart des recherches [2, 9, 10, 21]. De plus, des accélérations linéaires sont rarement obtenues dans les expériences menées avec ces arbres. Les valeurs Negamax des nœuds terminaux sont prises dans l'intervalle $[0, 127]$ [21].

Nos arbres générés ont pour la plupart un degré de 20 et une profondeur de 5, mis à part ceux destinés à mesurer les influences du degré et de la profondeur sur notre algorithme dans la deuxième implémentation. Ces paramétres ont été choisis en tenant compte de la capacité de la machine

cible, des conditions réelles des jeux et des résultats théoriques évoqués dans le paragraphe 2.2.

En effet, notre machine a seulement 9 processeurs et si une profondeur de 10 a été choisie, il est facile alors d'obtenir des accélérations linéaires. Aussi, pour avoir des arbres de tailles suffisamment grandes et approchant des conditions réelles des jeux [1], un degré de 20 a été choisi.

Deux séries de résultats sont présentées dans les paragraphes suivants. La première concerne ceux obtenus avec la file non-critique implémentée sous forme de liste chaînée à accès exclusif. Dans cette série de résultats, seuls les temps de résolution sont présentés. Aussi une première analyse dégagera les principales caractéristiques de notre algorithme parallèle.

La seconde série des résultats est obtenue avec la file non-critique implémentée sous forme de tas à accès concurrent (Skew-Heap Concurrent). Pour cette série, les temps de résolution et les totaux des nœuds terminaux évalués sont présentés. Une comparaison sera faite avec les résultats de la première série pour discerner les améliorations.

Pour l'ensemble des résultats de nos programmes parallèles, un rapprochement par rapport à ceux des deux versions séquentielles de l'algorithme $\alpha - \beta$ est systématiquement établi.

5.1 Résultats de la première implémentation

Les résultats présentés ici sont ceux obtenus avec la première implémentation de notre algorithme et un générateur d'arbres incorporé. La file non-critique est implémentée sous forme de liste chaînée à accès exclusif.

Les accélérations sont non-linéaires et vite bornées. Ceci est provoqué par l'accès exclusif et des insertions en $O(n)$ dans la file non-critique. Néanmoins, ils mettent en évidence deux points :

1. l'importance de l'accès concurrent et la complexité des insertions de la file non-critique,
2. l'intérêt du degré d'élagages parallèles k.

[1] Aux échecs, les arbres sont de degré 38 et de profondeur 10 au milieu des parties.

5.1.1 Les accélérations (speed-up)

Nous faisons ici varier le nombre de processeurs (1 à 9) pour un degré d'élagages parallèles $k = 1$. Deux accélérations ont été calculées, l'une par rapport à la version classique de $\alpha - \beta$ séquentiel (cla.) Figure 5.1, l'autre par rapport à la version affaiblie (aff.) Figure 5.2.

Les courbes des figures 5.1 et 5.2 schématisent ces résultats. Elles montrent respectivement les évolutions des accélérations pour les arbres parfaitement ordonnés (Ord.), fortement ordonnés (Frt.) et aléatoirement ordonnés (Alé.). Les moyennes des accélérations (Moy.) sont également présentées.

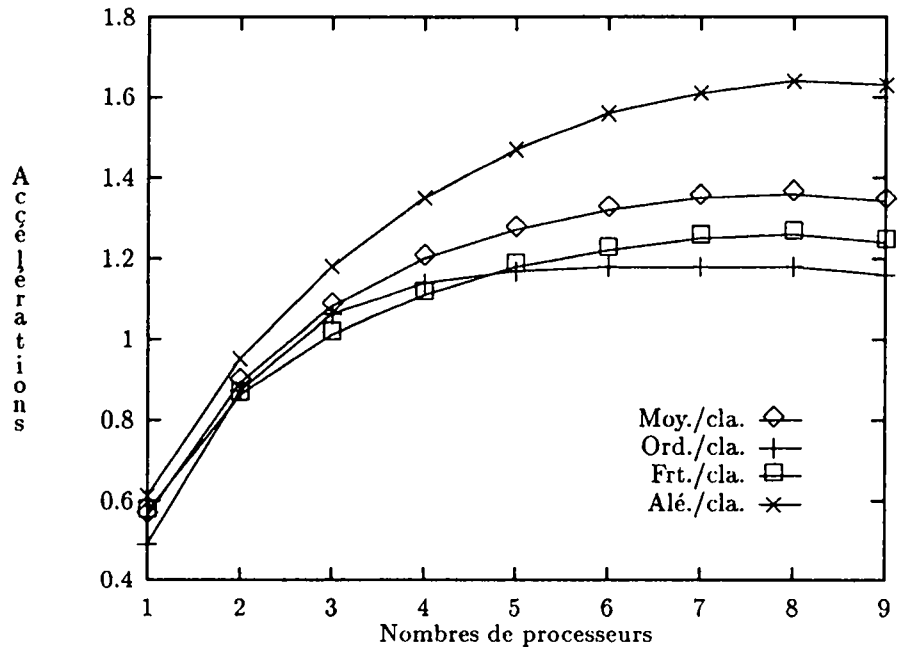

Figure 5.1 : L'évolution des accélérations par rapport à la version classique.

Nous pouvons tirer les premières remarques sur ces résultats :

1. les accélérations sont non-linéaires et rapidement bornées à partir de 5 processeurs parallèles et ceci quel que soit le type d'arbre,
2. des décélérations existent pour 1 à 2 processeurs,
3. les accélérations par rapport à la version affaiblie de $\alpha - \beta$ séquentiel sont plus importantes que celles par rapport à la version classique.

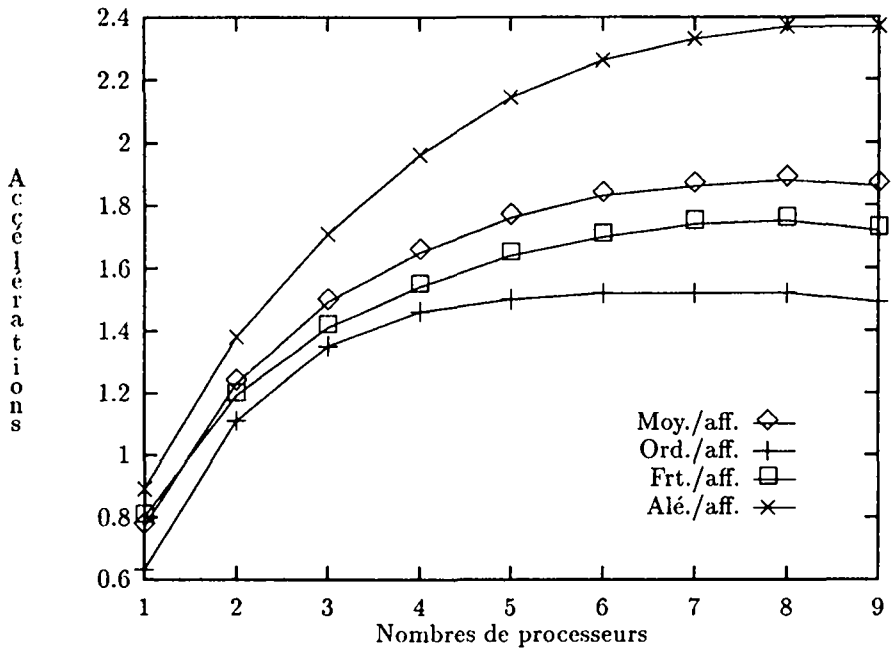

Figure 5.2 : L'évolution des accélérations par rapport à la version affaiblie.

La première remarque confirme les contentions d'accès aux structures de données. L'allure des courbes caractérise parfaitement ce problème. Alors que la deuxième remarque s'explique d'une part par les insertions en $O(n)$ dans la file non-critique, et d'autre part par les surcoûts du parallélisme dûs aux divers verrous et aux distributions de travail non existants en séquentiel.

Ces deux problèmes sont pris en compte dans la seconde implémentation de notre algorithme avec la file non-critique en tas à accès concurrent.

Quant à la dernière des remarques, la version affaiblie est moins performante que la version classique de l'algorithme $\alpha - \beta$ séquentiel.

5.1.2 L'influence du degré d'élagages parallèles k

Dans cette partie, nous allons analyser l'influence du degré d'élagages parallèles k introduit dans le but d'augmenter le parallélisme de l'algorithme.

Les expériences ont été menées avec 9 processeurs sur les mêmes arbres que précédemment. Les Figures 5.3 et 5.4 schématisent respectivement l'évolution des accélérations.

