# Algoritmos Paralelos para Busca em Árvores de Jogo

Claudio Santos Pinhanez

DISSERTAÇÃO APRESENTADA
AO
INSTITUTO DE MATEMÁTICA E ESTATÍSTICA
DA
UNIVERSIDADE DE SÃO PAULO

PARA OBTENÇÃO DO GRAU DE

MESTRE EM
MATEMÁTICA APLICADA

Área de Concentração: **Ciência da Computação**
Orientador: **Prof. Dr. Routo Terada**

— São Paulo, Agosto de 1989 —

# Abstract


## Parallel Algorithms for Game Tree Search

In this decade, several algorithms were proposed for parallel game tree search. This work studies these algorithms deeply, and suggests that they are, in essence, parallelizations of the sequential algorithms. The two main methods of parallelization are identified, the first based on Baudet's idea of Window Reduction, and the second on Akl's Mandatory Work First method.

The two methods are analyzed through extensive computer simulation, and the results conclusively show that parallelizations based on Window Reduction, although not much effective, can be used in any situation of game search. By its turn, the Mandatory Work First method was discovered to be very efficient, but extremely vulnerable to particular characteristics of the game tree.

This work starts with a detailed description of the nature of the game tree search problem (chapter 2), including an original proof of its lower bound. The sequential algorithms are then carefully examined, using a common framework named *description by messages* (chapter 3). Their complexity analysis is also done (chapter 4), which reveals their basic characteristics and flaws.

An extensive review of parallel algorithms for game tree search is given (chapter 5), again using *description by messages*, which contributes to clarify significantly their similarities and differences. The motivation, description and experimental analysis of the two main methods of parallelization are given (chapter 6), and some conclusions and conjectures about parallel computing analysis are listed at the end of the work (chapter 7).

# Agradecimentos

Ao meu orientador Prof. Routo Terada, pelas idéias e pelo tempo dispendidos neste trabalho.

À Biblioteca do IME e aos seus funcionários, sem a qual nada seria possível.

Aos meus colegas de trabalho, por me escutarem nas minhas empolgações e decepções.

Ao Prof. Marcos "Gubi", pela interminável paciência na caça de *bugs* da máquina WORM.

À Mônica e à minha mãe, pelos doces, pelo carinho, pelo apoio e pela compreensão.

Ao meu pai, que sempre esteve presente.

Meu Sincero Obrigado.

# Índice


**1 Introdução** **1**
- 1.1 Jogos e Máquinas . . . . . 1
- 1.2 Computadores Paralelos . . . . . 3
- 1.3 Objetivos da Dissertação . . . . . 4
- 1.4 Organização da Dissertação . . . . . 5

**2 Busca em Árvores de Jogo** **6**
- 2.1 Representação de Jogos por Árvores AND/OR . . . . . 7
- 2.2 Valoração dos Nós Terminais . . . . . 7
- 2.3 O Processo de Aproximação à Estratégia Defensiva Ótima . . . . . 8
- 2.4 Árvores Solução . . . . . 9
- 2.5 Estratégia Defensiva Ótima . . . . . 12
- 2.6 O Limite Inferior . . . . . 14
- 2.7 Conceitos Equivalentes à Estratégia Defensiva Ótima . . . . . 19
- 2.8 A Máquina de Turing Alternante . . . . . 21

**3 Algoritmos Seqüenciais para Busca em Árvores de Jogo** **22**
- 3.1 O Formalismo de Descrição dos Algoritmos . . . . . 22
- 3.2 Algoritmo Minimax . . . . . 24
- 3.3 Algoritmo Bound . . . . . 25
- 3.4 Algoritmo Alpha-Beta . . . . . 28
- 3.5 Cortes Rasos e Profundos . . . . . 32
- 3.6 Algoritmo SSS* . . . . . 34
- 3.7 Algoritmo SCOUT . . . . . 37
- 3.8 Considerações sobre a Implementação dos Algoritmos . . . . . 42

**4 Análise dos Algoritmos Seqüenciais** **43**
- 4.1 Modelos de Árvores de Jogo . . . . . 43
  - 4.1.1 O Modelo Maniqueísta . . . . . 44
  - 4.1.2 O Modelo Aleatório . . . . . 44
  - 4.1.3 Modelos de Valor Dependente da Ramificação . . . . . 44
  - 4.1.4 Modelo de Distribuição Fortemente Ordenada . . . . . 45
- 4.2 Métodos de Comparação . . . . . 45
- 4.3 Relações de Dominância . . . . . 46
- 4.4 Resultados sobre Árvores Maniqueístas . . . . . 51

4.5 Resultados sobre Árvores Aleatórias . . . 53
4.5.1 O Fator de Ramificação do Algoritmo Alpha-Beta . . . 54
4.5.2 O Limite Inferior do Fator de Ramificação de Árvores Aleatórias . . . 55
4.5.3 O Fator de Ramificação de SSS* e SCOUT . . . 55
4.5.4 Crítica ao Uso do Fator de Ramificação em Modelos Aleatórios . . . 56
4.6 Resultados sobre Outros Modelos . . . 57
4.7 Algumas Otimizações nos Algoritmos Seqüenciais . . . 58

**5 Algoritmos Paralelos para Busca em Árvores de Jogo 60**
5.1 Arquiteturas Paralelas . . . 60
5.2 Métodos Básicos de Paralelização . . . 62
5.2.1 Método de Paralelização por Redução de Janelas . . . 62
5.2.2 Método de Paralelização por Priorização do Trabalho Obrigatório . . . 64
5.3 Algoritmos para Processadores em Árvore . . . 65
5.3.1 Algoritmo Tree-Splitting . . . 66
5.3.2 Algoritmo PV-Splitting . . . 68
5.3.3 Algoritmo MWF-Tree . . . 70
5.4 Algoritmos para Processadores com Memória Compartilhada . . . 73
5.4.1 Algoritmo Aspiration Search . . . 73
5.4.2 Algoritmo MWF . . . 75
5.4.3 Algoritmo Parallel SCOUT . . . 78
5.4.4 Algoritmo KNM . . . 78
5.5 A Paralelização do Algoritmo SSS* . . . 81
5.5.1 Extensão do Conceito de Janela de Busca para SSS* . . . 83
5.5.2 Utilização de Valores Temporários do SSS* . . . 84
5.5.3 Algoritmo Pool-SSS* . . . 85
5.6 Considerações sobre a Implementação dos Algoritmos Paralelos . . . 86

**6 Análise dos Algoritmos Paralelos 87**
6.1 Métodos de Comparação de Algoritmos Paralelos . . . 87
6.2 Resultados Analíticos . . . 89
6.2.1 Análise de Aspiration Search . . . 89
6.2.2 Análise de Tree-Splitting . . . 91
6.2.3 Análise de MWF-Tree . . . 92
6.3 Resultados Experimentais . . . 92
6.3.1 Experimentos com Tree-Splitting . . . 93
6.3.2 Experimentos com PV-Splitting . . . 93
6.3.3 Experimentos com KNM . . . 94
6.3.4 Experimentos com SSS* - I/II . . . 96
6.4 Avaliação da Eficiência do Método de Redução de Janelas . . . 96
6.5 Avaliação da Eficiência do Método de Priorização do Trabalho Obrigatório . . . 101
6.6 Observações sobre Algoritmos para Processadores com Memória Compartilhada 107

**7 Conclusão** **108**

7.1 A Componente Serial do Algoritmo Alpha-Beta . . . . . . . . 108

7.2 A Dependência entre o Método de Paralelização e o Tipo de Corte Obtido . . . 109

7.3 A Influência do Modelo de Árvore na Escolha do Algoritmo Paralelo . . . . . . . 111

7.4 A Relação Tempo × Comunicação . . . . . . . . 112

7.5 A Relação Espaço × Comunicação . . . . . . . . 113

7.6 Problemas em Aberto . . . . . . . . 114

7.7 Observações Finais . . . . . . . . 116

**A Metodogia Experimental** **118**

A.1 Geração das Árvores . . . . . . . . 118

A.2 A Máquina WORM . . . . . . . . 120

A.3 A Árvore de Processadores . . . . . . . . 121

A.4 A Confecção das Tabelas . . . . . . . . 123

# Algoritmos Paralelos para Busca em Árvores de Jogo

*Yes, I believe that is possible (in principle) for me to be beaten by a computer... But they have a long way to go.*

*Bobby Fischer, 1972*

# Capítulo 1

# Introdução

Jogos têm sido, por várias centenas de anos, objeto de fascínio e interesse da humanidade. Jogos de azar, de inteligência, de perspicácia, de amor e de guerra têm proporcionado aos homens momentos de diversão e prazer ao longo de gerações e gerações de jogadores.

Talvez uma propriedade fundamental atraia tanto os homens para o jogo: ele é sempre *desvinculado da realidade*. Um jogo é, normalmente, um conjunto de regras sem qualquer justificativa concreta que não a de manter o jogo interessante. Trata-se de limitar os sentidos e os movimentos, explodir o número de alternativas muito além da capacidade humana, obliterar de todas as maneiras o julgamento baseado no senso comum. O interessante em um jogo é a dificuldade — e o desafio — que ele proporciona aos contendores.

Outrossim, esta característica desvinculação da realidade torna os jogos uma atividade tentadoramente passível de mecanização. É razoável supor que deve ser mais difícil automatizar uma atividade que envolva experiência, sensações e conhecimento acumulado, do que um jogo, baseado que é em regras e estruturas bem definidas e sem contrapartida concreta. Além disso, a complexidade e a diversidade de fatores que intervém em um jogo são drasticamente inferiores às presentes em qualquer outra atividade humana.

## 1.1 Jogos e Máquinas

Em 1769, o Barão Wolfgang von Kempelen, gentil-homem da Bratislava, Hungria, criou um Autômato Jogador de Xadrez que consistia em uma caixa do tamanho de uma mesa, dotada de complexos dispositivos mecânicos (?), na qual se via uma figura de metal, vestida de turco, sentada defronte a um tabuleiro de xadrez. O Autômato mexia as peças do tabuleiro através de um braço mecânico, e quase sempre derrotava seus oponentes, entre os quais figurou o Imperador Napoleão II. Mais tarde, um empresário de nome Maelzel comprou o engenho (e seu segredo) e com ele excursionou por toda a Europa e América.

No ensaio "O jogador de xadrez de Maelzel", Edgar Allan Poe dissecou a operação de exame do interior da máquina feita preliminarmente a toda exibição por Maelzel e chegou a conclusão que devia se tratar de um truque. Em 1827, dois garotos, espiando por uma fresta, comprovaram a fraude: um homem, de nome Schlumberger, exímio enxadrista, habilmente se escondia no interior do Autômato. A pretensa "máquina" foi destruída em 1854 por um incêndio no Museu Chinês da Filadélfia, pouco após a morte de Maelzel e de Schlumberger.

O sonho de máquinas automáticas de jogar xadrez contaminou também o inventor inglês

Charles Babbage, idealizador dos primeiros computadores mecânicos. Em 1864, afirmou que sua *máquina analítica* [1], se construída e adequadamente programada, seria capaz de jogar xadrez, sem, contudo, mostrar claramente como isso poderia ser feito.

No século passado, jogadores mecânicos não povoavam somente os sonhos das cabeças dos homens: eles também faziam parte de seus pesadelos. Publicado em 1894, o conto "O feitiço e o feiticeiro", do jornalista americano Ambrose Bierce [2], relata a estória de um inventor chamado Moxon, que cria um robô capaz de jogar xadrez. Certa feita, o brilhante inventor derrota sua criatura em uma partida, que resolve, então, simplesmente, estrangular seu criador. Um amigo (e confidente) descreve a cena da morte: *" ... Moxon por baixo, a garganta ainda nas garras daquela mão de ferro, a cabeça empurrada para trás, os olhos saltados, a boca escancarada e a língua de fora; e — horrendo contraste! — na cara pintada do assassino uma expressão pensativa, tranqüila e profunda, de quem contempla a solução de um problema de xadrez! "* [3]

Com uma visão menos horripilante do problema, e tendo em perspectiva os primeiros computadores eletrônicos, diversos pesquisadores de cibernética trabalharam na década de 40 no assunto, entre eles John von Neumann, Oskar Morgenstern e Norbert Wiener. Em 1950, Claude Shannon lançou os conceitos básicos para a programação de jogos em um computador. Praticamente ao mesmo tempo, Alan Turing publicou uma estratégia, baseada no valor das peças e na sua posição no tabuleiro, que permitia, através de cálculos feitos à mão, a determinação do próximo lance a ser feito, dada uma certa configuração. A estratégia era bastante simples, e o jogo obtido por meio dela, bastante fraco.

O primeiro programa efetivamente executado em um computador foi elaborado por James Kister, Paul Stein, Stanislaw Ulam, William Walden e Mark Wells, para o MANIAC-I. Por problemas de memória, o programa jogava xadrez em um tabuleiro de $6 \times 6$ casas, sem bispos. Coube ao programa de Alex Bernsteinm, Michael de V. Roberts, Thomas Arbuckle e Martin A. Belsky a primazia de ser o primeiro a jogar com o tabuleiro normal, rodando em um IBM 704 (à válvula). Como era de se esperar, os dois programas eram bastante ruins, conseguindo derrotar, no máximo, pessoas que tinham aprendido xadrez muito recentemente.

Dois trabalhos, no fim da década de 50, foram decisivos para o lançamento das bases dos modernos programas de jogo. O primeiro, realizado em 1955 e publicado em [NSS58], de autoria de Alan Newell, John Shaw e Herbert Simon, foi fundamental no estabelecimento de princípios orientadores para o jogo, bem como para o desenvolvimento dos primeiros algoritmos eficientes. O segundo trabalho, publicado por A.L. Samuel em 1959 ([Sam59]), utiliza, para um jogo de damas, diversos conceitos novos, incluindo o uso de *aberturas* pré-programadas e de aprendizado automático de variações ganhadoras. A este programa coube a honra de derrotar, pela primeira vez, um jogador "de nível", o ex-campeão de damas de Connecticut, Robert W. Nesley.

Por volta de 1980, vários programas já conseguiam performances em torneios mistos (homens e computadores) que os colocavam em uma posição superior a 99% dos jogadores humanos de xadrez. Em 1977, o mestre internacional David Levy desafiou e derrotou um dos mais sofisticados programas da época — o *Chess 4.7* da Northwestern University — elaborado por Christopher Evans, a quem o enxadrista mais tarde confidenciou que os movimentos da máquina eram tão astuciosos que ele tivera problemas imaginando se não se tratava de uma farsa, e que

[1] Do original, em inglês, *analyctical engine*.

[2] O conto "O feitiço e o feiticeiro", de Ambrose Bierce, pode ser encontrado na antologia *Máquinas que Pensam*, editada no Brasil pela L&PM.

[3] Extraído de "Máquinas que Pensam", L&PM, p. 23-24.

ele, na verdade, estaria jogando contra um homem.

Na última década, diversos programas para xadrez, damas e outros jogos têm sido elaborados e postos à prova, tanto em campeonatos exclusivos de computadores como em campeonatos mistos. Toda universidade americana que possui um bom departamento de computação busca, anualmente, conquistar o Campeonato Mundial de Xadrez por Computador. Esta láurea, de certa forma, é encarada como uma demonstração de competência e excelência da universidade, sendo alvo de intensa e exaustiva disputa.

O último Campeonato Mundial de Xadrez entre Computadores foi realizado de 27 a 31 de maio de 1989, em Edmonton, Canadá, com 24 concorrentes. O grande vencedor foi o programa *"Deep Thought"*, de Carnegie-Mellon University, capaz de analisar, em média, 720 mil lances por segundo.

Além desta conquista, o programa *"Deep Thought"* venceu, recentemente, o Torneio Aberto de Los Angeles (6,5 pontos em 8 possíveis), incluindo-se aí uma vitória, jogando com as pretas, sobre o Grande Mestre Bent Larsen (33º do *ranking* mundial). Em 1987, o atual Campeão Mundial de Xadrez, o soviético Gary Kasparov, afirmou que "nunca" um computador venceria um Grande Mestre de primeira linha. Em vista da derrota de Larsen, o campeão fez, recentemente, uma dura autocrítica — ao melhor estilo soviético — declarando que, embora não saiba quem será o 14º Campeão Mundial de Xadrez (Kasparov é o 13º), está seguro de que o 15º não será humano.

De qualquer forma, é bastante provável que seja necessária ao menos toda a próxima década até que um computador — ou melhor, seu programador — consiga ganhar o prêmio de US$100 mil, oferecido pelo Massachusetts Institute of Technology, ao primeiro programa que vencer uma partida contra um Campeão Mundial de Xadrez.

## 1.2 Computadores Paralelos

Apesar de toda a pesquisa realizada em torno de programas para jogos, a sua performance, comparada com a de bons jogadores, é ainda bastante inferior. Um dos elementos limitantes do aperfeiçoamento da qualidade do jogo é a potência computacional. Mesmo com os supercomputadores, que conseguem operar a taxas de vários Gigaflops [4], a quantidade de trabalho exigida pelos programas superaria várias vezes esse valor. Este é um problema que atinge diversas áreas da computação, onde cada vez mais *"... aplicações demandam computadores que são muitas* ordens de magnitude *mais rápidos que os mais rápidos computadores hoje disponíveis."* [5]

Na medida em que se esgotam as possibilidades de aumento de desempenho dos computadores convencionais por meios puramente eletrônicos, a idéia de utilizar múltiplos processadores vem se tornando a única saída viável, técnica e economicamente, para o problema. Muitos pesquisadores sustentam que está se iniciando *a década do computador paralelo*, e que a tão decantada 5ª geração dos computadores é a geração do paralelismo.

É um campo de pesquisa ainda novo: o primeiro computador paralelo, o Illiac IV, é de 1975, e o Cray-I foi lançado em 1976. Aos poucos, porém, a tecnologia vai tomando conta do mercado mundial, e hoje pode-se encontrar disponíveis no mercado desde uma Connection Machine (65536 processadores), ao preço de US$3 milhões, até multiprocessadores de baixa

[4] Bilhões de operações em ponto flutuante por segundo.

[5] Traduzido de [Qui87], p. 1.

potência a um custo semelhante ao de uma estação de trabalho profissional.

Enquanto o *hardware* parece estar dando passos pequenos, porém seguros, na direção do paralelismo, o mesmo não se pode falar a respeito do *software*. Se o problema de escrever e corrigir algoritmos já é delicado em computadores convencionais, o advento do paralelismo deve causar um verdadeiro pandemônio na engenharia de *software*. Em parte porque pensar em múltiplos processos trabalhando juntos é inerentemente complicado para a mente humana, mas também porque várias das técnicas conhecidas, compreendidas e assimiladas ao longo de 30 anos de computação deixam de ser válidas, enquanto que, concomitantemente, métodos e algoritmos considerados ineficientes tornam-se atrativos por apresentarem fortes possibilidades de paralelização.

Embora a criação de programas para jogos esteja longe de ser uma prioridade de mercado, é interessante notar que os primeiros algoritmos paralelos para tal finalidade datam de 1978, exatamente quando os primeiros computadores paralelos tornaram-se disponíveis. Desde então, diversos algoritmos surgiram e foram analisados, explorando tanto velhas idéias dos algoritmos seqüenciais como novos conceitos e sistemas.

De certa forma, pode-se dizer que as máquinas paralelas parecem reaquecer as esperanças de se conseguir programas de excelente performance em jogos.

## 1.3 Objetivos da Dissertação

Considerando a variedade e diversidade de algoritmos paralelos para jogos que foram propostos ao longo desta década, é oportuna a realização de um estudo mais profundo e exaustivo sobre os mesmos. Deste modo, o principal objetivo desta dissertação de mestrado é **sistematizar e avaliar os algoritmos paralelos para jogos, determinando as propriedades fundamentais nas quais seu funcionamento é baseado.**

Durante o processo de elaboração da tese, foi-se também percebendo que vários problemas que apareciam não eram típicos de jogos, mas sim de qualquer processo de transformação de algoritmos seqüenciais para algoritmos paralelos (processos de *paralelização*). Dessa forma, objetiva-se também, através da análise do caso de algoritmos para jogos, **estudar problemas e propriedades emergentes em processos genéricos de paralelização, bem como suas conseqüências típicas em termos de desempenho e eficiência.**

É claro que não se pretende aprofundar demais este último tópico, na medida que por si só ele já seria suficientemente amplo para um dissertação inteira. Entretanto, crê-se que as observações feitas a respeito de processos genéricos de paralelização estejam entre as mais importantes conclusões deste trabalho, tendo em vista a crescente importância do desenvolvimento e análise de algoritmos paralelos.

Por outro lado, vale a pena ressaltar que programas de jogos têm cada vez maior importância comercial, justificando-se assim toda tentativa de assimilação dessa tecnologia, tanto a nível de programas simples para microcomputadores como de gigantescos programas campeões de xadrez. Acrescente-se ainda que os algoritmos para jogos descritos nesta dissertação podem ser empregados na resolução de problemas de outras áreas importantes da computação, como, por exemplo:

- Reconhecimento de Padrões Seqüenciais , usado na determinação de caracteres e em diagnósticos médicos, conforme descrito em [SL71];

- Análise Lingüística de Ondas , com aplicações em reconhecimento de linguagem falada e em redução de ruídos; detalhes e referências podem ser encontrados em [SK83].

## 1.4 Organização da Dissertação

A fim de se conseguir realmente avaliar os algoritmos paralelos para jogos, é necessário antes percorrer um longo caminho. É fundamental uma prévia compreensão dos algoritmos seqüenciais, bem como de suas limitações e propriedades, pois são os desdobramentos destas que determinarão a estrutura fundamental dos algoritmos paralelos. Um certo rigor matemático será utilizado, inclusive com demonstrações de corretude e de dominância, pois algumas características cruciais do problema só são reveladas por uma análise matemática mais profunda.

Os três primeiros capítulos (exclusive este) têm a finalidade de dissecar o mais possível o problema e as suas soluções seqüenciais. O capítulo 2 busca um melhor entendimento do problema e de sua complexidade intrínseca, utilizando uma abordagem não convencional, porém mais esclarecedora na opinião do autor. No capítulo 3 são apresentados os principais algoritmos seqüenciais, sendo suas corretudes demonstradas e algumas de suas propriedades analisadas. O capítulo 4 é dedicado à análise desses algoritmos, que têm então suas relações de dominância averiguadas, seguindo-se a enumeração dos principais resultados analíticos conhecidos. Estes últimos são bastante elucidativos da real natureza do problema e do significado da busca de sua solução.

Os capítulos 5 e 6 são integralmente dedicados aos algoritmos paralelos e constituem-se, por assim dizer, no "coração" desta dissertação. O capítulo 5 começa descrevendo os diversos tipos de arquiteturas paralelas, e segue examinando os dois principais métodos de paralelização identificados pelo autor (baseados em duas características fundamentais do problema). Com base nesses métodos, os diversos algoritmos paralelos são apresentados e explicados, incluindo-se aqui um algoritmo original deste trabalho. Por seu lado, o capítulo 6 começa com a descrição das metodologias de comparação utilizadas e com a análise dos poucos resultados, teóricos e experimentais, existentes na literatura. A seguir são apresentados os resultados das simulações computacionais realizadas pelo autor, concebidas com o intuito de avaliar a eficiência dos dois métodos de paralelização identificados.

No capítulo 7 é feita a conclusão do trabalho, onde são expostas e comentadas diversas relações importantes encontradas, incluindo-se também uma enumeração dos problemas em aberto levantados ao longo da pesquisa. A dissertação inclui ainda um apêndice que descreve a metodologia e o sistema utilizados nas simulações, e a bibliografia completa do trabalho.

# Capítulo 2

# Busca em Árvores de Jogo

Neste trabalho, a palavra *jogo* é utilizada em um sentido mais restrito que o usual, referindo-se apenas a *jogos finitos entre dois jogadores*, nos quais inexista a presença do acaso.

Todos os algoritmos que serão apresentados a seguir aplicam-se somente a jogos de dois jogadores como, por exemplo, o xadrez e as damas; não é óbvia a sua extensão a um número maior de participantes (como o pôquer), ou a jogos com acaso (como o gamão). A restrição de que os jogos sejam finitos é fundamental do ponto de vista teórico, ainda que possa não ocorrer na prática em jogos como o xadrez. Contudo, estes jogos infinitos normalmente possuem um tal grau de explosão exponencial no tempo de computação, que mesmo os melhores algoritmos estão limitados ao exame de poucas jogadas à frente, forçando que estimativas de sucesso sejam consideradas como o "resultado" do jogo. Em outras palavras, as limitações do processo computacional modificam, de certa forma, esses jogos infinitos, transformando-os em variações finitas simplificadas do jogo original.

Por *jogo* se entende todas as possíveis situações geradas por um conjunto de regras, seguidas alternadamente por dois jogadores a partir de uma situação inicial. A palavra *partida* é reservada a uma seqüência particular de lances alternados dos dois jogadores. Por *lance*, entende-se uma ação tomada por qualquer jogador na vez que lhe cabe, seguindo estritamente as regras, e por *jogada*, a seqüência de dois lances, um do 1º jogador, seguido da resposta do 2º jogador.

O objetivo ideal dos algoritmos que serão apresentados é encontrar uma estratégia para o 1º jogador que garanta o melhor jogo *possível* a ele, considerando a situação hipotética em que o 2º jogador atua também de maneira perfeita, e que será, por isso, denominada *estratégia defensiva ótima*. Em poucas palavras, busca-se uma seqüência de lances na qual ambos os jogadores atuam de maneira ótima.

Contudo, devido a limitações de tempo e espaço no processamento, os algoritmos, na prática, buscam somente uma *aproximação* à estratégia defensiva ótima, através de uma busca local, conforme será visto mais adiante.

Conforme ressaltam Knuth e Moore, em [KM75], esta é uma estratégia bastante conservadora, pois não contempla toda a sutileza de um jogo real. No transcurso normal de partidas entre humanos, muitas vezes são feitos movimentos na esperança de erro do adversário, confiando na sua incapacidade de perceber claramente os desdobramentos da posição atual. Surpreendentemente, os autores observam que "*... humanos parecem derrotar programas que jogam xadrez adotando tal estratégia [de confiar no erro dos adversários].*" [1] .

[1] Traduzido de [KM75], p. 296.

## 2.1 Representação de Jogos por Árvores AND/OR

Uma maneira simples e útil para representar um jogo de maneira integral, incluindo todas as suas possíveis variações, é através de árvores AND/OR. Esta representação deriva da originalmente proposta por Stockman em [Sto79].

DEFINIÇÃO 2.1 *Uma* árvore AND/OR $G$ *é uma árvore orientada, não vazia, na qual existem somente dois tipos de nós, denominados* nós AND e nós OR, *de forma que um nó AND possui todos seus sucessores do tipo OR, e vice-versa, e que todo nó (AND ou OR) possui no máximo um único pai.*

Graficamente, representaremos uma árvore AND/OR utilizando quadrados para os nós AND e círculos para os nós OR; as setas, características de árvores orientadas, são suprimidas, e as arestas que partem de um nó OR são interligadas através de um arco, conforme a figura 2.1.

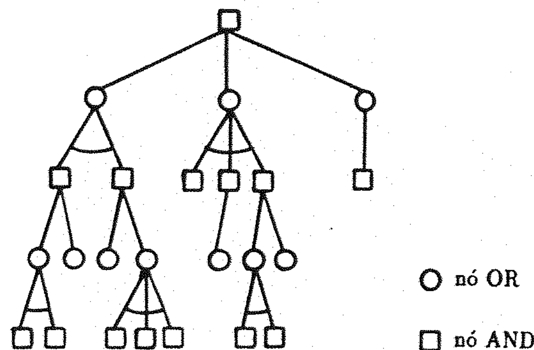



Figura 2.1: Árvore AND/OR.

Na representação de jogos, convenciona-se que os nós identificam posições ou situações do jogo, e as arestas os lances permitidos a partir daquela situação. Uma *partida* é, portanto, qualquer *caminho* da raiz até uma folha. É ainda convencional marcar a raiz como sendo um nó AND.

## 2.2 Valoração dos Nós Terminais

Para se representar convenientemente um jogo, deve-se necessariamente incluir os conceitos de vitória e derrota. Como todo jogo *finito* se reduz a determinação de vitória, empate ou derrota [2], convenciona-se a atribuição dos seguintes valores aos diferentes caminhos (que são as diferentes partidas), de uma árvore AND/OR :

$$\begin{aligned} +\infty &\rightarrow \text{vitória do 1º jogador} \\ 0 &\rightarrow \text{empate} \\ -\infty &\rightarrow \text{derrota do 1º jogador} \end{aligned}$$

[2]Certos jogos constituem-se de muitas "partidas", com contagem de pontos ao final de cada uma delas; neste caso, o objetivo do jogo não é um maior número de pontos a cada "partida", mas sim a vitória na contagem total dos pontos.

Devido à bijeção existente entre caminhos e nós terminais, estes valores são colocados como atribuições dos nós terminais.

Na medida em que, na prática, é geralmente impossível a geração integral da árvore que representa um jogo, opta-se por sua geração parcial e pela determinação de uma estratégia defensiva ótima para esta árvore parcial.

Dessa forma, aos nós terminais destas árvores de jogo parciais são atribuídos valores arbitrários, de modo a permitir o cálculo da estratégia defensiva ótima. Estes valores são, via de regra, determinados *heuristicamente* em função da situação do jogo correspondente ao nó, com valores no intervalo fechado $[-\infty, +\infty]$ (discreto ou contínuo), e de caráter probabilista: a um nó com valor 20 deve corresponder uma posição com maiores chances de vitória do que a de um nó de valor 10.

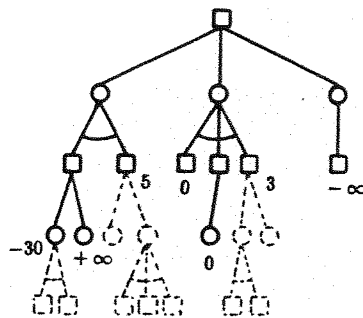



Figura 2.2: Árvore de jogo parcial com valores nos nós terminais.

## 2.3 O Processo de Aproximação à Estratégia Defensiva Ótima

A determinação da estratégia defensiva ótima para uma árvore de jogo parcial é o objetivo de todos os algoritmos vistos neste trabalho. Cumpre-se, no entanto, descrever o algoritmo completo de jogo, que basicamente consiste em :

- determinar, a cada jogada, o lance a ser realizado, através do cálculo da estratégia defensiva ótima da árvore parcial cuja raiz é o nó que representa a situação atual;
- de posse da resposta do adversário (ótima ou não), repete-se o processo, até o fim do jogo.

É evidente que este método não assegura, de forma alguma, que os lances escolhidos correspondam a lances de uma estratégia defensiva ótima *para um jogo completo*. Pelo contrário, é uma típica técnica *hill-climbing* para busca em sistemas *não comutativos*, de acordo com as definições feitas por Nilsson em [Nil82]. Considera-se, contudo, uma aproximação razoável, pois assemelha-se em parte ao processo humano de jogar e tem apresentado bons resultados nas implementações realizadas.

Tipicamente, a árvore de jogo parcial é determinada utilizando-se uma profundidade fixa de jogadas $h$, até a qual todas as possíveis partidas são consideradas, com valores (heurísticos)

atribuídos a elas da maneira vista na seção anterior. Porém, há programas que determinam a árvore parcial de maneira mais informada, utilizando conhecimento específico sobre o jogo, geralmente às custas da diminuição da altura $h$ de busca. Em [ABD82], Akl, Barnard e Doran notam que, dentro da comunidade de Inteligência Artificial, ainda é objeto de polêmica qual é a melhor das duas abordagens.

Existem ainda técnicas de reutilização do trabalho de busca feito em um lance anterior, como, por exemplo, a utilização de *transposition tables*, *killer heuristics* e *iterative deepening*, cujas descrições podem ser encontradas em [MC82]. Contudo, tais dispositivos não serão aqui considerados, e assim pode-se definir mais precisamente que o objetivo deste trabalho é, essencialmente, estudar *algoritmos para determinação do melhor lance de uma árvore de jogo parcial, considerando-se uma estratégia defensiva ótima.*

Em vista do exposto, se passará, a partir de agora, a denominar uma árvore de jogo parcial simplesmente de *árvore de jogo*, para a qual são atribuídos, a todos os nós terminais, valores no intervalo $[-\infty, +\infty]$ (discreto ou contínuo). O processo de determinação do melhor lance de uma árvore nessas condições será denominado, genericamente, *busca em árvore de jogo.*

## 2.4 Árvores Solução

Escrevendo formalmente o que foi enunciado acima,

DEFINIÇÃO 2.2 *Uma* árvore de jogo $G$ *é um par* $(\overline{G}, c)$, *onde* $\overline{G}$ *é uma árvore AND/OR com nós terminais somente do tipo AND, e* $c : TIPS(G) \mapsto [a, b]$ *é uma função qualquer de*

$$TIPS(G) = \{t \in V(\overline{G}) \mid t \text{ é terminal}\}$$

*em*

$$\begin{cases} [a,b] \subseteq \mathcal{R} \cup \{-\infty, +\infty\} & \text{(caso contínuo)} \\ \text{ou} & \\ [a,b] \subseteq \mathcal{Z} \cup \{-\infty, +\infty\} & \text{(caso discreto)} \end{cases}$$

*onde* $V(\overline{G})$ *denota o conjunto de nós de* $\overline{G}$.

Um exemplo de árvore de jogo é dado pela figura 2.3.

A restrição ao tipo AND dos nós terminais é extremamente útil, conforme se verá a seguir, e não se constitui em problema, visto que toda posição terminal OR pode ser estendida a uma posição terminal AND através de uma aresta "boba", conforme a figura 2.4.

Para tentar capturar o ponto de vista de cada um dos dois jogadores, utilizam-se os conceitos de *AND-árvore solução* e *OR-árvore solução*, que representam um conjunto de possíveis lances de cada um dos jogadores, considerando-se todas as respostas de seu adversário, conforme as seguintes definições:

DEFINIÇÃO 2.3 *Uma* AND-árvore solução $T$ *de uma árvore de jogo* $G$ *é uma sub-árvore de jogo contida em* $G$, *na qual:*

*i . A raiz* $p$ *da sub-árvore* $T$ *é a mesma da árvore* $G$;

*ii . Se um nó não-terminal* $n$ *de* $G$ *está em* $T$, *então*

- *se* $n$ *é do tipo AND, exatamente* <u>*um*</u> *de seus sucessores está em* $T$;
- *se* $n$ *é do tipo OR,* <u>*todos*</u> *os seus sucessores estão em* $T$.

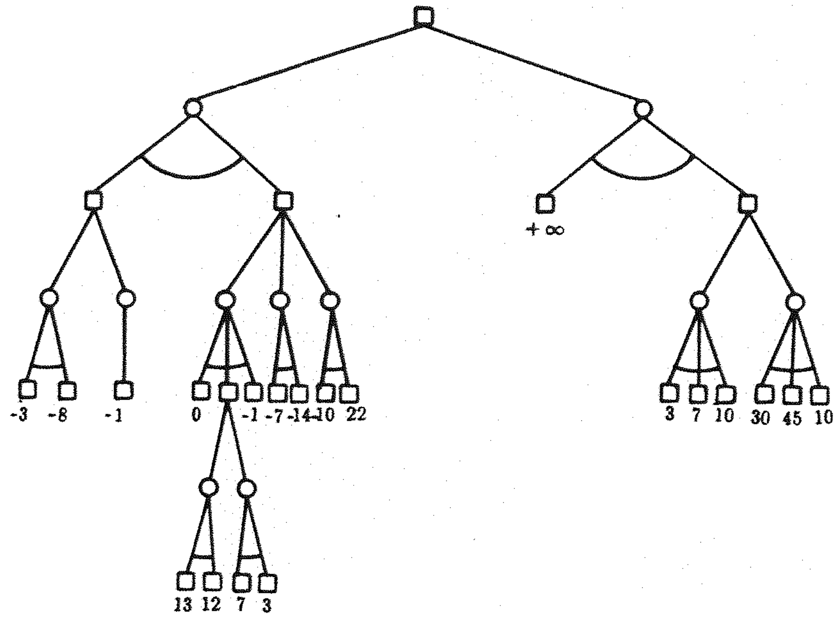



Figura 2.3: Árvore de jogo.

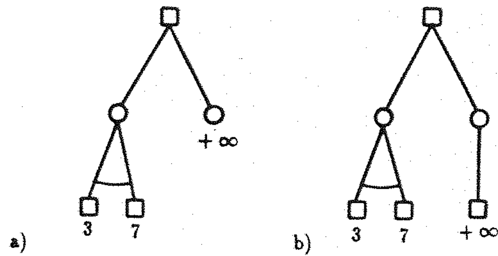



Figura 2.4: a)Árvore AND/OR com posição terminal OR; b)Árvore AND/OR equivalente.

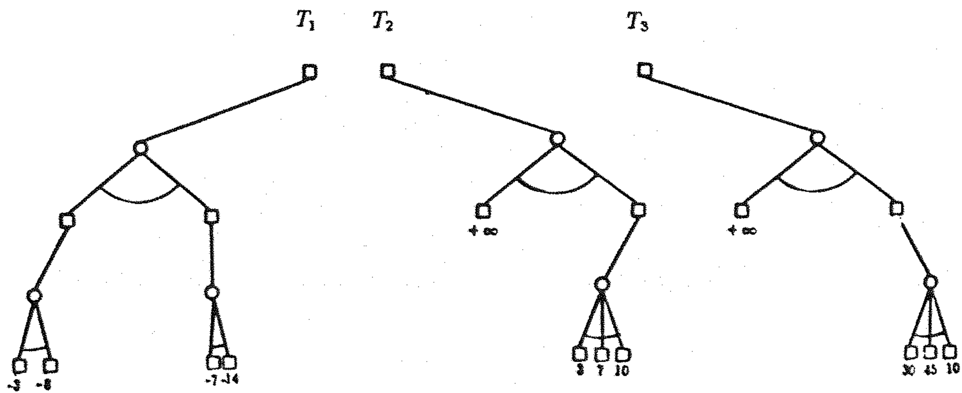



Figura 2.5: Algumas AND-árvores solução.