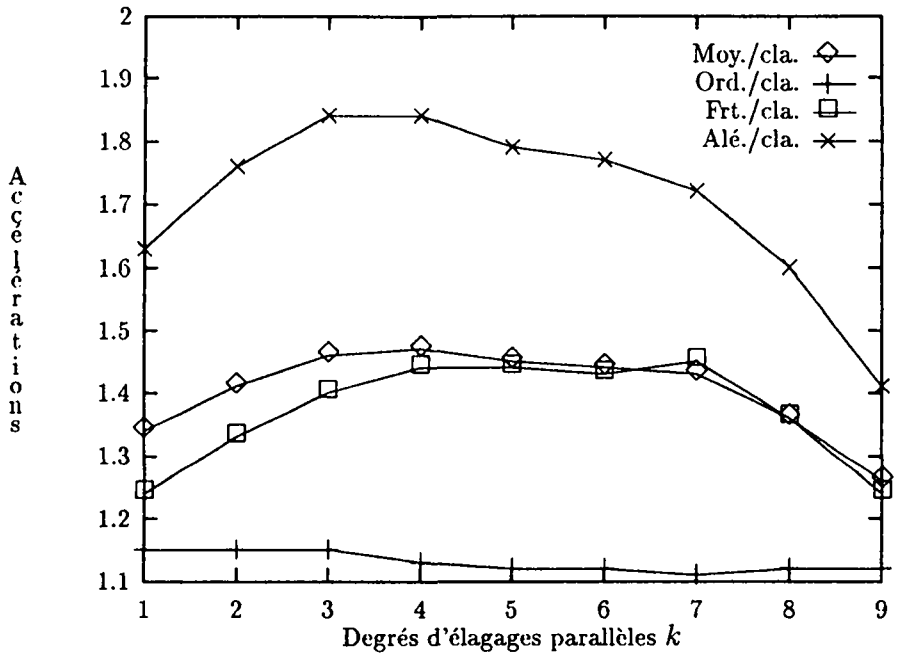

Figure 5.3 : L'évolution des accélérations par rapport à la version classique.

Nous pouvons constater que l'influence du degré d'élagages parallèles k sur le comportement de l'algorithme pour les arbres parfaitement ordonnés est minime voire inexistante. Ceci est tout à fait prévisible. Le paramètre k agit uniquement sur la file non-critique pour les élagages des nœuds non-critiques; or pour les arbres parfaitement ordonnés, l'exploration des nœuds critiques suffit à élaguer tous les nœuds non-critiques, la file non-critique n'est donc pas explorée. k a en conséquence aucune influence.

Par contre pour les arbres fortement ordonnés ou aléatoirement ordonnés, les accélérations croissent avec le degré d'élagages parallèles k jusqu'à $k = 4$. Pour des valeurs de k supérieures à 4, les accélérations décroissent. Des surcoûts de travail apparaissent et des noeuds non-critiques sont examinés inutilement.

Ceci confirme l'importance du degré d'élagages parallèles k pour les arbres non parfaitement ordonnés. Il importe donc de fixer convenablement k.

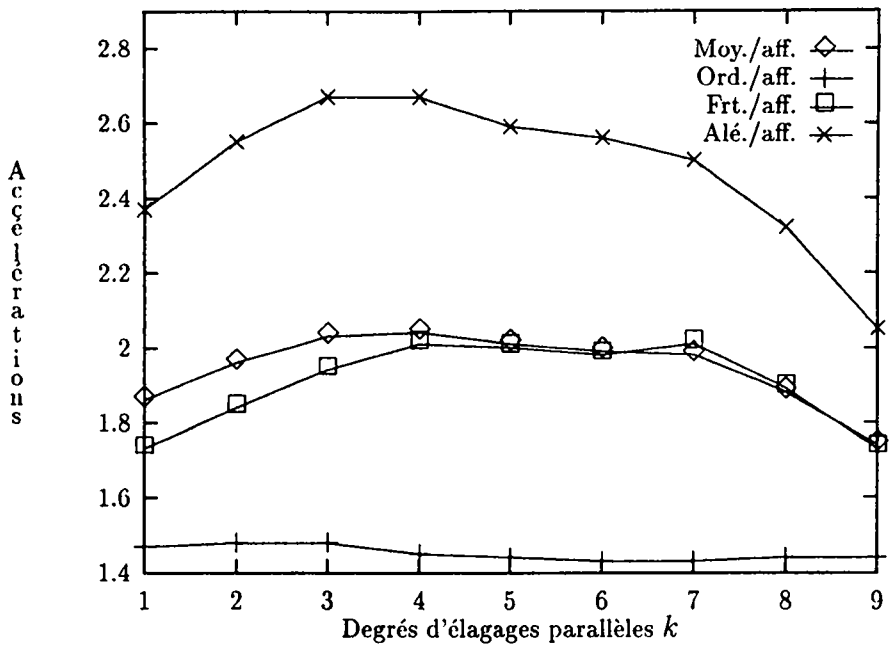

Figure 5.4 : L'évolution des accélérations par rapport à la version affaiblie.

5.2 Résultats de la seconde implémentation

En tenant compte des remarques faites à la suite de la première série de résultats, nous avons effectué une seconde implémentation de notre algorithme parallèle en transformant la file non-critique en tas à accès concurrent (Skew-Heap Concurrent).

Nous allons présenter pour ces résultats, comme précédemment, les accélérations obtenues sur les trois types d'arbres simulés de degré 20 et de profondeur 5. Puis nous ferons une étude complémentaire du comportement de notre algorithme parallèle sur des arbres de tailles différentes, en faisant varier le degré (5, 10, 15 et 20) et la profondeur (2, 3, 4 et 5).

Ensuite nous analyserons les nombres de nœuds terminaux évalués, car ils constituent aussi une mesure de performance (facteur de branchement [18]).

5.2.1 Les accélérations

La présentation des courbes est faite dans le même ordre que la première série de résultats.

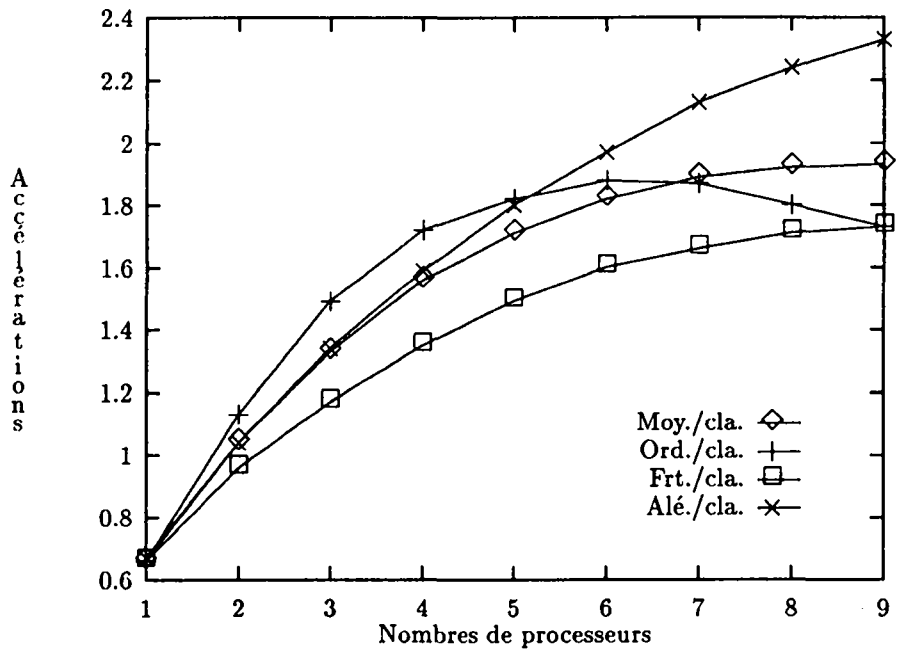

Figure 5.5 : L'évolution des accélérations par rapport à la version classique.

Avec la nouvelle implémentation de la file non-critique en tas à accès concurrent (Skew-Heap Concurrent), nous pouvons constater que les accélérations ont toutes nettement progressé par rapport à celles de la première implémentation. Les courbes d'accélérations atteignent des palliers pour un nombre de processeurs supérieurs, excepté le cas des arbres aléatoirement ordonnés (Alé./cla. et Alé./aff.) où il n'y a pas encore de limites.