A figura 2.5 mostra algumas árvores solução da árvore de jogo da figura 2.3. Pode-se perceber que uma AND-árvore solução representa uma estratégia fixa do 1º jogador, com todas as possíveis respostas do seu adversário. Note-se que, dada uma árvore solução $T$, qualquer nó terminal de $T$ pode ser atingido por uma seqüência apropriada de lances do 2º jogador. Portanto, se este último jogar de maneira ótima em resposta à estratégia do 1º, o nó terminal de *menor valor* será atingido. Este valor é associado então à AND-árvore solução T, pois representa quanto vale essa estratégia para o 1º jogador, considerando um adversário ótimo. Formalmente, usando a notação proposta por Kumar e Kanal em [KK84],

DEFINIÇÃO 2.4 *Seja $T$ uma AND-árvore solução de uma árvore de jogo $G$. Então o* mérito $f$ *de $T$ é*

$$f(T) = \min\{c(t) \mid t \in TIPS(T)\}$$

Para as AND-árvores solução da figura 2.5, temos :

$$f(T_1) = -14 \qquad f(T_2) = 3 \qquad f(T_3) = 10$$

Analogamente, definem-se OR-árvores solução, correspondendo às estratégias do 2º jogador. Exemplos na figura 2.6.

DEFINIÇÃO 2.5 *Uma* OR-árvore solução $\overline{T}$ *de uma árvore de jogo $G$ é uma sub-árvore de jogo contida em $G$, na qual:*

*i . A raiz $p$ da sub-árvore $\overline{T}$ é a mesma da árvore $G$;*

*ii . Se um nó não-terminal $n$ de $G$ está em $\overline{T}$, então*

- *se $n$ é do tipo OR, exatamente um de seus sucessores está em $\overline{T}$;*
- *se $n$ é do tipo AND, todos os seus sucessores estão em $\overline{T}$.*

DEFINIÇÃO 2.6 *Seja $\overline{T}$ uma OR-árvore solução de uma árvore de jogo $G$. Então o* mérito $\overline{f}$ *de $\overline{T}$ é*

$$\overline{f}(\overline{T}) = \max\{c(t) \mid t \in TIPS(\overline{T})\}$$

É importante destacar a relação entre AND e OR-árvores solução. Na medida em que o mérito de uma AND-árvore solução representa o melhor valor alcançável pelo 1º jogador segundo uma dada estratégia, é bastante razoável supor que este valor seja menor ou igual a qualquer estratégia do 2º jogador, sugerindo então,

LEMA 2.7 *Seja $G$ uma árvore de jogo, $T$ uma AND-árvore solução de $G$ e $\overline{T}$ uma OR-árvore solução de $G$. Então*

$$f(T) \leq \overline{f}(\overline{T})$$

DEMONSTRAÇÃO : basta mostrar que $T$ e $\overline{T}$ têm um nó terminal em comum. Para tanto, seja $t_0$ a raiz de $T$ e, portanto de $\overline{T}$. Sem perda de generalidade, suponha $t_0$ do tipo AND e seja $t_1$ o único sucessor de $t_0$ em $T$. Como $t_0$ é um nó AND, todos os seus sucessores estão na OR-árvore solução $\overline{T}$, e em particular, $t_1 \in \overline{T}$. Analogamente, seja $t_2 \in \overline{T}$ o único sucessor de $t_1$ em $\overline{T}$, e

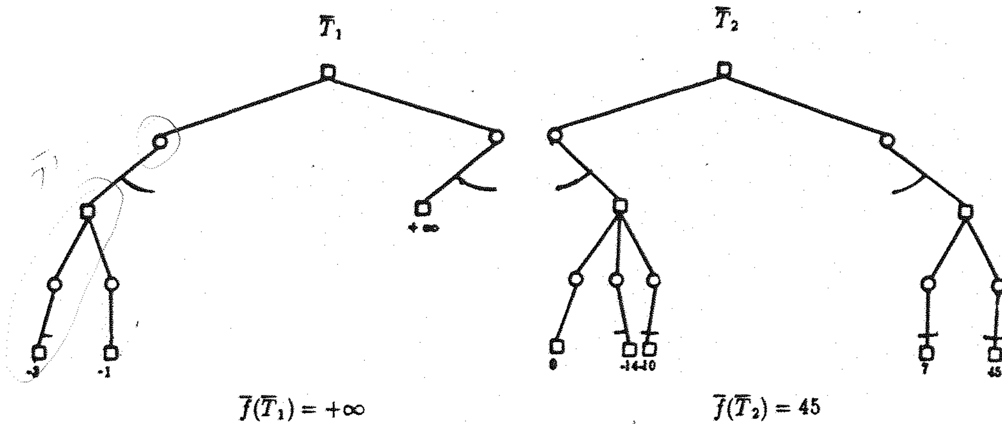



Figura 2.6: Algumas OR-árvores solução.

como $t_1$ é um nó do tipo OR, $t_2$ pertence também a $T$. Desta forma, é possível construir um caminho $(t_0, t_1, \ldots, t_h)$ contido em $T \cap \overline{T}$, onde $t_h$ é nó terminal. Lembrando que

$$c(t_h) \leq \max\{c(t) \mid t \in TIPS(\overline{T})\} = \overline{f}(\overline{T})$$

e

$$c(t_h) \geq \min\{c(t) \mid t \in TIPS(T)\} = f(T)$$

conclui-se

$$f(T) \leq \overline{f}(\overline{T})$$

□

## 2.5 Estratégia Defensiva Ótima

De posse dos conceitos de árvore solução e de mérito, pode-se então definir formalmente o conceito de estratégia defensiva ótima.

DEFINIÇÃO 2.8 *Seja $G$ uma árvore de jogo; uma* AND-estratégia defensiva ótima *de $G$ é uma AND-árvore solução $T$ tal que, para toda AND-árvore $T'$ de $G$*

$$f(T) \geq f(T')$$

Evidentemente, toda árvore de jogo possui alguma AND-estratégia defensiva ótima, e ela pode não ser única. Como exemplo, na figura 2.5, a AND-árvore $T_3$ é uma AND-estratégia defensiva ótima, o que pode ser verificado pela exaustão de todas as possibilidades.

Analogamente para OR-árvores,

DEFINIÇÃO 2.9 *Seja $G$ uma árvore de jogo; uma* OR-estratégia defensiva ótima *de $G$ é uma OR-árvore solução $\overline{T}$ tal que, para toda OR-árvore $\overline{T}'$ de $G$, temos*

$$\overline{f}(\overline{T}) \leq \overline{f}(\overline{T}')$$

Como se está lidando com estratégias defensivas, é de se esperar que a melhor estratégia do 1º jogador tenha o mesmo mérito da melhor estratégia do 2º jogador, conforme

TEOREMA 2.10 *Seja $G$ uma árvore de jogo, $T$ uma AND-estratégia defensiva ótima e $\overline{T}$ uma OR-estratégia defensiva ótima. Então*

$$f(T) = \overline{f}(\overline{T})$$

DEMONSTRAÇÃO : por indução na altura $h$ de $G$. Se $h$=0, então $G$ é constituído por um único nó $t_0$, que é, ao mesmo tempo, as únicas AND e OR-árvores solução, e portanto com méritos idênticos.

Suponha, por indução, $h$>0, e a proposição válida para toda árvore de jogo de altura $h' \leq h-1$. Seja $G$ uma árvore de jogo de altura $h$ e raiz $t$ :

**1º caso :** $t$ é do tipo AND.
Sejam $t_1, t_2, \ldots, t_d$, $d$>0, os sucessores de $t$, e $G_1, G_2, \ldots, G_d$ as árvores de jogo cujas raízes são exatamente $t_1, t_2, \ldots, t_d$ . Pela hipótese de indução, se $T_i$ é uma AND-estratégia defensiva ótima e $\overline{T_i}$ é uma OR-estratégia defensiva ótima para cada árvore $G_i$, $1 \leq i \leq d$, então

$$f(T_i) = \overline{f}(\overline{T_i})$$

Existe $j$, $1 \leq j \leq d$ tal que $f(T_j) \geq f(T_i)$, para todo $i$, $1 \leq i \leq d$. É fácil ver que a AND-árvore solução $T$ que liga $t$ à $T_j$ é uma AND-estratégia defensiva ótima, e que

$$f(T) = \max\{f(T_1), f(T_2), \ldots, f(T_d)\} = f(T_j)$$

Por outro lado, a OR-árvore solução $\overline{T}$ que liga $t$ a todas as outras OR-árvores $\overline{T_i}$ , $1 \leq i \leq d$, é uma OR-estratégia defensiva ótima, onde, pela definição de $\overline{f}$,

$$\overline{f}(\overline{T}) = \max\{\overline{f}(\overline{T_1}), \overline{f}(\overline{T_2}), \ldots, \overline{f}(\overline{T_d})\}$$

Como $f(T_i) = \overline{f}(\overline{T_i})$, $1 \leq i \leq d$,

$$f(T) = \overline{f}(\overline{T})$$

**2º caso :** $t$ é do tipo OR, de maneira estritamente análoga. □

Dado que os valores de estratégias defensivas ótimas AND e OR de uma mesma árvore de jogo $G$ são iguais, pode-se associar este valor à própria árvore de jogo; formalmente,

DEFINIÇÃO 2.11 *O* mérito $m$ *de uma árvore de jogo $G$ é o mérito de uma AND-estratégia defensiva ótima $T$, ou, equivalentemente, de uma OR-estratégia defensiva ótima $\overline{T}$, expresso por*

$$m(G) = f(T) = \overline{f}(\overline{T})$$

A proposição seguinte é imediata,

PROPOSIÇÃO 2.12 *Seja $G$ uma árvore de jogo, $T$ uma AND-árvore solução de $G$ e $\overline{T}$ uma OR-árvore solução de $G$. Então*

$$f(T) \leq m(G) \leq \overline{f}(\overline{T})$$

DEMONSTRAÇÃO : evidente a partir da combinação do teorema 2.10 , do lema 2.7 e das definições de mérito dadas acima. □

É importante ressaltar que a determinação do mérito de uma árvore de jogo é um problema que, de certa forma, "contém" o problema já colocado de determinar, a cada jogada, o próximo lance a ser realizado. Para tanto, basta que se considere como melhor lance a aresta que sai da raiz e que pertença a um caminho principal, isto é, a um caminho que ligue a raiz a um nó terminal $t$ cujo valor $c(t)$ é igual ao valor de uma AND ou OR-estratégia defensiva ótima.

Por outro lado, o conhecimento do melhor lance em um conjunto de nós irmãos não é suficiente para determinar o melhor lance de seu pai, ao passo que quando se tem o mérito das sub-árvores de raiz nos nós irmãos pode-se facilmente determinar o mérito da árvore cuja raiz é o pai desses nós. Assim, calcular o mérito de uma árvore de jogo é sempre uma maneira mais informada do que a simples determinação do próximo lance a ser realizado. Além disso, o valor do mérito pode ser usado — e o é — para diminuir a quantidade de cálculo necessária, conforme será visto a seguir.

Todos os algoritmos que serão apresentados neste trabalho calculam somente o mérito de uma árvore de jogo. No entanto, será fácil ver que modificações triviais nos mesmos possibilitam-lhes também a determinação do melhor lance a partir da raiz.

## 2.6 O Limite Inferior

Uma característica importante de árvores de jogo é que, para determinar seu mérito *não é sempre necessário o exame do valor de todos os seus nós terminais.* De fato, o principal objetivo dos algoritmos de busca em árvores de jogo é exatamente determinar o mérito de uma árvore considerando o menor número possível de nós terminais.

Para se entender como isto é possível, é necessário compreender primeiro o conceito de refutação:

DEFINIÇÃO 2.13 *Seja $G$ uma árvore de jogo com raiz $t$ do tipo AND e $T$ uma AND-árvore solução de $G$. Seja $t'$ um sucessor de $t$ tal que $t' \notin T$, e $G'$ a sub-árvore de jogo de $G$ com raiz em $t'$. Uma OR-árvore solução $\overline{T}^t$ de $G'$ é uma* refutação *de $T$ se, e somente se,*

$$f(T) \geq \overline{f}(\overline{T}^t)$$

.

A figura 2.7 exemplifica uma refutação. Note-se que a existência de uma refutação indica que qualquer AND-árvore solução que passe pelo nó $t'$ terá um valor menor que o de $T$, pois nesse nó o outro jogador já possui uma estratégia (representada pela OR-árvore $\overline{T}^t$) que garante a ele pelo menos um valor menor que $T$. Formalmente,

PROPOSIÇÃO 2.14 *Seja $G$ uma árvore de jogo com raiz $t$ do tipo AND e sucessores $t_1, t_2, \ldots, t_d$, $d>0$. Seja $T$ uma AND-árvore solução de $G$, na qual, sem perda de generalidade, $t_1 \in T$. Sejam $G_i^t$, $2 \leq i \leq d$ as sub-árvores de $G$ com raiz em $t_i$. Para toda sub-árvore $G_i^t$ de $G$, se existe $\overline{T}_i^t \subseteq G_i^t$, refutação de $T$, então*

$$m(G) \geq f(T) \geq m(G_i^t)$$

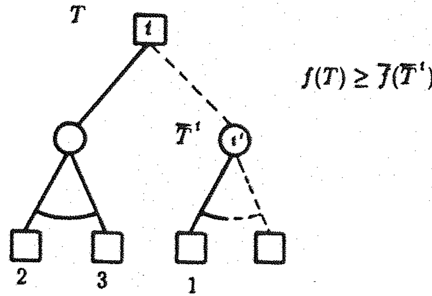



Figura 2.7: AND-árvore solução $T$ e uma refutação $\overline{T}^t$ de $T$.

DEMONSTRAÇÃO : pela proposição 2.12 $\bar{f}(\overline{T}_i^t) \geq m(G_i^t)$. Mas $\overline{T}_i^t$ é refutação de $T$, logo $f(T) \geq \bar{f}(\overline{T}_i^t)$ e portanto $f(T) \geq m(G_i^t)$. Por sua vez, a desigualdade $m(G) \geq f(T)$ é assegurada pela proposição 2.12. □

Este resultado sinaliza uma importante idéia para a determinação de uma condição necessária e suficiente para que uma dada AND-árvore solução seja considerada AND-estratégia defensiva ótima. Para facilitar este enunciado e sua demonstração, será utilizado um tipo especial de árvore de jogo.

DEFINIÇÃO 2.15 *Uma árvore de jogo* uniforme *G é uma árvore de jogo com raiz do tipo AND, altura $h \geq 0$ e grau $d > 0$ constante para todos os seus nós não terminais.*

Dessa forma, toda árvore de jogo uniforme tem altura $h$ par (pela definição 2.2), e tem-se

TEOREMA 2.16 *Seja G uma árvore de jogo uniforme de grau d e altura h, par. Seja T uma AND-árvore solução de G. Então T é uma AND-estratégia defensiva ótima* se, e somente se,

> *Existe um caminho $(t_0, t_1, \ldots, t_h) \subset T$, onde $t_0$ é raiz de $G$ e $c(t_h) = f(T)$ (ou seja, é um* caminho principal *de T) tal que :*
> *Seja $t_j^i$, $0 < i \leq h$, $1 \leq j \leq d$, o j-ésimo irmão do nó $t_i$, de forma que $t_1^i = t_i$; seja ainda $G_j^i$ a sub-árvore de jogo de G de raiz $t_j^i$ , e, se $t_j^i \in T$, seja também $T_j^i$ a sub-árvore de T com raiz em $t_j^i$.*
> *Então, para cada $t_j^i$, i ímpar, $0 < i < h$, $2 \leq j \leq d$, existe uma* refutação *$\overline{T}_j^i \subseteq G_j^i$ de $T_1^{i-1}$.*

DEMONSTRAÇÃO :
($\Longleftarrow$) Seja $T$ uma AND-árvore solução que possue um caminho principal $C = (t_0, t_1, \ldots, t_h) \subset T$ que satisfaz a condição do teorema. Suponha, por absurdo, que $T$ não é estratégia defensiva ótima, ou seja, que existe $V$, AND-árvore solução de $G$ tal que $f(V) > f(T)$. É claro que $t_0 \in V$, e que $t_h \notin V$ (do contrário, $f(V) \leq c(t_h) = f(T)$, absurdo).
Seja $t_j$, $0 \leq j \leq h-2$, $j$ par, tal que $t_j \in V$ mas $t_{j+1} \notin V$. Então existe $k$, $2 \leq k \leq d$ tal que $t_k^{j+1} \in V$. Pela hipótese, existe uma $\overline{T}_k^{j+1}$ refutação de $T_1^j$, conforme mostrado na figura 2.9.

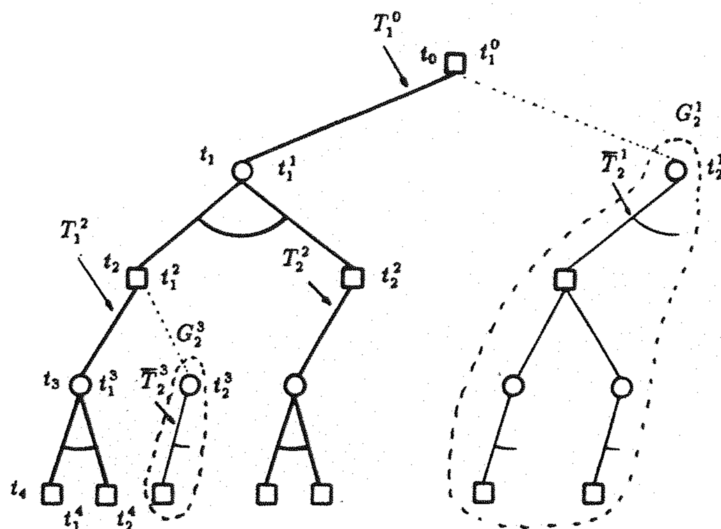



Figura 2.8: Exemplo da notação utilizada.

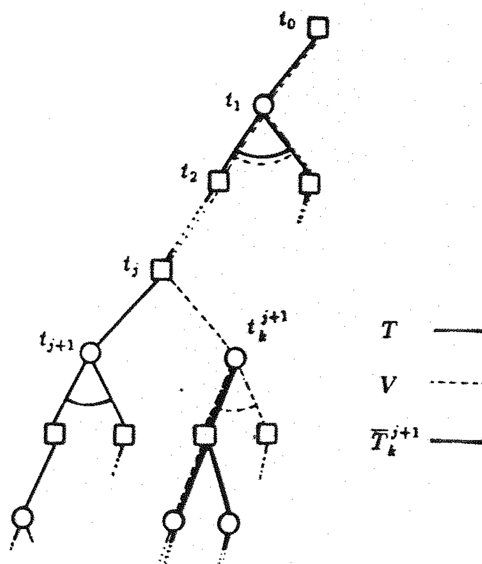



Figura 2.9: Árvore solução $V$ e refutação $\overline{T}_k^{j+1}$.

Pela proposição 2.14, $m(G_1^j) \geq f(T_1^j) \geq m(G_k^{j+1})$. Seja $V_k^{j+1}$ a sub-árvore de $V$ cuja raiz é $t_k^{j+1}$. Pela proposição 2.12, $m(G_k^{j+1}) \geq f(V_k^{j+1})$. Mas o mérito de uma AND-árvore solução é menor ou igual ao mérito de qualquer de suas sub-árvores, logo $f(V_k^{j+1}) \geq f(V)$ e, portanto,

$$f(T_1^j) \geq m(G_k^{j+1}) \geq f(V_k^{j+1}) \geq f(V)$$

Como $T_1^j$ contém o caminho $(t_j, t_{j+1}, \ldots, t_h) \subseteq \mathcal{C}$, então $f(T) = c(t_h) \geq f(T_1^j)$ e, utilizando a desigualdade anterior,

$$f(T) \geq f(V) \text{ , absurdo!}$$

($\Longrightarrow$) Seja $T$ uma AND-estratégia defensiva ótima de $G$. Suponha, por absurdo, que para todo caminho principal $\mathcal{C} = (t_0, t_1, \ldots, t_h) \subset T$, não se verifica a implicação do teorema.
Tome-se então um caminho principal $\mathcal{C}$; como a hipótese não vale, existe $t_j^i$, $i$ ímpar, $0<i<h$, $2 \leq j \leq d$, no qual não existe refutação $\overline{T}_j^i \subseteq G_j^i$ de $T_1^{i-1}$.
Isto significa que toda OR-árvore solução $\overline{T}_j^i$ de $G_j^i$ satisfaz $\overline{f}(\overline{T}_j^i) > f(T_1^{i-1})$, e em particular uma $\overline{T}_j^{i\,\prime}$ estratégia defensiva ótima de $G_j^i$. Como $\overline{f}(\overline{T}_j^{i\,\prime}) = m(G_j^i)$, temos

$$m(G_j^i) > f(T_1^{i-1}).$$

Tome-se então uma <u>AND</u>-estratégia defensiva ótima $T_j^{i\,\prime}$ de $G_j^i$ e construa-se a AND-árvore solução $T\,'$ de $G$, que consiste essencialmente na árvore $T$ onde se substitui $T_1^i$ por $T_j^{i\,\prime}$, conforme mostrado na figura 2.10.

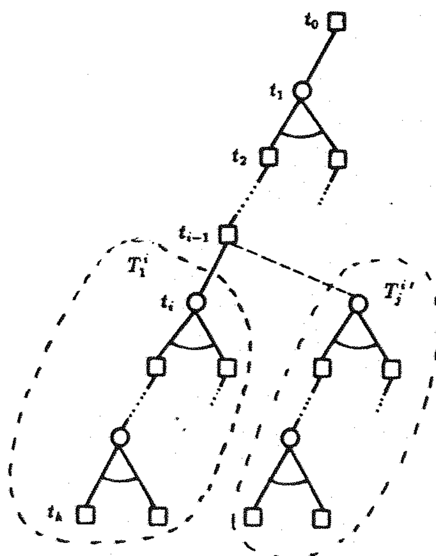



Figura 2.10: Árvores solução $T$ e $T\,'$.

Como, por construção,

$$f(T_j^{i\,\prime}) = m(G_j^i) > f(T_1^{i-1}) = f(T_1^i),$$

e $T$ e $T\,'$ diferem apenas nos ramos $T_1^i$ e $T_j^{i\,\prime}$, respectivamente, então $f(T\,') \geq f(T)$ e *pelo menos um dos caminhos principais de* $T$ não está em $T\,'$. Mais que isto, como $f(T_j^{i\,\prime})$ é *estritamente*

*maior* que $f(T_1^i)$, *nenhum* nó terminal de $T_j^{i\,\prime}$ tem valor igual a $f(T)$, e portanto o número de caminhos principais de $T'$ é *estritamente menor* que o de $T$.
Repetindo-se finitamente o processo para $T'$ até se esgotarem todos os caminhos principais de $T$, será obtida, ao final, uma árvore $T^*$ tal que $f(T^*) > f(T)$, absurdo! □

É fácil perceber que a utilização de árvores de jogo uniformes não influencia em nada a demonstração deste teorema, e que, portanto pode ser omitida da hipótese. Ressalte-se ainda que é possível um raciocínio inteiramente análogo para o caso de uma OR-estratégia defensiva ótima.

Um importante corolário é extraído do teorema 2.16, e que determina o limite inferior de complexidade para algoritmos de busca em árvores de jogo.

COROLÁRIO 2.17 *Seja $G$ uma árvore de jogo uniforme de altura $h$ e grau $d$, e tal que $-\infty < m(G) < +\infty$; seja também $T$ uma AND-árvore solução de $G$. Para garantir que $T$ é uma AND-estratégia defensiva ótima, é necessário, no mínimo, considerar*

$$2d^{\frac{h}{2}} - 1 \text{ nós terminais.}$$

DEMONSTRAÇÃO : relembre-se inicialmente que em uma árvore de jogo uniforme de altura $h$ (que é par) e grau $d$ toda AND-árvore solução e toda OR-árvore solução têm exatamente $d^{\frac{h}{2}}$ nós terminais.
Pois bem, seja $G$ uma árvore de jogo uniforme de altura $h$ e grau $d$ tal que $-\infty < m(G) < +\infty$. Como o mérito de $G$ não é $\pm\infty$, todos os $d^{\frac{h}{2}}$ nós terminais de qualquer AND-árvore solução $T$ têm de ser examinados para o cálculo de $f(T)$.
Além disso, necessita-se garantir que $T$ é realmente uma estratégia defensiva ótima. Suponha conhecido um caminho principal $C = (t_0, t_1, \ldots, t_h) \subset T$ que satisfaz a condição $i$ do teorema 2.16; para se poder afirmar tal fato, terão ainda que ser considerados os nós terminais das $(d-1)$ refutações que devem ser encontradas em cada nível ímpar.

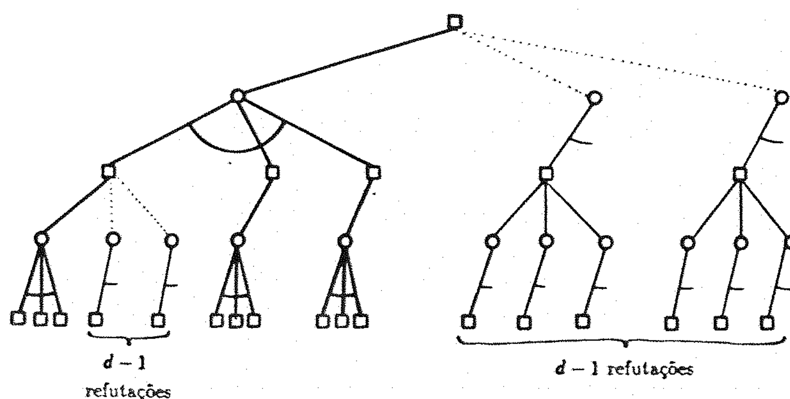



Figura 2.11: Árvore $T$ e as refutações necessárias

Para um dado nível $i$ fixo, haverão a considerar, portanto,

$$(d-1)d^{\frac{h-(i+1)}{2}} \text{nós terminais,}$$

e, portanto, considerando-se todos os níveis ímpares,

$$\sum_{i=1,3,\ldots,h-1} (d-1)d^{\frac{h-(i+1)}{2}} \text{ nós terminais.}$$

Fazendo $j = \frac{h-(i+1)}{2}$, temos,

$$\sum_{j=0}^{\frac{(h-2)}{2}} (d-1)d^j = (d-1)\frac{d^{(\frac{h-2}{2})+1}-1}{(d-1)} = d^{\frac{h}{2}} - 1$$

Somando-se aos $d^{\frac{h}{2}}$ nós terminais de $T$, temos finalmente

$$2d^{\frac{h}{2}} - 1 \text{ nós terminais.}$$

□

Uma versão equivalente a este corolário é creditada por Knuth e Moore a Adelson-Velskiy *et alli* em [KM75]; também aparece em [SD69], sendo que nas demonstrações apresentadas nesses trabalhos é utilizado sempre o conceito de *valor minimax* que será definido a seguir. A formulação deste resultado usando árvores AND/OR, através da determinação de condições necessárias e suficientes, conforme foi feito no teorema 2.16, é original deste trabalho.

Finalmente, não é difícil demonstrar que para o caso em que $m(G)=\pm\infty$ bastam $d^{\frac{h}{2}}$ nós terminais. Conclui-se, portanto, que o problema de busca em árvores de jogo uniformes pertence à categoria dos problemas *intrinsecamente exponenciais.*

## 2.7 Conceitos Equivalentes à Estratégia Defensiva Ótima

Embora o conceito de estratégia defensiva ótima seja um modelo convincente do problema de determinação do próximo lance de um jogo, a literatura técnica do assunto privilegia, via de regra, duas outras abordagens equivalentes. Estas abordagens, ainda que matemática e computacionalmente mais sucintas, foram preteridas nesta abordagem pelo conceito de estratégia defensiva ótima porque este último possibilita uma compreensão do significado de mérito de uma árvore de jogo de um ponto de vista *global.*

Pensando localmente em cada nó, pode-se definir o valor para um dado nó considerando-se que, no caso AND, ao melhor lance corresponde o nó sucessor de valor máximo, e, equivalentemente, no caso OR, o nó sucessor de valor mínimo. Isto sugere a definição de *valor minimax* de uma árvore de jogo, conforme sugerida por Knuth e Moore em [KM75], e exemplificada pela figura 2.12:

DEFINIÇÃO 2.18 *Seja $G$ uma árvore de jogo, e as funções $H : V_{AND}(G) \mapsto [a,b]$ e $I : V_{OR}(G) \mapsto [a,b]$, definidas por :*

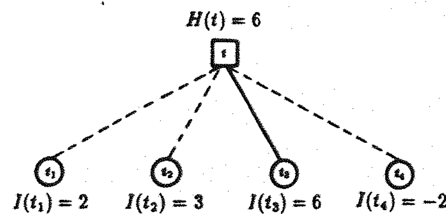
$$H(t) = \begin{cases} c(t) & \text{, se } t \text{ é terminal de } G \\ \max\{I(t_1), I(t_2), \ldots, I(t_d)\} & \text{, se } t_1, t_2, \ldots, t_d \text{ são sucessores de } t \end{cases}$$

$$I(t) = \min\{H(t_1), H(t_2), \ldots, H(t_d)\} \quad , \textit{ se } t_1, t_2, \ldots, t_d \textit{ são sucessores de } t$$

*onde $V_{AND}(G)$ (respectivamente $V_{OR}(G)$) denota o conjunto de nós AND de G (respectivamente OR).*

*Nestas condições, o* valor minimax *de G é, para $t_0$ raiz de G,*

$$minimax(G) = \begin{cases} H(t_0) & se\ t_0 \in V_{AND}(G) \\ I(t_0) & se\ t_0 \in V_{OR}(G) \end{cases}$$



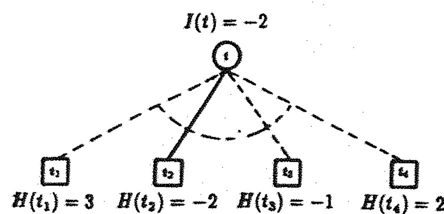




Figura 2.12: Cálculo dos valores $H$ e $I$ de dois nós.

Esta definição pode ser simplificada através de uma abordagem ainda mais local, na qual a alternância de máximos e mínimos é substituída por uma troca de sinal, originando,

DEFINIÇÃO 2.19 *Seja G uma árvore de jogo, $F : V(G) \mapsto [a,b]$ a função :*

$$F(t) = \begin{cases} c(t) & , \textit{se } t \textit{ é terminal de } G \\ \max\{-F(t_1), -F(t_2), \ldots, -F(t_d)\} & , \textit{se } t_1, t_2, \ldots, t_d \textit{ são sucessores de } t \end{cases}$$

*Nestas condições, o* valor negamax *de G é, para $t_0$ raiz de G,*

$$negamax(G) = \begin{cases} F(t_0) & se\ t_0 \in V_{AND}(G) \\ -F(t_0) & se\ t_0 \in V_{OR}(G) \end{cases}$$

É fácil demonstrar, por indução no comprimento da árvore, que, para todo nó $t$ de uma árvore de jogo $G$, vale

$$\begin{aligned} F(t) &= H(t) \quad \text{se } t \in V_{AND}(G) \\ F(t) &= -I(t) \quad \text{se } t \in V_{OR}(G) \end{aligned}$$

e, portanto, que

$$minimax(G) = negamax(G).$$

A observação de Knuth e Moore, em [KM75], ajuda a compreender melhor a relação entre $F$, $H$ e $I$: *"[a função F] é análoga à operação NOR que aparece no projeto de circuitos; dois níveis de lógica NOR são equivalentes a um nível de AND [função H] seguido por um nível de OR [função I]."* [3]

Menos evidente, porém extremamente razoável, é o fato que o mérito de uma árvore de jogo coincide com o seu valor minimax (e, por conseqüência, com o valor negamax). De fato,

TEOREMA 2.20 *Seja G uma árvore de jogo. Então,*

$$m(G) = minimax(G) = negamax(G)$$

DEMONSTRAÇÃO : a demonstração da igualdade entre $minimax(G)$ e $negamax(G)$ foi feita acima. Logo, resta verificar que $m(G){=}minimax(G)$, o que é feito por indução no comprimento da árvore $G$. Para tanto, faz-se corresponder a função $I$ (que calcula *mínimos* no caso minimax) à função $f$ (cujo valor é o *mínimo* dos nós terminais de AND-árvores soluções); igualmente, a função $H$ (que calcula *máximos* no caso minimax) é mimetizada pelo processo de tomar a AND-árvore solução de $f$ *máximo* como AND-estratégia defensiva ótima.
A demonstração completa e rigorosa deste teorema é aqui omitida por razões de interesse e espaço, e pode ser encontrada em [Sto79]. □

## 2.8 A Máquina de Turing Alternante

Cumpre-se citar outra representação para jogos que não a utilização de árvores AND/OR : a *máquina de Turing alternante* [4] . Estas máquinas teóricas, estudadas em [CKS81], são essencialmente uma generalização das máquinas de Turing não-determinísticas, e utilizam conceitos de quantificadores existenciais e universais que se alternam ao longo da computação.

Em [SC79], Stockmeyer e Chandra detalham a maneira de representação de jogos em máquinas de Turing alternantes. Contudo, esta representação deve ser feita para o jogo de maneira *integral*, considerando-se somente os resultados finais de vitória ou de derrota.

Embora nenhum sistema de computação *parcial* de uma posição seja mencionado pelos autores (o que inviabiliza sua utilização prática), este método de representação é simples, poderoso e provavelmente bastante capaz de auxiliar na demonstração de proposições difíceis a respeito de jogos.

[3]Traduzido de [KM75], p. 295.
[4]Do original, em inglês, *alternating Turing machine*.

# Capítulo 3

# Algoritmos Seqüenciais para Busca em Árvores de Jogo

Conforme foi visto no capítulo anterior, o problema de se determinar o mérito de uma árvore de jogo é intrinsecamente exponencial. Ao mesmo tempo, o teorema 2.16 afirma que pode não ser necessário o exame de todos os nós terminais, chegando, no melhor caso de árvores uniformes, a se conseguir diminuir da ordem de raiz quadrada o número total de nós.

Esse resultado, entretanto, não é atingido deterministicamente, pois o caminho principal que satisfaz as condições do teorema 2.16, embora necessariamente presente, não é escolhido algoritmicamente. A busca deste caminho, que daqui em diante será denominado *caminho de refutações*, pode exigir, conforme a ordem em que a árvore for percorrida, o exame de todos os demais nós da árvore.

Os diversos algoritmos existentes para busca em árvore de jogos refletem, então, diferentes maneiras de percorrer a árvore e de utilizar as informações coletadas anteriormente. Serão examinados neste capítulo os principais algoritmos existentes, bem como as idéias fundamentais que os orientam. Preferiu-se o estudo cuidadoso à enumeração exaustiva dos mesmos, visando-se à compreensão global do problema na sua dimensão seqüencial, para, a partir da análise dos algoritmos, dispor-se de ferramental e conhecimento adequados para o tratamento do caso paralelo.

## 3.1 O Formalismo de Descrição dos Algoritmos

Para uniformizar os diferentes algoritmos tratados, adotou-se um formalismo, denominado *descrição por mensagens*, no qual se imagina que os nós da árvore emitem e recebem mensagens, sendo o resultado final do algoritmo obtido tanto pelo tratamento do conteúdo das mensagens, como da utilização de valores previamente armazenados em cada nó.

Desse modo, cada algoritmo será descrito na forma de uma tabela, na qual os diferentes tipos de mensagens recebidas são considerados; na tabela, a cada mensagem é associada uma série de ações, dependendo da mensagem recebida e dos valores atuais do nó receptor, e terminando pela emissão de uma ou mais mensagens a outros nós da árvore.

A figura 3.1 ilustra a descrição de um algoritmo para percurso em *pré-ordem* de uma árvore [1].

[1] Uma árvore é percorrida em pré-ordem quando todo nó pai é visitado *antes* de seus nós filhos.

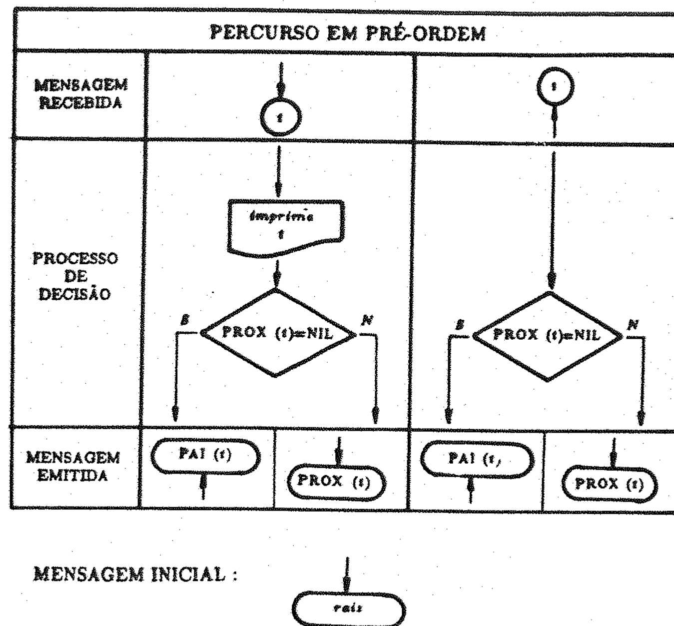



Figura 3.1: Descrição por mensagens de um algoritmo para percurso em pré-ordem de uma árvore.

O algoritmo é iniciado supondo-se somente a existência da mensagem inicial indicada. A cada passo, uma única mensagem é interpretada, e todo o processo termina quando o *hipotético* pai da raiz recebe uma mensagem. As funções PAI e PROX são:

PAI $(t)$ : gera o pai do nó $t$ na árvore considerada;

PROX $(t)$ : gerencia os sucessores do nó $t$ da seguinte forma: na primeira vez que a função é chamada com o valor do nó $t$, é gerado o sucessor mais à esquerda $t_1$ de $t$; na segunda vez, o sucessor $t_2$ e assim por diante até que se acabem os mesmos, quando então retorna NIL ; no caso de nós terminais, o valor retornado é *sempre* NIL .

Duas outras possibilidades de formalismo também foram investigadas e preteridas a esta:

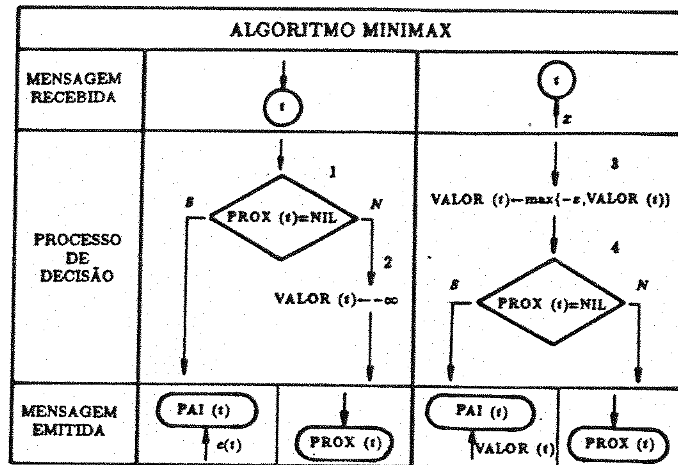
- **descrição procedimental**, ou seja, a descrição dos algoritmos em linguagens de programação como o Pascal; comparativamente, a descrição por mensagens adotada atenua diferenças não significativas, que são bastante realçadas pela descrição procedimental; além disso, a descrição procedimental não se adapta muito bem à diversidade estrutural típica dos sistemas paralelos;

- **descrição Branch & Bound**, desenvolvida por Kumar e Kanal em [KK83], que utiliza a representação dos estados de busca por meio de árvores de jogo parciais; embora seja admirável a clareza com que as diferenças entre certos algoritmos seja mostrada, este modelo não se adapta de forma natural à cerca de metade dos algoritmos que serão apresentados.

Cumpre-se destacar ainda algumas das deficiências do método utilizado de descrição por mensagens, entre elas, um certo ocultamento da quantidade de memória realmente necessária e uma aparente "desestruturação" do algoritmo. Estas questões, quando necessário, serão analisadas em separado, juntamente com os aspectos importantes da implementação dos algoritmos.

## 3.2 Algoritmo Minimax

O algoritmo **Minimax** caracteriza-se pela ***busca exaustiva e completa*** da árvore de jogo. Embora classicamente denominado de Minimax, este algoritmo calcula, na verdade, o valor *negamax* de uma árvore de jogo (usando o método da definição 2.19), e sua formulação original pode ser encontrada em [KM75].



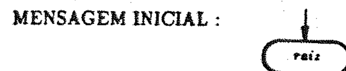




Figura 3.2: Algoritmo Minimax.

A descrição por mensagens do algoritmo Minimax é dada pela figura 3.2, onde $c(t)$ é a função descrita na definição 2.2, que retorna o valor associado a um nó terminal $t$ e VALOR $(t)$ é uma variável indexada pelo número do nó (tipicamente, um vetor). A proposição a seguir garante a corretude do algoritmo Minimax, ou seja, que o valor retornado na busca de uma árvore de jogo $G$ de raiz $t_0$ é, precisamente, $F(t_0)=m(G)$, conforme a definição 2.19.

PROPOSIÇÃO 3.1 *Seja $G$ uma árvore de jogo de raiz $t_0$ cujo pai (hipotético ou não) é $\bar{t}$. Então, se for enviada a $t_0$ uma mensagem descendente, após um tempo de processamento finito será enviada uma mensagem ascendente a $\bar{t}$ cujo valor é, exatamente $F(t_0)$.*

DEMONSTRAÇÃO : por indução na altura $h$ de $G$.

Suponha $h=0$.
Pela definição 2.19, $F(t_0)=c(t_0)$. Por outro lado, a mensagem descendente para $t_0$ conduz ao teste (1) do algoritmo, que tem, neste caso, resposta afirmativa, o que resulta na emissão de mensagem ascendente para PAI $(t_0)=\bar{t}$, com valor $c(t_0)$. Neste ponto o algoritmo cessa, em tempo obviamente finito, e com valor $c(t_0)=F(t_0)$.

Suponha a proposição válida para árvores de jogo de altura $h' \leq h-1$.
Sejam $t_1, t_2, \ldots, t_d$ os sucessores de $t_0$, e $G_1, G_2, \ldots, G_d$ as sub-árvores de respectivas raízes.
Nesta situação, o teste (1) é negativo, causando a inicialização de VALOR $(t_0)$ com $-\infty$ e a emissão de uma mensagem descendente para PROX $(t_0) = t_1$. Pela hipótese de indução, após

um tempo finito ocorre uma mensagem ascendente para PAI $(t_1) = t_0$, de valor $F(t_1)$. Por sua vez, esta mensagem causa a atualização (3) de VALOR $(t_0)$ e, se PROX $(t_0) \neq$ NIL , há uma nova mensagem descencente, desta vez para $t_2$, causada pela condição (4).
É claro que após $d$ mensagens ascendentes para $t_0$ tem-se PROX $(t_0) =$ NIL e

$$\begin{aligned} \text{VALOR } (t_0) &= \max\{-F(t_d), \max\{-F(t_{d-1}), \max\{\ldots, \max\{-F(t_1), -\infty\}\}\ldots\}\} \\ &= \max\{-F(t_1), -F(t_2), \ldots, -F(t_d)\} \\ &= F(t_0) \end{aligned}$$

Como PROX $(t_0) =$ NIL , este é precisamente o valor enviado a $\bar{t} =$ PAI $(t_0)$, concluindo, portanto, a demonstração. □

Embora o algoritmo Minimax descrito calcule somente $F(t_0)$, um simples teste do tipo (AND ou OR) de $t_0$ e, se necessário, uma troca de sinal, produz imediatamente o valor $negamax(G)$, conforme a definição 2.19.

Finalmente, observe-se que a condição (4) assegura que, independentemente dos valores retornados, todos os sucessores de um nó recebem mensagens descendentes antes que o mesmo emita uma mensagem ascendente para seu pai, caracterizando assim, uma busca exaustiva (e ordenada da esquerda para a direita) de $G$ em pós-ordem. A figura 3.3 mostra o resultado da execução do algoritmo Minimax sobre árvore uniforme de altura 4 e grau 3: os valores finais de VALOR $(t)$, em cada nó $t$, são colocados ao lado dos nós.

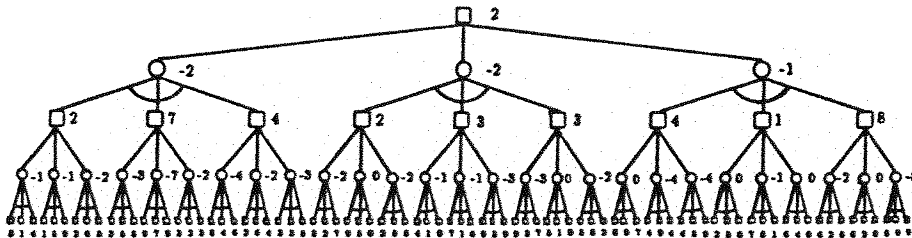
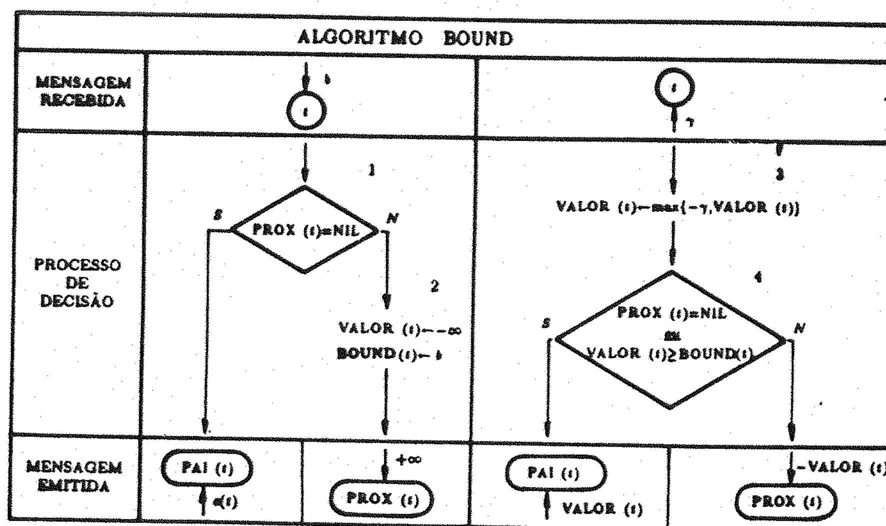


Figura 3.3: Árvore de jogo examinada pelo algoritmo Minimax.

## 3.3 Algoritmo Bound

Knuth e Moore, em [KM75], consideram que foi John McCarthy o primeiro pesquisador a se dar conta de que era possível a determinação do valor de uma árvore de jogo sem fazer uma busca exaustiva. Segundo alguns registros, a idéia surgiu em 1956, na pioneira Dartmouth Summer Research Conference on Artificial Intelligence, onde McCarthy criticou um dos primeiros programas de xadrez, concebido por A. Bernstein, por não utilizar nenhum mecanismo de redução de busca.

Considera-se que os dois primeiros programas para jogos que tiveram sucesso, concebidos por A. Newell, J.C. Shaw e H.A. Simon (xadrez) e A.L. Samuel (damas), utilizavam a técnica descrita a seguir. No entanto, os artigos que os descrevem, respectivamente [NSS58] e [Sam59], são omissos em relação a este fato.



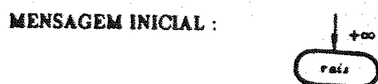




Figura 3.4: Algoritmo Bound.

O algoritmo **Bound** é baseado no algoritmo **F1** descrito por Knuth e Moore em [KM75], sendo sua denominação emprestada de Finkel e Fishburn (em [FF83]).

A descrição por mensagens do algoritmo Bound é dada pela figura 3.4, onde se utiliza a mesma estrutura do algoritmo Minimax, acrescida do vetor BOUND $(t)$, semelhante ao vetor VALOR $(t)$. A corretude é obtida com auxílio da proposição:

PROPOSIÇÃO 3.2 *Seja $G$ uma árvore de jogo de raiz $t_0$ cujo pai (hipotético ou não) é $\bar{t}$. Então, se for enviada a $t_0$ uma mensagem descendente de valor $b$, após um tempo de processamento finito será enviada uma mensagem a $\bar{t}$ cujo valor $\gamma$ é tal que*

$$\begin{array}{ll} \gamma = F(t_0) & \text{se } F(t_0) < b \text{ ou } t_0 \text{ é terminal} \\ \gamma \geq b & \text{se } F(t_0) \geq b \text{ e } t_0 \text{ não é terminal.} \end{array}$$

DEMONSTRAÇÃO : por indução na altura $h$ de $G$.

Suponha $h$=0. Neste caso, a demonstração é analóga à utilizada na proposição 3.1, e obtém-se $\gamma$=$c(t_0)$=$F(t_0)$.

Suponha a proposição válida para árvores de altura $h' \leq h-1$.
Sejam $t_1, t_2, \ldots, t_d$ os sucessores de $t_0$ e $G_1, G_2, \ldots, G_d$ as sub-árvores de $G$ de respectivas raízes. Considere-se então uma mensagem descendente para $t_0$ de valor $b$. Como $h$>0, o teste (1) é negativo, causando a inicialização de VALOR $(t_0)$ e BOUND $(t_0)$ com $-\infty$ e $b$, respectivamente. Na seqüência, é enviada a mensagem descendente de valor $+\infty$ para $t_1$.
Pela hipótese de indução, após um tempo finito de processamento o nó $t_0$=PAI $(t_1)$ recebe uma mensagem de valor $\gamma_1$. Como a $t_1$ foi enviado o valor $+\infty$, tem-se $\gamma_1$=$F(t_1)$, e assim

$$\text{VALOR } (t_0) \leftarrow \max\{-F(t_1), -\infty\} = -F(t_1)$$

Chamando de $\gamma_i$ o valor da mensagem ascendente emitida pelo nó $t_i$, seja $v_i$ o conteúdo da variável VALOR $(t_0)$ após o comando (3),

$$\text{VALOR } (t_0) \leftarrow \max\{-\gamma_i, \text{VALOR } (t_0)\}$$

É correto afirmar que

$$v_i = \max\{-F(t_1), -F(t_2), \ldots, -F(t_i)\} \ , \quad \forall i \ 1 \leq i \leq d$$

De fato, faça-se indução finita em $i$. Já foi mostrado que $v_1 = -F(t_1)$. Suponha então que $v_{i-1} = \max\{-F(t_1), -F(t_2), \ldots, -F(t_{i-1})\}$; ora, o nó $t_i$ recebe a mensagem descendente $-v_{i-1}$. Se $F(t_i) < -v_{i-1}$, e $t_i$ recebe a mensagem ascendente $\gamma_i = F(t_i)$, tem-se, como $-F(t_i) > v_{i-1}$,

$$\begin{aligned} v_i &= -F(t_i) = \max\{-F(t_i), v_{i-1}\} \\ &= \max\{-F(t_1), -F(t_2), \ldots, -F(t_i)\} \end{aligned}$$

No caso contrário, $F(t_i) \geq -v_{i-1}$, a hipótese global de indução garante que a mensagem $\gamma_i$ enviada por $t_i$ satisfaz $\gamma_i \geq -v_{i-1}$, ou $-\gamma_i \leq v_{i-1}$. Então $v_i = v_{i-1}$. Por outro lado, como $-F(t_i) \leq v_{i-1}$,

$$v_{i-1} = \max\{v_{i-1}, -F(t_i)\} = \max\{\underbrace{-F(t_1), -F(t_2), \ldots, -F(t_{i-1})}_{v_{i-1}}, -F(t_i)\}$$

e assim, como $v_i = v_{i-1}$,

$$v_i = \max\{-F(t_1), -F(t_2), \ldots, -F(t_i)\}$$

Analisem-se então os dois casos da proposição:

$F(t_0) \geq b$ : neste caso existe $i$, $1 \leq i \leq d$, tal que $-F(t_i) \geq b$, pois do contrário $F(t_0) = \max\{-F(t_1), -F(t_2), \ldots, -F(t_d)\} < b$.

Seja $j$, $1 \leq j \leq d$, o menor índice para o qual $-F(t_j) \geq b$. Nestas condições, $v_j = \max\{-F(t_1), -F(t_2), \ldots, -F(t_j)\} = -F(t_j)$, e portanto quando o nó $t_0$ recebe a mensagem de $t_j$, tem-se $v_j \geq b$. Como, pela definição, $v_j =$ VALOR $(t_0)$ e $b =$ BOUND $(t_0)$, obtém-se

$$\text{VALOR } (t_0) \geq \text{BOUND } (t_0)$$

e assim a condição (4) é positiva, causando o envio de $\gamma = v_j =$ VALOR $(t_0)$ para $\bar{t} =$ PAI $(t_0)$. Como $v_j \geq b$, então $\gamma \geq b$, confirmando o enunciado da proposição.

$F(t_0) < b$ : nesta outra situação, tem-se que $-F(t_i) < b$, para todo $i$, $1 \leq i \leq d$. Logo, os máximos parciais $v_j$ satisfazem sempre $v_j < b$, ou, equivalentemente, que, em todos os casos

$$\text{VALOR } (t_0) < \text{BOUND } (t_0)$$

Dessa forma, todos os nós $t_i$ são examinados, um a um, pois a condição (4) só é positiva quando todos os nós tiverem sido examinados, isto é, PROX $(t_0) =$ NIL . Nestas condições, é retornado o valor $v_d =$ VALOR $(t_0)$.

Já foi demonstrado acima que

$$v_d = \max\{-F(t_1), -F(t_2), \ldots, -F(t_d)\}$$

logo $v_d = F(t_0)$. Portanto, a mensagem $\gamma = v_d$ é precisamente $F(t_0)$, conforme o enunciado da proposição.

Este último resultado completa a indução, estabelecendo a correção da proposição.□

Finalmente, se a mensagem inicial descendente para a raiz tiver valor $+\infty$, conforme indicado na figura 3.4, a proposição 3.2 garante que a mensagem ascendente para $\bar{t}$=PAI $(t_0)$ tem valor $F(t_0)$, pois

$$\begin{array}{llll} \text{se} & F(t_0) < +\infty & \text{então} & \gamma = F(t_0) \\ \text{e se} & F(t_0) = +\infty & \text{então} & \gamma \geq F(t_0) \quad , \gamma = +\infty = F(t_0) \end{array}$$

Olhando-se com cuidado a condição (4), nota-se o poder deste algoritmo de, em cada nó, e com base somente no valor BOUND recebido de seu pai (proveniente de irmãos à esquerda), cortar a busca em ramos inteiros da árvore de jogo *sem que seja alterado o resultado global.*

A figura 3.5 mostra, comparativamente, o algoritmo Bound executado sobre a árvore da figura 3.3. Na figura, os ramos marcados com tracejado indicam aqueles que não foram examinados pelo algoritmo. Repare-se que somente 36 dos 81 nós terminais da árvore são examinados, e que o valor da raiz é idêntico.

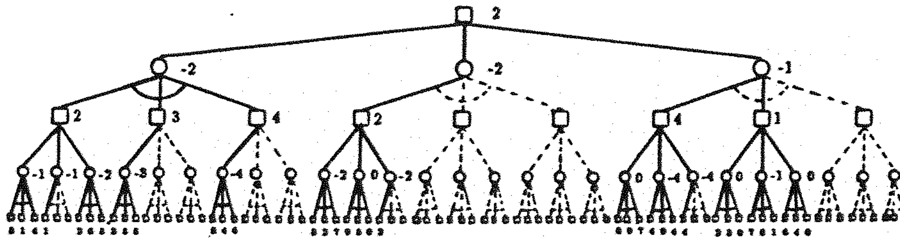


Figura 3.5: Árvore de jogo examinada pelo algoritmo Bound.

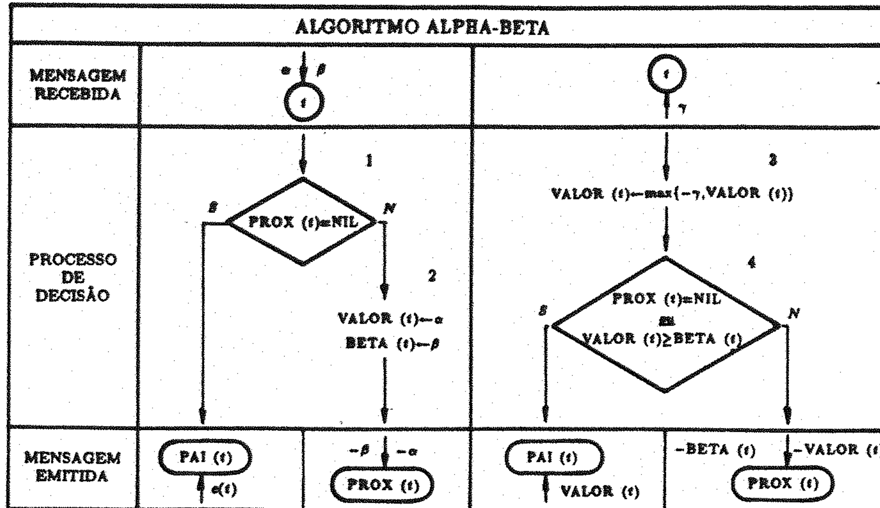
## 3.4 Algoritmo Alpha-Beta

Na seção anterior foi visto que a idéia de realizar "cortes" de ramos das árvores de jogo é atribuída a John McCarthy. A este pesquisador também cabe o mérito de ter concebido um posterior refinamento do algoritmo Bound, batizado por ele mesmo de **Alpha-Beta**.

No entanto, o próprio McCarthy tinha dúvidas se o algoritmo era correto e, por isso, credita a descoberta a Hart e Edwards, que em 1961 escreveram um *memorandum* sobre o assunto ([HE61]) sem, contudo, demonstrar a sua corretude. Esta demonstração, segundo Knuth e Moore, aparece somente em 1963 em um artigo, em russo, de Brudno ([Bru63]), e em inglês nos trabalhos de Slagle e Bursky ([SB68]) e Slagle e Dixon ([SD69]).

Nesta seção será feito um primeiro contato com o algoritmo Alpha-Beta, incluindo a demonstração de sua corretude e um exemplo, e à próxima seção caberá uma melhor diferenciação deste algoritmo com o Bound. A descrição por mensagens do algoritmo Alpha-Beta é dada pela figura 3.6, onde é utilizada uma estrutura semelhante a do algoritmo Bound, substituindo-se o vetor BOUND $(t)$ por BETA $(t)$.

A proposição a seguir possibilita a demonstração da corretude do algoritmo Alpha-Beta, ou seja, que o valor retornado pelo mesmo, quando do envio de mensagem inicial de valor $-\infty$:$+\infty$, é exatamente o valor do mérito da árvore de jogo examinada.



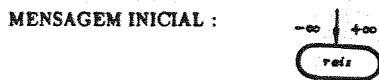




Figura 3.6: Algoritmo Alpha-Beta.

PROPOSIÇÃO 3.3 *Seja G uma árvore de jogo de raiz $t_0$ cujo pai (hipotético ou não) é $\bar{t}$. Então, se for enviada a $t_0$ uma mensagem descendente de valor $\alpha : \beta$ (onde $\alpha \leq \beta$), após um tempo de processamento finito será enviada uma mensagem a $\bar{t}$ cujo valor $\gamma$ é tal que*

$$\begin{array}{lll} \gamma \leq \alpha & \textit{se } F(t_0) \leq \alpha & \textit{e } t_0 \textit{ não é terminal} \\ \gamma = F(t_0) & \textit{se } \alpha < F(t_0) < \beta & \textit{ou } t_0 \textit{ é terminal} \\ \gamma \geq \beta & \textit{se } F(t_0) \geq \beta & \textit{e } t_0 \textit{ não é terminal.} \end{array}$$

DEMONSTRAÇÃO : a demonstração segue analogamente a da proposição 3.2, por indução na altura $h$ de $G$.

Suponha $h=0$. Novamente, $\gamma = c(t_0) = F(t_0)$.

Suponha a proposição válida para árvores de altura $h' \leq h-1$.
Sejam $t_1, t_2, \ldots, t_d$ os sucessores de $t_0$ e $G_1, G_2, \ldots, G_d$ as sub-árvores de $G$ de respectivas raízes. Como $h>0$, a mensagem descendente de valores $\alpha : \beta$ para $t_0$ causa a inicialização de VALOR $(t_0)$ e de BETA $(t_0)$ com $\alpha$ e $\beta$, respectivamente, e o envio de mensagem descendente para $t_1$, de valores $-\beta : -\alpha$ $(-\beta \leq -\alpha)$.
Pela hipótese de indução, o valor $\gamma_1$ retornado por $t_1$ satisfaz:

$$\begin{array}{lll} \gamma_1 \leq -\beta & \text{se } F(t_1) \leq -\beta & \\ \gamma_1 = F(t_1) & \text{se } -\beta < F(t_1) < -\alpha & (3.1) \\ \gamma_1 \geq -\alpha & \text{se } F(t_1) \geq -\alpha & \end{array}$$

Novamente, seja $\gamma_i$ o valor da mensagem ascendente emitida pelo nó $t_i$, $1 \leq i \leq d$, e $v_i$ o conteúdo de VALOR $(t_i)$ após o comando (3). Então, para todo $i$, $1 \leq i \leq d$, é verdade que:

i . Se para todo $j$, $1 \leq j \leq i$, vale que $-F(t_j) < \beta$, então

$$v_i = \max\{\alpha, -F(t_1), -F(t_2), \ldots, -F(t_i)\}$$

ii . **Se existe $j$, $1\leq j\leq i$, tal que $-F(t_j)\geq\beta$, então**

$$v_i \geq \beta$$

De fato, faça-se indução finita em $i$.
Para $i$=1, as equações (3.1) acima garantem que se $-F(t_1)\geq\beta$, então $-\gamma_1\geq\beta\geq\alpha$, logo $v_1=\max\{\alpha,-\gamma_1\}=-\gamma_1$, e assim $v_1\geq\beta$. Se, caso contrário, $-F(t_1)<\beta$, então, tanto no caso $-F(t_1)>\alpha$ (com $\gamma_1=F(t_1)$), como no caso $-F(t_1)\leq\alpha$ (com $\gamma_1\leq-F(t_1)\leq\alpha$), tem-se

$$v_1 = \max\{\alpha,-\gamma_1\} = \max\{\alpha,-F(t_1)\}$$

Suponha-se, então, a afirmação válida para $v_{i-1}$. Como ao nó $t_i$ foi enviada a mensagem de valor $-\beta$:$-v_{i-1}$, a hipótese global de indução garante que, para a árvore $G_i$, de altura $h_i<h$, vale

$$\begin{array}{ll} \gamma_i \leq -\beta & \text{se } F(t_i)\leq-\beta \\ \gamma_i = F(t_i) & \text{se } -\beta<F(t_i)<-v_{i-1} \\ \gamma_i \geq -v_{i-1} & \text{se } F(t_i)\geq-v_{i-1} \end{array}$$

É evidente que, se já existe $j$, $1\leq j\leq i-1$ tal que $-F(t_j)\geq\beta$ (com $v_{i-1}\geq\beta$), obtém-se $v_i\geq\beta$, pois $v_i=\max\{-\gamma_i,v_{i-1}\}$. [2].
Suponha-se então que para todo $1\leq j\leq i-1$, vale que $-F(t_j)<\beta$, e portanto que, pela hipótese de indução,

$$v_{i-1} = \max\{\alpha,-F(t_1),-F(t_2),\ldots,-F(t_{i-1})\}$$

Se $-F(t_i)\geq\beta$, então $-\gamma_i\geq\beta$, e logo $v_i=\max\{-\gamma_i,v_{i-1}\}\geq\beta$, caracterizando o caso ($ii$) da afirmação.
Restam duas opções a considerar, $v_{i-1}<-F(t_i)<\beta$ e $-F(t_i)\leq v_{i-1}$ (com $v_{i-1}<\beta$). Ambas as situações enquadram-se no caso ($i$), pois $-F(t_i)<\beta$. Na primeira opção, tem-se que

$$\begin{aligned} v_i &= \max\{-\gamma_i,v_{i-1}\} \\ &= \max\{-F(t_i),\max\{\alpha,-F(t_1),-F(t_2),\ldots,-F(t_{i-1})\}\} \\ &= \max\{\alpha,-F(t_1),-F(t_2),\ldots,-F(t_i)\} \end{aligned}$$

Na segunda opção, $-F(t_i)\leq v_{i-1}$, com $-\gamma_i\leq v_{i-1}$, verifica-se que

$$v_i = \max\{-\gamma_i,v_{i-1}\} = v_{i-1}$$

e, por outro lado, devido ao fato que $-F(t_i)\leq v_{i-1}$,

$$\begin{aligned} v_{i-1} &= \max\{-\gamma_i,v_{i-1}\} \\ &= \max\{\alpha,-F(t_1),-F(t_2),\ldots,-F(t_{i-1}),-F(t_i)\} \end{aligned}$$

e, assim

$$v_i = \max\{\alpha,-F(t_1),-F(t_2),\ldots,-F(t_i)\}$$

Demonstrada a afirmação sobre o invariante $v_i$, resta examinar os três casos da proposição:

[2]Embora, na prática, este caso *nunca* venha a ocorrer durante a execução do algoritmo, pois a condição (4) impede que seja enviada qualquer mensagem a $t_i$.

$F(t_0) \leq \alpha$ : logo, $-F(t_i) \leq \alpha$, para todo $i$, $1 \leq i \leq d$. O caso $\alpha = \beta$ é evidente, tome-se pois $\alpha < \beta$.

Como $v_i = \max\{\alpha, -F(t_1), -F(t_2), \ldots, -F(t_i)\} < \beta$, para todo $i$, $1 \leq i \leq d$, então na condição (4) o teste

$$v_i = \text{VALOR } (t_0) \geq \text{BOUND } (t_0) = \beta$$

é sempre falso, e o algoritmo só envia mensagem ascendente para $\bar{t} = \text{PAI } (t_0)$ quando $\text{PROX } (t_0) = \text{NIL}$ .

Como $v_d = \max\{\alpha, -F(t_1), -F(t_2), \ldots, -F(t_d)\}$, e também, para todo $i$, é verdade que $-F(t_i) \leq \alpha$, então $\gamma = v_d = \alpha$, satisfazendo a desigualdade $\gamma \leq \alpha$ da proposição. [3]

$\alpha < F(t_0) < \beta$ : análogo ao caso anterior, exceto que

$$\begin{aligned} v_d &= \max\{\alpha, \underbrace{-F(t_1), -F(t_2), \ldots, -F(t_d)}_{F(t_0)}\} \\ &= F(t_0) \end{aligned}$$

confirmando o valor $\gamma = v_d = F(t_0)$ da proposição.

$F(t_0) \geq \beta$ : neste caso, existe $t_i$, $1 \leq i \leq d$, $i$ índice mínimo, tal que $-F(t_i) > \beta$. Como $i$ é tomado como o menor índice no qual tal fato ocorre, é verdade que $v_j < \beta$, para todo $1 \leq j \leq i-1$; além disso, $v_i \geq \beta$, e assim o teste (4),

$$v_j = \text{VALOR } (t_0) \geq \text{BOUND } (t_0) = \beta$$

é verdadeiro pela primeira vez com $j = i$.

Nessas condições, o algoritmo retorna $\gamma = v_i$, que o caso ($ii$) da afirmação garante que satisfaz $\gamma = v_i \geq \beta$, confirmando a proposição.

Desta forma, verifica-se o passo da indução, finalizando-se a demonstração. □

Analogamente aos algoritmos anteriores, se a mensagem descendente para a raiz for $-\infty{:}+\infty$, conforme indica a figura 3.6, a proposição 3.3 garante que a mensagem ascendente para $\bar{t} = \text{PAI } (t_0)$ tem valor $F(t_0)$.

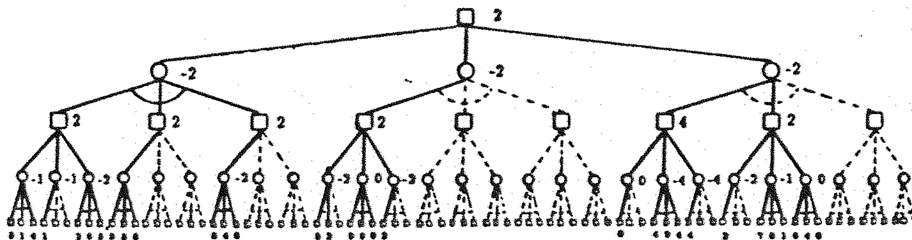


Figura 3.7: Árvore de jogo examinada pelo algoritmo Alpha-Beta.

A figura 3.7 mostra o algoritmo Alpha-Beta executado sobre a árvore da figura 3.3, com somente 31 dos 81 nós terminais da árvore sendo examinados.

[3] Na verdade, um exame rápido do algoritmo Alpha-Beta é suficiente para perceber que, em todos os casos, $\gamma \geq \alpha$, e que, portanto, se $F(t_0) \leq \alpha$ tem-se obrigatoriamente $\gamma = \alpha$.

## 3.5 Cortes Rasos e Profundos

Antes de prosseguir com a descrição dos algoritmos seqüenciais, é essencial um estudo mais cuidadoso dos algoritmos Bound e Alpha-Beta. Como foi visto, ambos os algoritmos são capazes de realizar "cortes" na árvore de jogo em exame. Há dois tipos diferentes de corte, *raso* e *profundo* [4], sendo que o algoritmo Bound só realiza cortes rasos, enquanto que o Alpha-Beta executa os dois.

A figura 3.8.a mostra um corte raso, onde os nós com valores numéricos são nós já explorados, e os nós com letras são nós em exploração ou a explorar.

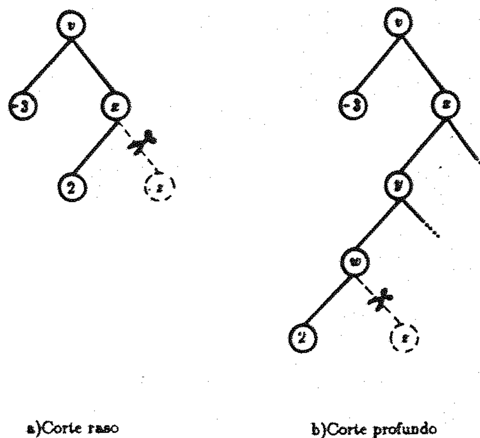



Figura 3.8: Corte raso (a) e corte profundo (b).

Em um **corte raso**, um nó — no caso, $x$ — utiliza a informação proveniente de um irmão ($-3$) e de um de seus filhos (2) para evitar a busca em outro filho ($z$) mais à direita. É fácil ver que o valor de $z$ não afeta o valor de $v$ nessa situação, pois

$$v = \max\{3, -x\} \quad \text{e} \quad x = \max\{-2, -z\}$$

logo $x \geq -2$, ou $2 \geq -x$, obtendo-se sempre $v$=3 qualquer que seja o valor de $z$. Conclui-se então que, em situações como essa, o valor final de $v$ *independente* do valor de $z$.

O **corte profundo** é uma variação desta idéia, na qual ao invés de informação proveniente de um irmão, utiliza-se valores de um "tio-avô", ou mesmo "parentes" mais distantes. Examine-se o caso mais simples, ilustrado na figura 3.8.b . Neste caso, o que se deseja mostrar é que o valor real de $v$ independe do valor de $z$, dado que um filho de $w$ tem o valor 2. De fato,

$$v = \max\{3, -x\} \qquad x \geq -y \qquad y \geq -w \qquad w = \max\{-2, -z\}$$

logo $2 \geq -w$. Se o valor de $y$ for estritamente maior que $-w$, $y > -w$, então o valor de $y$ é determinado por um outro seu sucessor que não $w$, e consequentemente, o valor de $v$ independe da exploração de $z$.

Por outro lado, se $y = -w$, então como $w \geq -2$, tem-se $y \leq 2$, e logo $-x \leq 2$. Desse modo, é garantido que $v$=3, novamente sem necessidade de considerar o valor de $z$.

[4] Do original, em inglês, *shallow and deep cutoff*.

O raciocínio acima é facilmente estendido para cortes mais "profundos", onde o valor de um nó situado a um nível $h$ da árvore causa o corte de um outro nó de nível $h+i$, $i$ ímpar, $i \geq 3$. A diferença de níveis, $i$, é necessariamente ímpar, pois do contrário a sucessão de desigualdades não garante o corte. O caso $i=1$ é, como foi visto, o corte raso.

O algoritmo Bound, embora execute cortes rasos, é incapaz de "localizar" cortes profundos. Isto porque nós à esquerda recebem de seu pai sempre o valor $+\infty$, sendo portanto incapazes de realizar cortes (ver figura 3.9).

O algoritmo Alpha-Beta, por sua vez, tem a capacidade de fazer tanto cortes rasos como profundos, pois os valores $\alpha : \beta$, alternadamente, levam aos nós dos níveis inferiores os valores já calculados nos ramos superiores mais à esquerda da árvore (ver figura 3.10).

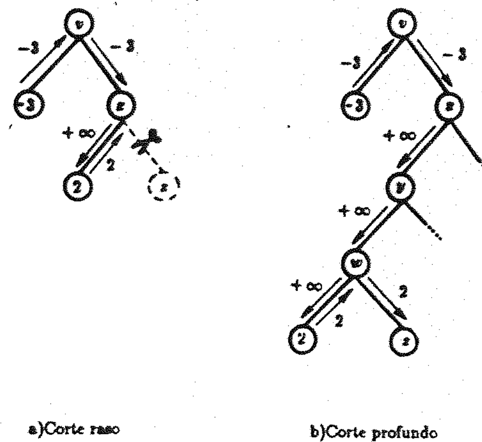



Figura 3.9: Algoritmo Bound em situações de corte raso e profundo.

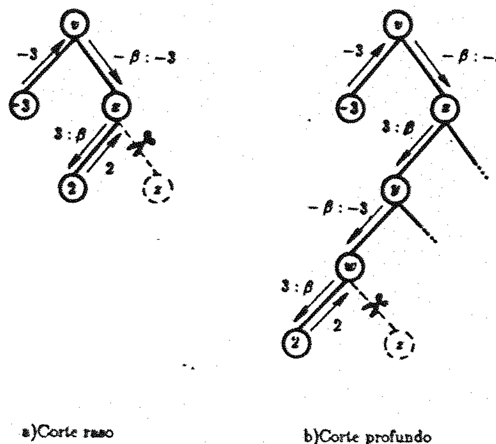



Figura 3.10: Algoritmo Alpha-Beta em situações de corte raso e profundo.

É interessante lembrar que, historicamente, muita confusão existiu entre os algoritmos Bound e Alpha-Beta, causada em parte pela similaridade aparente entre cortes rasos e profundos. Como exemplo, o computólogo Donald E. Knuth, um dos autores de um dos artigos clássicos sobre Alpha-Beta ([KM75]), comenta que durante aproximadamente 5 anos trabalhou em busca em árvores de jogo sem se dar conta que cortes profundos eram possíveis. Ele próprio afirma, textualmente:

*"É fácil entender-se o procedimento F1 [Bound] e associá-lo com o termo "poda Alpha-Beta" do qual nossos colegas estão falando sem descobrir F2 [Alpha-Beta]."* [5]

## 3.6 Algoritmo SSS*

Todos os algoritmos examinados até agora guardam uma característica em comum: o exame da árvore de jogo da esquerda para a direita. Em melhores termos, os algoritmos Minimax, Bound e Alpha-Beta são *direcionais*, de acordo com a definição abaixo, extraída de [Pea80]:

DEFINIÇÃO 3.4 *Um algoritmo para busca em árvores de jogo A é dito* direcional *se, para algum arranjo linear fixo (para qualquer árvore de jogo) dos nós terminais , nunca é selecionado para exame um nó situado à esquerda de outro nó já examinado.*

A figura 3.7 ilustra bem as conseqüências da direcionalidade de um algoritmo: Alpha-Beta examina as porções esquerdas da árvore mais minuciosamente que as direitas. Ocorre que, se a "solução" encontra-se mais à direita, a busca realizada à esquerda não traz redução no trabalho necessário na parte direita da árvore, causando uma deterioração na performance do algoritmo direcional.

Em 1979, Stockman propôs um algoritmo não-direcional, baseado na idéia de *branch & bound*, ou seja, na concentração do esforço de busca nas regiões da árvore mais promissoras (considerando a informação disponível até o momento). O algoritmo foi denominado **State Space Search**, ou, abreviadamente, **SSS*** [6].

No artigo original ([Sto79]), toda a teoria que dá suporte ao algoritmo é baseada no conceito de AND-árvore solução. Na verdade, o algoritmo é estruturado na forma de uma busca "paralela" ao longo de todas as AND-árvores solução possíveis, considerando-se sempre a árvore mais promissora, e eliminando da busca as árvores que são, garantidamente, refutações de outras já existentes.

A figura 3.11 mostra a descrição por mensagens do algoritmo SSS*, onde a função TIPO $(t)$ retorna o tipo (AND ou OR) de um nó $t$. Ao contrário dos demais algoritmos já examinados, existem situações , por exemplo, no comando (4), em que mais de uma mensagem pode ser emitida a partir de uma única mensagem recebida.