Cependant les allures des courbes d'accélérations restent semblables à celles observées dans la première série d'expériences. Pourtant les contentions d'accès et les insertions en $O(n)$ de la file non-critique ont été résolues par l'utilisation d'un tas à accès concurrent. La non-linéarité des accélérations est liée à trois facteurs que nous avons sous-estimé dans les résultats antérieurs :

1. les allocations dynamiques des données en mémoire partagée sont coûteuses en temps,
2. les conditions d'expérience sont extrêmes dans le sens que la fonction d'évaluation des nœuds terminaux est très simple,

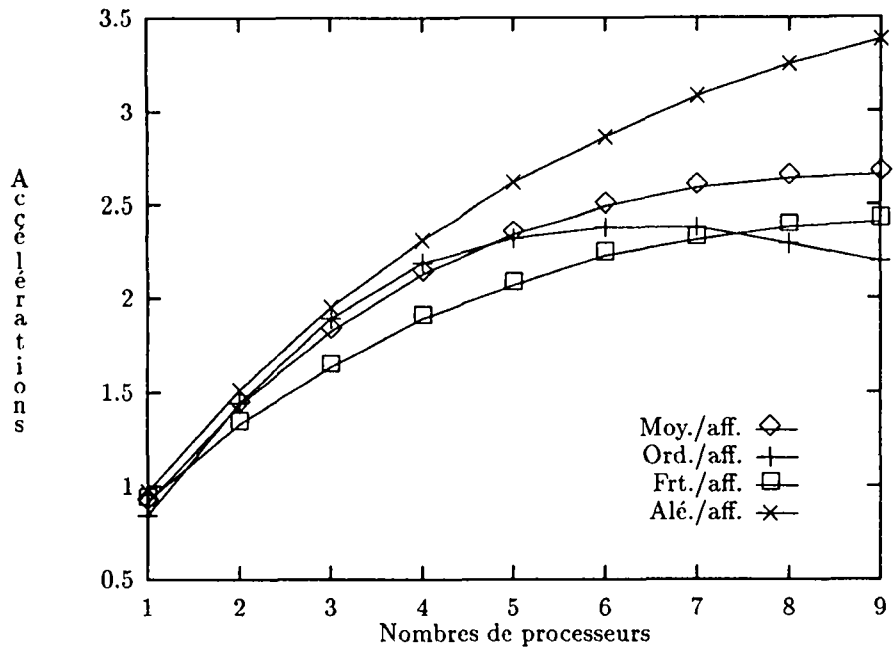

Figure 5.6 : L'évolution des accélérations par rapport à la version affaiblie.

3. la génération des nœuds des arbres de jeux simulés est très rapide.

Le premier facteur s'explique par le fait que, pour la même taille de mémoire, les temps d'allocation en mémoire partagée de notre machine parallèle est une fonction croissante du nombre de processeurs utilisés dans les résolutions. Ce fait est particulièrement sensible sur les figures 5.1 et 5.5. En effet, pour les arbres parfaitement ordonnés, seuls les nœuds de la file critique sont explorés. Nous pouvons alors nous attendre à des accélérations linéaires pour un nombre fixe de nœuds critiques à explorer. Or nous constatons que les accélérations décroissent à partir de cinq à six processeurs.

Aussi, pour une grande taille de mémoire à allouer, les problèmes de défauts de pages[2] apparaissent.

Sous le deuxième et le troisième facteurs se cache le problème de la granularité. Notre machine parallèle à mémoire partagée est mieux adaptée aux algorithmes parallèles à gros grain. Cependant notre algorithme est à granularité fine, surtout lorsque le travail d'évaluation des nœuds terminaux par une fonction heuristique (cf. définition 2) est infime dans le cas des arbres simulés; de même que la durée de génération des nœuds. Dans les conditions

[2]La machine utilisant une partie des disques comme mémoire virtuelle, elle effectue alors énormément d'accès aux disques pour trouver des mémoires disponibles.

d'applications réelles (les échecs, l'Othello, ou le Go), le travail plus important des fonctions heuristiques d'évaluation et des générations de nœuds permet de grossir la granularité de notre algorithme et d'obtenir des accélérations supérieures.

L'influence du degré d'élagages parallèles k

Comme pour les expériences menées jusqu'ici, celles-ci sont faites également sur les arbres de degré 20 et de profondeur 5. Nous avons utilisé les 9 processeurs disponibles de notre machine pour bénéficier d'un intervalle assez grand pour k.

Les courbes des figures 5.7 et 5.8 schématisent ces résultats. Elles montrent respectivement les évolutions des accélérations pour les arbres parfaitement ordonnés (Ord.), fortement ordonnés (Frt.) et aléatoirement ordonnés (Alé.). Les moyennes des accélérations (Moy.) sont également présentées.

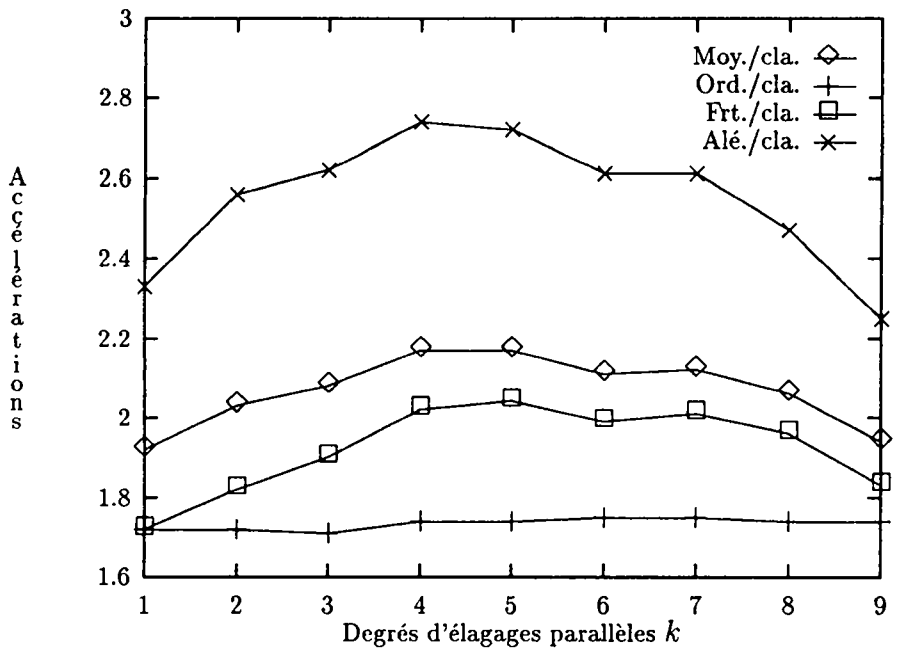

Figure 5.7 : L'évolution des accélérations par rapport à la version classique.

Ces résultats confirment tout ce que nous avons dit au sujet du degré d'élagages parallèles k pour la première série de résultats. De plus, l'écart entre les accélérations de $k = 1$ et $k = 4$ est relativement plus important que la première implémentation (de l'ordre de 0,25 contre 0,13 en moyenne).

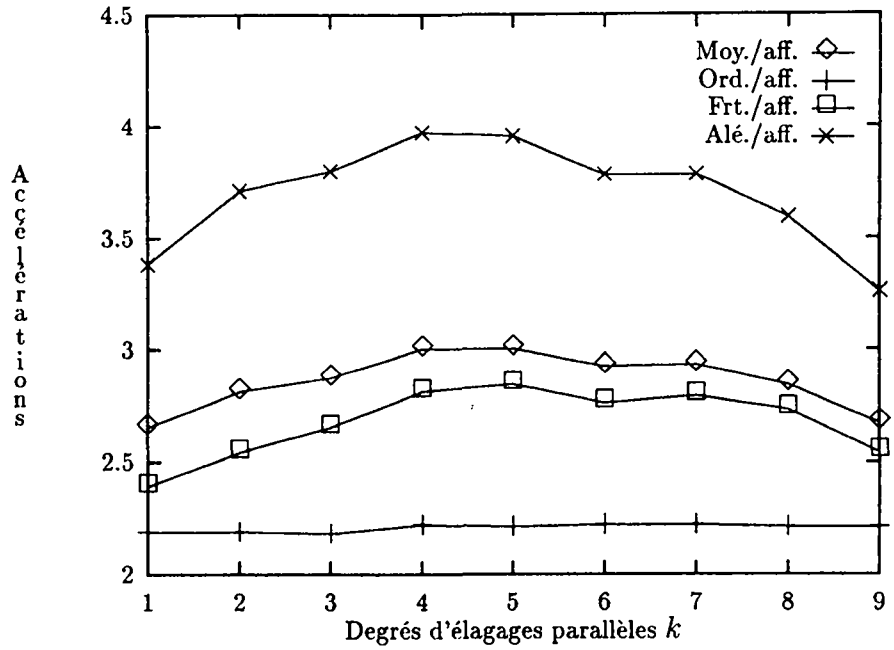

Figure 5.8 : L'évolution des accélérations par rapport à la version affaiblie.

Nous verrons un peu plus loin avec le nombre de nœuds terminaux évalués, l'importance de la valeur du degré d'élagages parallèles k dans le compromis entre les accélérations et le nombre de nœuds terminaux évalués.

L'influence de la profondeur des arbres simulés

Pour connaître le comportement de notre algorithme face aux problèmes de petite taille, nous avons varié la profondeur des arbres simulés (de profondeur 2 à profondeur 5).

Nous avons choisi de présenter uniquement les accélérations obtenues avec 9 processeurs et un degré d'élagages parallèles $k = 4$. C'est la configuration de notre algorithme qui a obtenu les meilleures accélérations. Les figures 5.9 et 5.10 synthétisent ces résultats.

Les accélérations sont beaucoup plus faibles voire mêmes préjudiciables (*speedup* < 1) avec des arbres de petite taille. Elles sont nettement plus importantes pour les arbres de profondeur 3 et 4. Cependant, une limite semble exister pour les arbres de grande taille (à partir de profondeur 5) à cause des problèmes de défauts de pages déjà évoqués (cf. paragraphe 5.2.1).

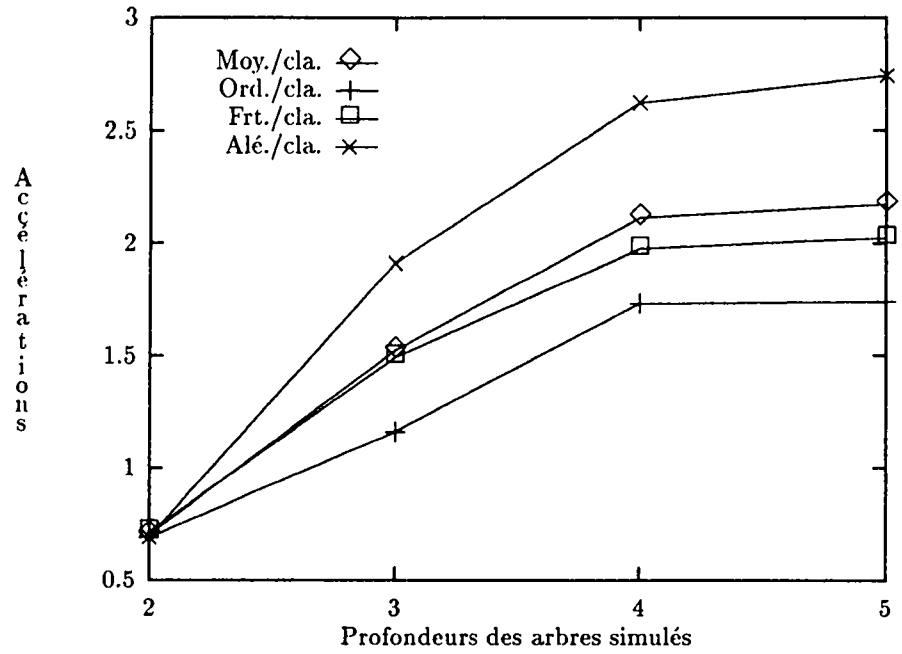

Figure 5.9 : L'évolution des accélérations par rapport à la version classique.

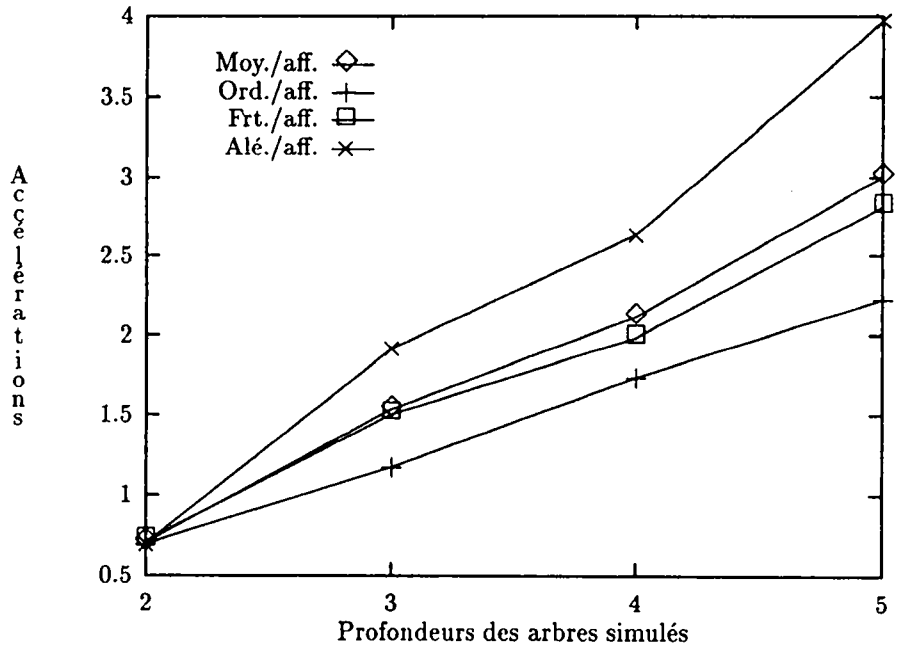

Figure 5.10 : L'évolution des accélérations par rapport à la version affaiblie.

L'influence du degré des arbres simulés

Au lieu de l'influence de la profondeur, nous étudions ici l'influence du degré des arbres de jeux simulés. Nous avons varié le degré (de degré 5 au degré 20 en passant par 10 et 15).

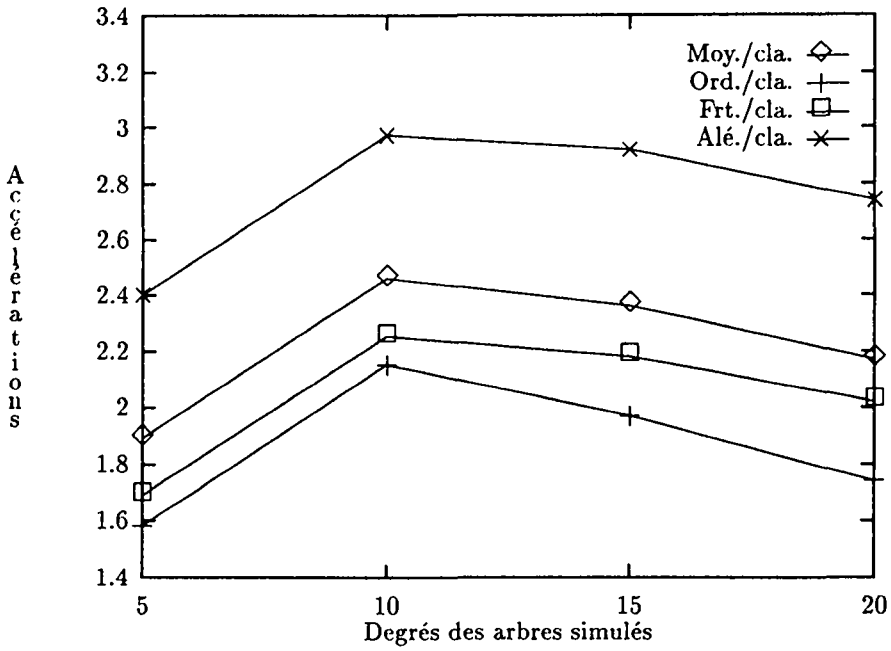

Figure 5.11 : L'évolution des accélérations par rapport à la version classique.

La figure 5.11 montre que les accélérations croissent également avec l'augmentation du degré des arbres simulés, tout au moins du degré 5 au degré 10. Cela confirme que notre algorithme est plus efficace pour les arbres de taille relativement grande.

Cependant, nous observons une diminution des accélérations pour les arbres de degré 15 et 20. C'est le problème des défauts de pages pour les arbres de grande taille. L'influence de la profondeur des arbres ne nous a pas permis de voir que les accélérations pouvaient être plus grandes avec les arbres de taille plus petite. Car les arbres de degré 20 et de profondeur 4 sont de taille supérieure à ceux de degré 10 et de profondeur 5. Tandis que ceux de degré 20 et de profondeur 3 sont de taille inférieure.