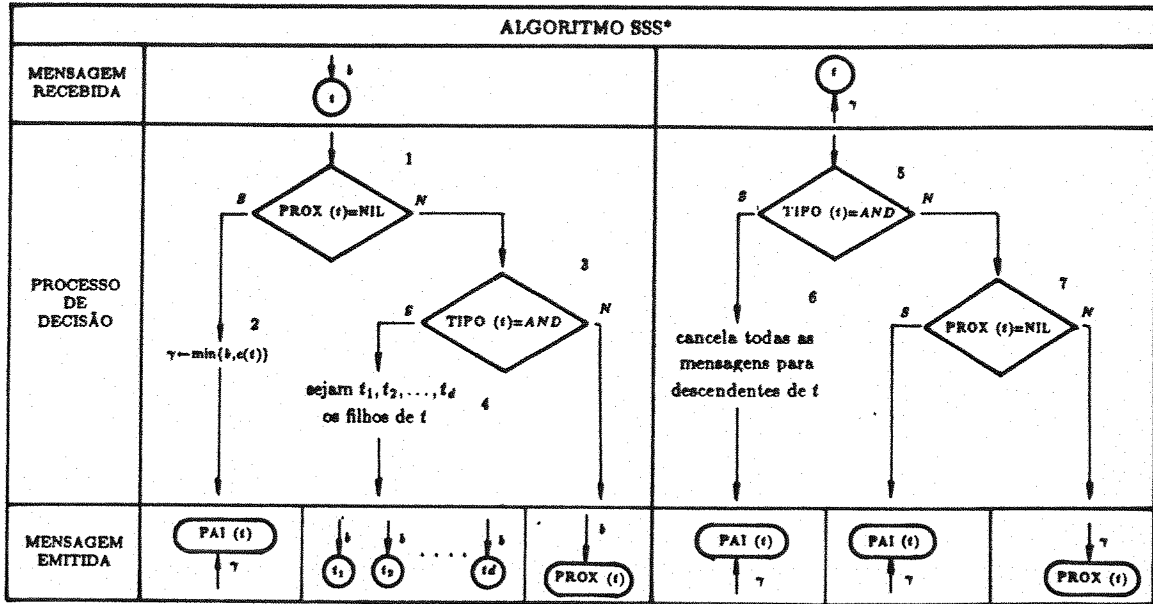
O gerenciamento das mensagens é feito através de uma lista $L$, ordenada pelos seguintes critérios:

- 1º critério : maior valor ($b$ ou $\gamma$), ou seja, $L$ é uma lista decrescente em relação ao valor da mensagem;
- 2º critério (*tie-breaker*) : se os valores de duas mensagens forem iguais, coloca-se na frente a mensagem enviada para o nó de menor valor lexicográfico (grosseiramente, o nó mais à esquerda, e, de maneira precisa, conforme [RP83], p. 206).

A cada passo do algoritmo, escolhe-se a mensagem do início de $L$ (ou seja, de valor maximal). A inserção e remoção de mensagens, respectivamente casos (2) e (6), é feita de maneira que se

[5] Traduzido de [KM75], p. 304.

[6] O * é proveniente das semelhanças entre o algoritmo de Stockman e o algoritmo A*, de Hart, Nilsson e Raphael, cujas descrição e referências podem ser encontrados em [Nil82].




MENSAGEM INICIAL :

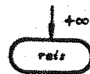



Figura 3.11: Algoritmo SSS*.

preserve a ordem da lista; as demais operações são feitas exclusivamente no topo da lista (sem, contudo, alterar sua ordem). A corretude do algoritmo é dada por:

PROPOSIÇÃO 3.5 *Seja $G$ uma árvore de jogo de raiz $t_0$ cujo pai (hipotético ou não) é $\bar{t}$. Então, se for enviada a $t_0$ uma mensagem descendente de valor $b$, após um tempo de processamento finito será enviada uma mensagem a $\bar{t}$ cujo valor $\gamma$ é tal que*

$$\begin{array}{lll} \gamma = m(G) & se\ m(G)<b & (i) \\ \gamma = b & se\ m(G)\geq b & (ii) \end{array}$$

DEMONSTRAÇÃO : por indução na altura $h$ de $G$.

Suponha $h$=0. Claramente, a condição (1) garante que $\gamma=\min\{b, c(t_0) = m(G)\}$, e portanto satisfaz a proposição.

Suponha a proposição válida para árvores de altura $h' \leq h - 1$.
Sejam $t_1, t_2, \ldots, t_d$ os sucessores de $t_0$ e $G_1, G_2, \ldots, G_d$ as sub-árvores de $G$ de respectivas raízes.
<u>1º Caso</u> $t_0 \in V_{AND}(G)$.
A condição (3) produz o envio de mensagens descendentes para $t_1, t_2, \ldots, t_d$ de valor $b$. A hipótese de indução garante que, potencialmente, todos os nós $t_i$, $1 \leq i \leq d$, enviam uma mensagem $\gamma_i$ correspondente para $t_0$. Seja $t_j$, $1 \leq j \leq d$, o nó que envia uma mensagem ascendente $\gamma_j$ em primeiro lugar.
A execução do comando (6) ocasiona a "morte" de todos os demais nós descendentes de $t_0$, e logo a única mensagem ascendente enviada por $t_0$ tem valor $\gamma = \gamma_j$.

Examine-se o caso $m(G)<b$. Então, como $m(G_i)\leq m(G)$, para todo $1\leq i\leq d$, e, em particular, $m(G_j)\leq m(G)$, a hipótese de indução garante que $\gamma_j=m(G_j)$.
Por absurdo, suponha que $m(G)>m(G_j)$. Então existe $i$, $1\leq i\leq d$, tal que $m(G_i)>m(G_j)$. Considere-se, agora, uma AND-estratégia defensiva ótima $T_i$ de $G_i$. Ora, todas as mensagens para nós de $T_i$, de valor $\dot{b}$, ascendentes ou descendentes, satisfazem $m(G_i)\leq\dot{b}\leq b$. Como $m(G_i)>m(G_j)$, então $\dot{b}>\gamma_j$, e, na medida em que estas mensagens não podem ser canceladas por mensagens de $t_j$ (ver comando (6)), então a lista $L$ não estaria ordenada, absurdo!
Logo $m(G_j)\geq m(G_i)$, para todo $i$, $1\leq i\leq d$, e portanto

$$m(G)=m(G_j)=\gamma_j=\gamma$$

Considere-se o caso em que $m(G)\geq b$. Se $m(G_j)\geq b$, então $\gamma_j=b$, e logo a mensagem enviada por $t_0$, $\gamma=\gamma_j=b$, satisfaz a condição $(ii)$ da proposição. A outra opção, $m(G_j)<b$ não pode ocorrer, pois gera uma contradição semelhante à encontrada acima.

2º Caso $t_0\in V_{OR}(G)$.
Observe-se inicialmente que, no caso OR, os nós sucessores são percorridos da esquerda para a direita, de forma que o retorno $\gamma_i$ de um nó $t_i$ é o valor enviado ao próximo nó $t_{i+1}$. Pela hipótese de indução, e considerando $\gamma_0=b$,

$$\begin{array}{ll}\gamma_i=m(G_i) & \text{se } m(G_i)<\gamma_{i-1}\\ \gamma_i=\gamma_{i-1} & \text{se } m(G_i)\geq\gamma_{i-1}\end{array}$$

Se $m(G)\geq b$, como $t_0$ é nó OR, tem-se também $m(G_i)\geq m(G)\geq b$, para todo $i$, $1\leq i\leq d$, logo $\gamma_i=b$ para todo $i$, e em particular para $\gamma_d=\gamma$, valor da mensagem ascendente enviada para $\bar{t}$.
Se, por outro lado, $m(G)<b$, então existe $t_j$, $1\leq j\leq d$, tal que $m(G_j)=m(G)<b$, e $\gamma_d=\gamma_j$. De fato, os nós $t_1,t_2,\ldots,t_{j-1}$ produzem sempre mensagens que satisfazem $\gamma_i\geq\gamma_j$, $1\leq i\leq j-1$, pois $m(G_i)\geq m(G_j)$. O nó $t_j$ obviamente retorna $\gamma_j=m(G_j)$, e como para $j+1\leq i\leq d$ tem-se $m(G_i)\geq m(G_j)\geq\gamma_j$, $\gamma_{j+1}=\gamma_{j+2}=\ldots=\gamma_d=\gamma_j$. Então a mensagem $\gamma=\gamma_j$ enviada à $\bar{t}$ tem valor $\gamma_d=\gamma_j=m(G_j)=m(G)$, satisfazendo a afirmação $(i)$. □

Olhando-se novamente esta demonstração e a figura 3.11, chega-se a algumas conclusões sobre o modo pelo qual o algoritmo SSS* funciona:

- o corte de ramos da árvore é feito pelo comando (6) e é garantido pela ordenação da lista;
- a busca pelos ramos mais promissores ocorre como resultado do comando (4) e de se tomar sempre a mensagem de maior valor.

Se, analogamente, for enviada ao nó $t_0$ de uma árvore de jogo $G$ uma mensagem inicial de valor $+\infty$, então a proposição 3.5 garante que o valor enviado ao seu pai terá exatamente o valor $m(G)$.
A figura 3.12 mostra a execução do algoritmo SSS* sobre a árvore da figura 3.3. Note-se que se examinam somente 30 nós terminais, e que *todo nó examinado por SSS* é também examinado por Alpha-Beta*.
Na verdade, este último fato vale para qualquer árvore de jogo, conforme será visto no próximo capítulo.

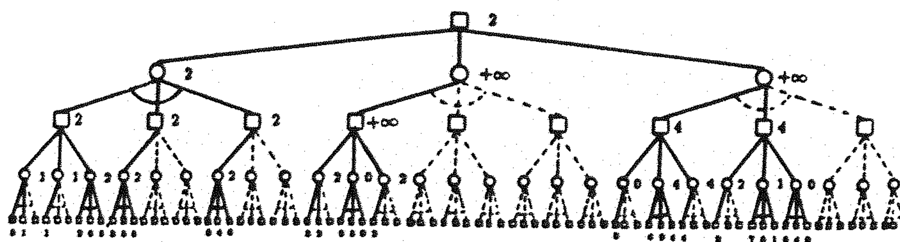


Figura 3.12: Árvore de jogo examinada pelo algoritmo SSS*.

## 3.7 Algoritmo SCOUT

No capítulo anterior, verificou-se que a busca em árvores de jogo cujo mérito é $\pm\infty$ tem limite inferior de cerca da metade do encontrado para árvores cujo mérito está no intervalo aberto $]-\infty,+\infty[$. Intuitivamente, pode-se entender o porquê examinando-se a figura 3.13. No caso (a), o fato $c(t_1)$=$-\infty$, com $t_1$ do tipo OR implica automaticamente que $m(G)$=$-\infty$, independentemente dos valores de $t_2$, $t_3$ ou $t_4$. O caso (b) ilustra uma situação mais complexa, na qual o exame de apenas dois nós terminais, $t_3$ e $t_4$, garante que a AND-árvore solução $T$ é estratégia defensiva ótima, não importa que valores tenham $t_5$ e $t_6$.

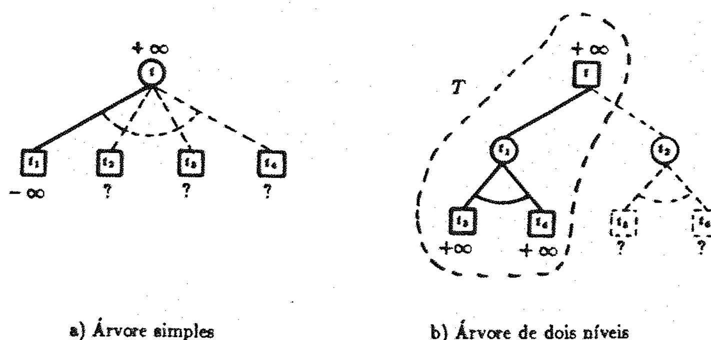



Figura 3.13: Exemplos de árvores cujo mérito é $\pm\infty$.

Neste caso, o que se observa é que é possível se efetuar cortes sem a informação de valores dos irmãos, avôs, tios, etc. Tais cortes são denominados *cortes no nível* e são, como pode ser visto, extremamente econômicos.

Baseando-se nesta idéia, J. Pearl propôs em [Pea80] um algoritmo de busca em árvores de jogo que calcula o nó mais à esquerda e, para cada nó direito, primeiro testa, e somente se necessário calcula o nó. Este algoritmo foi por ele batizado como *SCOUT*, mas para seu entendimento é preciso antes compreender dois algoritmos mais simples, denominados $\text{TEST}_{\text{AND}}$ e $\text{TEST}_{\text{OR}}$ .

Inicialmente, perceba-se que um teste $m(G)<k$ em uma árvore de jogo $G$ é, basicamente, equivalente a uma busca em uma árvore de jogo $G'$ na qual os nós terminais de $G$ têm seus valores convertidos pela seguinte regra:

$$\begin{array}{llll} \text{se} & c(t) < k, & \text{então} & c'(t) = +\infty \\ \text{se} & c(t) \geq k, & \text{então} & c'(t) = -\infty \end{array}$$

conforme ilusta a figura 3.14.

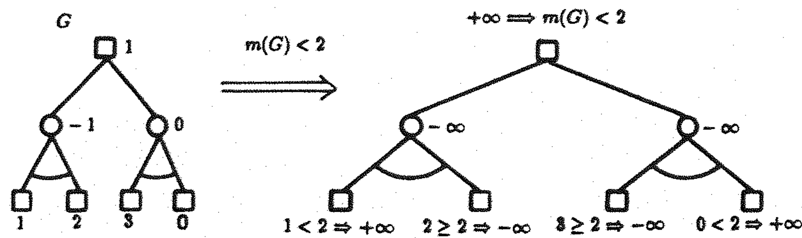



Figura 3.14: Equivalência básica de um teste sobre uma árvore de jogo $G$ com uma árvore de jogo $G'$ do tipo $m(G')=\pm\infty$.

O algoritmo TEST$_{\text{AND}}$ , concebido por Pearl, é mostrado na figura 3.15, com a adaptação ao esquema negamax conforme a sugestão de Campbell e Marsland em [CM83] [7]. Nesta descrição, PROXT $(t)$ tem comportamento idêntico a PROX $(t)$, e $V$ (respectivamente $F$) indica resultado do teste *verdadeiro* (respectivamente *falso*). A corretude do algoritmo TEST$_{\text{AND}}$ é assegurada pela proposição seguinte:

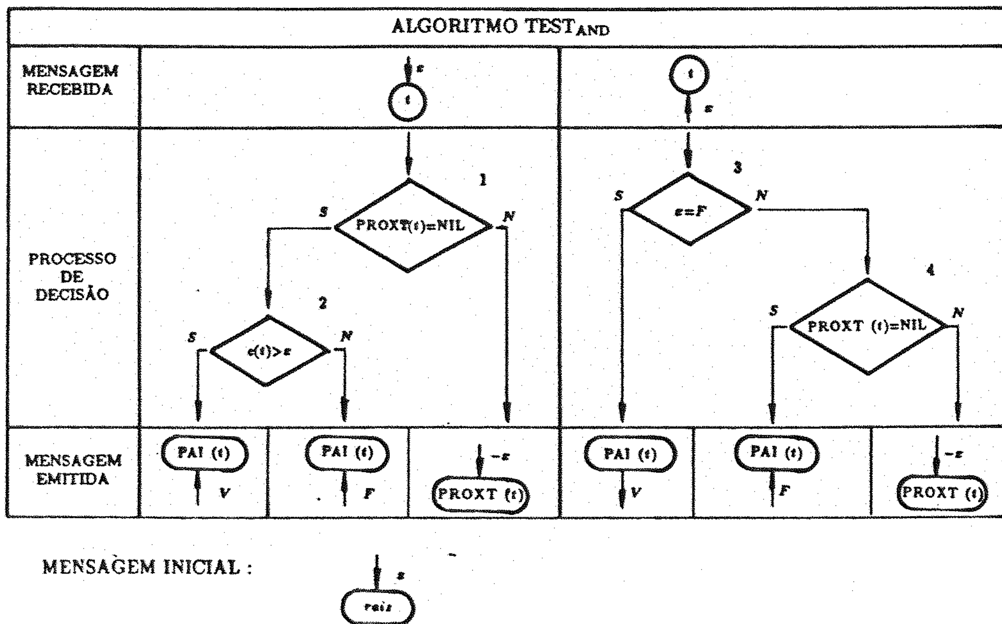



Figura 3.15: Algoritmo TEST$_{\text{AND}}$ .

PROPOSIÇÃO 3.6 *Seja $G$ uma árvore de jogo de raiz $t_0$ cujo pai (hipotético ou não) é $\bar{t}$. Então, se for enviada a $t_0$ uma mensagem descendente de valor $x$, após um tempo de processamento*

[7] Embora o algoritmo descrito nessa referência contenha incorreções.

*finito o algoritmo* $\text{TEST}_{\text{AND}}$ *enviará a* $\bar{t}$ *uma mensagem* $v$*, de valor* $V$ *ou* $F$*, que satisfaz:*

$$\begin{array}{lll} se\ t_0 \in V_{AND}(G) & v = V & se\ F(t_0) > x \\ & v = F & se\ F(t_0) \leq x \\ \\ se\ t_0 \in V_{OR}(G) & v = V & se\ F(t_0) \geq x \\ & v = F & se\ F(t_0) < x \end{array}$$

DEMONSTRAÇÃO : por indução na altura $h$ de $G$.

Suponha $h$=0. Então PROXT $(t_0)$=NIL e, como $t_0 \in V_{AND}(G)$, a condição (2) garante que se $c(t_0) = F(t_0) > x$, $\bar{t}$ recebe $V$, e caso contrário, $F$.

Suponha a proposição válida para árvores de altura $h' \leq h - 1$.
Sejam $t_1, t_2, \ldots, t_d$ os sucessores de $t_0$ e assuma $t_0 \in V_{AND}(G)$.
Examinando-se as condições (1) e (4), vê-se que a cada nó $t_i$ pode ser enviada uma mensagem descendente de valor $-x$. Pela hipótese de indução, cada um destes nós, $t_i \in V_{OR}(G)$, se for "chamado", "responde" com uma mensagem ascendente $v_i$ que satisfaz:

$$\begin{array}{ll} v = V & \text{se } F(t_i) \geq -x \\ v = F & \text{se } F(t_i) < -x \end{array}$$

Se algum $v_i$ for falso, então $-F(t_i) > x$, logo $F(t_0) \geq -F(t_i) > x$ e portanto $F(t_0) > x$, e o algoritmo retorna $V$.
Se todos os nós $v_i$ forem verdadeiros, então

$$-F(t_i) \leq x \qquad \forall i,\ 1 \leq i \leq d$$

logo $F(t_0) \leq x$ e o algoritmo, passando pela condição (4), retorna $F$.
O caso $t_0 \in V_{OR}(G)$ é inteiramente análogo. □

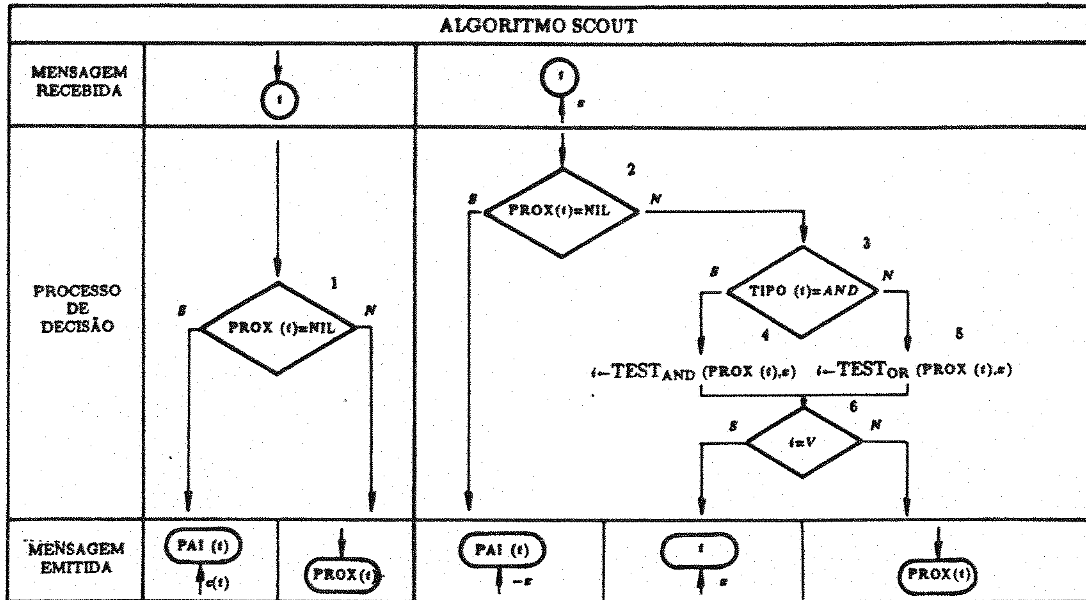
É necessário ainda considerar a existência de um outro algoritmo denominado $\text{TEST}_{\text{OR}}$ , que consiste essencialmente no algoritmo $\text{TEST}_{\text{AND}}$ modificado, onde a condição (2) é trocada por:

$$c(t) > x \longrightarrow c(t) \geq x$$

Tal troca, embora pequena, produz uma pequena modificação na proposição 3.6, descrita na proposição 3.7 abaixo, cuja demonstração é inteiramente análoga à anterior.

PROPOSIÇÃO 3.7 *Seja* $G$ *uma árvore de jogo de raiz* $t_0$ *cujo pai (hipotético ou não) é* $\bar{t}$*. Então, se for enviada a* $t_0$ *uma mensagem descendente de valor* $x$*, após um tempo de processamento finito o algoritmo* $\text{TEST}_{\text{OR}}$ *enviará a* $\bar{t}$ *uma mensagem* $v$*, de valor* $V$ *ou* $F$*, que satisfaz:*

$$\begin{array}{lll} se\ t_0 \in V_{AND}(G) & v = V & se\ F(t_0) \geq x \\ & v = F & se\ F(t_0) < x \\ \\ se\ t_0 \in V_{OR}(G) & v = V & se\ F(t_0) > x \\ & v = F & se\ F(t_0) \leq x \end{array}$$




MENSAGEM INICIAL :

Figura 3.16: Algoritmo SCOUT.

Posto isso, o algoritmo **SCOUT**, apresentado por Pearl em [Pea80] é mostrado na figura 3.16, onde os comandos (4) e (5) são chamadas para as rotinas $\text{TEST}_{\text{AND}}$ e $\text{TEST}_{\text{OR}}$ , respectivamente. A corretude do algoritmo é dada pela proposição:

PROPOSIÇÃO 3.8 *Seja G uma árvore de jogo de raiz $t_0$ cujo pai (hipotético ou não) é $\bar{t}$. Então, se for enviada a $t_0$ uma mensagem descendente, após um tempo de processamento finito o algoritmo SCOUT enviará a $\bar{t}$ uma mensagem $x$, tal que*

$$F(t_0) = x$$

DEMONSTRAÇÃO : por indução na altura $h$ de $G$.

Suponha $h=0$. É evidente que a condição (1) garante $c(t_0)=F(t_0)=x$.

Suponha a proposição válida para árvores de altura $h' \leq h-1$.
Sejam, novamente, $t_1, t_2, \ldots, t_d$ os sucessores de $t_0$. Como $h>0$, a condição (1) provoca uma mensagem descendente para $t_1=\text{PROX}\ (t_0)$. Pela hipótese de indução, $t_1$ envia em tempo finito a $t_0$ uma mensagem de valor $x_1=F(t_1)$.
Se $d=1$, é enviada a $\bar{t}=\text{PAI}\ (t_0)$ a mensagem $-x_1=-F(t_1)=F(t_0)$. Caso contrário, faz-se o teste (3). Suponha, sem perda de generalidade, que $t_0 \in V_{OR}(G)$. A proposição 3.7 garante que, dado que $t_2 \in V_{AND}(G)$,

$$\begin{aligned} i = V &\quad \text{se } F(t_2) \geq x_1 \\ i = F &\quad \text{se } F(t_2) < x_1 \end{aligned}$$

No caso $i=V$, temos $-F(t_2) \leq -x_1 = -F(t_1)$, e não há necessidade de se determinar o valor exato de $F(t_2)$, pois tal valor *não influencia o valor de* $F(t_0) = \max\{-F(t_1), -F(t_2), \ldots, -F(t_d)\}$.

Assim, é enviada a $t_0$ nova mensagem ascendente de valor $x_2=F(t_1)$, para continuar o processamento de $t_3, t_4, \ldots, t_d$. [8]

Quando $i=F$, está garantido $-F(t_2)>-F(t_1)$, e a mensagem descendente enviada para $t_2$, pela hipótese de indução, garante que $t_0$ receberá uma mensagem ascendente de valor $F(t_2)$.

Considerando que o processo repete-se sem alterações até o esgotamento dos sucessores, tem-se, ao final, o envio da mensagem ascendente de valor, para algum $j$ tal que $1 \leq j \leq d$,

$$-F(t_j) = \max\{-F(t_1), -F(t_2), \ldots, -F(t_d)\} = F(t_0)$$

o que conclui a demonstração, pois o caso $t_0 \in V_{AND}(G)$ é inteiramente análogo. □

De acordo com a definição 3.4, este algoritmo é não-direcional; há, contudo, que se reparar que este algoritmo faz busca da esquerda para a direita. Ocorre que, às vezes, a condição (6) faz com que nós sejam reexaminados. Embora isto possa parecer perda de tempo, uma análise matemática cuidadosa revela que a perda não é substancial (conforme será visto no próximo capítulo). Na verdade, o processo de teste e, se necessário, cálculo, é capaz de realizar cortes que o algoritmo Alpha-Beta não consegue (ainda que o contrário também ocorra, conforme se verá no próximo capítulo).

Quando do reexame de um nó terminal, pode-se evitar o seu recálculo se o seu valor tiver sido armazenado, em memória, no momento do seu primeiro exame. Isto evita o processo, normalmente custoso, de calcular o valor da função heurística $c(t)$ sobre esse nó; contudo, este método pode ser usado somente quando há grande quantidade de memória disponível.

Quando os valores dos nós terminais são discretos, os procedimentos $\text{TEST}_{\text{AND}}$ e $\text{TEST}_{\text{OR}}$ podem ser substituídos por uma chamada ao algoritmo Alpha-Beta, com valores $\alpha=x-1$ e $\beta=x$. Examinando-se a proposição 3.3, vê-se que facilmente o teste (6), $i=V$ é substituído por $i \geq x$, com resultados idênticos.

A figura 3.17 mostra a execução do algoritmo SCOUT sobre a árvore da figura 3.3. No exemplo são examinados 30 dos 81 nós, sendo que 4 deles duplamente, perfazendo portanto, 34 nós terminais.

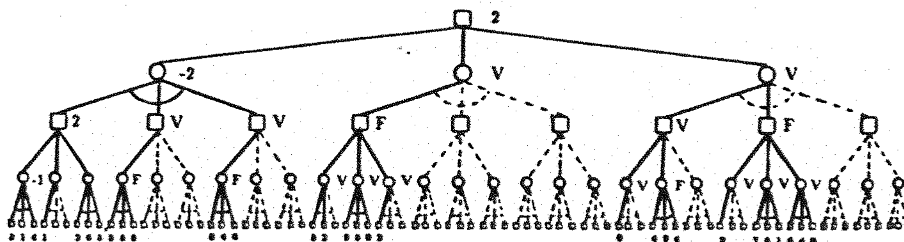


Figura 3.17: Árvore de jogo examinada pelo algoritmo SCOUT.

[8] Esta mensagem é, na verdade, um artifício para continuar o processamento.

## 3.8 Considerações sobre a Implementação dos Algoritmos

Conforme já foi dito, a descrição por mensagens dos algoritmos pode ocultar questões importantes de implementação. Neste caso específico, um fator importante é a quantidade de memória necessária para cada algoritmo.

Na medida em que os gastos de memória com o gerenciamento da árvore de jogo são comuns a todos os algoritmos, será analisada somente a memória gasta pelas demais variáveis, notadamente os vetores VALOR , BOUND , BETA e a lista $L$.

Os algoritmos Minimax, Bound e Alpha-Beta necessitam uma quantidade de memória proporcional ao tamanho do maior caminho da raiz a um nó terminal. Isto porque os mesmos percorrem a árvore de jogo direcionalmente: observe-se que é preciso armazenar um valor em uma posição $t$ do vetor VALOR somente enquanto se busca o valor dos descendentes de $t$; a busca de um irmão $t'$ de $t$ necessita somente guardar os valores atuais dos antecessores de $t'$. E este raciocínio é estendido diretamente para os vetores BOUND e BETA .

Será visto mais adiante que o algoritmo SSS* é o mais poderoso de todos os algoritmos vistos. Mas o seu "calcanhar de Aquiles" é justamente a quantidade de memória: a lista $L$ pode, no pior caso, ter o tamanho da "maior" OR-árvore solução contida na árvore de jogo. Por exemplo, em uma árvore de jogo uniforme de altura $h$ e grau $d$, a lista $L$ chega a possuir, ao mesmo tempo $d^{\frac{h}{2}}$ mensagens!

Por fim, analisando-se o algoritmo SCOUT, vê-se que não existem vetores auxiliares, logo sua implementação necessita de uma quantidade de memória que independe do tamanho da árvore de jogo em exame. Todavia, se for utilizado o método de armazenamento, descrito acima, que evita o recálculo de nós terminais, então pode ser necessária uma memória suplementar proporcional ao tamanho da maior AND-árvore solução contida na árvore, o que, como foi visto acima, significa $d^{\frac{h}{2}}$ posições (para uma árvore uniforme de grau $d$ e altura $h$).

# Capítulo 4

# Análise dos Algoritmos Seqüenciais

Dada a diversidade de algoritmos para busca em árvores de jogo, faz-se necessária a comparação da eficiência dos mesmos. Diversos problemas então aparecem: o tipo de árvore a ser considerado, as relações de dominância entre os algoritmos, o critério de comparação, o método de análise de complexidade, a validade de simulações experimentais e mesmo a otimização dos algoritmos propostos.

Um dos objetivos desta análise é preparar terreno para o estudo dos algoritmos paralelos. Assim, será evitado um exame por demais detalhado dos resultados analíticos e, notadamente, de suas demonstrações, por estas últimas serem, em geral, excessivamente longas e técnicas (muitas das quais resolvidas somente com a utilização de resultados fortes de matemática e estatística). Compreensivamente, será dada maior ênfase aos motivos e às situações nas quais um algoritmo consegue melhor desempenho que outro, e às potencialidades de cada diferente tipo de análise.

## 4.1 Modelos de Árvores de Jogo

Embora a maneira mais simples de comparar algoritmos para busca em árvores de jogo seja a sua implementação em programas reais, há de se convir que tal método é demasiado instável, dada a dificuldade de controle das situações apresentadas em um jogo real. Ademais, a irregularidade típica de um jogo torna quase impossível a análise da complexidade dos algoritmos.

Diversos *modelos de árvores de jogo* foram então propostos pelos pesquisadores da área. Os modelos utilizam *árvores uniformes*, e diferem entre si na maneira de atribuir valores aos nós terminais. Os valores são atribuídos antes do algoritmo de busca ser executado, e incorporam sempre algum mecanismo aleatório. De um modo geral, o objetivo dos modelos é criar árvores de jogo que se assemelhem às árvores provenientes de jogos reais, considerando-se inclusive, em alguns casos, as otimizações e heurísticas utilizadas pelos algoritmos de cada jogo. Por exemplo, é comum que programas reais de jogos determinem a ordem dos filhos de um determinado nó através de uma função heurística, que atribui valor mais alto a nós mais promissores, considerando somente a informação local. Alguns modelos, conforme se verá, incorporam mecanismos aleatórios que dão à árvore gerada uma distribuição que, de certa forma, equivale à árvore *efetivamente percorrida* por tais algoritmos.

Há muita disputa sobre qual é o modelo mais realista, ou seja, qual modelo tem maior capacidade de produzir árvores semelhantes às de um jogo real. Esta discussão, de natureza

estatística (e extremamente conturbada pela diversidade típica dos jogos), não será considerada aqui em profundidade. É, importante, no entanto, lembrar que alguns modelos prestam-se melhor à análise computacional que outros, e que por esse motivo são explorados em maior profundidade. Os principais modelos de árvores de jogo são mostrados a seguir.

### 4.1.1 O Modelo Maniqueísta

O *modelo maniqueísta* [1] de árvores de jogo, formalizado por Pearl em [Pea80], atribui aos nós terminais somente os valores $-\infty$ e $+\infty$ (derrota e vitória do 1º jogador, respectivamente), aleatoriamente, de forma que cada terminal tem uma probabilidade $P_0$ de receber $+\infty$ (e, portanto, $1 - P_0$ de receber $-\infty$).

É interessante perceber que este modelo tipifica jogos vantajosos para o 2º jogador. De fato, este jogador, por executar o último lance, é capaz de derrotar o primeiro abruptamente : basta que, entre os seus $d$ nós sucessores, somente *um* deles tenha o valor $-\infty$. O 1º jogador, ao contrário, só é capaz de ganhar se o seu adversário tiver todos os seus sucessores valorados com $+\infty$.

### 4.1.2 O Modelo Aleatório

No *modelo aleatório*[2] , conforme definido por Knuth e Moore no seu clássico artigo [KM75], os valores dos nós terminais são extraídos aleatoriamente de um intervalo $[-\infty, +\infty]$, discreto ou contínuo. Em simulações computacionais reais, os valores $-\infty$ e $+\infty$ são, respectivamente, o mínimo e o máximo número representável.

A simplicidade deste modelo tornou-o objeto favorito para análises de complexidade. Contudo, certos autores questionam sua validade como modelo para um jogo real, como em [MC82]. Os próprios autores Knuth e Moore apontam o fato de que experimentos comparativos realizados por Fuller, Gasching e Gillogly, descritos em [FGG73], mostraram que a busca em uma árvore de jogo real era mais rápida, em média, do que em uma árvore com modelo aleatório (utilizando o algoritmo Alpha-Beta descrito no capítulo anterior).

Note-se que a utilização de uma distribuição contínua de valores dos nós terminais é equivalente a atribuir a cada nó um elemento diferente do conjunto $\{1, 2, \ldots, d^h\}$.

### 4.1.3 Modelos de Valor Dependente da Ramificação

Alguns modelos de árvores de jogo foram propostos objetivando que nós com maior grau de parentesco também tivessem seus valores mais próximos. Espera-se, com isso, uma aproximação maior com o caráter posicional da maioria dos jogos, nos quais a sucessão de jogadas vai melhorando as chances de um jogador e diminuindo as do outro. São denominados, genericamente, *modelos de valor dependente da ramificação* [3].

**Modelo de Knuth e Moore** [4] : para cada uma das $d$ arestas que partem de um nó, atribui-se aleatoriamente um valor distinto do conjunto $\Omega = \{1/d^k, 2/d^k, \ldots, d/d^k\}$, onde $k$ é o número

[1] Do original, em inglês, *WIN-LOSS*.
[2] Do original, em inglês, *random uniform game tree*, ou, abreviadamente, *rug-tree*.
[3] Do original, em inglês, *branch-dependent score*.
[4] Conforme descrito por Newborn em [New77].

da jogada correspondente à aresta. O valor de cada um dos nós terminais é determinado somando-se o peso de todas as arestas do caminho que liga o nó à raiz da árvore.

**Modelo de Fuller, Gasching e Gillogly**[5] : idêntico ao modelo de Knuth, com o conjunto $\{1, 2, \ldots, d\}$ utilizado no lugar do conjunto $\Omega$.

**Modelo de Lindstrom** [6] : idêntico ao modelo de Knuth, com o conjunto $\Omega$ sendo substituído por valores extraídos aleatoriamente do intervalo $]-\infty, +\infty[$ , de forma que a soma não "estoure" jamais os valores mínimo e máximo representáveis.

### 4.1.4 Modelo de Distribuição Fortemente Ordenada

Conforme já foi comentado, diversos programas de jogos heuristicamente pré-ordenam os lances possíveis a partir de uma dada situação antes de examiná-los. Devido ao sucesso aparente desta heurística, um modelo teórico foi desenvolvido que *simula* o efeito dessa ordenação de lances nas árvores geradas artificialmente. Referências podem ser encontradas em [MC82].

O *modelo de distribuição fortemente ordenada* [7], é tal que a distribuição dos valores ao longo dos nós terminais satisfaz, para todos os nós da árvore, os seguintes critérios :

- 70% das vezes, o melhor movimento coincide com o 1º sucessor do nó;
- 90% das vezes, o melhor movimento está entre os 25% primeiros sucessores do nó.

Marsland e Campbell apontam em [MC82] evidências de que estes números aproximam-se dos conseguidos em jogos reais com auxílio de pré-ordenação de movimentos.

## 4.2 Métodos de Comparação

Uma questão central em qualquer análise de algoritmos é o método pelo qual se compara a performance dos algoritmos. Nos caso de busca em árvores de jogo, quatro métodos são normalmente utilizados:

**Número de Nós Terminais (NBP)** [8] : é um método extremamente comum, que consiste em se contar o número de nós terminais examinados pelo algoritmo. O fundamento motivador do NBP é a consideração de que a maior parte do tempo de processamento, no caso de algoritmos de busca em jogos reais, é gasto no cálculo da função heurística $c(t)$, que atribui a um nó terminal um "valor" quantitativo. Deste ponto de vista, o trabalho interno do algoritmo usaria um tempo proporcionalmente desprezível comparado ao da determinação dos valores terminais, e portanto um algoritmo que examina menos posições terminais é melhor do que outro que examina mais.
Este método é o mais largamente utilizado, tanto em estudos empíricos como em teóricos, embora, conforme foi visto, não compare diretamente a eficiência *per si* dos algoritmos.

[5] Idem.
[6] Conforme descrito em [Lin83].
[7] Do original, em inglês, *strongly ordered game tree*.
[8] Do original, em inglês, *number of bottom positions*.