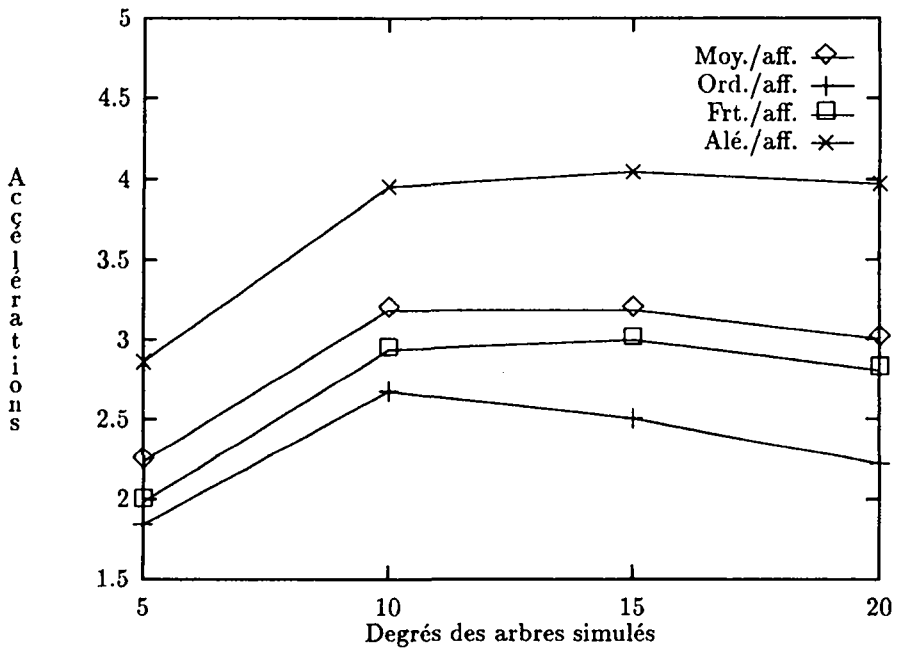

Figure 5.12 : L'évolution des accélérations par rapport à la version affaiblie.

5.2.2 Les nœuds terminaux évalués

Comme les accélérations, il est intéressant d'étudier les totaux des nœuds terminaux évalués par notre algorithme en comparaison avec ceux des deux versions de l'algorithme $\alpha - \beta$ séquentiel. D'autant plus intéressant qu'une mesure d'efficacité en terme de coût (facteur de branchement), introduite par Knuth et Moore [18], est fondée justement sur le total des nœuds terminaux évalués par un algorithme. Car il faut se rappeler que dans l'algorithme $\alpha - \beta$ seuls les nœuds terminaux sont évalués par une fonction heuristique.

Aussi, nous essayerons d'établir des relations entre les accélérations observées dans les paragraphes précédents et les totaux des nœuds terminaux évalués présentés ici.

Les résultats sont obtenus à partir des mêmes arbres simulés que pour les accélérations. Les figures 5.13 et 5.14 schématisent ces résultats.

Nous voyons bien, que par rapport aux deux versions de l'algorithme $\alpha-\beta$ séquentiel, les totaux des nœuds terminaux évalués par notre algorithme croissent avec le nombre de processeurs utilisés; notamment pour les arbres fortement et aléatoirement ordonnés. Ceci s'explique par le fait que l'explo-

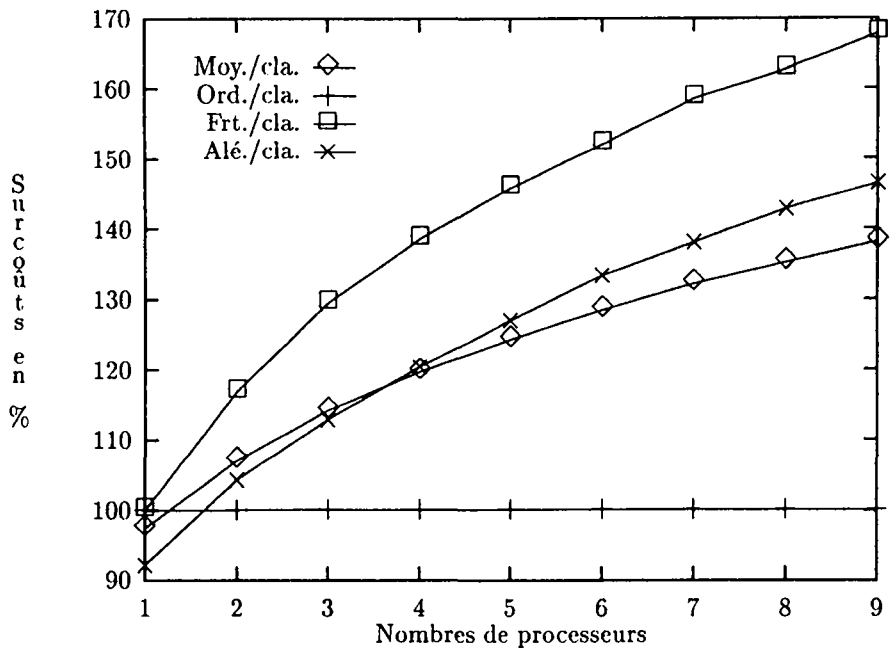

Figure 5.13 : Les totaux des nœuds terminaux évalués par rapport à la version classique.

ration des nœuds critiques ne suffit pas à élaguer les nœuds non-critiques. Si bien que l'exploration des nœuds non-critiques par plusieurs processus provoque des surcoûts de recherche par rapport aux versions séquentielles. Ces surcoûts expliquent en partie les limites des accélérations observées précédemment.

Cependant, ces surcoûts ne sont pas linéaires quant au nombre de processeurs utilisés. Les courbes semblent s'infléchir pour un nombre de processeurs utilisés assez grand.

Alors que les totaux des nœuds terminaux évalués restent constants pour les arbres ordonnés. Dans ce cas, l'exploration des nœuds critiques permet d'élaguer immédiatement les autres, d'où la stabilité de ces totaux.

Il est à noter que notre algorithme réalise, avec un seul processeur, des totaux inférieurs à ceux de la version classique de l'algorithme $\alpha - \beta$ séquentiel. Ceci prouve et confirme que l'algorithme $\alpha - \beta$ n'est pas le meilleur algorithme séquentiel [6].

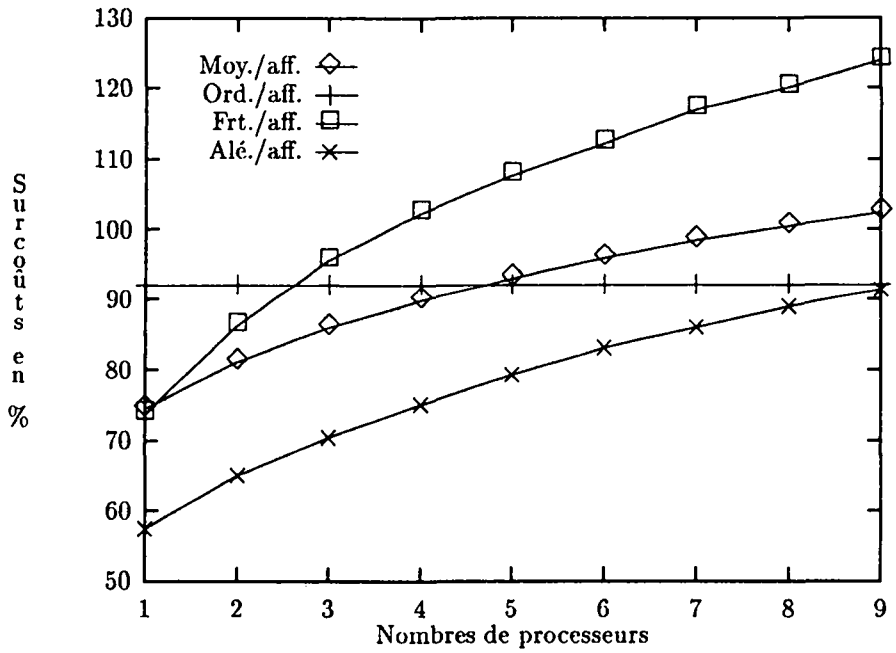

Figure 5.14 : Les totaux des nœuds terminaux évalués par rapport à la version affaiblie.

Aussi en comparant les figures 5.13 et 5.14, nous remarquons clairement que la version affaiblie de l'algorithme $\alpha - \beta$ séquentiel évalue plus de nœuds terminaux que la version classique.

L'influence du degré d'élagages parallèles k

Voyons à présent l'influence du degré d'élagages parallèles k sur les totaux de nœuds terminaux évalués. Nous avons vu précédemment que notre algorithme réalise ses meilleures accélérations pour $k = 4$. Or une partie des explications se trouve dans les figures 5.15 et 5.16.

En effet, nous pouvons constater qu'il y a une baisse importante du nombre de nœuds terminaux évalués lorsque k passe de 1 à 4. Pourtant en augmentant le degré d'élagages parallèles k, nous attendons plutôt une croissance du nombre de nœuds terminaux évalués. Cependant, lorsque k est grand, nous explorons k nœuds terminaux en parallèles, et nous arrivons souvent en peu d'itérations au nœud qui permet l'élagage des autres nœuds frères. Par conséquent, nous évitons d'évaluer bon nombre de nœuds terminaux.

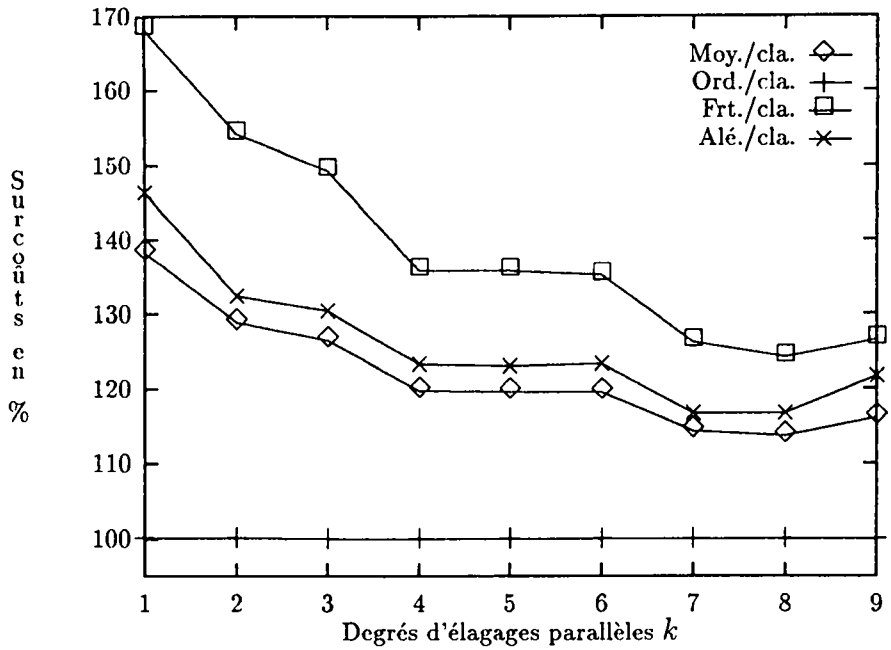

Figure 5.15 : Les totaux des nœuds terminaux évalués par rapport à la version classique.

Pour $4 < k < 9$, les totaux de nœuds terminaux évalués continuent de baisser. Mais cette diminution est faite au détriment de la disponibilité des processus pour l'exploration d'autres nœuds. Parce que trop de processus sont occupés à élaguer des nœuds frères, alors que le processus qui explore le nœud possédant la valeur Negamax suffisante pour élaguer l'ensemble des nœuds frères aurait suffit aux élagages. Ceci explique pourquoi les accélérations diminuent.

Lorsque $k = 9$, nous observons même une légère hausse des totaux des nœuds terminaux évalués. Car le processus possédant le nœud avec la valeur Negamax suffisante pour les élagages n'a pas le temps de la communiquer aux autres processus, d'où des surcoûts d'évaluation et des accélérations moindres.

Il est à noter que pour les arbres ordonnés, les nœuds critiques suffisent à élaguer les autres. Comme la communication est faite immédiatement aux autres processus, les surcoûts n'ont pas lieu et les accélérations sont constantes.

Il importe donc de bien choisir la valeur de k pour trouver un compromis entre les accélérations et le total des nœuds terminaux évalués.

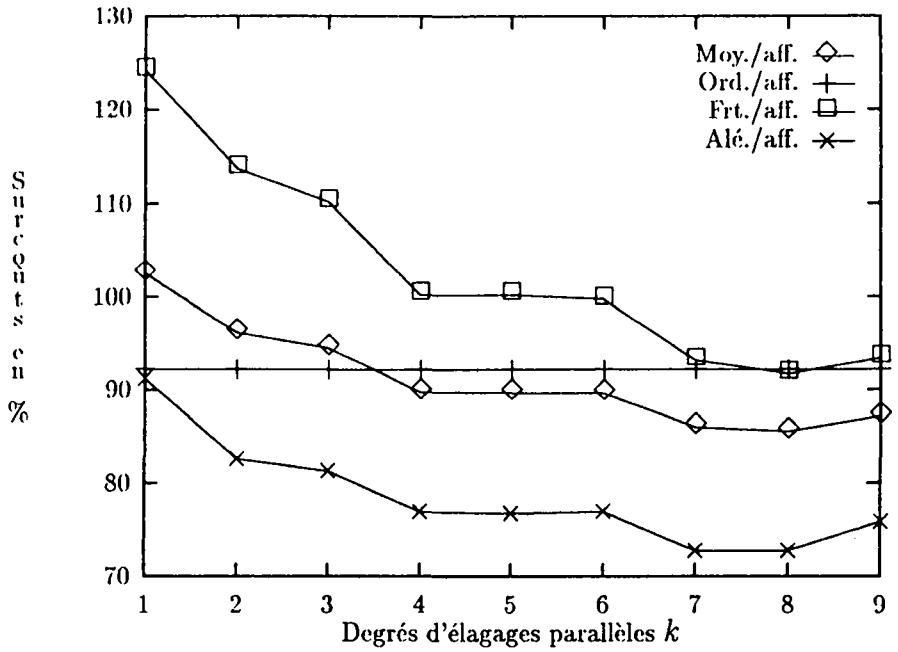

Figure 5.16 : Les totaux des nœuds terminaux évalués par rapport à la version affaiblie.

L'influence de la profondeur des arbres simulés

Par rapport à la version classique (Figure 5.17), les totaux de nœuds terminaux évalués progressent avec l'augmentation de la profondeur des arbres fortement et aléatoirement ordonnés.

Pour les arbres ordonnés, il y a un léger surcoût de recherche lorsque les valeurs Negamax suffisantes pour élaguer des nœuds n'ont pu être communiquées aux autres processus.

Quant à la version affaiblie, nous remarquons que lorsque les élagages profonds deviennent possibles (i.e. à partir de la profondeur 4), notre algorithme évalue moins de nœuds terminaux.

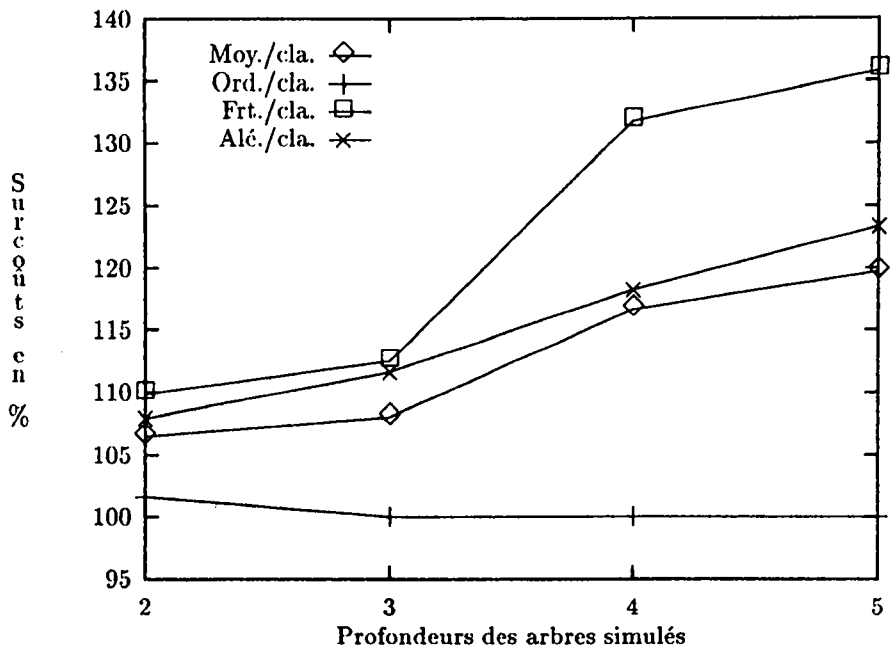

Figure 5.17 : Les totaux des nœuds terminaux évalués par rapport à la version classique.