**Fator de Ramificação** [9] : é, pode-se dizer, a versão assintótica do NBP, definida da seguinte maneira: seja um algoritmo $A$ e uma classe $\Phi$ de árvores de jogo; seja ainda $T_{A,d,h}$ o número médio de nós terminais examinado pelo algoritmo $A$ em árvores uniformes de grau $d$ e altura $h$, pertencentes à $\Phi$. Então o fator de ramificação do algoritmo $A$ sobre a classe $\Phi$, $\tau_A(d)$, é dado por:

$$\tau_A(d) = \lim_{h\to\infty} (T_{A,d,h})^{\frac{1}{h}}$$

Intuitivamente, $\tau_A(d)$ é uma medida do poder de redução do algoritmo $A$, no sentido de que $\tau_A(d)^h$ é o número médio de nós terminais examinados em uma árvore de grau $d$ e altura $h$, para $h$ suficientemente grande. Em outras palavras, o algoritmo $A$ seria equivalente a um algoritmo que examina *todos* os nós terminais de uma árvore hipotética de grau $\tau_A(d)$ e altura $h$.
Ainda, com base nesta definição, será dito que um algoritmo $A$ é *assintoticamente ótimo* sobre uma classe $C$ de algoritmos se

$$\tau_A(d) \leq \tau_B(d) \quad \forall B \in C$$

Embora este seja um dos métodos favoritos para a análise teórica, possui reduzida aplicação na prática, na medida em que a altura $h$ necessária para que $\tau_A(d)$ comece a convergir para o seu valor teórico pode, facilmente, ser impraticável computacionalmente.

**Número de Nós Visitados** : é uma variação do NBP na qual são considerados *todos* os nós pelos quais o algoritmo passa. É pouco utilizado devido à semelhança de caráter com NBP, e também porque mistura nós com tempos de processamento drasticamente diferentes.

**Tempo de CPU** : é o método que compara algoritmos diretamente pelo tempo gasto no processamento. Assim, os resultados são dependentes de diversos fatores indesejáveis, como, por exemplo, o tempo de processamento de um nó terminal, a codificação do algoritmo, as características da máquina, etc. Deste modo, este método é normalmente empregado somente para teste de um mesmo algoritmo sobre árvores de diferentes modelos e tamanhos.
Ressalte-se que este é o único dos métodos descritos que realmente detecta se um algoritmo é, intrinsecamente, mais *rápido* do que outro. Todos os demais métodos são indiretos, pois associam o tempo de processamento ao custo do cálculo da função $c(t)$ sobre os nós terminais.

Neste trabalho optou-se basicamente pelo uso do NBP, tendo em vista a simplicidade de medição e ainda o fato de que a medida de tempo de CPU em algoritmos paralelos é muito mais delicada. O fator de ramificação também será examinado, notadamente neste capítulo, pois o limite na altura da árvore simplifica sobremaneira o cálculo de complexidade, auxiliando, pois, a clarear o real poder dos algoritmos seqüenciais estudados.

## 4.3 Relações de Dominância

O primeiro aspecto a ser considerado na análise dos algoritmos seqüenciais são as relações de dominância entre eles. Sob este ponto de vista, o método de comparação utilizado é o NBP, número de posições terminais examinadas.

Será dito que um algoritmo $A$ *domina* um algoritmo $B$ se todos os nós terminais examinados por $A$ são também examinados por $B$. Esta definição procura captar o sentido de que $B$ pode

[9] Do original, em inglês, *branching factor*.

estar fazendo um trabalho desnecessário, embora, às vezes, o esforço extra dispendido para um algoritmo dominar outro implique em um custo muito elevado, quer em tempo de processamento, quer em memória ocupada. É evidente que esta relação de dominância é reflexiva, anti-simétrica (a menos de classes de equivalência) e transitiva, definindo portanto uma *ordem parcial* sobre a classe dos algoritmos de busca em árvores de jogo.

O algoritmo SCOUT merece atenção especial: em certos casos, este algoritmo pode examinar duas vezes o mesmo nó terminal; embora, estritamente, este fato não altere nosso conceito de dominância, caso realmente se deseje manter a associação entre dominância e desempenho, faz-se necessário considerar o algoritmo SCOUT que utiliza o método de armazenamento de valores de nós terminais descrito no capítulo anterior. Além disso, serão considerados como nós examinados pelo algoritmo SCOUT tanto os nós explorados por ele mesmo como os explorados pelos algoritmos $\text{TEST}_{\text{AND}}$ e $\text{TEST}_{\text{OR}}$ .

Considere-se inicialmente, então, o algoritmo Minimax. Já foi visto que o mesmo examina sempre todos os nós terminais da árvore, logo é evidente que todos os demais algoritmos dominam o Minimax.

A próxima relação a ser considerada é entre os algoritmos Bound e Alpha-Beta. Conforme foi dito na diferenciação de cortes rasos e profundos, os valores $b$ e $\beta$ têm a mesma função, enquanto que o valor $\alpha$ acrescenta ao Alpha-Beta a capacidade de informar níveis inferiores de resultados já conseguidos em níveis acima dos mesmos. A dominância de Bound por Alpha-Beta é formalmente demonstrada pelo teorema a seguir:

TEOREMA 4.1 *Seja $G$ uma árvore de jogo de raiz $t_0$. Suponha a execução do algoritmo Bound com mensagem inicial de valor $b$ e também a execução do algoritmo Alpha-Beta com mensagem inicial $\alpha : \beta$, $(\alpha \leq \beta)$. Se $b \geq \beta$, então todo nó terminal explorado por Alpha-Beta é também explorado por Bound.*

DEMONSTRAÇÃO : por indução na altura $h$ de $G$.

Se $d$=0, a raiz é explorada por ambos os algoritmos.

Suponha o teorema válido para árvores de altura $h' \leq h-1$.

Sejam $t_1, t_2, \ldots, t_d$ os sucessores de $t_0$ e $G_1, G_2, \ldots, G_d$ as sub-árvores de respectivas raízes. Então ao nó $t_1$ são enviadas, por Bound e Alpha-Beta, as mensagens de valores $+\infty$ e $-\beta : -\alpha$, respectivamente, logo $+\infty \geq -\alpha$ e, pela hipótese de indução, todo nó terminal de $G_1$ explorado por Alpha-Beta é também explorado por Bound.

Sejam $v_i$ e $u_i$ os valores do vetor VALOR $(t_0)$, após o comando (3) de Bound e Alpha-Beta, respectivamente. A demonstração das proposições 3.2 e 3.3 garantem que, para todo $i$, $1 \leq i \leq d$, se vale que $-F(t_j) < \beta$, qualquer que seja $1 \leq j \leq i$, então

$$u_i = \max\{\alpha, -F(t_1), -F(t_2), \ldots, -F(t_i)\}$$

e, que, sempre

$$v_i = \max\{-F(t_1), -F(t_2), \ldots, -F(t_i)\}$$

Se, nessas condições, não ocorre um corte em Alpha-Beta, então

$$v_i \leq u_i < \beta \leq b$$

não se caracterizando nenhum tipo de corte em Bound. A árvore $G_{i+1}$, se existir, será, então, examinada tanto por Bound como por Alpha-Beta, com mensagens $-v_i$ e $-\beta$:$-u_i$, respectivamente. Como $-u_i \leq -v_i$, vale a hipótese de indução, e Bound examina todos os nós explorados por Alpha-Beta.
Por outro lado, suponha-se um caso de corte em Bound, ou seja, $v_i \geq b$. Como $v_i = \max\{-F(t_1), -F(t_2), \ldots, -F(t_i)\}$, então existe $j$, $1 \leq j \leq i$, tal que $-F(t_j) \geq b$. Lembrando que $b \geq \beta$, a demonstração da proposição 3.3 garante que $u_i \geq \beta$, e logo o algoritmo Alpha-Beta também executa um corte. □

Observe-se ainda que, de acordo com a demonstração acima, podem ocorrer casos em que Alpha-Beta realiza cortes sem que Bound os faça. Tipicamente, isto ocorre quando $b > \beta$, $v_i = u_i = \max\{-F(t_1), -F(t_2), \ldots, -F(t_i)\}$, e se $b > v_i = u_i \geq \beta$.

É claro que as mensagens iniciais típicas dos dois algoritmos, $+\infty$ para o Bound e $-\infty$:$+\infty$ para o Alpha-Beta, satisfazem as condições do teorema 4.1, estabelecendo claramente a dominância do primeiro pelo último. O inverso não é válido, ou seja, Bound não domina Alpha-Beta (caso de algoritmos equivalentes), conforme mostra o estudo do capítulo anterior de cortes rasos e profundos, nas quais ocorre exatamente a situação descrita acima de ocorrência de corte em Alpha-Beta sem contrapartida em Bound.

Considerando-se todo o trabalho extra realizado por SSS* (manutenção da lista, priorização do ramo mais promissor, etc), era de se esperar que o mesmo apresentasse um desempenho melhor que outros algoritmos. E, de fato, demonstra-se que SSS* domina Alpha-Beta, conforme será visto a seguir.

Há que, primeiramente, fazer-se uma ressalva, para, inclusive, propiciar-se um melhor entendimento dos mecanismos do SSS*. Deve-se notar que, enquanto os cortes no nível AND são realizados pelo comando de cancelamento de mensagens, um corte no nível OR é proveniente de mensagens presentes na lista $L$ cujo valor é maior que o das mensagens de descendentes do nível OR. Em outras palavras, o valor das mensagens é que realiza os cortes rasos, enquanto que os cortes profundos são causados pelo sistema de ordenação da lista $L$.

Suponha-se que se deseja comparar as performances de SSS* e Alpha-Beta sobre, por exemplo, um nó $t \in V_{AND}(G)$, em uma situação na qual uma mensagem de valor $\alpha : \beta$ foi enviada para $t$. Para manter o mesmo nível de informação, SSS* deve tanto enviar a $t$ uma mensagem descendente de valor $\beta$, como "incluir" na lista $L$ uma mensagem ascendente *fictícia* para o pai de $t$ de valor $\alpha$. Esta mensagem é tal que, sempre que o valor de todas as mensagens entre descendentes de $t$ ficar abaixo de $\alpha$ (e, portanto $F(t) \leq \alpha$), então a mesma será processada (pois seu valor é o máximo da lista $L$), garantindo o retorno do valor $\alpha$, que claramente satisfaz $F(t) \leq \alpha$.

É importante observar que, a menos da raiz, essas mensagens aparecem naturalmente, dando ao SSS* o poder de realizar cortes rasos e profundos. Ainda, quando se utilizam as mensagens iniciais padrão, ou seja, $+\infty$ para SSS* e $-\infty : +\infty$ para Alpha-Beta, não há necessidade da mensagem fictícia de valor $\alpha = -\infty$, mesmo porque ela jamais seria processada.

Para a demonstração da dominância, contudo, é necessário considerar tal mensagem, a fim de garantir a indução, conforme o teorema a seguir. Demonstrações equivalentes encontram-se no artigo original de Stockman ([Sto79]), no artigo de Roizen e Pearl que analisa SSS* ([RP83]), e de uma maneira extremamente simples e elegante por Kumar e Kanal em [KK83], devido ao uso do formalismo de representação Branch & Bound.

TEOREMA 4.2 *Seja $G$ uma árvore de jogo de raiz $t_0$. Suponha a execução do algoritmo Alpha-Beta com mensagem inicial de valor $\alpha : \beta$, $(\alpha \leq \beta)$, e também a execução do algoritmo SSS* com mensagem inicial $b$. Suponha ainda que, quando a mensagem inicial de valor $b$ é processada por SSS*, existe uma mensagem ascendente de valor $a$ (fictícia ou não) na lista $L$, que satisfaz:*

$$\begin{aligned} \alpha \leq a \leq b \leq \beta \quad & se \quad t_0 \in V_{AND}(G) \\ e \quad -\beta \leq a \leq b \leq -\alpha \quad & se \quad t_0 \in V_{OR}(G) \end{aligned}$$

*Então todo nó terminal de $G$ explorado por SSS* é também examinado por Alpha-Beta.*

DEMONSTRAÇÃO : por indução na altura $h$ de $G$.

Se $d=0$, a raiz é explorada por ambos os algoritmos.

Suponha o teorema válido para árvores de altura $h' \leq h-1$.

Sejam $t_1, t_2, \ldots, t_d$ os sucessores de $t_0$ e $G_1, G_2, \ldots, G_d$ as sub-árvores de respectivas raízes; lembre-se ainda que o valor do topo da lista $L$ *nunca* cresce.

1º Caso $t_0 \in V_{AND}(G)$.

O nó $t_1$ é examinado através da mensagem $\alpha_1 : \beta_1$, onde $\alpha_1=-\beta$ e $\beta_1=-\alpha$. Por outro lado, o processamento da mensagem SSS* de valor $b$ para $t_0$ produz $d$ mensagens, de valor também igual a $b$, para os sucessores de $t_0$. Pelo 2º critério de ordenação (lexicográfico) da lista $L$, a mensagem para $t_1$ é a primeira a ser interpretada. Então vale que,

$$-\beta_1 \leq a \leq b \leq -\alpha_1 \quad , \text{pois} \quad -\beta_1 = \alpha \quad e -\alpha_1 = \beta \, .$$

Logo, pela hipótese de indução, todo nó terminal de $G_1$ examinado por SSS* é também explorado por Alpha-Beta. Seja a mensagem $\gamma_1$ enviada a $t_0$ por $t_1$. Se $F(t_1) \geq -\alpha$, tem-se $-\gamma_1 \leq \alpha$, e então ao nó $t_2$ é enviada a mensagem $\alpha_2 : \beta_2$, $\alpha_2=-\beta$ e $\beta_2=-\alpha$. Quanto ao SSS*, como $m(G_1) = -F(t_1) \leq \alpha \leq a \leq b$ está garantida a interpretação prioritária da mensagem de valor $b$ para $t_2$, novamente satisfazendo a hipótese do teorema.

Por outro lado, se $F(t_1) \leq -\beta$, então $-\gamma_1 \geq \beta$, o que faz Alpha-Beta enviar mensagem ascendente de $t_0$ para seu pai $\bar{t}$. O algoritmo SSS* comporta-se de maneira idêntica, pois como $m(G_1) = -F(t_1) \geq \beta \geq b$, então a mensagem ascendente de $t_1$ tem valor $\gamma_1=b$, sendo assim a próxima a ser interpretada (considerando-se o desempate lexicográfico), e causando o cancelamento de todas as demais mensagens. Em ambos os casos, os dois algoritmos examinam somente os nós terminais da sub-árvore $G_1$, mantendo a dominância de SSS*.

Finalmente, se $-\beta < F(t_1) < -\alpha$, então o nó $t_2$ é explorado com os valores $\alpha_2 : \beta_2$, onde $\alpha_2=-\beta$ e $\beta_2=-F(t_1)$. Se $m(G_1)=-F(t_1) \geq b$, o algoritmo SSS* terminará (conforme foi visto acima), confirmando, portanto, a hipótese. Caso contrário, a mensagem ascendente $\gamma_1=m(G_1)$ enviada por $t_1$ satisfaz:

$$-\beta_2 = -F(t_1) = \gamma_1 < b \leq \beta = -\alpha_2$$

e a hipótese de indução garante que Alpha-Beta examina todos os nós terminais de $G_2$ que forem explorados por SSS*.

É evidente que o raciocínio acima é facilmente imitável para os nós $t_3, t_4, \ldots, t_d$.

2º Caso $t_0 \in V_{OR}(G)$.

Novamente o nó $t_1$ satisfaz as condições do teorema sem restrições, pois a mensagem $\alpha_1 : \beta_1$, onde $\alpha_1=-\beta$ e $\beta_1=-\alpha$ é tal que

$$-\beta = \alpha_1 \leq a \leq b \leq \beta_1 = -\alpha$$

Como $t_0$ é nó do tipo OR, o algoritmo SSS* comporta-se como o Alpha-Beta, ou seja, só envia mensagem para $t_{i+1}$ depois de receber mensagem $\gamma_i$ de $t_i$.
Considerando-se separadamente as situações $\gamma_i \leq \alpha_i$, $\alpha_i < \gamma_i < \beta_i$ e $\beta_i \leq \gamma_i$, e utilizando raciocínios análogos ao caso anterior, completa-se sem maiores dificuldades esta demonstração. □

Não é excessivo repetir aqui que a condição do teorema 4.2 é satisfeita pelas mensagens iniciais padrão de SSS* e Alpha-Beta, de valores $+\infty$ e $-\infty : +\infty$, respectivamente. Além disso, a dominância também é estrita, como mostra a figura 4.1. Na árvore da figura, é fácil ver que Alpha-Beta obrigatoriamente examina o nó $t_1$, devido à direcionalidade, enquanto que SSS* não o faz.

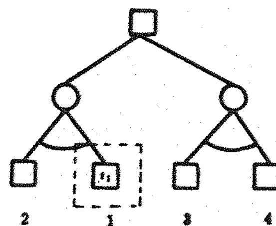



Figura 4.1: Contra-exemplo da dominação de SSS* por Alpha-Beta.

Pode-se demonstrar que o algoritmo SCOUT domina o algoritmo Bound, sendo essa prova aqui omitida por razões de espaço e de interesse. Mais interessante, contudo, é observar as relações entre SCOUT, Alpha-Beta e SSS*.

A figura 4.2 mostra uma árvore de jogo que possui um nó $t_1$ que é examinado tanto por Alpha-Beta como por SSS* (observe-se que $t_0$ é do tipo OR, logo o nó $t_2$ só é explorado depois de completado o exame do nó $t_1$ e de seus sucessores). Por outro lado, quando é feita a busca $\text{TEST}_{\text{AND}}$ $(t_0, 5)$, o nó $t_1$ retorna $V$ sem examinar $t_{11}$, e o nó $t_2$, por retornar $F$, garante que todo o ramo não será posteriormente explorado. Assim, nem Alpha-Beta nem SSS* dominam SCOUT.

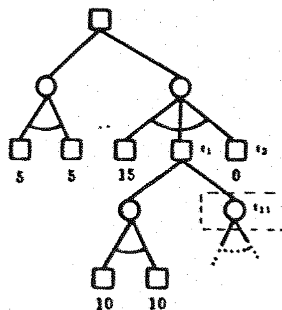



Figura 4.2: Contra-exemplo da dominação de SCOUT por Alpha-Beta.

O oposto também é verdadeiro, conforme mostra a figura 4.3. O nó $t_1$ é explorado durante $\text{TEST}_{\text{AND}}$ $(t_0, 0)$, mas o algoritmo Alpha-Beta não o explora, pois o nó $t_0$ recebe a mensagem $-25 : -10$ de seu pai, e logo o valor 5 do irmão de $t_1$ causa seu corte. Como SSS* domina

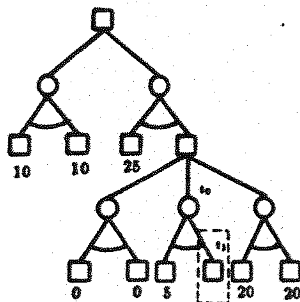
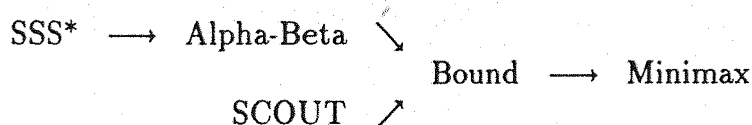



Figura 4.3: Contra-exemplo da dominação de Alpha-Beta por SCOUT.

Alpha-Beta, então SCOUT não o domina (pois a dominância é transitiva). Conclui-se, então, que SCOUT não domina nem Alpha-Beta nem SSS*.

Se for adotada a notação $A \longrightarrow B$ , para indicar que o algoritmo $A$ domina o algoritmo $B$, as relações de dominância são resumidas pelo esquema abaixo:




## 4.4 Resultados sobre Árvores Maniqueístas

Árvores de jogo baseadas no modelo maniqueísta de distribuição de nós terminais têm pouca aplicação prática, na medida em que quase nenhum jogo pode ser simulado por árvores cujos valores são sempre $-\infty$ ou $+\infty$. No entanto, os resultados teóricos provenientes do estudo das árvores maniqueístas são determinantes dos resultados sobre árvores aleatórias examinados na próxima seção.

Considere-se pois uma árvore maniqueísta com probabilidade $P_0$ de que um nó terminal receba o valor $+\infty$. Em 1978, G.M. Baudet determinou o número médio de nós terminais examinados pelo algoritmo Alpha-Beta sobre uma árvore com tais características ([Bau78b], p. 186, teorema 3.2). Denominando-se esse valor como $T_{d,h}(P_0)$ Baudet nele observou um estranho comportamento em relação a $P_0$, qual seja, a existência de um valor específico de $P_0$, $\xi_d$, para o qual havia uma significativa degradação da performance. Esse valor $\xi_d$ é a única raiz positiva, no intervalo $]0,1[$, da equação

$$x^d + x - 1 = 0$$

e a relação observada foi, para $h$ suficientemente grande,

$$T_{d,h}(0) = T_{d,h}(1) = \Theta\left(d^{\lceil \frac{h}{2} \rceil}\right)$$

enquanto que

$$T_{d,h}(\xi_d) \sim \left(\frac{d}{\ln d}\right)^h$$

No seu artigo [Pea80], Pearl fez uma análise profunda da questão. Um primeiro resultado (p. 115), diz respeito à natureza do jogo: *seja $P_n(P_0)$ a probabilidade do 1º jogador ganhar na jogada n (h=2n), dada uma árvore maniqueísta de distribuição $P_0$. Então,*

$$\lim_{n\to\infty} P_n(P_0) = \begin{cases} 1 & se\ P_0 > \xi_d \\ \xi_d & se\ P_0 = \xi_d \\ 0 & se\ P_0 < \xi_d \end{cases}$$

Isso mostra que as chances de vitória de cada um dos jogadores são drasticamente alteradas pelos valores de $P_0$. A figura 4.4 [10] mostra os valores de $P_n(P_0)$ em uma árvore binária. O comportamento de "função escada" de $P_n(P_0)$ é reforçado por outro resultado de Pearl, que mostra que a inclinação no ponto $\xi_d$ cresce *exponencialmente* com $n$.

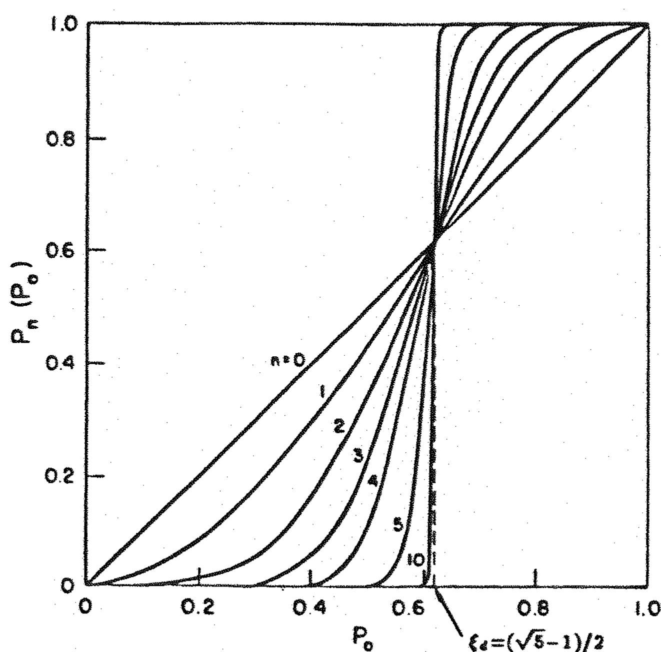



Figura 4.4: Probabilidade de vitória do 1º jogador em árvores maniqueístas binárias de distribuição $P_0$ (fonte: [Pea80], p. 116).

A continuação da análise depende da definição do algoritmo **SOLVE**, específico para busca em árvores maniqueístas:

[10] Extraída de [Pea80], p. 116.

Algoritmo SOLVE: um nó $t$ envia aos seus sucessores, da esquerda para a direita, mensagens descendentes até que:

- se $t$ é terminal, $t$ retorna $c(t)$ ao seu pai;
- se $t \in V_{AND}(G)$ e um de seus sucessores retorna $+\infty$, então $t$ envia $+\infty$ a seu pai, e no caso contrário, envia $-\infty$;
- se $t \in V_{OR}(G)$ e um de seus sucessores retorna $-\infty$, então $t$ envia $-\infty$ a seu pai, e no caso contrário, envia $+\infty$.

Como conseqüência do resultado sobre a probabilidade de vitória do 1º jogador, Pearl mostra ([Pea80], p. 124, corolário 1), que o fator de ramificação do algoritmo SOLVE, $r_{SOLVE}(d)$, é tal que

$$r_{SOLVE}(d) = \begin{cases} d^{\frac{1}{2}} & se \ \ P_0 \neq \xi_d \\ \frac{\xi_d}{1-\xi_d} & se \ \ P_0 = \xi_d \end{cases}$$

onde, conforme resultado de Baudet,

$$\frac{\xi_d}{1-\xi_d} \sim \frac{d}{\ln d}$$

Como já foi visto, no capítulo 2, que o limite inferior de complexidade para árvores de mérito $\pm\infty$ é $d^{\frac{h}{2}}$, conclui-se que o *algoritmo SOLVE é assintoticamente ótimo* para $P_0 \neq \xi_d$. Além disso, Pearl mostra que todos os algoritmos *direcionais* têm fator de ramificação maior ou igual a SOLVE.

Finalmente, M. Tarsi, no artigo [Tar83], colocou um ponto final na questão, demonstrando que SOLVE é tão bom quanto *qualquer* outro algoritmo, ou seja, que o *fator de ramificação de qualquer algoritmo A sobre uma árvore maniqueísta de distribuição $P_0$ satisfaz:*

$$r_A(d) \geq \begin{cases} d^{\frac{1}{2}} & se \ \ P_0 \neq \xi_d \\ \frac{\xi_d}{1-\xi_d} & se \ \ P_0 = \xi_d \end{cases}$$

## 4.5 Resultados sobre Árvores Aleatórias

O modelo aleatório é o mais comumente utilizado nas análises dos algoritmos para busca em árvores de jogo. De uma forma geral, a pesquisa científica concentrou-se na determinação do fator de ramificação dos algoritmos, tanto para o caso discreto como para o caso contínuo.

Em seu artigo [KM75], Knuth e Moore realizam o primeiro grande estudo sobre o assunto, examinando o algoritmo Bound sobre *árvores aleatórias com distribuição contínua de valores nos nós terminais*, e chegando à conclusão que:

$$r_{Bound}(d) = \Theta\left(\frac{d}{\ln d}\right)$$

Knuth e Moore não acreditavam que os fatores de ramificação de Bound e Alpha-Beta fossem de ordens diferentes; não conseguem, no entanto, provar essa afirmação, e lançam assim

sua famosa conjectura acerca da natureza da relação entre os dois algoritmos: *"... cortes profundos provavelmente têm somente um efeito de segunda ordem na eficiência [do algoritmo Alpha-Beta]."* [11].

### 4.5.1 O Fator de Ramificação do Algoritmo Alpha-Beta

A conjectura de Knuth e Moore foi investigada passo a passo por diversos pesquisadores, e comprovada ao cabo de alguns anos. Os primeiros resultados foram obtidos por Baudet em [Bau78b], no qual são inicialmente apresentadas fórmulas para o número médio de nós terminais examinados pelo algoritmo Alpha-Beta, tanto no caso discreto como no contínuo (ver [Bau78b], p. 184 e p. 191, respectivamente).

A complexidade das fórmulas dificulta bastante sua simplificação, mas mesmo assim é provado que, tanto para o caso discreto como para o contínuo,

$$\frac{d}{\ln d} \sim \frac{\xi_d}{1-\xi_d} \leq \tau_{\alpha-\beta}(d) \leq \sqrt{\frac{d.\xi_d}{(1-\xi_d)}} \sim \frac{d}{\sqrt{\ln d}}$$

Como Alpha-Beta domina Bound, estreita-se ainda mais a relação, obtendo-se

$$\frac{d}{\ln d} \sim \frac{\xi_d}{1-\xi_d} \leq \tau_{\alpha-\beta}(d) \leq \tau_{Bound}(d) = \Theta\left(\frac{d}{\ln d}\right)$$

donde se conclui que

$$\tau_{\alpha-\beta}(d) = \Theta\left(\frac{d}{\ln d}\right)$$

É J. Pearl quem, em seus artigos de 1980 e 1982 ([Pea80] e [Pea82]), resolve, de forma conclusiva, a questão:

- **caso discreto**: (ver [Pea80], p. 137, teorema 7) se existe um valor $v^*$ da distribuição que possua probabilidade $\xi_d$ de ser atribuído a um nó terminal, então

  $$\tau_{\alpha-\beta}(d) = \frac{\xi_d}{1-\xi_d} \sim \frac{d}{\ln d}$$

  senão

  $$\tau_{\alpha-\beta}(d) = d^{\frac{1}{2}}$$

- **caso contínuo**: (ver [Pea82], p. 563, teorema 2)

  $$\tau_{\alpha-\beta}(d) = \frac{\xi_d}{1-\xi_d} \sim \frac{d}{\ln d}$$

Observe-se que é possível que o fator de ramificação de Alpha-Beta no caso discreto não seja $d^{\frac{1}{2}}$ (embora não seja comum), o que confirma a desigualdade de Baudet.

[11] Traduzido de [KM75], p. 310.

### 4.5.2 O Limite Inferior do Fator de Ramificação de Árvores Aleatórias

Conhecendo-se, pois, o fator de ramificação do algoritmo Alpha-Beta, e sabendo-se ainda que ele é dominado pelo SSS* e que possui uma relação indeterminada com o SCOUT, é natural que surja a questão: será que estes algoritmos (ou outros quaisquer) possuem um fator de ramificação inferior ao de Alpha-Beta ?

Considere-se inicialmente uma árvore de jogo aleatória $G$, um valor $v$ da distribuição de $G$, e uma árvore de jogo maniqueísta $G'$ de estrutura idêntica a $G$, cuja função de valoração dos nós terminais $c'(t)$ obedece:

$$c'(t) = \begin{cases} +\infty & se \;\; c(t) > v \\ -\infty & se \;\; c(t) \leq v \end{cases}$$

Ora, a execução do algoritmo SOLVE sobre $G'$ examina exatamente os mesmos nós que a execução de $TEST_{AND}$ sobre $G$. Logo ambos têm o mesmo fator de ramificação.

A observação fundamental é feita por Pearl: *"Todo algoritmo que calcula uma árvore de jogo precisa examinar ao menos tantos nós quantos os requeridos para testar se o valor da raiz é maior que alguma referência v."* [12]. Assim, nenhum algoritmo pode ter um fator de ramificação inferior ao de SOLVE. Como este, de acordo com os resultados de Tarsi, é assintoticamente ótimo sobre qualquer algoritmo, nenhum outro algoritmo de busca em árvores aleatórias tem fator de ramificação menor do que o de SOLVE.

No caso contínuo, existe $v^*$ tal que a probabilidade de um nó terminal ter um valor $v$ satisfazendo a desigualdade $v \leq v^*$ é exatamente $\xi_d$ (pois a distribuição é contínua). Logo existe uma situação de ocorrência do pior caso de SOLVE e, portanto, o fator de ramificação de qualquer algoritmo $A$ sobre uma árvore aleatória com distribuição contínua satisfaz:

$$r_A(d) \geq \frac{\xi_d}{1 - \xi_d}$$

Similarmente, no caso discreto, se existe $v^*$ com probabilidade $\xi_d$ de atribuição, então qualquer algoritmo de busca $A$ executado sobre árvores com este tipo de distribuição é tal que:

$$r_A(d) \geq \frac{\xi_d}{1 - \xi_d}$$

Caso este valor não exista, há uma melhora de performance, que passa a ser:

$$r_A(d) \geq d^{\frac{1}{2}}$$

### 4.5.3 O Fator de Ramificação de SSS* e SCOUT

Os resultados acima garantem que o algoritmo Alpha-Beta é assintoticamente ótimo em relação a qualquer algoritmo, considerando-se busca em árvores de jogo com modelo aleatório de valoração de nós terminais.

Já foi demonstrado que SSS* domina Alpha-Beta, logo, para toda árvore aleatória:

$$r_{SSS*}(d) = r_{\alpha-\beta}(d)$$

[12] Traduzido de [Pea80], p. 128.

A demonstração deste fato, sem utilizar diretamente a otimalidade de Alpha-Beta, foi feita por Roizen e Pearl em [RP83], incluindo ainda uma fórmula para o cálculo do número médio de nós terminais examinados por SSS* (p. 212, teorema 4.2).

Quanto ao algoritmo SCOUT, a determinação de seu fator de ramificação é anterior a do Alpha-Beta, tendo um valor idêntico ao deste:

$$r_{SCOUT}(d) = r_{\alpha-\beta}(d)$$

A demonstração dessa assertiva, feita por Pearl em [Pea80], foi, na verdade, o caminho utilizado pelo autor para garantir que o limite inferior do fator de ramificação era exatamente o calculado. É interessante notar que, nessas contas, o número de nós terminais examinados por SCOUT *considera o reexame* dos nós, o que mostra que essa duplicação de tarefas, ainda que possa ocorrer, tem efeito marginal sobre a performance do algoritmo.

### 4.5.4 Crítica ao Uso do Fator de Ramificação em Modelos Aleatórios

É sempre perigoso trabalhar, em computação, com limites para o infinito, e mais ainda, em se tratando da altura de árvores uniformes. Desse modo, os fatores de ramificação de algoritmos, conforme foram calculados acima, devem ser usados criteriosamente, pois "acobertam" algumas questões bastante importantes.

A primeira delas diz respeito às conseqüências de expandir a altura de uma árvore aleatória até valores muito grandes: surpreendentemente, o mérito da árvore de jogo converge para um valor fixo! Este fato é demonstrado também por Pearl ([Pea80], p. 117, teorema 1), que determina o valor fixo de convergência:

- **caso contínuo:** à medida que aumenta a altura $h$ de uma árvore aleatória com distribuição contínua no intervalo $[a,b]$, o mérito dessa árvore tende ao valor $v^*=(1-\xi_d)(b-a)+a$, ou seja, a $(1-\xi_d)$-fração de $[a,b]$;

- **caso discreto:** se os valores dos nós terminais são aleatoriamente escolhidos de um conjunto $v_1 < v_2 < \ldots < v_m$ então, para $h$ suficientemente grande, o mérito dessas árvores $G$ tende para o valor $v_i$ cuja probabilidade de ocorrência $P(m(G) \leq v_i)$ satisfaça:

  $$P(m(G) \leq v_{i-1}) < 1 - \xi_d < P(m(G) \leq v_i) \;;$$

  se existir $v_i$ tal que $P(m(G) \leq v_i)=1-\xi_d$, então o mérito de $G$ pode convergir, com igual probabilidade, para $v_i$ ou para $v_{i+1}$.

O próprio Pearl comenta:

> *"Parece que a aplicação repetitiva de operações MIN-MAX alternantes sobre os valores terminais tem o efeito de filtrar suas incertezas, até que o resultado emergente nos níveis mais altos da árvore é quase um fixo, pré-determinado valor."* [13]

[13] Traduzido de [Pea80], p. 117.

Em outras palavras, expandir tais árvores transforma-as em "jogos com cartas marcadas"; e toda a análise baseada no fator de ramificação de algoritmos de busca em árvores de jogo uniformes, geradas no modelo aleatórios, engloba, de certa forma, esta propriedade pouco comum a jogos reais.

Outro fato que deve causar estranheza é a brusca transição do fator de ramificação do caso discreto para o contínuo. Essa descontinuidade não deve ser tomada como "real", na medida em que, quanto maior for o número de valores diferentes em uma distribuição discreta, maior será a altura $h$ necessária para que o fator de ramificação "assuma" o valor $d^{\frac{1}{2}}$. Por outro lado, para árvores com grau razoável (do ponto de vista de jogos), a razão entre esses dois valores não ultrapassa jamais o dobro, conforme a relação abaixo:

$$\text{se } 3 \leq d \leq 74 \quad \text{então} \quad 1 \leq \frac{\left(\frac{d}{\ln d}\right)}{d^{\frac{1}{2}}} \leq 2$$

Finalmente, quando são comparados algoritmos diferentes com o mesmo fator de ramificação, há que se levar em conta a eficiência de cada um deles sobre árvores com tamanhos práticos. Por exemplo, a proporção entre o número médio de nós terminais examinados por Alpha-Beta e SSS* é limitada por:

$$1,1 \leq \frac{T_{\alpha-\beta,d,h}}{T_{SSS*,d,h}} \leq 3 \quad \text{para} \quad 2 \leq d \leq 20 \;\text{ e }\; 2 \leq h \leq 20 \;.$$

Estes resultados são fruto das fórmulas exatas de número médio de nós examinados que foram citadas ao longo desta seção, não se tratando de resultados de simulações.

Deste último ponto de vista, alguns resultados interessantes podem ser encontrados em [CM83]. A tabela 4.1 foi construída a partir das figuras 10 e 12 dessa referência, e comparam os três algoritmos assintoticamente ótimos sobre árvores aleatórias com o algoritmo Minimax e com o fator de ramificação.

| Número de Nós Terminais Examinados (NBP) | | |
|---|---|---|
| | Tamanho da Árvore | |
| ALGORITMO | $h = 2$ | $h = 4$ |
| Minimax | 567 | 331776 |
| SCOUT | 191 | 11313 |
| Alpha-Beta | 161 | 10857 |
| SSS* | 100 | 5740 |
| $(d/\ln d)^h$ | 57 | 3252 |

Tabela 4.1: Simulação dos algoritmos ótimos sobre dois tipos de árvores aleatórias de grau $d = 24$ (fonte: [CM83], p. 10 e 12).

## 4.6 Resultados sobre Outros Modelos

Árvores de jogo com modelo de valoração dos nós terminais dependente da ramificação foram,

de um modo geral, pouco estudadas pelos pesquisadores da área. Contudo, pode-se destacar alguns resultados:

**Modelo de Knuth e Moore** : os próprios autores, em [KM75], determinam o número médio de nós terminais examinados pelo algoritmo Alpha-Beta (p. 323, teorema 6), que é mostrado satisfazer um crescimento da ordem de $C_d.d^{\frac{h}{2}}$, onde $C_d$ é uma constante que depende exclusivamente do grau da árvore $d$.

**Modelo de Fuller, Gasching e Gillogly** : em [New77], Newborn faz algumas explorações em cima deste modelo, esbarrando, contudo, em dificuldades consideráveis. De qualquer forma, é determinada a ordem do número médio de nós terminais examinados pelo algoritmo Alpha-Beta para árvores de altura igual a 2, que é $O(dH_d)$, de altura igual a 3, $O(d^2)$, e de altura igual a 4, que se situa entre $O(d^2)$ e $O(d^2 {H_d}^2)$, onde $H_d = 1 + \frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{d-1} + \frac{1}{d}$.

Árvores de jogo com distribuição fortemente ordenada carecem de resultados analíticos. Resultados experimentais (simulações) são apresentados por Campbell e Marsland em [CM83], que neste artigo chegam à conclusão que árvores com tais características devem ser examinadas pelos algoritmos Alpha-Beta e SCOUT e não pelo SSS*. Ocorre que, neste caso, suas performances são praticamente equivalentes a do algoritmo SSS* e, portanto, são melhores do que este pois usam significativamente menos memória.

## 4.7 Algumas Otimizações nos Algoritmos Seqüenciais

Após a análise dos algoritmos seqüenciais, é interessante se discutir algumas otimizações que podem ser feitas nos mesmos, visando a diminuição, em alguns casos, do tempo de processamento, e, em outros, da quantidade de memória utilizada.

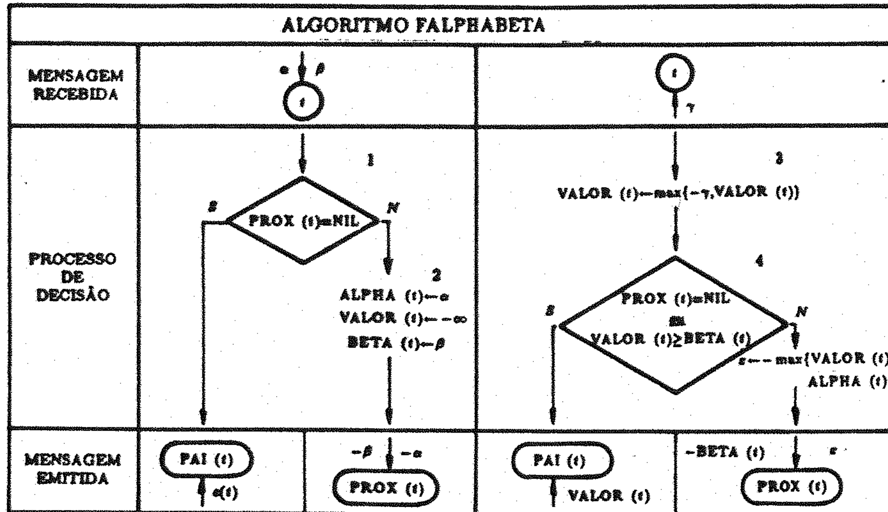
**Falphabeta** [14] : esta otimização, sugerida por Fishburn e Finkel em [FF80], diz respeito a aumentar a quantidade de informação disponível sempre que um nó examinado pelo algoritmo Alpha-Beta com *janela* $\alpha : \beta$ retorna um valor $\gamma$ *fora* da janela de busca. Para tanto, utiliza-se um vetor ALPHA ($t$), e se modifica o algoritmo conforme mostra a figura 4.5.
Nestas condições, demonstra-se que o valor $\gamma$ enviado ao pai do nó inicial $t_0$ satisfaz a desigualdade abaixo, mais forte do que a da proposição 3.3:

$$\begin{array}{ll} \gamma \leq F(t_0) & \text{se } F(t_0) \leq \alpha \\ \gamma = F(t_0) & \text{se } \alpha < F(t_0) < \beta \\ \gamma \geq F(t_0) & \text{se } F(t_0) \geq \beta \end{array}$$

Esta otimização não afeta o número de nós examinados. Contudo, o fato do valor $\gamma$ poder ser menor que $\alpha$ ou maior que $\beta$ permite uma exploração com vantagens por alguns algoritmos paralelos, conforme se verá no próximo capítulo.

**Lalphabeta** [15] : é uma otimização simples mas muito útil em programas reais. Como nestes casos o que se deseja realmente descobrir é o próximo lance a ser realizado (e não explicitamente

[14] De *fail-soft alpha-beta search*.
[15] De *last-move-with-minimal-window*.



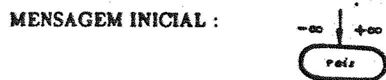




Figura 4.5: Algoritmo Falphabeta

calcular o mérito da árvore), o *último sucessor da raiz* é examinado somente pelo algoritmo $\mathrm{TEST}_{\mathrm{AND}}$ . Caso receba uma mensagem de valor $F$, então o último lance é o escolhido, sem que haja necessidade de calcular o seu valor exato; caso contrário, sabe-se que o valor da sub-árvore correspondente não é maior que o valor de alguma já encontrada, à qual corresponde, então, o melhor lance. Esta idéia, original também de Fishburn e Finkel, pode ser encontrada, com outra apresentação, em [FF80].

**Alpha-Beta/SSS*** : esta otimização, que visa diminuir a quantidade de memória necessária, é sugerida pelo próprio Stockman (criador do SSS*) em [Sto79]. A idéia central consiste em utilizar Alpha-Beta até uma determinada profundidade da árvore em exame, e SSS* para as partes mais "baixas" da árvore. Evita-se assim um excessivo gasto de memória, mas ainda se utiliza a redução, proporcionada por SSS*, no número de nós terminais examinados. Resultados obtidos por Campbel e Marsland ([CM83]) mostram que esta versão híbrida de Alpha-Beta e SSS* situa-se a meio termo entre as performances de ambos.

# Capítulo 5

# Algoritmos Paralelos para Busca em Árvores de Jogo

O advento de máquinas de múltiplos processadores tem gerado muito interesse, em todos os campos da computação, nos modos pelos quais as tarefas realizadas nos computadores convencionais podem ser transportadas, com ganho significativo de tempo, para os novos equipamentos.

Na área de jogos, os primeiros trabalhos datam de 1978, com um grande volume de artigos publicados em 1982 e 1983. É interessante notar que estas pesquisas foram feitas *antes* que os seus autores tivessem acesso a máquinas paralelas concretas: quase sempre foram utilizados simuladores de paralelismo, o que, de certa forma, "idealiza" o problema. Assim, tanto o desenvolvimento como a análise dos algoritmos propostos não se preocuparam com problemas típicos de multiprocessadores, tais como, comunicação, acesso concorrente à memória, *deadlock*, etc.

Essa caracterização idealizada dos algoritmos é mantida neste trabalho, em parte pela impossibilidade de utilização de máquinas paralelas reais, mas também pelo seu valor teórico: conforme será visto mais adiante, paralelizar algoritmos de busca em árvores de jogo traz à tona diversos problemas e propriedades importantes de qualquer processo de paralelização, independentemente do dispositivo empregado.

Para facilitar o entendimento dos algoritmos propostos, este capítulo se inicia com uma descrição sucinta das arquiteturas paralelas utilizadas; segue-se a explicação dos dois métodos básicos empregados no processo de paralelização de todos os algoritmos e, finalmente, a descrição dos principais algoritmos paralelos para busca em árvores de jogo.

## 5.1 Arquiteturas Paralelas

Dois tipos básicos de arquitetura de computadores com múltiplos processadores são considerados neste trabalho, ambos dos quais pertencentes à categoria MIMD [1]:

**Processadores em Árvore** : é a arquitetura na qual os processadores são dispostos na forma

[1] Do original, em inglês, *Multiple-Instruction Stream, Multiple Data Stream*, de acordo com a taxonomia proposta por Flynn em [Fly66].

de uma árvore uniforme de grau $f$ e altura $n$ (portanto $k=\frac{f^{n+1}-1}{f-1}$ processadores), considerando-se as arestas da árvore como canais bi-direcionais de comunicação conforme ilustra a figura 5.1.

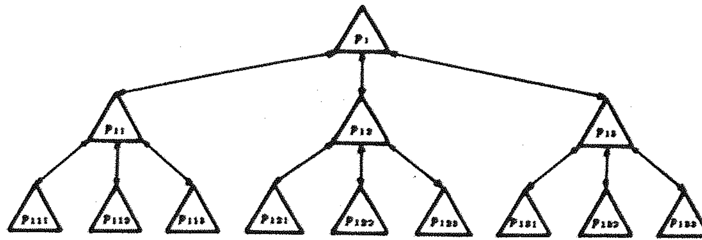


Figura 5.1: Arquitetura de processadores em árvore de grau $f$=3 e altura $n$=2.

Dados um processador $p_i$ e seus sucessores $p_{i1}, p_{i2}, \ldots, p_{if}$, será dito que $p_i$ é o *processador mestre* de cada um dos $p_{ij}$, e que cada $p_{ij}$ é um *processador escravo* de $p_i$. Além disso, cada unidade processadora $p_i$ será suposta constituída de:

- processador;
- memória de acesso exclusivo do processador, de forma que os demais processadores só obtêm dados dessa memória através de mensagens a $p_i$;
- sistema de gerenciamento de mensagens, tanto com os seus processadores escravos, como com seu processador mestre.

A topologia de árvore é utilizada em virtude de sua similaridade com o problema; é claro que todos os algoritmos para processadores em árvore mostrados podem ser executados em máquinas de outras topologias, desde que a constituição de cada unidade seja semelhante, e que a topologia da máquina seja eficientemente configurável na forma de árvore (como, por exemplo, se for uma topologia de *grade* ou de *hipercubo*).

**Processadores com Memória Compartilhada** : nesta arquitetura, os canais de comunicação são substituídos por uma memória comum a todos os processadores (ver figura 5.2). Neste caso, não há topologia de ligação entre os processadores, mas mecanismos de gerenciamento da memória compartilhada devem ser providos, de forma a evitar alteração e/ou acesso simultâneo a uma mesma região.

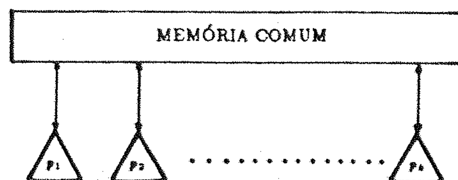



Figura 5.2: Arquitetura de processadores com memória compartilhada.

Usualmente, os processadores executam o mesmo processo, mas assincronamente. Cada unidade processadora possui, também neste caso, alguma memória de uso exclusivamente local.

## 5.2 Métodos Básicos de Paralelização

No decorrer do trabalho de pesquisa e entendimento dos algoritmos paralelos para busca em árvores de jogo, foi-se tornando evidente que, apesar da enorme diversidade aparente, todos os algoritmos se fundamentam em somente dois princípios básicos. Estes princípios ligam-se a uma característica marcante dos algoritmos paralelos, que é a de não conterem métodos originais, constituindo-se somente em *paralelizações* dos algoritmos seqüenciais.

Talvez um dos mais importantes resultados desta dissertação seja exatamente a redução dos fundamentos dos diversos algoritmos paralelos a somente dois conceitos básicos. Na medida em que estes conceitos referem-se à maneira pela qual é feita a transformação do caso serial para o paralelo, será dito que há dois *métodos de paralelização* básicos de algoritmos para busca em árvores de jogo.

Esta classificação é particularmente útil porque, com sua utilização, pode-se prever o grau e a qualidade da eficiência de um algoritmo paralelo, desde que se conheça o método de paralelização que o fundamenta. O próximo capítulo conduz a análise nesse sentido, com a indicação de alguns resultados interessantes sobre o poder e o emprego dos dois métodos.

Para se compreender melhor as diferenças entre os dois métodos, é necessária uma investigação preliminar da natureza dos problemas envolvidos na paralelização de algoritmos de busca. Para simplificar a terminologia, é conveniente examinar apenas um algoritmo seqüencial, sendo no caso utilizado o Alpha-Beta. Uma característica importante deste algoritmo é que, à medida que a árvore de jogo é percorrida, menos trabalho é feito na parte restante da árvore de jogo. Isto é conseqüência da maior ocorrência de cortes rasos e profundos, advinda da propagação de valores $\alpha{:}\beta$ cada vez mais "informados"; note-se que boa parte da eficiência de Alpha-Beta provêm exatamente de cortes de ramos mais à direita da árvore, que são processados somente após o exame dos ramos mais à esquerda.

Quando a busca é feita por vários processadores, parte dos cortes (aqueles devidos a valores determinados mais à esquerda) pode ser perdida: na divisão do trabalho entre os processadores, a seqüência de busca em pós-ordem do algoritmo Alpha-Beta é quebrada; tipicamente, duas sub-árvores de um mesmo nó são examinadas por processadores distintos, o que ocasiona o exame da sub-árvore mais à direita antes que o valor da sub-árvore esquerda possa, eventualmente, determinar o corte da direita. Em outras palavras, a paralelização da busca pode causar um exame desnecessário de nós, fruto da indisponibilidade dos valores necessários ao corte no momento do início das buscas nas árvores mais à direita.

Evitar o exame desnecessário de nós terminais é, assim, o problema central da paralelização de algoritmos para busca em árvores de jogo. Os dois métodos básicos descritos a seguir refletem exatamente duas soluções distintas para este problema, visando ambos evitar a perda da informação utilizada pelo algoritmo Alpha-Beta.

### 5.2.1 Método de Paralelização por Redução de Janelas

Define-se por *janela* de execução do algoritmo Alpha-Beta sobre uma árvore de jogo $G$ ao intervalo $[\alpha, \beta]$, determinado pela mensagem inicial de valor $\alpha : \beta$ enviada à raiz $t_0$ de $G$. De acordo com a proposição 3.3, o valor $\gamma$ resultante da execução completa de Alpha-Beta sobre

$G$ satisfaz:

$$\begin{array}{lll} \gamma \leq \alpha & \text{se} & F(t_0) \leq \alpha \\ \gamma = F(t_0) & \text{se} & \alpha < F(t_0) < \beta \\ \gamma \geq \beta & \text{se} & F(t_0) \geq \beta \end{array}$$

Ora, é claro que quanto menor for o comprimento da janela $(\beta - \alpha)$, mais rápida será a busca (no sentido NBP), pois se $\alpha$ e $\beta$ são menos espaçados, maiores são as chances de ocorrer um corte. Esta idéia é captada pela proposição a seguir, encontrada em [Bau78a]:

PROPOSIÇÃO 5.1 *Seja uma árvore de jogo $G$ de raiz $t_0$. Suponha a execução do algoritmo Alpha-Beta duas vezes, a primeira com mensagem inicial $A' : B'$, a segunda com $A : B$, onde $A' \leq A \leq B \leq B'$. Seja um nó $t$ de $G$, que recebe uma mensagem descencente de valor $\alpha : \beta$ durante a segunda execução. Então $t$ também recebe uma mensagem descendente quando da primeira execução, cujo valor $\alpha' : \beta'$ satisfaz:*

$$\begin{array}{llllll} se & d(t_0,t) & \text{é par,} & \alpha = \max\{\alpha', A\} & e & \beta = \min\{\beta', B\} \\ se & d(t_0,t) & \text{é impar,} & \alpha = \max\{\alpha', -B\} & e & \beta = \min\{\beta', -A\} \end{array}$$

*onde $d(t_0, t)$ é a diferença entre o nível de $t$ e o de $t_0$.*

DEMONSTRAÇÃO : por indução na altura $h$ de $G$, de maneira análoga à demonstração da dominância de Bound por Alpha-Beta (teorema 4.1). □

É importante observar nesta proposição que, embora a busca com janela reduzida melhore a performance de Alpha-Beta, ela não assegura que o valor retornado pela segunda execução seja igual ao da primeira. Pode, por exemplo, ocorrer que

$$A' \leq A < B < F(t_0) < B'$$

e portanto a execução com janela $A' : B'$ retorna precisamente $\gamma' = F(t_0)$, enquanto que no outro caso, se terá somente $F(t_0) \geq B$, com a garantia que o valor $\gamma$ satisfaz $\gamma \geq B$. Esta propriedade é, assim, desprovida de utilidade no caso seqüencial, embora possa ser explorada no caso paralelo conforme o método descrito a seguir.

Imagine-se, pois, um caso simplificado de paralelização de busca sobre uma árvore $G$ de raiz $t_0$, no qual coloca-se um processador executando, paralelamente, o algoritmo Alpha-Beta sobre cada sub-árvore $G_i$ de raiz $t_i$, onde $t_i$ é sucessor de $t_0$, conforme a figura 5.3.

Pode-se imaginar cada processador $p_{1i}$ executando Alpha-Beta com mensagem descendente inicial para $t_i$ de valor $-\infty : +\infty$. Suponha-se então que o processador $p_{11}$ termine seu trabalho antes dos demais, enviando o valor $F(t_1)$ para $p_1$. Se esta busca fosse seqüencial, o nó $t_2$ receberia a mensagem $-\infty : F(t_1)$; contudo, a busca na sub-árvore $G_2$ já está em andamento (através de $p_{12}$), com janela inicial $-\infty : +\infty$.

Visando apressar o trabalho de $p_{12}$, o processador $p_1$ lhe informa da janela já calculada $-\infty : F(t_1)$. A partir deste momento, o algoritmo Alpha-Beta executado em $p_{12}$ converte a busca da janela anterior $-\infty : +\infty$ para a janela $-\infty : F(t_1)$ através do artifício: antes de iniciar o processo de decisão de uma mensagem enviada para um nó $t$ de $G_2$, os valores de VALOR $(t)$ e BETA $(t)$ são atualizados por:

$$\begin{array}{llllll} \text{se} & d(t_0,t) & \text{é par,} & \text{BETA}\ (t) & = & \min\{\text{BETA}\ (t), F(t_1)\} \\ \text{se} & d(t_0,t) & \text{é ímpar,} & \text{VALOR}\ (t) & = & \max\{\text{VALOR}\ (t), -F(t_1)\} \end{array}$$

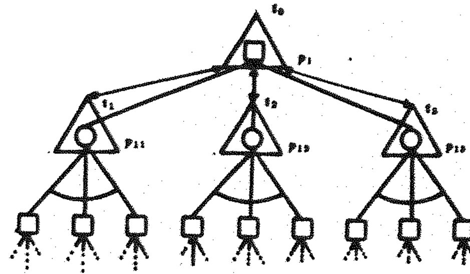


Figura 5.3: Árvore de processadores sobreposta a uma árvore de jogo.

Observe-se que, nesta situação, a redução da janela mantém a segurança de que o valor de $t_0$ seja correto, pois ela é feita usando-se a janela que seria efetivamente utilizada na busca seqüencial. Ou, sucintamente, $p_{12}$ realiza um trabalho ***desinformado*** (comparativamente ao trabalho seqüencial) até que a informação necessária de redução da janela lhe chegue "às mãos".

Dessa forma, algum trabalho inútil é feito, mas a comunicação entre os processadores garante que a busca seja realizada de forma cada vez mais semelhante a que seria feita pelo algoritmo Alpha-Beta. Além disso, se a solução está em um ramo mais à direita, é possível que se realize menos trabalho que no caso seqüencial: por exemplo, se o processador $p_{13}$ termina rapidamente, parte do trabalho em $p_{11}$ e $p_{12}$ pode ser substancialmente reduzido.

O raciocínio empregado para um nível de processadores é facilmente estendido para um número maior de níveis. Este método de paralelização por *redução de janelas* pode ser então resumido, informalmente, em:

> Sempre que um nó $t$ da árvore tem seu valor alterado, pode-se acelerar o processo de busca atualizando-se as janelas de seus descendentes (na forma da proposição 5.1).

## 5.2.2 Método de Paralelização por Priorização do Trabalho Obrigatório

O outro método empregado em algoritmos paralelos para busca em árvores de jogo envolve a noção de refutação desenvolvida no capítulo 2.

A figura 5.4 mostra o caso de busca no qual menos nós são examinados pelo algoritmo Alpha-Beta. Este caso se baseia, conforme visto no capítulo 2, na determinação de uma AND-árvore solução e de refutações para os nós de altura par do caminho principal. Uma característica importante de qualquer algoritmo *direcional* [2] é que sempre ele examina, no mínimo, todos os nós da AND-árvore solução mais à esquerda e de suas refutações.

Ora, como se pode ver pela figura 5.4, estes nós cuja exame é obrigatório não se distribuem uniformemente pelos ramos da árvore, pelo contrário, concentram-se à esquerda. Assim, se, por exemplo, fossem colocados três processadores, $p_1$, $p_2$ e $p_3$, em $t_1$, $t_2$ e $t_3$, respectivamente, $p_1$ examinaria 11 nós terminais antes de retornar. Considerando-se tempos semelhantes para os três processadores, $p_2$ e $p_3$ examinariam, ao menos, $11 + 11 = 22$ outros nós, e com sorte, a busca terminaria por aqui, totalizando 33 nós terminais. Como a busca seqüencial examina somente 17 nós, a eficiência desta paralelização seria somente 51%.

[2] Conforme definição 3.4.

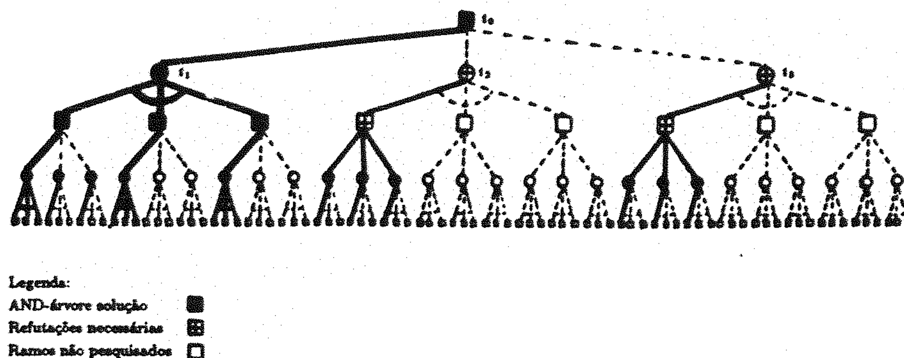



Figura 5.4: Exemplo de melhor caso de busca em uma árvore de jogo por um algoritmo direcional.

Para evitar esta degradação do caso ótimo, alguns algoritmos priorizam o exame desses *nós obrigatórios*, de forma a garantir maior potência de processamento para os nós da esquerda da árvore. Assim, evita-se realizar trabalho desinformado, através da concentração de forças na busca que também é realizada com menos informação pelo algoritmo Alpha-Beta. Pode-se dizer que esses algoritmos buscam melhorar a eficiência da paralelização por um método de *priorização do trabalho obrigatório* [3]. É fácil perceber que estes algoritmos são particularmente eficientes quando pouco trabalho <u>além</u> do obrigatório necessita ser feito, ou seja, quando a estratégia defensiva ótima está mais para o lado esquerda da árvore de jogo.

Os algoritmos descritos a seguir utilizam algum desses dois métodos (às vezes ambos). Mais adiante, será estudada mais profundamente a correlação entre o método empregado e a eficiência advinda de seu uso, considerando-se, também, as diferentes posições da árvore solução dentro da árvore de jogo.

## 5.3 Algoritmos para Processadores em Árvore

Conforme já foi dito, uma máquina paralela com processadores arranjados em uma topologia de árvore parece ser bastante adequada às características do problema de busca em árvores de jogo. De fato, diversos dos algoritmos propostos foram idealizados tendo em vista tal topologia.

Na tentativa de possibilitar uma melhor compreensão da estrutura e das diferenças entre os algoritmos, será utilizado novamente o formalismo de *descrição por mensagens*. A fim de se evitar confusão entre mensagens "reais" entre processadores e mensagens "algorítmicas" entre os nós da árvore, será utilizada a seguinte representação:

| | | |
|---|---|---|
| $\Downarrow$ | (seta dupla) | : mensagens entre processadores |
| $\downarrow$ | (seta simples) | : mensagens entre nós da árvore |

Cada algoritmo será descrito pelo processo que é executado em cada processador, denominando-se de *origem* ao processador da raiz da árvore, e de *folhas* os localizados nos nós terminais daquela. Atenção especial será dada ao modo pelo qual os processadores são distribuídos ao longo da árvore de jogo: este é o principal fator diferenciador dos algoritmos.

[3] Do original, em inglês, *mandatory-work-first*.

### 5.3.1 Algoritmo Tree-Splitting

O primeiro algoritmo paralelo para processadores em árvore foi concebido por Fishburn e Finkel em 1980 (ver [FF80] e [FF82]). Baseado no método de redução de janelas, o algoritmo **Tree-Splitting** justapõe o topo da árvore de jogo que está sendo buscada com o topo da árvore de processadores; depois, os $f$-primeiros sucessores da raiz aos $f$ escravos do processador origem, e assim por diante. A figura 5.5 mostra um exemplo de configuração inicial dos processadores.

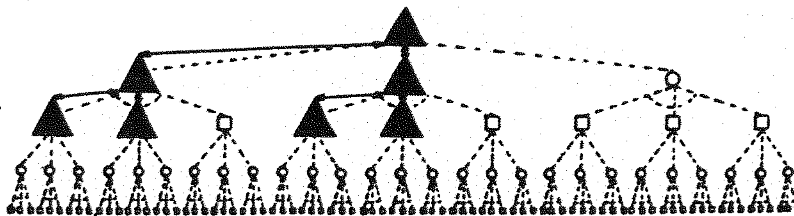
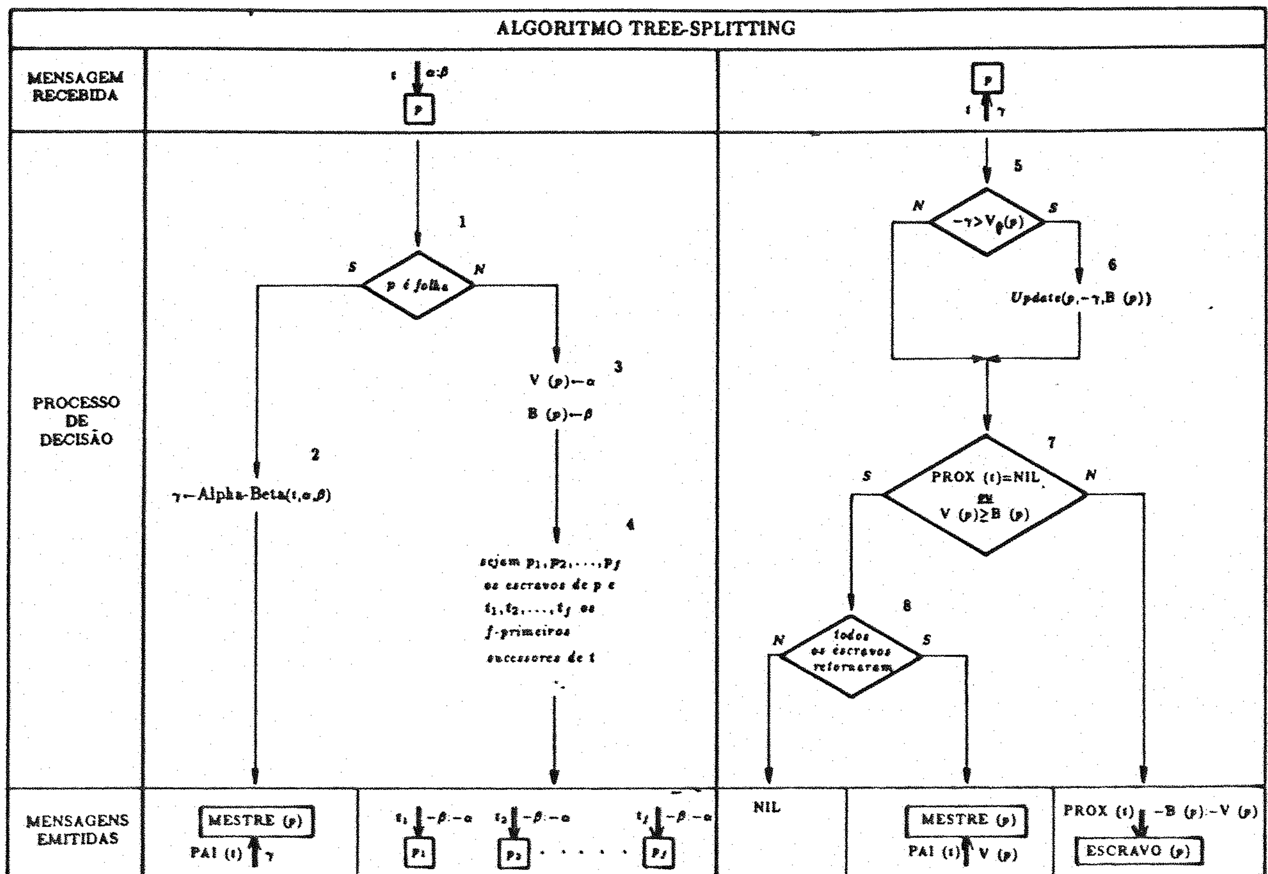


Figura 5.5: Configuração inicial do algoritmo Tree-Splitting.




MENSAGEM INICIAL :

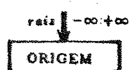
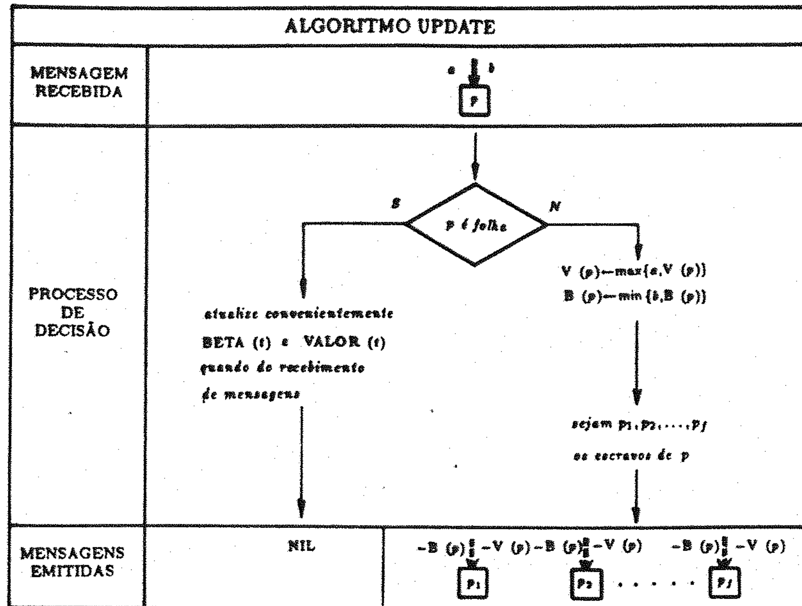



Figura 5.6: Algoritmo Tree-Splitting.



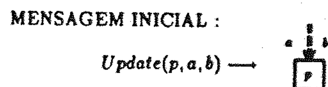




Figura 5.7: Interrupção Update de atualização.

As figuras 5.6 e 5.7 descrevem os diversos processos envolvidos no algoritmo Tree-Splitting. Diversas estruturas merecem uma melhor explicação:

MESTRE $(p)$ : gera o processador que é mestre de $p$;

ESCRAVO $(p)$ : gerencia os escravos de um processador $p$, retornando um processador livre, se houver, ou esperando sua liberação de maneira apropriada;

ORIGEM : raiz da árvore de processadores;

Alpha-Beta$(t, a, b)$ : chamada do algoritmo Alpha-Beta na sub-árvore de raiz $t$, com mensagem inicial (janela) $a$:$b$ ;

NIL (mensagem) : o processador não emite mensagem, e passa a decodificar a próxima mensagem enviada;

Update$(p, a, b)$ : chamada da interrupção de atualização descrita na figura 5.7, que produz mensagens entre os processadores capazes de gerar interrupções (marcadas com setas tracejadas) nos processos em execução; tipicamente, a interrupção pode ocorrer somente na fase anterior à da decodificação das mensagens normais, a fim de se evitarem colisões;

V $(p)$ e B $(p)$ : equivalentes aos vetores VALOR $(t)$ e BETA $(t)$, embora pertençam à memória local de cada processador.

Note-se que o algoritmo acaba quando algum processador (no caso, o ORIGEM ) recebe uma mensagem para $\bar{t}$=PAI $(t_0)$, $t_0$ raiz da árvore de jogo. A implementação prática do conceito de

redução de janelas é feita pela interrupção **Update**. Ela é chamada sempre que o processador recebe uma mensagem ascendente que melhora o valor da variável V ($p$), causando a *redução das janelas de busca de todos os descendentes de t*, onde $t$ é o nó da sub-árvore sobre a qual o processador está baseado.

Note-se que, como normalmente $f<d$ (o grau da árvore de processadores é menor que o grau da árvore de jogo), a parte mais à direita da árvore de jogo não sofre qualquer tipo de pesquisa na fase inicial. A garantia de busca nesses ramos é dada pela condição (7), que envia mensagens aos ramos não examinados, utilizando os processadores que ficam livres, a menos da ocorrência de cortes. A árvore de processadores, com o passar do tempo, move-se horizontalmente da esquerda para a direita, mantendo-se *sempre* igual o nível em que um determinado processador trabalha. Há, portanto, que se admitir que aqui existe uma aplicação "casual" também do método de priorização do trabalho obrigatório.

Os processadores interiores têm como função exclusiva o gerenciamento das mensagens de seus escravos. Isto acarreta que boa parte do tempo eles permanecem desocupados, sugerindo um caminho pelo qual otimizações podem ser feitas.

### 5.3.2 Algoritmo PV-Splitting

Contrastando com o anterior, o algoritmo **PV-Splitting**, proposto por Marsland e Campbell em [MC82], utiliza o método de priorização do trabalho obrigatório (embora seja uma versão "fraca" do mesmo). A idéia básica consiste em basear a origem da árvore de processadores nos nós ao longo do caminho mais à esquerda da árvore de jogo (*variação principal*), de forma que na posição inicial os processadores folha coincidam com nós terminais da árvore (ou estejam a uma distância pré-fixada dos mesmos). Sempre que o processador ORIGEM resolver o nó sobre o qual está baseado, ele "sobe" um nível na árvore de jogo, rearranjando seus escravos ao longo dos ramos não pesquisados, até atingir a raiz, conforme mostra a figura 5.8.

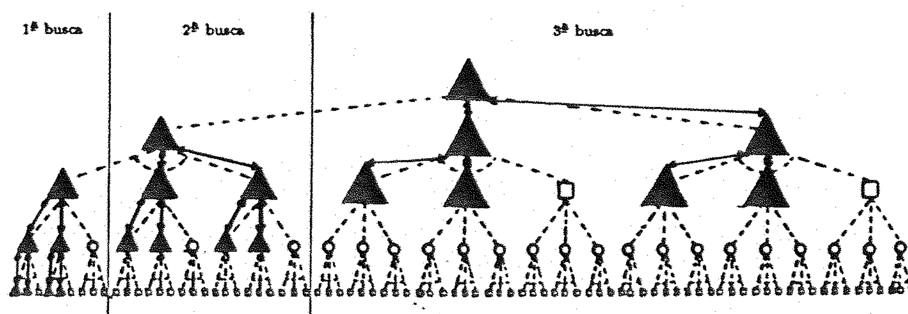



Figura 5.8: Movimento da árvore de processadores ao longo da árvore de jogo no algoritmo PV-Splitting.

Para se conseguir esse efeito, é necessário modificar-se o algoritmo Tree-Splitting, acrescentando o teste (1) para verificação se a origem já atingiu a posição inicial, e um mecanismo (testes (7) e (9), comandos (8) e (10) ), que permite à origem da árvore de processadores subir de volta à raiz da árvore, conforme mostra a figura 5.9.

A função D ($t$) retorna a altura da sub-árvore de raiz $t$, logo o teste $n \leq$D ($t$) verifica se as folhas da árvore de processadores já "atingiram o fundo". Analogamente ao anterior, este

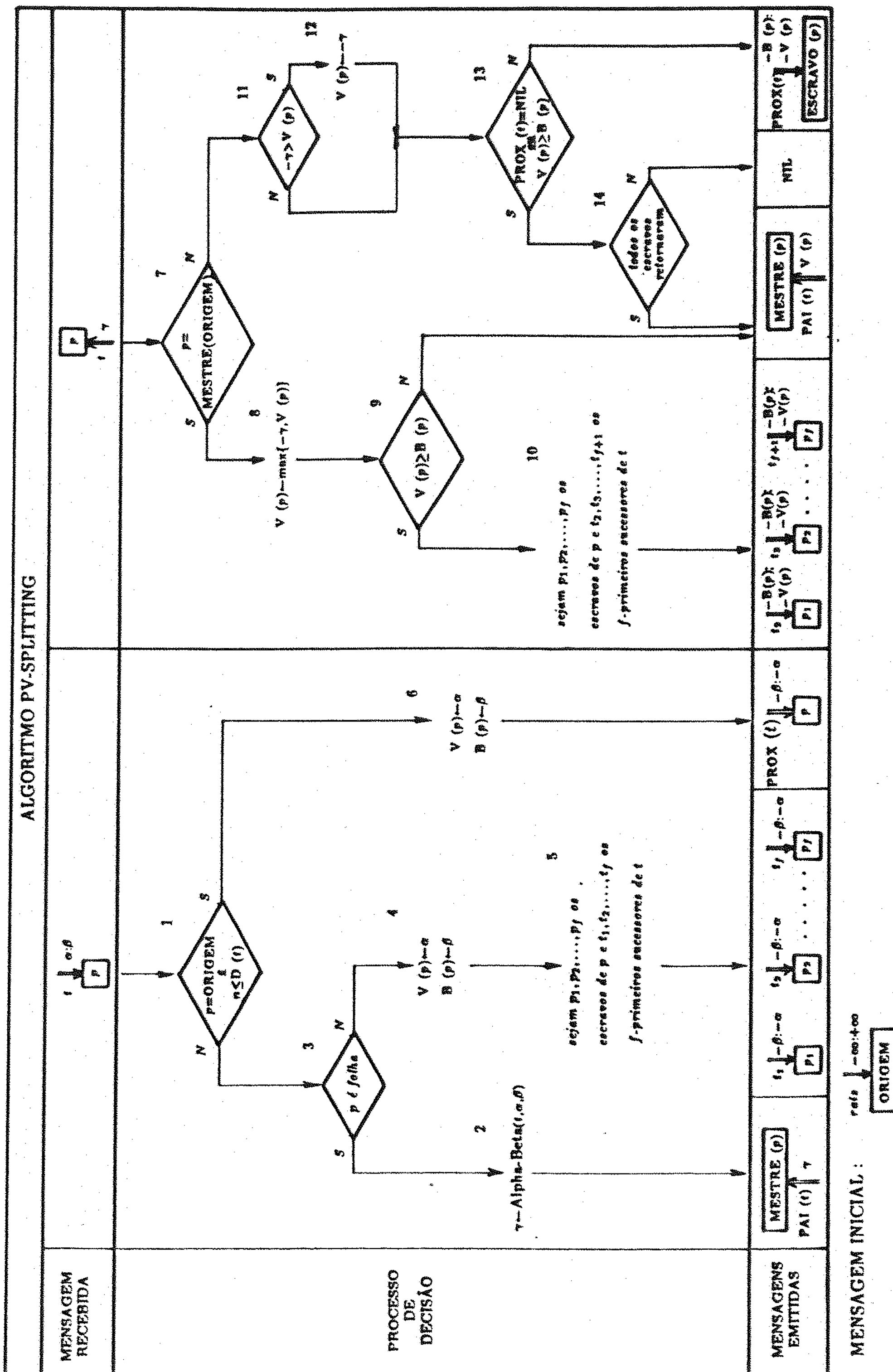



Figura 5.9: Algoritmo PV-Splitting.

algoritmo pára quando o processador ORIGEM recebe uma mensagem para $\bar{t}$=PAI $(t_0)$, $t_0$ raiz de $G$.