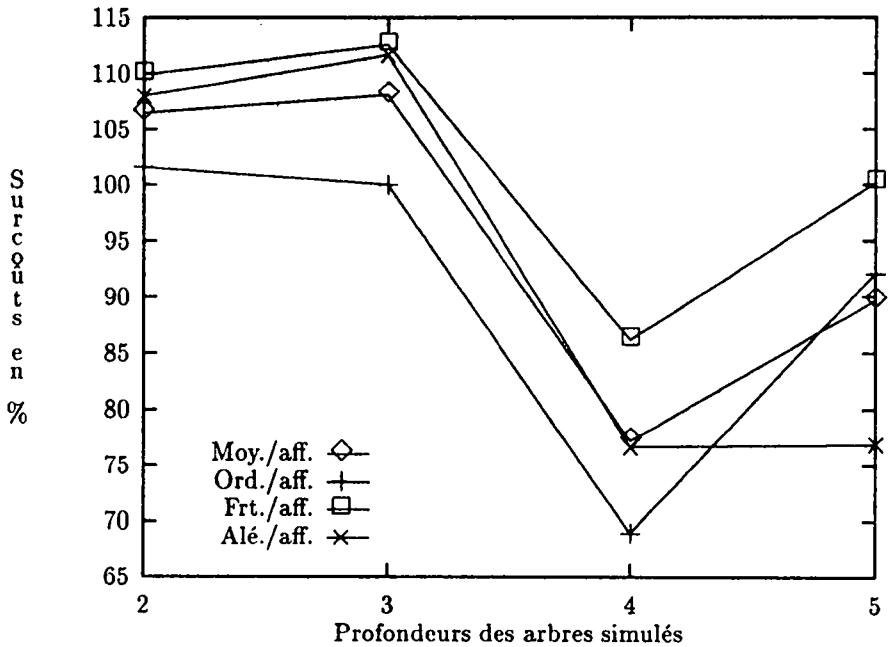

Figure 5.18 : Les totaux des nœuds terminaux évalués par rapport à la version affaiblie.

L'influence du degré des arbres simulés

Les figures 5.19 et 5.20 montrent l'évolution des totaux des nœuds terminaux évalués suivant le degré des arbres simulés.

Nous observons, pour les arbres fortement et aléatoirement ordonnés, les totaux des nœuds terminaux évalués diminuent avec l'accroissement du degré des arbres (de 10 à 20). En effet, avec l'accroissement du degré, la quantité de travail augmente et les surcoûts de recherche sont moindres.

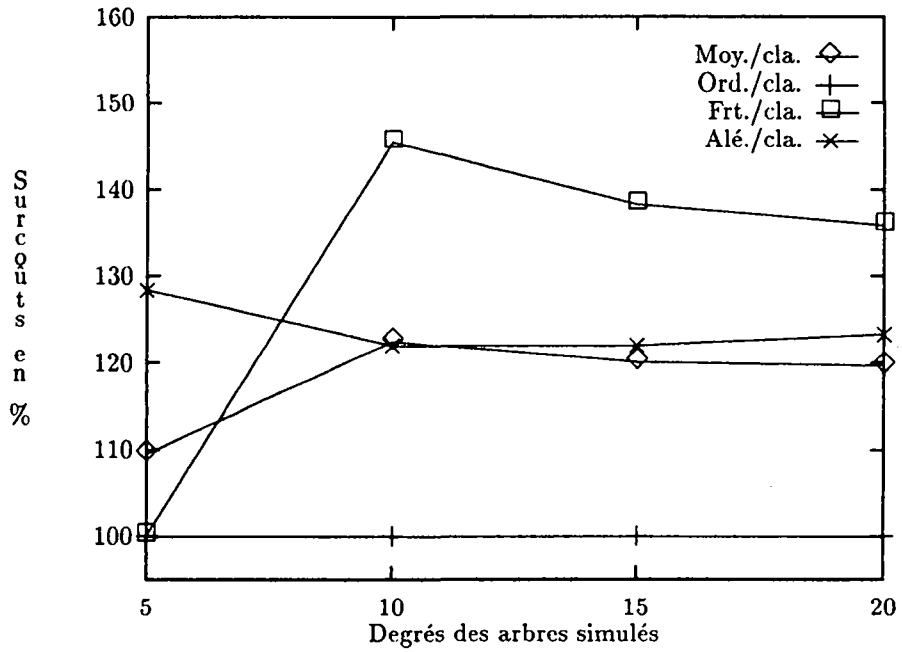

Figure 5.19 : Les totaux des nœuds terminaux évalués par rapport à la version classique.

Pour les arbres fortement ordonnés de degré 5, du fait de leur caractéristique, nous pouvons les considérer comme des arbres parfaitement ordonnés. C'est la raison pour laquelle, le total des nœuds terminaux évalués pour les arbres fortement ordonnés de degré 5 est le même que celui des arbres parfaitement ordonnés.

Pour les arbres parfaitement ordonnés, il est à remarquer sur la figure 5.20 que les totaux des nœuds terminaux progressent. Alors que ces totaux restent constants dans la figure 5.19. Cela est dû au fait que la version affaiblie de l'algorithme $\alpha - \beta$ séquentiel évalue de moins en moins de nœuds terminaux. Les totaux de ces derniers restent cependant supérieurs à ceux évalués par la version classique.

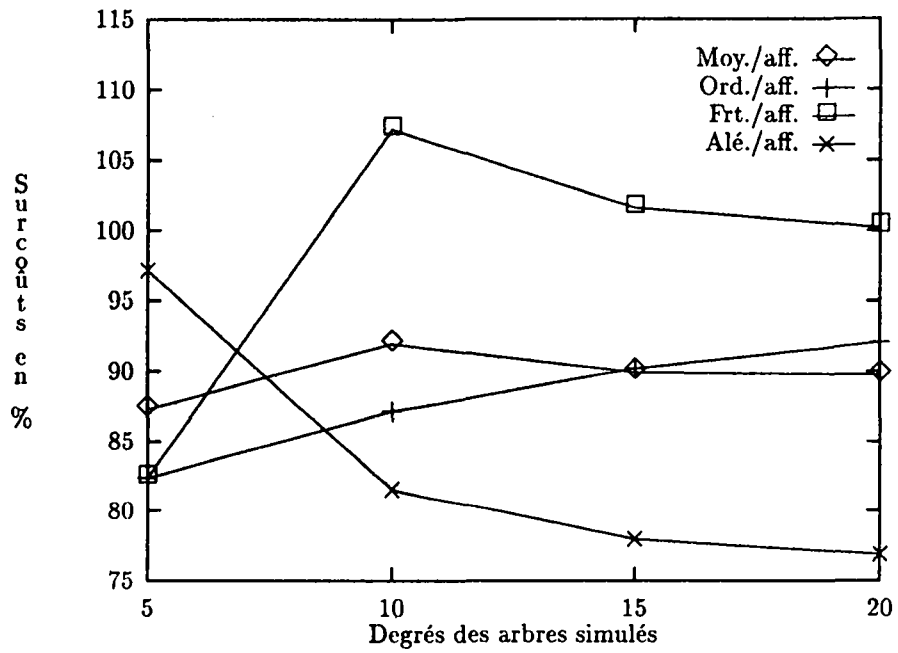

Figure 5.20 : Les totaux des nœuds terminaux évalués par rapport à la version affaiblie.

Conclusions

Les apports de cette étude sont multiples. Ils sont concrétisés par :

1. la formulation des critères de choix entre les deux versions de l'algorithme $\alpha-\beta$, et la connaissance de la taille mémoire minimale nécessaire à l'implémentation de l'algorithme parallèle;
2. la proposition d'un algorithme parallèle asynchrone avec grain fin (chaque processus n'explore qu'un nœud à la fois) fondé sur les algorithmes Branch and Bound parallèles [26] et sur la notion d'arbre critique;
3. l'utilisation d'un arbre de score partagé, qui rend les élagages profonds possibles et résoud les difficultés dans la remontée des valeurs Negamax vers la racine de l'arbre de jeux;
4. l'utilisation de deux files d'attente dont l'une à accès concurrent pour résoudre les problèmes de contention d'accès aux nœuds des arbres à explorer,
5. l'introduction du degré d'élagages parallèles k pour augmenter le parallélisme de l'exploration des arbres.

Le degré d'élagages parallèles k donne des résultats intéressants. Il a permis une nette amélioration des accélérations de notre algorithme parallèle dans les conditions d'expérience extrêmes. Il autorise également la recherche d'un compromis, dans l'utilisation des processus, entre les accélérations et le nombre de nœuds terminaux évalués.

Les travaux futurs seront centrés essentiellement sur quatre points :

1. rechercher ou développer des structures de données mieux adaptées aux spécificités des élagages de $\alpha - \beta$,

2. étudier le comportement de notre algorithme sur d'autres machines parallèles,

3. tester l'efficacité réelle de notre algorithme avec une application comme l'Othello ou les échecs,

4. appliquer le capital de connaissances acquis aux problèmes que l'on peut trouver dans des méthodes voisines de l'algorithme $\alpha - \beta$. En effet, cet algorithme est un cas particulier de la classe des algorithmes diviser pour régner, évaluation et séparation [23].

   L'étude de la parallélisation de $\alpha - \beta$ dépasse le cadre restreint d'un simple algorithme de jeux, elle peut servir pour la parallélisation d'autres algorithmes :

   - où il faut obligatoirement explorer un ensemble de tâches,
   - et conserver les optimisations qui assuraient en séquentiel une diminution de la recherche, donc un gain de temps.