Note-se que este algoritmo, na forma originalmente proposta por Marsland e Campbell, não utiliza o método de redução de janelas. No entanto, tal otimização pode ser facilmente feita, substituindo-se:

$$\text{comando (12)}: \ \text{V}\ (p) \leftarrow -\gamma \quad \text{por} \quad \text{Update}(p, -\gamma, \text{V}\ (p))$$

Ainda, uma variação deste algoritmo consiste em fazer com que os processadores folha não coincidam com os terminais, mas que fiquem a uma altura pré-determinada $m$. Para tanto, modifica-se

$$\text{teste (1)}: \ n \leq \text{D}\ (t) \quad \text{por} \quad n + m \leq \text{D}\ (t)$$

### 5.3.3 Algoritmo MWF-Tree

Em 1983, Finkel e Fishburn publicaram um artigo, [FF83], no qual propõe um novo algoritmo, que é provado superior ao algoritmo Tree-Splitting, desenvolvido anteriormente pelos dois pesquisadores. O algoritmo, denominado **MWF-Tree**, é fortemente baseado no algoritmo MWF de Akl, Barnard e Doran, que será visto na próxima seção (pois é um algoritmo para processadores com memória compartilhada).

Na verdade, o MWF-Tree é uma conversão do MWF para uma topologia de processadores em árvore. O método de paralelização utilizado, como o próprio nome diz, é o de priorização do trabalho obrigatório, no qual a busca da árvore de jogo, em cada processador, caracteriza-se por duas fases:

- 1ª Fase: um processador sobre o nó $t$ coloca o primeiro de seus escravos fazendo a busca do seu sucessor mais à esquerda, (nó $t_1$, no exemplo da figura 5.10), e os demais fazendo a busca do sucessor mais à esquerda de cada outro sucessor, (no exemplo, $t_{21}$ e $t_{31}$).

- 2ª Fase: após o retorno das mensagens de todos os seus escravos (completada, pois, a busca de todos os sucessores esquerdos de seus filhos direitos), o processador tem um valor exato para o seu sucessor esquerdo, e valores temporários para os demais; estes valores, contudo, são capazes de realizar cortes (pois podem ter encontrado refutações do caminho principal); aqueles nós cujo valor preliminar é insuficiente para indicar uma refutação são reexaminados, desta vez com o processador escravo colocado diretamente sobre o nó sucessor de seu mestre (e não sobre o sucessor esquerdo deste), conforme mostra a figura 5.10.

Repare-se na figura que os nós da 1ª fase marcados com tracejado são examinados somente quando algum processador escravo fica livre, e que, usualmente, este processador não é o que busca o sucessor esquerdo da raiz, pois a busca, para este processador, involve $d$ vezes mais nós que a dos outros processadores. Outro detalhe importante do exemplo é a sub-árvore central, cujo valor é suficiente para realizar o corte, e, desse modo, não é examinada na 2ª fase.

Este algoritmo de busca em 2 fases lembra, de certa forma, o algoritmo SCOUT, embora, na prática, tenha somente o efeito do Bound: é incapaz de realizar cortes profundos. A figura 5.11 mostra a descrição por mensagens do algorimo MWF-Tree.

Diversas estruturas dessa figura necessitam melhor explicação:

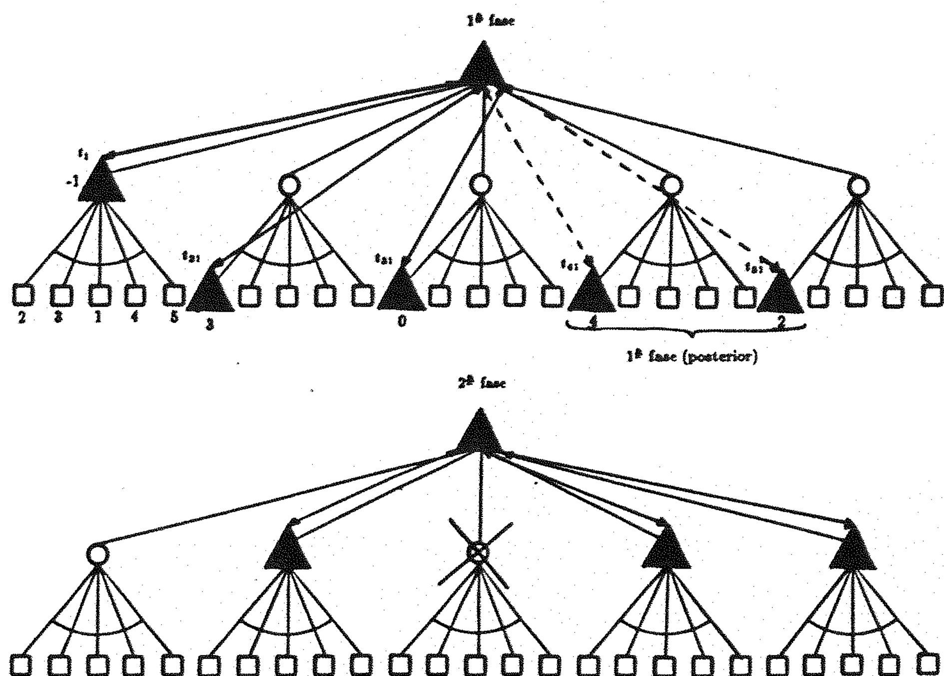



Figura 5.10: Fases 1 e 2 do algoritmo MWF-Tree.

NO $(p)$ : vetor que armazena o nó que está na base do processador $p$;

T1 $(p)$ : vetor que armazena o 1º sucessor do nó armazenado em NO $(p)$;

FASE $(p)$ : vetor que indica qual fase (1 ou 2) está o processador $p$;

SOL $(p)$ : vetor que armazena a solução temporária obtida para o nó base do processador $p$;

TEMP $(p)(t)$ : matriz que, para cada processador $p$, indica o valor temporário para os filhos $t = t_1, t_2, \ldots, t_d$ do nó armazenado em NO $(p)$.

Ainda, denota-se por $t_{i1}$ o primeiro sucessor (mais à esquerda) de $t_i$.

Algumas características do algoritmo MWF-Tree merecem maior detalhamento:

- o algoritmo realiza cortes rasos utilizando os valores dos nós esquerdos, evitando o reexame daqueles que já garantiram uma refutação;

- o algoritmo não realiza cortes profundos, pois não há mecanismos de propagação de valores alcançados a níveis superiores;

- não há utilização do método de redução de janelas;

- os sinais de $\gamma$ nos comandos (8) e (9) são diferentes porque T1 $(p)$ é filho de NO $(p)$, enquanto que os demais são netos de NO $(p)$.

Embora o algoritmo MWF-Tree não utilize nem o conceito de valor $\alpha$, nem redução de janelas, é importante se perceber que os mesmos podem ser usados, embora os autores do algoritmo não apontem esse fato.

Figura 5.11: Algoritmo MWF-Tree.

## 5.4 Algoritmos para Processadores com Memória Compartilhada

Os algoritmos para busca em árvores de jogo que utilizam processadores com memória compartilhada, são, em geral, adaptáveis a outras topologias. Contudo, o compartilhamento de memória evita problemas com a administração direta de mensagens, o que, em muitos casos, é de extrema utilidade no processo de desenvolvimento dos algoritmos.

### 5.4.1 Algoritmo Aspiration Search

O primeiro algoritmo paralelo proposto para busca em árvores de jogo foi elaborado por G. Baudet, em sua tese de doutoramento, [Bau78a], e denominado **Aspiration Search**. Nesta tese, Baudet argumenta tanto contra a decomposição *estática* da árvore de jogo (como faz o Tree-Splitting), como contra a decomposição *dinâmica* (por exemplo, MWF-Tree). Embora algoritmos que explorassem tais tipos de decomposição ainda não tivessem sido formulados na época, Baudet sugere um método radicalmente diferente de dividir tarefas entre processadores, utilizando sempre como base o algoritmo Alpha-Beta, e explorando o método de redução de janelas.

Dado um intervalo $[a, b]$ (tipicamente $[-\infty, +\infty]$), Baudet sugere que cada processador $p_i$ procure o valor da árvore de jogo considerando-a totalmente, desde a raiz, mas *trabalhando com uma janela reduzida* $[a_i, b_i] \subset [a, b]$. Conforme foi mostrado, janelas menores produzem busca mais rápida, e assim todo o processo seria acelerado.

A maneira mais simples de particionar o intervalo inicial $[a, b]$ é visto no exemplo da figura 5.12.a ; coloca-se um processador em cada intervalo e, com base nos valores retornados pelos processadores, calcula-se facilmente o valor da árvore.

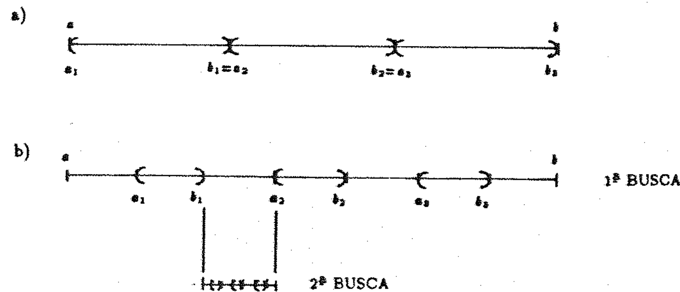



Figura 5.12: a)Partição simples; b)Partição recursiva.

A figura 5.12.b mostra uma maneira melhor de particionamento: a união dos intervalos $[a_i, b_i]$ não cobre todo o invervalo $[a, b]$. Já foi visto várias vezes neste trabalho que o algoritmo Alpha-Beta tem a propriedade de retornar o valor exato da árvore de jogo, caso este valor seja interior a sua janela de busca $[\alpha, \beta]$; senão, o valor retornado indica se o mérito da árvore é valor real é menor que $\alpha$, ou maior que $\beta$. Considerando-se tais informações, é possível ao algoritmo Aspiration Search determinar, no caso em que o valor exato não pertence aos sub-intervalos iniciais, em qual dos sub-intervalos não explorados este valor se encontra. Recursivamente este novo intervalo é submetido ao particionamento, até a determinação exata da solução.

Dada a possibilidade de diferentes tempos de busca em cada processador, o algoritmo Aspiration Search utiliza uma versão dinâmica desta idéia, mostrada na figura 5.13: utilizando duas variáveis globais (compartilhadas) GALPHA e GBETA , sempre que a busca em um certo intervalo $[a_j, b_j]$ termina (comando (3)), testa-se o retorno $v_j$. Caso $v_j \leq a_j$, então o valor da árvore é inferior à $a_j$, e a variável global GBETA é atualizada; se $v_j \geq b_j$, atualiza-se GALPHA ; e, no último caso, $a_j < v_j < b_j$, $v_j$ é o valor exato da árvore.

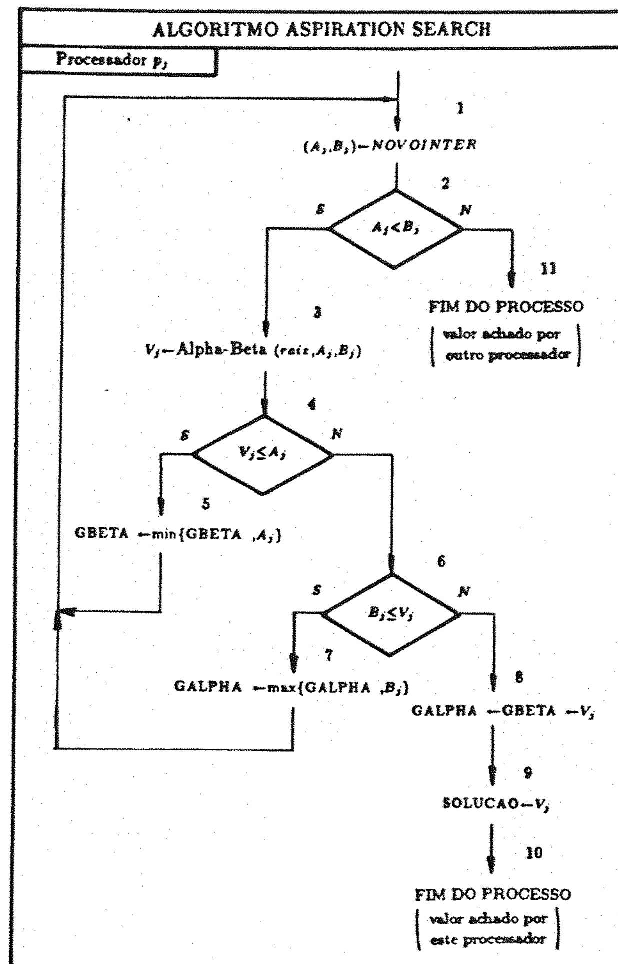



Figura 5.13: Algoritmo Aspiration Search.

A rotina NOVOINTER retorna um intervalo $[a_j, b_j]$, $a_j < b_j$, se GALPHA <GBETA , e caso contrário, $a_j = b_j =$ GALPHA $=$ GBETA ; neste último caso, a condição (2) assegura o término do trabalho no processador. É claro que todas as consultas e alterações de GALPHA e GBETA devem ser realizadas com proteção, evitando-se o acesso concomitante de diferentes processadores.

Note-se que um processador $p_j$ só é afetado por uma diminuição da janela global *após* o término da busca em sua janela $[a_j, b_j]$. Ora, como os resultados de outros processadores podem já ter indicado que o valor procurado não pertence à essa janela, o próprio Baudet

sugere uma otimização no seu algoritmo, utilizando o método de redução de janelas: sempre que GALPHA ou GBETA são alterados, todos os processadores atualizam suas janelas de busca, da maneira prescrita pela proposição 5.1.

Esta versão "informada" do algoritmo Aspiration Search é mais eficiente que a primeira, em particular quando se utiliza o algoritmo Falphabeta, de Fishburn e Finkel ([FF80]), conforme notam Marsland e Campbell em [MC82]: como foi visto, o valor retornado por Falphabeta é capaz de "alargar" a janela de busca inicial, com ganhos evidentes quando empregado, no lugar de Alpha-Beta, pelo algoritmo Aspiration Search.

Como última otimização, pode-se adaptar a sugestão de Kumar e Kanal, em [KK84] [4] , e utilizar um processador de segurança, ou seja, manter um processador fazendo a busca com janela inicial $[-\infty, +\infty]$, o que garante que a busca com múltiplos processadores não terá um tempo real pior do que aquela com um único processador.

### 5.4.2 Algoritmo MWF

O algoritmo **MWF** foi proposto por Akl, Barnard e Doran em 1980, ([ABD80]), sendo que uma descrição detalhada pode ser encontrada em [ABD82]. Este algoritmo utilizou pela primeira vez o *método de priorização do trabalho obrigatório*, que, na verdade, foi idealizado pelos pesquisadores acima citados.

Quando da explicação do algoritmo MWF-Tree nesta tese, foi feita uma introdução aos conceitos utilizados pelo MWF. Justiça seja feita, o algoritmo agora descrito, além de ser o introdutor do conceito, é também mais sofisticado que o anterior. Isto porque o algoritmo MWF *não reexamina nós terminais já examinados.* A figura 5.14 mostra claramente o processo utilizado: na impossibilidade de cortes, o algoritmo instala um "processo" não sobre o sucessor com problemas, mas sim diretamente sobre os sucessores deste que não foram pesquisados (compare-se com a figura 5.10).

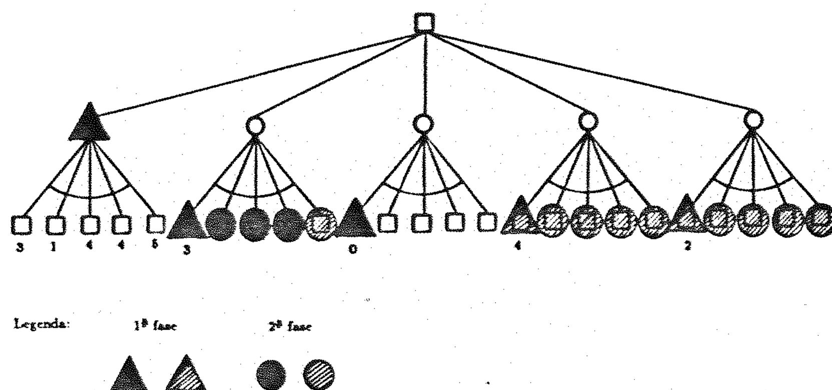



Figura 5.14: Fases 1 e 2 do algoritmo MWF, utilizando-se 3 processadores.

A figura 5.14, embora ilustrativa da técnica utilizada, não deve, contudo, ser tomada como real. O algoritmo MWF não instala processadores em nós da árvore de jogo, mas sim configura-os em um sistema de *pool*. Cada processador executa, assincronamente, "mensagens" enviadas

[4] Esta sugestão é feita, neste artigo, para uma versão do Aspiration Search que utiliza o algoritmo SSS*.

por nós da árvore a seus antecessores e sucessores, que, por sua vez, provocam novas mensagens. Estas mensagens são organizadas em uma fila de prioridades, e sempre que um processador termina a decodificação de uma mensagem, o topo dessa fila é extraído e entregue, para interpretação, a esse processador livre.

A figura 5.15 mostra o algoritmo executado em cada processador. Na figura, LADO $(t)$ é uma função que indica se o nó $t$ é um nó esquerdo $E$ (1º sucessor mais à esquerda, incluindo a raiz), ou direito $D$ (os demais). Para melhor entendimento, a função de algumas estruturas é explicada a seguir:

teste (2) : verifica se o nó que recebeu a mensagem descendente é esquerdo; se verdade, todos os sucessores serão examinados simultaneamente, senão somente o 1º sucessor;

teste (6) : no caso de mensagem ascendente proveniente de um nó esquerdo (teste (5) afirmativo), dois tipos de ações são tomadas: se o nó receptor é esquerdo, então deve-se esperar o retorno de todos os nós, caso contrário, como somente este sucessor foi examinado (de acordo com o teste (2)), o retorno é imediato;

teste (10) : se o nó emissor $f$ é nó direito, o nó receptor é esquerdo e $-\gamma >$ VALOR $(t)$, então os sucessores de $f$ precisam ser examinados, com excessão do 1º que já o foi.

A prioridade de processamento de uma mensagem é função do nó $t$, destino da mensagem, e dada recursivamente pela fórmula:

$$\text{PRIORIDADE } (t) = \text{PRIORIDADE } (\text{PAI } (t)) - (d + 1 - i) \times 10^{ff \times (h-w)}$$

e

$$\text{PRIORIDADE } (raiz) = \sum_{x=1}^{h} (d+1) \times 10^{ff \times (h-x)}$$

onde $d$ = grau máximo da árvore
$h$ = altura máxima da árvore
$i$ = $t$ é o $i$-ésimo sucessor de PAI $(t)$
$w$ = nível de $t$
$ff$ $\doteq$ valor tal que $10^{ff-1} < d < 10^{ff}$

A figura 5.16 mostra as diferentes prioridades dos nós de uma árvore uniforme de grau $d$=3 e altura $h$=2.

Note-se ainda que o algoritmo MWF não propaga valores descendentemente, logo é incapaz de realizar cortes profundos. Os próprios autores afirmam que é possível reformulá-lo de forma a trabalhar com janelas Alpha-Beta, a partir da utilização de um mecanismo de redução de janelas. Tal extensão deve ser útil, em particular, para o caso da busca dos nós direitos sucessores de um nó direito (comando (13)), pois nesta situação uma boa janela já está disponível (proveniente do valor do 1º sucessor e de seu pai).

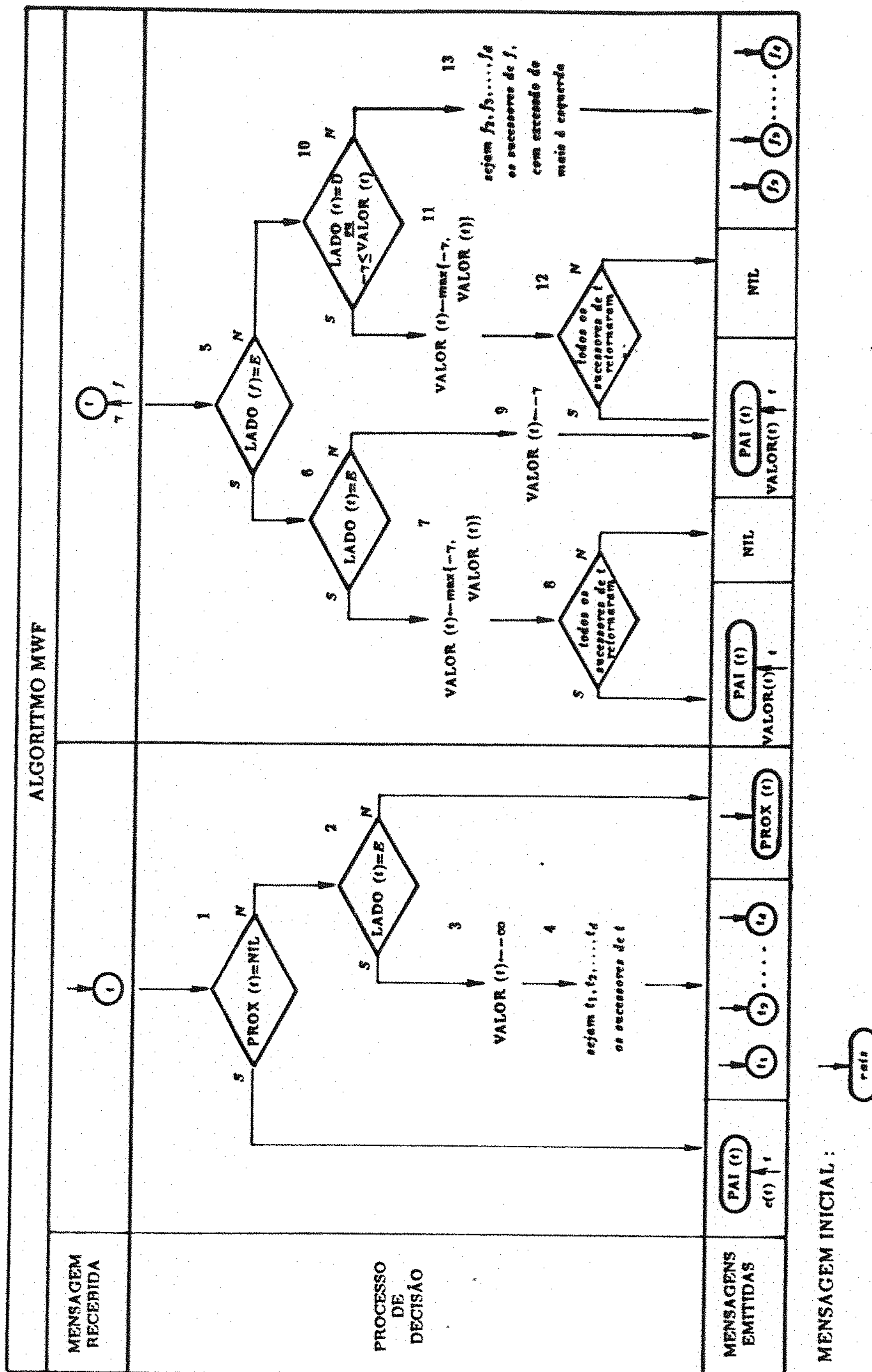



Figura 5.15: Algoritmo MWF.

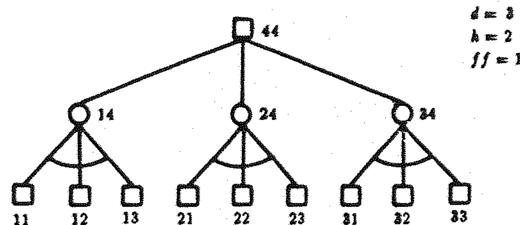



Figura 5.16: Atribuição de prioridades aos nós de uma árvore de jogo.

### 5.4.3 Algoritmo Parallel SCOUT

Em [AD83], Akl e Doran propõe uma versão paralela do algoritmo SCOUT, por eles denominada **Parallel SCOUT.** Embora a referência dada acima (e a única disponível) não apresente claramente os detalhes do algoritmo, pode-se dizer que o método empregado é, novamente, o de priorização do trabalho obrigatório.

Como no algoritmo MWF, os processadores estão arranjados na forma de um *pool*, alimentado por mensagens entre nós. A figura 5.17 mostra o algoritmo executado em cada processador.

Nesta figura, as mensagens indicadas com linha traço-e-ponto devem ser entendidas como "chamadas" para os algoritmos $\text{TEST}_{\text{AND}}$ e $\text{TEST}_{\text{OR}}$ . O ponto crucial no algoritmo consiste em não paralelizar a chamada dessas rotinas, de forma que a busca, nestes casos, é feita seqüencialmente. Assim, o algoritmo inicialmente atravessa a variação mais à esquerda da árvore até atingir o nó terminal. A condição (7) faz, então, um nó $t$ enviar mensagens testes para os seus nós direitos; estas mensagens testes têm a propriedade de garantir que sempre a mensagem $\gamma$ retornada satisfaz $\gamma \leq -\text{VALOR}\ (t)$, se necessário explorando as sub-árvores mais cuidadosamente (ver condição (4), caso negativo). Assim, como o SCOUT original, o algoritmo Parallel SCOUT possui a desvantagem de, por vezes, examinar mais de uma vez um mesmo nó terminal.

### 5.4.4 Algoritmo KNM

Quando da apresentação do método de priorização do trabalho obrigatório, foi ressaltado que os nós examinados refletem a busca mínima realizada por um algoritmo *direcional.* O algoritmo KNM [5], proposto por G. Lindstrom em [Lin83], procura conciliar o conceito de priorização do trabalho obrigatório com a possibilidade do caminho principal se encontrar mais à direita da árvore.

A figura 5.18 mostra os nós visitados prioritariamente pelo algoritmo MWF, constituindo o que será denominado *árvore mwf* (em traço contínuo). A outra sub-árvore mostrada na figura possui a mesma "forma" da árvore mwf, mas com o sentido de nó esquerdo "perturbado"; será denominada, assim, uma *árvore mwf-deslocada.* Formalmente,

DEFINIÇÃO 5.2 *Seja $G$ uma árvore de jogo de raiz $t_0$. Uma sub-árvore $T$ de $G$ é uma* árvore mwf *se, e somente se, ela é constituída somente de nós* chave [6], *que são os nós de $G$ que satisfazem:*

[5] Do original, em inglês, *Key Node Method.*
[6] Do original, em inglês, *key node.*

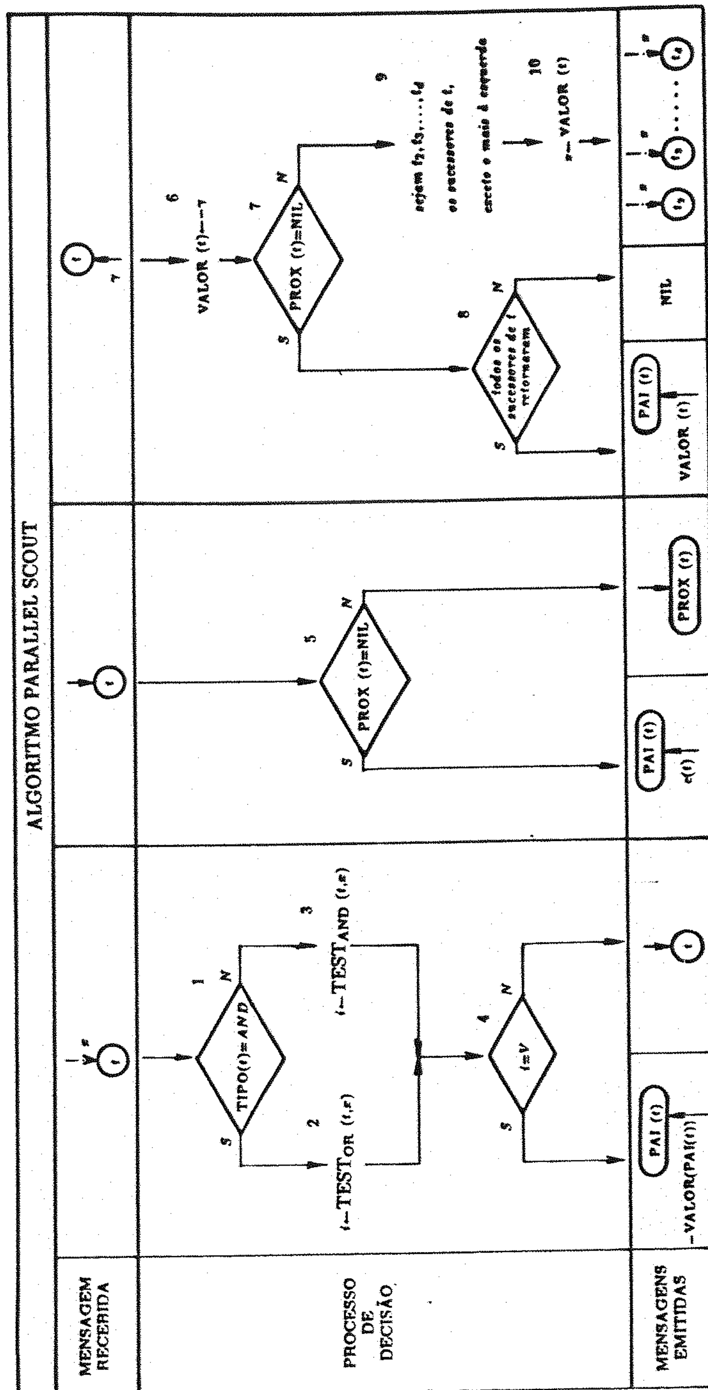



Figura 5.17: Algoritmo Parallel SCOUT.

*i . a raiz $t_0$ é um nó chave, e é considerado como sendo o primeiro sucessor de seu hipotético pai;*

*ii . se um nó chave é o 1º sucessor de seu pai, então todos os seus descendentes são também nós chave;*

*iii . se um nó chave não é o 1º sucessor de seu pai, então somente o seu 1º sucessor é um nó chave.*

*Uma* árvore mwf-deslocada *é uma sub-árvore de G com nós que satisfazem os três ítens acima, considerando-se um outro critério de ordenação para os primeiros sucessores dos nós da árvore G.*

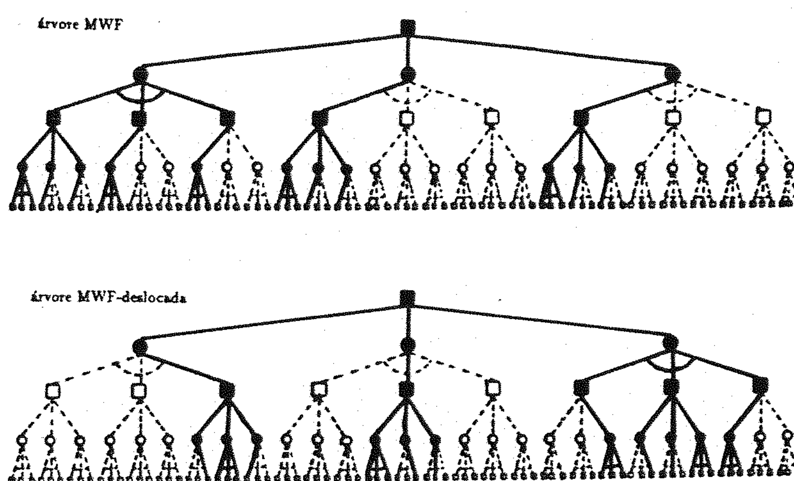



Figura 5.18: Árvores mwf e mwf-deslocada.

Pode-se dizer que uma árvore mwf (ou mwf-deslocada) está para o algoritmo Bound da mesma forma que uma AND-árvore solução está para o algoritmo Alpha-Beta: ambas representam um conjunto mínimo de nós terminais cujo respectivo algoritmo necessita examinar para determinar o mérito da árvore de jogo.

Assim como o algoritmo SSS* seqüencial procura dinamicamente a AND-árvore solução e suas refutações, o algoritmo KNM procura, também dinamicamente, ajustar a estrutura de uma árvore mwf-deslocada, a partir da árvore mwf inicial, e de forma que a árvore mwf-deslocada "tombe" para as partes consideradas mais promissoras em cada momento.

Este efeito é alcançado por uma estrutura de dados do tamanho de todos os nós da árvore, e de um sistema de mensagens entre os nós da árvore que altera o conceito de 1º sucessor dos nós (gerando, portanto, árvores mwf-deslocadas). Simplificadamente, toda vez que um nó $t$ melhora seu valor, ele se comunica com seu pai $\bar{t}$, informando-o. Caso este valor também se torne o melhor para seu pai, ocorrerá uma mudança na estrutura da árvore mwf-deslocada corrente, na qual o nó $t$ passa a ser considerado o 1º sucessor de $\bar{t}$, e fazendo com que vários de seus sucessores passem a ser considerados nós chaves; por outro lado, o antigo 1º sucessor perde sua "posição privilegiada", acarretando, assim, que vários dos seus sucessores deixem de ser nós chave.

Este processo de reestruturação é comandado por mensagens que percorrem a árvore de cima a baixo, convertendo nós chave em nós não-chave, e vice-versa. A figura 5.19 descreve o algoritmo executado por cada processador na decodificação de mensagens neste algoritmo.

As seguintes estruturas merecem explicação:

VIS $(t)$ : vetor booleano que indica se o nó terminal $t$ já foi visitado $(V)$; inicialmente, todos os nós não foram visitados $(F)$;

CHAVE $(t)$ e PRIM $(t)$ : vetores booleanos que indicam se o nó $t$ é chave e se $t$ é o 1º sucessor de seu pai (na ordenação original), respectivamente;

M $(t)(i)$ : matriz de estruturas de dois campos, *nome* e *est*; para um nó $t$, o valor M $(t)(1).nome$ é o sucessor de $t$ de maior valor estimado M $(t)(1).est$ até o momento, M $(t)(2).nome$ é o sucessor de $t$ com o 2º maior valor, e assim por diante;

POS $(f, \text{M } (t))$ : função que procura o índice $i$ de M $(t)$ tal que $f$=M $(t)(i).nome$;

REORDENE (M $(t)$) : ordena decrescentemente os sucessores de $t$ de acordo com seu valor estimado.

No algoritmo KNM, os comandos (1), (7) e (8), e os testes (5) e (6) têm a função de propagar a modificação de estrutura da árvore mwf-deslocada corrente. Quanto aos comandos (9) e (10), sua função é colocar o retorno de um sucessor na posição correspondente, salvar em $x$ e $v$ o 1º sucessor atual e seu valor, reordenando então o vetor M $(t)$. Mudanças no 1º sucessor (teste (11) negativo) causam reconfiguração na árvore mwf, e melhoras no valor estimado para o nó são reportadas ao pai pelo teste (12), caso negativo.

As mensagens geradas são organizadas na forma de uma *fila*, e são enviadas para os processadores na medida em que suas tarefas são completadas. O algoritmo termina quando não há mais mensagens a serem decodificadas, e o valor de árvore é encontrado em M $(raiz)(1)$.

Lindstrom sugere ainda que, a fim de se evitar um excesso de "inércia" na modificação da árvore mwf-deslocada corrente, pode-se substituir a fila de mensagens por uma *pilha*, desde que se acrescente a cada mensagem o instante de tempo na qual foi gerada, e garantindo-se que nenhum nó aceite uma mensagem mais velha do que a última recebida.

O algoritmo KNM não realiza cortes profundos, pois não há propagação descendente de valores. Novamente, tal estrutura pode ser incorporada ao algoritmo, com aumento considerável na complexidade de escrita do algoritmo.

## 5.5 A Paralelização do Algoritmo SSS*

No artigo original de descrição do algoritmo SSS* ([Sto79]), Stockman comenta: *"SSS* pode ser utilizado com vantagens em um sistema com autêntico processamento paralelo."* [7]. Contudo, poucas tentativas foram feitas de paralelização de SSS*, a mais importante sendo feita por Kumar e Kanal em [KK84].

A proposta desta seção é estender as idéias de Kumar e Kanal, aumentando sua aplicabilidade, bem como propor um novo algoritmo paralelo baseado em SSS*.

[7] Traduzido de [Sto79], pg 195.

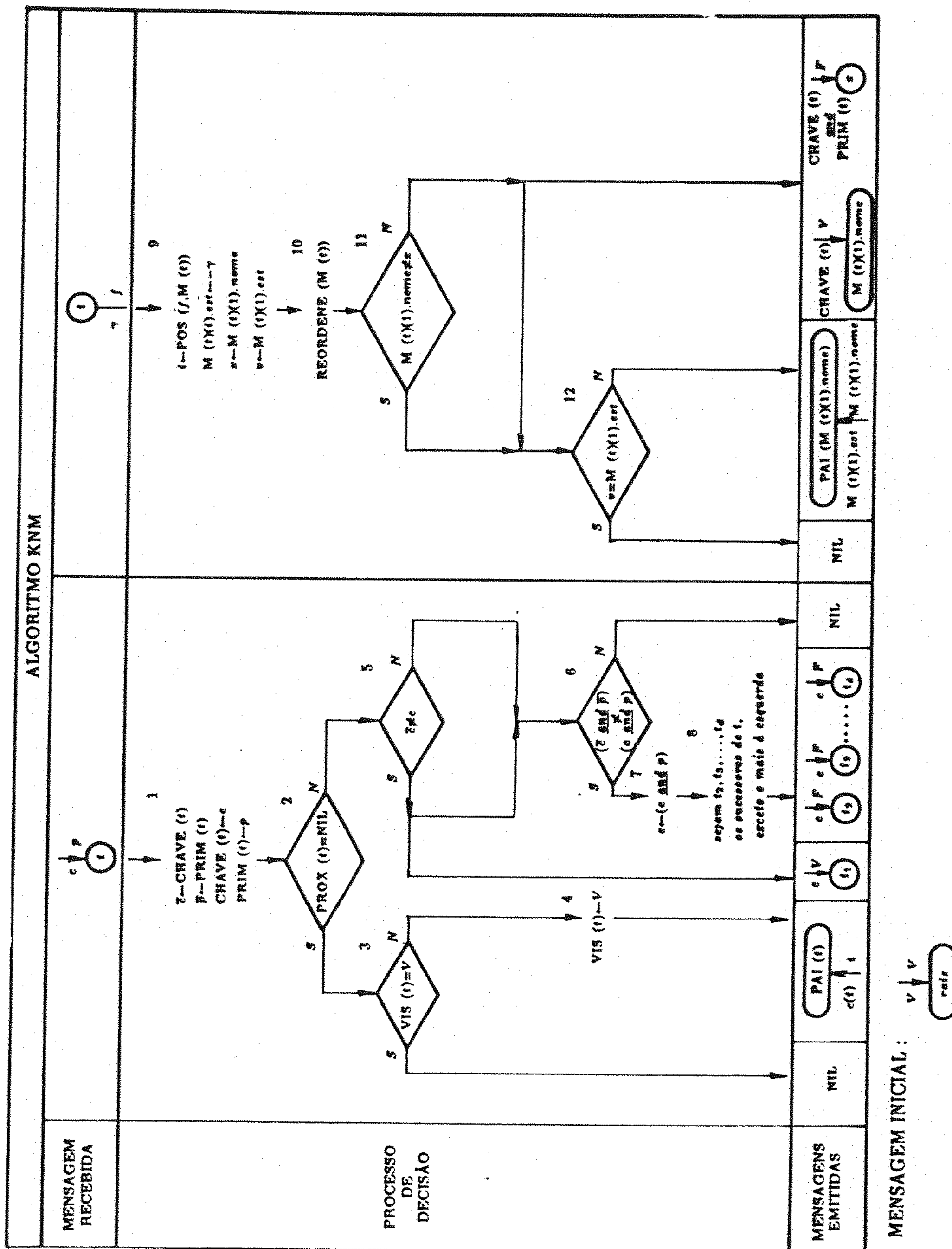



Figura 5.19: Algoritmo KNM.