Bibliographie

[1] Selim G. Akl. *The design and analysis of parallel algorithms.* Prentice Hall, 1989.

[2] S.G. Akl, D.T. Barnard, and R.J. Doran. Simulation and analysis in deriving time and storage requirements for a parallel alpha-beta algorithm. In *International Conference on Parallel Processing*, pages 231–234, 1980.

[3] S.G. Akl, D.T. Barnard, and R.J. Doran. Design, analysis, and implementation of a parallel tree search algorithm. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-4(2):192–203, March 1982.

[4] G.M. Baudet. *The design and analysis of algorithms for asynchronous multiprocessors.* PhD thesis, Carnegie-Mellon University, 1978.

[5] Andrei Z. Broder, Anna R. Karlin, Prabhakar Raghavan, and Eli Upfal. On the parallel complexity of evaluating game-trees. RR RJ 7729, IBM Research Division, October 1990.

[6] Murray S. Campbell and T.A. Marsland. A comparison of minimax tree search algorithms. *Artificial Intelligence*, 20(4):347–367, July 1983.

[7] E. Cousin, C. Coustet, M. Cubero-Castan, G. Durrieu, B. Lecussan, M. Lemaitre, D. Marre, and P. Ng. Algorithme parallèle en langage fonctionnel : application à mars_lisp. *BIGRE*, (66):147–162, May 1989. Journées AFCET-GROPLAN à Chamonix.

[8] E.W. Felten and S.W. Otto. Chess on a hypercube. In Geoffrey Fox, editor, *The Third Conference on Hypercube Concurrent Computers and Applications*, volume II-Applications, pages 1329–1341, 1988.

[9] R.A. Finkel and J.P. Fishburn. Parallelism in alpha-beta search. *Artificial Intelligence*, 19:89–106, 1982.

[10] R.A. Finkel and J.P. Fishburn. Improved speedup bounds for parallel alpha-beta search. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-5(1):89–92, January 1983.

[11] J.P. Fishburn and R.A. Finkel. Parallel alpha-beta search on arachne. Technical report, University of Wisconsin-Madison, Computer Sciences Dept., 1980.

[12] S.H. Fuller, J.G. Gaschnig, and J.J. Gillogly. An analysis of the alpha-beta pruning algorithm. Technical report, Carnegie-Mellon University, Dept. of Computer Science, Pittsburgh, July 1973.

[13] J.J. Gillogly. *Performance Analysis of the Technologie Chess Program.* PhD thesis, Carnegie-Mellon University, 1978.

[14] Usui Hiromoto, Yamashita Masafumi, Imai Masaharu, and Ibaraki Toshihide. Parallel searches of game trees. *Systems and Computers in Japan*, 18(8):97–109, 1987.

[15] Feng-Hsiung Hsu. *Large Scale Parallelization of Alpha-Beta search : An Algorithmic and Architectural Study with Computer Chess.* PhD thesis, Carnegie-Mellon University, School of Computer Science, February 1990.

[16] R.M. Hyatt and B.W. Suter. A parallel alpha/beta tree searching algorithm. *Parallel Computing*, 10:299–308, 1989.

[17] Richard M. Karp and Yanjun Zhang. On parallel evaluation of game trees. *First ACM Annual symposium on parallel algorithms and architectures*, pages 409–420, 1989.

[18] D.E. Knuth and R.W. Moore. An analysis of alpha-beta pruning. *Artificial Intelligence*, 6(4):293–326, 1975.

[19] Gary Lindstrom. The key node method : A highly-parallel alpha-beta algorithm. Technical Report UUCS 83-101, University of Utah, Department of Computer Science, Salt Lake City, March 1983.

[20] Bernard Mans and Catherine Roucairol. Concurrency in priority queues for branch and bound algorithms. RR 1311, INRIA-Rocquencourt, October 1990.

[21] T.A. Marsland and M. Campbell. Parallel search of strongly ordered game trees. *ACM Computing Surveys*, 14(4):533–551, December 1982.

[22] T.A. Marsland and F. Popowich. Parallel game-tree search. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-7(4):442–452, July 1985.

[23] Dana S. Nau, Vipin Kumar, and Laveen Kanal. General branch and bound, and its relation to a* and ao*. *Artificial Intelligence*, 23:29–58, 1984.

[24] Nils J. Nilsson. *Principles of Artificial Intelligence.* Tioga Publishing Co., 1980.

[25] Anita Osterhaug. *Guide to parallel programming on sequent computer systems.* Sequent technical publications. Sequent computer systems, Inc., 1987.

[26] Catherine Roucairol. Recherche arborescente en parallèle. RR M.A.S.I. 90.4, Institut Blaise Pascal - Paris VI, 1990.

[27] Igor Steinberg and Marvin Solomon. Searching game trees in parallel. Technical Report 877, University of Wisconsin-Madison, Computer Sciences Dept., September 1987.

[28] G.C. Stockman. A minimax algorithm better than alpha-beta ? *Artificial Intelligence*, 12:179–196, 1979.

[29] K. Thompson. Computer chess strength. *Advances in Computer Chess 3*, pages 55–56, 1982.

Annexe A

Génération des problèmes

La première solution consiste à utiliser un générateur statique d'arbres uniformément aléatoires. Il est qualifié de statique car les arbres générés seront tous stockés dans des fichiers disques. Cette méthode nous garantit par la suite d'avoir des données identiques pour les tests. Cependant, les arbres de jeux réels pour les échecs en milieu de partie sont en général de degré 38 et de profondeur 10, soient 38^{10} nœuds !! Un générateur statique est par conséquent inutilisable dans ce cas, mais il permet la mise au point des programmes dans les premières étapes des implémentations et des tests sur des arbres de petites tailles.

Pour approcher les conditions réelles de jeux nous avons implémenté un générateur dynamique qui est intégré aux algorithmes pour engendrer les nœuds d'un fils au fur et à mesure des explorations. L'arbre de jeux ne sera donc pas stocké sur fichier. Mais cette approche ne garantit pas des arbres identiques pour différentes exécutions des algorithmes, puisque ces dernières dépendent fortement du parcours. En effet, Il se peut que deux algorithmes différents ne visitent pas les mêmes nœuds, et même si c'était le cas sûrement pas dans le même ordre. De plus, le non déterminisme des algorithmes parallèles peut changer l'ordre des explorations de l'arbre. Il est possible d'associer des nombres identiques à chaque nœud non-terminal d'un arbre. Cette derrière solution nous permet d'avoir des arbres identiques, mais en chaque nœud il faut appeler $O(N)$ fois le générateur de nombres aléatoires si l'arbre est de degré N.

Comme nous cherchons à nous rapprocher le plus possible des conditions réelles de jeux, même si les arbres ne sont pas identiques à chaque exécution. Nous ne nous intéresserons qu'aux temps moyens des exécutions.

Ce générateur dynamique d'arbres proposé [15] peut être décrit comme

suit :

- des *poids* w seront affectés aux N fils d'un nœud. Ces poids déterminent le degré d'ordre des arbres à engendrer, ils doivent avoir un total de 100. Par exemple, pour engendrer un arbre ordonné meilleur d'abord, on affectera un poids de 100 au premier fils d'un nœud non-terminal et des poids de 0 aux autres fils. Pour avoir un arbre ordonné aléatoirement, on affectera un poids de $100/N$ aux N fils d'un nœud non-terminal.

- à la racine s, un nombre aléatoire est utilisé pour sélectionner sur la base des poids w quel fils parmi N aura la plus grande valeur Negamax. Supposons que le fils i soit sélectionné.
  La valeur Negamax v du fils i est généré aléatoirement dans l'intervalle $]-\infty, +\infty[$. Et les valeurs des fils $1 \ldots i-1$ sont générées aléatoirement dans l'intervalle $]-\infty, v-1]$, celles des fils $i+1 \ldots N$ dans l'intervalle $]-\infty, v]$.
  Toutes ces valeurs sont stockées dans une pile. Pour générer les fils d'un nœud non-terminal s_i, la valeur de s_i doit être dépilée. Le meilleur fils j de s_i est de nouveau choisi en fonction de son poids et sa valeur Negamax sera celle de s_i : $-v$. Cette méthode sera appliquée récursivement tout au long de la génération d'arbre.

Nous sommes certains que les valeurs Negamax des nœuds fils sont générées avec les valeurs Negamax connues des nœuds pères en suivant le type d'ordre souhaité.

Imprimé en France
par
. l'Institut National de Recherche en Informatique et en Automatique

ISSN 0249 - 6399