### 5.5.1 Extensão do Conceito de Janela de Busca para SSS*

Quando da demonstração da dominância de Alpha-Beta por SSS* foi utilizado o conceito de mensagem *fictícia*, que consistia em uma mensagem de um certo valor $a$, que era "artificialmente" colocada na lista $L$. O efeito obtido por essa mensagem era impedir que o valor retornado fosse menor que $a$, mesmo que o valor da árvore $G$ explorada fosse realmente inferior ao de $a$, ou seja,

$$\begin{array}{ll} \gamma = m(G) & \text{se} \quad a < m(G) \\ \gamma = a & \text{se} \quad m(G) \leq a \end{array}$$

Por outro lado, a demonstração da corretude de SSS* (proposição 3.5) mostrava que, se uma mensagem de valor $b$ é enviada a um nó de uma árvore $G$, o valor retornado $\gamma$ satisfaz:

$$\begin{array}{ll} \gamma = m(G) & \text{se} \quad m(G) < b \\ \gamma = b & \text{se} \quad m(G) \geq b \end{array}$$

Estas duas propriedades sugerem que os valores $a$ e $b$, utilizados desse modo, podem produzir o mesmo tipo de efeito em SSS* do que uma janela de busca provoca em Alpha-Beta. Pode-se simplificar o processo ainda mais se os valores $a$ e $b$ forem incorporados não à lista $L$, mas ao próprio algoritmo SSS*, da seguinte forma:

> O gerenciador da lista $L$ possui duas váriaveis, $infL$ e $supL$; antes da decodificação de uma dada mensagem (ascendente ou descendente) para um nó $t$ de valor $\gamma$, faz-se:
>
> i . se $\gamma \leq infL$, então a mensagem (e todas as mensagens para sucessores de $t$) é cancelada;
>
> ii . se $\gamma > supL$, então substitui-se o valor de $\gamma$ pelo de $supL$.
>
> Caso a lista $L$ fique vazia, é gerada uma mensagem para o pai da raiz da árvore que está sendo examinada, de valor $infL$.

É evidente que a mensagem $\gamma$ enviada ao pai da raiz da árvore explorada $G$ satisfaz, nestas condições,

$$\begin{array}{ll} \gamma = infL & \text{se} \quad m(G) \leq infL \\ \gamma = m(G) & \text{se} \quad infL < m(G) < supL \\ \gamma = supL & \text{se} \quad m(G) \geq supL \end{array}$$

Em outras palavras, $infL$ e $supL$ fazem perfeitamente o papel de *janela de busca* para o algoritmo SSS* [8]. Indo além, é fácil demonstrar um equivalente à proposição 5.1 para o algoritmo SSS*, e portanto, como no caso anterior, *pode-se utilizar o método de redução de janelas* no algoritmo SSS*.

Kumar e Kanal propõe uma paralelização de SSS* nos moldes do algoritmo Aspiration Search, em um algoritmo denominado **SSS*-I**. No entanto, é proposta a utilização de janelas de busca na forma $-\infty : b_i$, onde $b_1 < b_2 < \ldots < b_k = +\infty$, e cada uma destas janelas é atribuída a um processador diferente. O artigo [KK84] falha em não apontar que é possível a utilização da

[8] Em [KK83], Kumar e Kanal fazem um notável estudo comparativo entre Alpha-Beta e SSS*, utilizando o formalismo de descrição Branch & Bound; este estudo é elucidativo dos motivos da semelhança entre os significados das janelas de Alpha-Beta e de SSS*, conforme mostradas neste trabalho.

janela com ambos os limites, inferior e superior, o que, sem dúvida, deve aumentar a eficiência da paralelização.

Pode-se ainda utilizar SSS* no lugar de Alpha-Beta nos processadores *folha* dos algoritmos Tree-Splitting, PV-Splitting e MWF-Tree (com óbvias vantagens de tempo mas com maior gasto de memória). O algoritmo Update é modificado, no caso em que o processador é terminal, passando a simplesmente atualizar $infL$ e $supL$. Um detalhe, contudo, merece atenção: se o processador for colocado sobre um nó OR, o mesmo deverá inverter o sinal do valor calculado antes de enviá-lo ao seu antecessor.

### 5.5.2 Utilização de Valores Temporários do SSS*

Examinando-se o método de redução de janelas para o algoritmo Alpha-Beta, nota-se que só ocorre redução quando um nó completa a busca em sua sub-árvore. Não é, assim, possível utilizar os resultados intermediários obtidos de Alpha-Beta, pois eles não garantem a existência de nenhuma condição *utilizável* sobre o valor da sub-árvore pesquisada.

Em [KK84], Kumar e Kanal observam que o mesmo não ocorre com o SSS*. A figura 5.20.a exemplifica uma situação em que o algoritmo SSS* consegue utilizar valores temporários para diminuição do tempo de busca: três processadores distintos, $p_1$, $p_2$ e $p_3$, examinam concorrentemente as sub-árvores $G_1$, $G_2$ e $G_3$, respectivamente. Ao cabo de algum tempo, suponha-se que os valores das mensagens de ínicio de suas listas $L_1$, $L_2$ e $L_3$ são, respectivamente, 2,3 e 4.

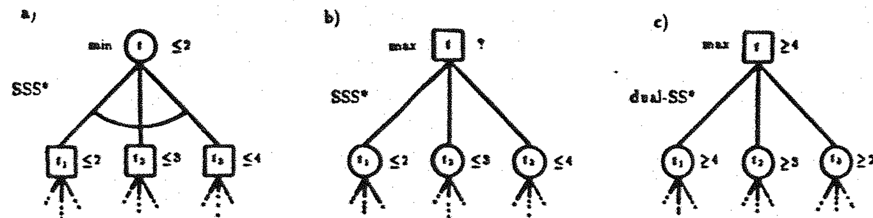



Figura 5.20: Exemplo de condições de utilização de valores temporários.

Nos capítulos anteriores foi visto que o valor de início da lista $L$ do SSS* *nunca cresce*, portanto 2, 3 e 4 são majorantes de $m(G_1)$, $m(G_2)$ e $m(G_3)$. Como o nó $t$ realiza uma operação de mínimo ($t$ é do tipo OR), qualquer valor de $m(G_2)$ que seja maior do que 2 não terá influência sobre seu valor, e idem para $m(G_3)$. Dessa forma, os valores $supL_2$ e $supL_3$ podem ser alterados para $2{=}supL_1$.

Simplificadamente, se forem colocados processadores com o algoritmo SSS* examinando os sucessores de um nó OR, pode-se aumentar a eficiência garantindo-se que, sempre que diminuir o valor do início da lista $L$ de um deles, altere-se o valor de $supL$ dos demais.

A figura 5.20.b mostra que se $t$ for um nó AND, então tal raciocínio *não* é válido: operar máximos sobre majorantes não produz efeito.

A solução é proposta por Kumar e Kanal no artigo supra-citado: alterar o algoritmo SSS* de forma que o valor do início da lista $L$ *nunca decresça*. Isto requer uma inversão completa do algoritmo, substituindo as operações de mínimo por máximo, teste de nós AND por teste de nós OR, e inversão no sistema de ordenamento de $L$. Este algoritmo é denominado dual-SS*,

e é baseado no conceito de OR-árvore solução, podendo ser indistintamente empregado em substituição a SSS* no caso seqüencial.

A figura 5.20.c exibe um exemplo de busca paralela em nós do tipo OR, utilizando o algoritmo dual-SS* : como os valores do início das listas passam a ser minorantes, a operação de máximo garante que, dualmente em relação ao SSS*, todos os processadores podem ter seus valores de $infL$ igualados, por meio de maximização.

A utilização destas propriedades dos valores temporários de SSS* pode ser feita diretamente, colocando-se os processadores exatamente sobre os sucessores da raiz, no algoritmo denominado **SSS*-II**; ou indiretamente, modificando-se apropriadamente os algoritmos Tree-Splitting ou PV-Splitting. Observe-se, entretanto, que em nenhum dos casos o algoritmo SSS* é usado *internamente* na árvore de processadores.

Embora de efeito reduzido, estas propriedades de SSS* e dual-SS* não são compartilhadas por nenhum outro algoritmo seqüencial; justifica-se, desse modo, seu emprego em casos de necessidade de máxima eficiência.

### 5.5.3 Algoritmo Pool-SSS*

Até agora não foi visto nenhum algoritmo que realmente paralelize o SSS* (com excessão do caso direto citado acima, e somente quando o número de processadores é menor que o grau da raiz da árvore sendo examinada). O algoritmo **Pool-SSS*** proposto a seguir é uma tentativa simples de adaptação dos conceitos do algoritmo MWF para o SSS*.

> **Algoritmo Pool-SSS*** : em uma máquina de processadores com memória compartilhada, cada processador executa o algoritmo SSS* (de acordo com a definição deste). A lista de mensagens $L$ é, no entanto, global, e gerenciada por um único processador. Sempre que um processador fica livre, a mensagem do início da lista lhe é enviada; ao término, as mensagens emitidas são apropriadamente colocadas em $L$. Caso haja um pedido de cancelamento de mensagens para descendentes de um nó $t$, estas são expurgadas da lista, e se algum processador estiver analisando tal tipo de mensagem, então sua execução é também abortada. O algoritmo termina quando uma mensagem para o pai fictício da raiz é enviada.

Este algoritmo possui, como o KNM, a característica de alterar dinamicamente o foco de concentração da busca. Por outro lado, possui uma estrutura de dados mais simples e econômica, e muito menos "inércia". Além disso, se o cancelamento de mensagens for eficiente, não há o risco de reexame de nós terminais e, tampouco, a possibilidade de mensagens desfazendo o trabalho de mensagens anteriores como o KNM. É admirável, ainda, que este algoritmo possua a capacidade de realizar tanto cortes profundos como rasos, e que facilmente possa utilizar uma janela de busca. Embora simples, este algoritmo jamais foi proposto pelos pesquisadores da área, embora Monien e Vornberger pareçam sugerir algo nesse estilo em [MV87].

## 5.6 Considerações sobre a Implementação dos Algoritmos Paralelos

É importante lembrar que neste trabalho não serão considerados problemas relativos à sincronização, gerenciamento de mensagens e proteção da memória; tal análise está além dos objetivos deste trabalho. Contudo, é interessante examinar neste momento a questão da quantidade de memória necessária.

Em [ABD82] (p. 197–199), Akl, Barnard e Doran investigam as necessidades de memória do algoritmo MWF para a busca de uma árvore uniforme de grau $d$ e altura $h$ com $k$ processadores. Devido à prioridade de exame dos nós mais profundos, a explosão combinatorial é controlada pelo número de processadores, e a memória necessária é $O(k.h)$.

Já o algoritmo KNM paga um alto preço pela alteração dinâmica da busca: todos os $\frac{d^h-1}{d-1}$ nós não-terminais necessitam ser mantidos na memória, cada um deles ocupando espaço proporcional a $d$ (por causa do vetor M $(t)$). Logo, ocupa-se $O(d^h)$ posições de memória.

O algoritmo Pool-SSS* utiliza, com certeza, $d^{\frac{h}{2}}$ espaços correspondentes ao tamanho de uma mensagem. Contudo, a paralelização pode produzir um aumento colateral nesta área, embora o autor acredite que memória de tamanho $O(k.d^{\frac{h}{2}})$ seja plenamente satisfatória para o algoritmo.

# Capítulo 6

# Análise dos Algoritmos Paralelos

A definição de uma metodologia para análise de algoritmos paralelos tem causado muita controvérsia entre os pesquisadores atuantes na área. Além das dificuldades normais de um processo de análise, a multiplicidade de processadores implica em novos fatores impactantes sobre a eficiência, a saber, questões de sincronização, envio e recebimento de mensagens, possíveis *deadlocks* e a própria influência da arquitetura concreta da máquina.

Desse modo, não é de se estranhar a exigüidade de resultados analíticos sobre algoritmos paralelos para busca em árvores de jogo. Nem tampouco que os resultados disponíveis quase sempre *idealizem* o problema, por exemplo, desconsiderando a comunicação entre os processadores.

Resultados experimentais são mais freqüentes na literatura, embora não seja recomendada a confrontação entre valores obtidos por diferentes pesquisadores: as condições de simulação, os modelos de árvores de jogo, as arquiteturas e os simuladores apresentam contrastes decisivos. Há, contudo, maior quantidade de material disponível, possibilitando ao menos a apreensão, em linhas gerais, das principais propriedades de cada algoritmo.

Este capítulo propõe-se ainda a estender um pouco mais os conhecimentos sobre o assunto, através de um exame mais cuidadoso do *impacto* sobre a eficiência dos dois métodos básicos de paralelização de algoritmos para busca em árvores de jogo apresentados no capítulo anterior. Com esse objetivo alguns algoritmos foram implementados em um simulador de paralelismo, e executados sobre árvores geradas "artificialmente". Os resultados são apresentados neste mesmo capítulo, enquanto que a descrição do sistema de simulação é feita no apêndice A.

## 6.1 Métodos de Comparação de Algoritmos Paralelos

De acordo com Michael Quinn, em [Qui87], existem duas medidas importantes da qualidade de algoritmos paralelos: *aceleração* e *eficiência* [1].

Suponha-se um certo problema computacional, e uma máquina paralela com $k$ processadores. Sejam ainda $s$ o melhor algoritmo seqüencial e um algoritmo paralelo $p$. Se $T_s$ é o tempo gasto pela máquina paralela para executar $s$ com um único processador, e $T_p(k)$ o tempo do

[1] Do original, em inglês, *speedup* e *efficiency*.

algoritmo $p$ utilizando $k$ processadores, então a *aceleração* $S_p(k)$ do algoritmo $p$ é

$$S_p(k) = \frac{T_s}{T_p(k)}$$

e a *eficiência* $E_p(k)$ de $p$ é

$$E_p(k) = \frac{S_p(k)}{k} = \frac{T_s}{T_p(k).k}$$

Conseqüentemente, a definição de aceleração produz uma indicação da melhora esperada com a utilização de $k$ processadores, enquanto que o valor da eficiência aponta para a quantidade de trabalho extra realizado pela paralelização.

Se o algoritmo seqüencial $s$ for realmente o mais rápido para *todas* as instâncias possíveis do problema, então decorre imediatamente que, para todo algoritmo paralelo $p$,

$$S_p(k) \leq k \quad \text{e} \quad E_p(k) \leq 1$$

Embora as duas definições dadas acima possam ser aplicadas a qualquer algoritmo, é necessário adaptá-las para a análise de algoritmos de busca em árvores de jogo. Conforme foi visto no capítulo 4, o método de comparação normalmente empregado na análise de tais algoritmos é baseado no número de nós terminais examinados ($NBP$), e não no tempo de processamento.

Considere-se, então, $NBP_p^i$ o número de nós terminais examinados pelo processador $i$, $1 \leq i \leq k$, executando um algoritmo paralelo $p$ com $k$ processadores; define-se o *número de nós terminais examinados por um algoritmo paralelo* $p$, $NBP_p(k)$, como

$$NBP_p(k) = \max_{1 \leq i \leq k} \{NBP_p^i\}$$

e a aceleração por

$$S_p(k) = \frac{NBP_s}{NBP_p(k)}$$

onde $NBP_s$ é o número de nós terminais examinados pelo melhor algoritmo seqüencial. A eficiência $E_p(k)$ é facilmente obtida por analogia,

$$E_p(k) = \frac{S_p(k)}{k} = \frac{NBP_s}{NBP_p(k).k}$$

É importante observar que $NBP_p(k)$ é definido por um máximo e não por uma média: quando se extrai o máximo do conjunto $\{NBP_p^i\}_{1 \leq i \leq k}$, o objetivo é precisamente verificar se há sobrecarga de trabalho em algum processador, o que caracteriza algoritmos paralelos de baixa eficiência.

Contudo, para comparações e medidas simplificadas, pode-se utilizar um cálculo de média, obtendo-se assim:

$$\overline{NBP}_p(k) = \frac{1}{k} \sum_{i=1}^{k} NBP_p^i$$

$$\overline{S}_p(k) = \frac{NBP_s}{\overline{NBP}_p(k)}$$

$$\overline{E}_p(k) = \frac{\overline{S}_p(k)}{k} = \frac{NBP_s}{\sum_{i=1}^{k} NBP_p^i}$$

Vale também tecer algumas considerações sobre o algoritmo serial empregado no cálculo da aceleração. Segundo a definição, deveria ser utilizado o mais eficiente algoritmo, que, no caso de busca em árvores de jogo, é o SSS*. No entanto, não parece correto calcular a aceleração de um algoritmo paralelo que utiliza busca alpha-beta tomando-se por base o número de nós terminais examinados por SSS*, pois os algoritmos possuem necessidades de memória bastante distintas.

Por outro lado, já foi visto que a maioria dos algoritmos paralelos pode ser adaptado para a utilização de busca SSS*. Assim, a fim de se evitar o cálculo de aceleração baseado em comparações injustas ou inadequadas, o *algoritmo serial tomado como referência será sempre compatível com o algoritmo paralelo.* Por exemplo, a aceleração de Tree-Splitting é calculada com base no algoritmo Alpha-Beta, a aceleração de Pool-SSS* com base em SSS*, etc.

Estas considerações devem ser aplicadas tanto aos resultados experimentais como aos analíticos.

## 6.2 Resultados Analíticos

Conforme foi visto no capítulo 4, os algoritmos seqüenciais para busca em árvores de jogo foram objeto de extensiva análise, que, de certa forma, foi bem sucedida em seus intentos. O mesmo não se aplica aos algoritmos paralelos: em toda a literatura examinada, encontrou-se somente a análise de três algoritmos, e sempre com o auxílio de condições de contorno extremamente fortes.

A ausência de análise dos algoritmos deve ser creditada, basicamente, a dois fatores:

- à falta de uma metodologia e de um modelo matemático adequados ao estudo e análise de algoritmos paralelos;
- à própria complexidade do problema, na medida em que árvores de jogo, conforme a valoração de seus nós terminais, possuem propriedades bastante distintas; por exemplo, se a estratégia defensiva ótima estiver mais à esquerda, haverá ganho de eficiência em alguns algoritmos, e perda em outros, exigindo-se, assim, a análise não só do caso médio como também dos casos extremos.

De qualquer forma, são apresentados a seguir os principais resultados em análise de algoritmos paralelos para busca em árvores de jogo.

### 6.2.1 Análise de Aspiration Search

Em sua tese de doutoramento, [Bau78a], G. Baudet apresenta, além dos importantes resultados sobre o Alpha-Beta seqüencial mencionados no capítulo 4, um detalhado estudo sobre o algoritmo Aspiration Search. A análise é feita utilizando-se árvores de modelo aleatório, e o algoritmo descrito na figura 5.13. Para realizar a análise, Baudet encontra duas fórmulas para o custo de uma busca em uma janela $[a,b]$ de um intervalo de interesse normalizado $[0,1]$, descritas nos teoremas 7.2 e 7.3 ([Bau78a], p. 115-116).

Através de simplificações e normalizações, o custo $C$ de uma busca parcial em uma janela $[a, a+h] \subseteq [0,1]$ é determinado como sendo

$$C(a, a+h) = \lambda + h$$

onde $\lambda$ é um valor que depende do grau $d$ e da altura $h$ das árvores de jogo consideradas. Para valores típicos de, por exemplo, *xadrez* ($3 \leq d \leq 32$ e $2 \leq h \leq 8$), contudo, o valor $\lambda$ apresenta pouca variação, situando-se normalmente entre $0,2 \leq \lambda \leq 0,4$.

A seguir, Baudet apresenta, no teorema 7.5 de sua tese (p. 121), o custo $C_k(\lambda)$ de uma busca com $k$ processadores sobre uma árvore de jogo com característica $\lambda$. O cálculo é ainda complexo (razão pela qual é omitido aqui), mas os valores de $C_k(\lambda)$, para $\lambda$=0,2 e $\lambda$=0,4 são apresentados na tabela 6.1. Dado o valor do custo $C_k(\lambda)$, torna-se fácil calcular a aceleração do algoritmo Aspiration Search, $S_{asp}(k)$, dado que o custo normalizado de uma busca Alpha-Beta é $\lambda + 1$.

| | $C_k(\lambda)$ | | $S_{asp}(k)$ | |
|---|---|---|---|---|
| | $\lambda = 0,2$ | $\lambda = 0,4$ | $\lambda = 0,2$ | $\lambda = 0,4$ |
| $k = 2$ | 0,56 | 0,88 | 2,12 | 1,58 |
| $k = 3$ | 0,51 | 0,73 | 2,63 | 1,92 |
| $k = 4$ | 0,44 | 0,65 | 2,74 | 2,15 |
| $k = 5$ | 0,40 | 0,60 | 3,00 | 2,33 |
| $k > 5$ | $0,2 + \frac{1}{k}$ | $0,4 + \frac{1}{k}$ | $\frac{1,2}{0,2+1/k} < 6$ | $\frac{1,4}{0,4+1/k} < 3,5$ |

Tabela 6.1: Custo e aceleração teóricos para o algoritmo Aspiration Search.

A tabela 6.1 mostra duas conseqüências importantes da análise de Baudet. A primeira é vista logo na primeira linha, onde com 2 processadores consegue-se, se $\lambda$=0,2, uma aceleração de 2,12 ! Isto demonstra que há instâncias para as quais o algoritmo Alpha-Beta não é ótimo. Estas instâncias, conforme afirma Baudet, são mais eficientemente buscadas por uma versão seqüencial do algoritmo Aspiration Search do pelo Alpha-Beta, onde se observa *"... uma melhora de 15% a 25% (...) sobre o algoritmo original, e isto constitui-se em um ganho substancial."* [2]

A última linha da tabela apresenta outra importante propriedade do algoritmo Aspiration Search: a aceleração é, não importa o número $k$ de processadores, sempre inferior a 6. Em [KK84], Kumar e Kanal comentam este resultado:

> *"... mesmo que um processo for iniciado com os limites alpha e beta corretos, a quantidade de pesquisa necessária para estabelecer que esses limites estão corretos ainda é, em média, uma fração considerável da pesquisa exigida por um processo iniciado com valores alpha e beta desinformados* $[-\infty, +\infty]$*."* [3]

Em vista disso, o uso do algoritmo Aspiration Search é recomendado somente para casos nos quais poucos processadores são disponíveis (tipicamente, no máximo 6).

[2] Traduzido de [Bau78a], p. 124.
[3] Traduzido de [KK84], p. 769.

### 6.2.2 Análise de Tree-Splitting

Em [FF80] e [FF82], Fishburn e Finkel analisam a aceleração obtível por uma versão fraca do algoritmo Tree-Splitting, denominada **Palphabeta**, na qual ***não há redução de janelas*** (pois inexiste a interrupção Update). Dessa forma, muito da análise desenvolvida pelos autores é, na verdade, uma paralelização dos resultados de Knuth e Moore em [KM75], na qual se procura avaliar a quantidade de trabalho *extra* realizado pelo algoritmo Palphabeta.

Inicialmente é examinado o caso em que a ordenação da árvore é a pior possível. Nestas condições, se uma árvore de processadores de grau $f$ e altura $n$ examina uma árvore de jogo, nenhum exame de nó terminal é desnecessariamente feito, e obtém-se, para árvores de jogo profundas, uma aceleração $S_{p\alpha\beta}$ de:

$$S_{p\alpha\beta}(f, n) = f^n$$

Neste ponto, os autores assumem que

$$f^n \sim \frac{f^{n+1} - 1}{f - 1} = k$$

e, assim

$$S_{p\alpha\beta}(k) = k$$

No caso oposto, ou seja, quando a ordem é a melhor possível, a divisão de trabalho entre os processadores, sem comunicação entre si, ocasiona um aumento no número de nós terminais examinados (utilizando o mesmo raciocínio que o apresentado quando da justificação do método de priorização do trabalho obrigatório no capítulo anterior). A degradação de performance é considerável, e os autores demonstram que, para árvores de jogo de altura suficientemente grande, a aceleração é, aproximadamente,

$$S_{p\alpha\beta}(k) = k^{\frac{1}{2}}$$

Em [FF80], são estudadas também árvores de jogo com modelo aleatório, nas quais é considerada uma versão simplificada do Palphabeta, que não realiza cortes profundos. Com estas suposições, a aceleração é

$$S_{p\alpha\beta}(f, n) = \Theta\left(\left(\frac{f \log d}{\log d + \log f}\right)^n\right)$$

onde $d$ é o grau da árvore uniforme de jogo examinada.

Estes trabalhos de análise, embora interessantes, não medem uma característica fundamental do algoritmo Tree-Splitting, que é a utilização de mensagens entre processadores, capazes de reduzir as janelas de trabalho. Dessa forma, tais resultados devem ser considerados como uma demonstração de que muito trabalho extra é realizado se o algoritmo paralelo não é adequado à instância do problema, e não como uma análise real do algoritmo Tree-Splitting.

### 6.2.3 Análise de MWF-Tree

Em [FF83], os mesmos Fishburn e Finkel fazem a análise do algoritmo MWF-Tree, conforme é descrito neste trabalho pela figura 5.11. Novamente a análise é realizada para os casos de melhor e pior ordenação possível, para os quais são determinados limites inferior e superior de aceleração:

Melhor ordenação:

$$k^{1-\log_f(1+d^{-\frac{1}{2}}-d^{-1})} \leq S_{mtree}(k) \leq k^{1-\log_f(1+d^{-\frac{1}{2}}+(f-2)d^{-1})}$$

Pior ordenação:

$$k^{1-\log_f(1+d^{-1}-d^{-2})} \leq S_{mtree}(k) \leq k^{1-\log_f(1+fd^{-1}-d^{-2})}$$

onde $k$ = número total de processadores
$f$ = grau da árvore de processadores
$d$ = grau da árvore de jogo

Considerando-se um caso típico onde $f$=2 e $d$=38 (média de movimentos por posição em um jogo de xadrez), obtém-se que o algoritmo MWF-Tree possui uma aceleração dada por:

Melhor ordenação: $k^{0,78} \leq S_{mtree}(k) \leq k^{0,82}$
Pior ordenação: $k^{0,93} \leq S_{mtree}(k) \leq k^{0,96}$

e comparando-se com o algoritmo Palphabeta, no qual

Melhor ordenação: $S_{p\alpha\beta}(k) = k^{0,5}$
Pior ordenação: $S_{p\alpha\beta}(k) = k$

pode-se concluir, conforme o fazem os autores, que MWF-Tree *"... é uma melhora significativa sobre Palphabeta."* [4], pois embora haja alguma degradação de performance no pior caso, o ganho de aceleração no melhor caso é bastante significativo.

## 6.3 Resultados Experimentais

Há diversos artigos nos quais são apresentados resultados experimentais de algoritmos paralelos para busca em árvores de jogo. No entanto, esses dados são de difícil utilização, pois refletem diferentes condições de trabalho, e devem ser comparados entre si com extremo cuidado. Além disso, quase sempre os algoritmos são simulados com os modelos de distribuição de árvores para os quais são mais adaptados, o que, de certa forma, "falsifica" os resultados. Mesmo assim, estes dados são úteis para algumas análises, justificando sua apresentação a seguir.

[4] Traduzido de [FF83], p. 92.

### 6.3.1 Experimentos com Tree-Splitting

Finkel e Fishburn apresentam, em [FF82] (p. 96 e 97), os resultados da execução do algoritmo Tree-Splitting, considerando-se 10 árvores de jogo reais, provenientes de 10 posições comuns em um jogo de damas, e fazendo-se busca até a profundidade $h=8$. Um ponto interessante deste trabalho é a utilização de uma máquina paralela concreta, a *Arachne*, composta de 5 processadores LSI-11 ligados em forma de árvore. Além disso, é feita também a simulação em um ambiente UNIX, de forma a experimentar com configurações de maior porte da árvore de processadores. Os resultados obtidos são vistos na tabela 6.2, sendo que a aceleração é medida diretamente pelo tempo de execução.

| $f$ | $n$ | total de procs. | Aceleração (tempo) | | Eficiência | |
|---|---|---|---|---|---|---|
| | | | Arachne | Simul. | Arachne | Simul. |
| 1 | 2 | 3 | 1,81 | 1,57 | 0,60 | 0,58 |
| 1 | 3 | 4 | 2,34 | 2,04 | 0,59 | 0,51 |
| 2 | 2 | 7 | — | 2,37 | — | 0,34 |
| 2 | 3 | 15 | — | 3,55 | — | 0,24 |
| 3 | 2 | 13 | — | 3,12 | — | 0,24 |
| 3 | 3 | 40 | — | 5,31 | — | 0,13 |

Tabela 6.2: Resultados experimentais para o algoritmo Tree-Splitting (fonte: [FF82], p. 96 e 97).

As colunas de eficiência merecem alguma atenção: vê-se que este algoritmo tem sua eficiência degradada com o aumento do número de processadores, passando de 0,6 com 2 processadores a 0,13 com 40 processadores. Isto confirma a hipótese de que o algoritmo Tree-Splitting, com o crescimento da árvore de processadores, tende a realizar maior quantidade de trabalho extra, devido ao fato de não priorizar o trabalho obrigatório.

### 6.3.2 Experimentos com PV-Splitting

O algoritmo PV-Splitting também foi objeto de estudos empíricos. Em [MC82] (p. 550), Marsland e Campbell apresentam uma tabela comparativa entre os desempenhos dos algoritmos Tree-Splitting e PV-Splitting, reproduzida aqui na tabela 6.3.

Os resultados dessa tabela foram obtidos utilizando-se árvores uniformes de grau 24 e altura 4, geradas artificialmente em dois modelos, o de melhor ordenação possível e o de distribuição fortemente ordenada, e considerando-se como referência comparativa o número de nós visitados *máximo* por processador.

Observe-se que a comparação é amplamente favorável ao algoritmo PV-Splitting. Contudo, há que se levar em conta que a comparação se baseia em árvores nas quais a estratégia defensiva ótima se situa mais à esquerda. Em outras palavras, confirma-se o esperado, ou seja, que o algoritmo PV-Splitting deve ser preferencialmente usado em relação ao Tree-Splitting sempre que for possível fazer pré-ordenação de movimentos.

| $f$ | $n$ | total de procs. | número de nós visitados máximo por proc. | | | |
|---|---|---|---|---|---|---|
| | | | melhor ordenação | | fortemente ordenada | |
| | | | Tree-Split. | PV-Split. | Tree-Split. | PV-Split. |
| 1 | 2 | 3 | 1222 | 961 | 2700 | 2264 |
| 1 | 4 | 5 | 922 | 505 | 2030 | 1425 |
| 1 | 8 | 9 | 772 | 277 | 1859 | 1084 |
| 2 | 2 | 15 | 910 | 648 | 1724 | 1587 |
| 3 | 2 | 13 | 778 | — | 1172 | — |

Tabela 6.3: Resultados comparativos entre os algoritmos Tree-Splitting e PV-Splitting (fonte: [MC82], p. 550).

Esta hipótese poderia ser melhor verificada se houvessem também resultados de simulações com o modelo aleatório, o que, lamentavelmente, não foi feito pelos autores.

### 6.3.3 Experimentos com KNM

Em [Lin83], Lindstrom apresenta resultados de simulações obtidas com seu algoritmo KNM. Um resumo dos seus resultados é mostrado na tabela 6.4, obtidos com o uso de um simulador de paralelismo executando árvores geradas artificialmente, segundo o modelo de valor dependente da ramificação do próprio autor (descrito no capítulo 4).

As duas últimas colunas não são originais do trabalho de Lindstrom, e foram acrescentadas visando a um acerto metodológico: os valores de aceleração, da tabela 6.4 (indicados na coluna acel*), foram calculados dividindo-se o tempo de processamento do *algoritmo KNM* em um único processador pelo tempo com múltiplos processadores. Este método contraria a definição de aceleração dada no início deste capítulo, que exige no numerador o tempo do melhor algoritmo serial disponível (no caso, Alpha-Beta).

Desse modo, utilizando-se os valores de $NBP_{\alpha-\beta}$ e de número de terminais examinados por KNM, fornecidos no trabalho, pode-se aproximar a aceleração e a eficiência pela média:

$$\overline{S_{knm}} = \frac{NBP_{\alpha-\beta}}{NBP_{knm}} \times k \quad e \quad \overline{E_{knm}} = \frac{NBP_{\alpha-\beta}}{NBP_{knm}}$$

Este cálculo, que se aproxima dos demais feitos neste capítulo, reduz a aceleração obtida. Contudo, o aspecto mais importante revelado pelos resultados de Lindstrom não é alterado: a eficiência é praticamente constante em relação ao número de processadores. Esta propriedade é singular nos dados examinados até agora, e sinaliza a hipótese de que KNM é um algoritmo para ser utilizado com muitos processadores (em uma arquitetura com memória compartilhada).

É claro que essa constância de eficiência tem um limite, provavelmente relacionado ao número médio de mensagens aguardando processamento, pois se houverem mais processadores do que mensagens há clara perda de aceleração. No entanto é importante lembrar que há sempre uma árvore mwf-deslocada ativa (com aproximadamente $d^{\frac{h}{2}}$ nós trocando mensagens entre si), e que, portanto, o número médio de mensagens aguardando processamento é de ordem

| árvore | total de proc.($k$) | $NBP_{knm}$ | acel* | $\overline{S_{knm}}$ | $\overline{E_{knm}}$ |
|---|---|---|---|---|---|
| $d=3$ $h=5$ $NBP_{\alpha-\beta}=86$ | 2 | 137,6 | 1,88 | 1,25 | 0,63 |
| | 5 | 137,4 | 4,36 | 3,13 | 0,63 |
| | 10 | 136,2 | 8,17 | 6,31 | 0,63 |
| | 20 | 134,6 | 14,66 | 12,78 | 0,64 |
| $d=3$ $h=6$ $NBP_{\alpha-\beta}=196$ | 2 | 369,8 | 1,90 | 1,06 | 0,53 |
| | 5 | 368,0 | 4,47 | 2,66 | 0,53 |
| | 10 | 366,8 | 8,51 | 5,34 | 0,53 |
| | 20 | 365,6 | 16,04 | 10,72 | 0,54 |
| $d=4$ $h=4$ $NBP_{\alpha-\beta}=89$ | 2 | 142,9 | 1,83 | 1,25 | 0,62 |
| | 5 | 143,2 | 4,19 | 1,86 | 0,62 |
| | 10 | 141,7 | 7,76 | 6,28 | 0,63 |
| | 20 | 136,6 | 13,29 | 13,03 | 0,65 |
| $d=4$ $h=5$ $NBP_{\alpha-\beta}=247$ | 2 | 470,8 | 1,85 | 1,05 | 0,52 |
| | 5 | 474,1 | 4,33 | 1,56 | 0,52 |
| | 10 | 471,7 | 8,16 | 5,24 | 0,52 |
| | 20 | 473,5 | 15,45 | 10,43 | 0,52 |

Tabela 6.4: Resultados experimentais para o algoritmo KNM (fonte: [Lin83], p. 15).

exponencial. Assim, é provável que o número limite de processadores que mantenha a eficiência seja bastante alto.

### 6.3.4 Experimentos com SSS* - I/II

As duas paralelizações de SSS* propostas por Kumar e Kanal em [KK84] são também testadas pelos autores. Para o algoritmo SSS*-I (versão do algoritmo seqüencial SSS* nos moldes do Aspiration Search), é mencionada que uma aceleração de 1,25 foi obtida em um sistema com dois processadores (p. 775), sem serem apresentadas, no entanto, maiores explicações.

O algoritmo SSS*-II (versão simples de Tree-Splitting para a busca SSS*) é objeto de maiores estudos. Uma simulação com 50 árvores de jogo aleatórias de cada tipo foi feita (p. 776), e um resumo dos resultados é apresentado na tabela 6.5. A metodologia de cálculo é idêntica à adotada neste trabalho, sendo que a aceleração é, apropriadamente, calculada com base no tempo seqüencial do SSS*.

| total de procs. | árvore | | $NBP_{SSS*}$ | $NBP_{SSS*II}$ | $S_{SSS*II}$ | $E_{SSS*II}$ |
|---|---|---|---|---|---|---|
| | d | h | | | | |
| 3 | 3 | 7 | 456 | 203 | 2,29 | 0,76 |
| | 3 | 8 | 873 | 520 | 1,71 | 0,57 |
| | 3 | 9 | 2148 | 1043 | 2,11 | 0,70 |
| 5 | 5 | 5 | 626 | 203 | 3,14 | 0,63 |
| | 5 | 6 | 1662 | 755 | 2,25 | 0,45 |
| | 5 | 7 | 6037 | 2193 | 2,80 | 0,56 |
| 8 | 8 | 3 | 187 | 38 | 4,95 | 0,62 |
| | 8 | 4 | 693 | 239 | 2,92 | 0,37 |
| | 8 | 5 | 3631 | 897 | 4,09 | 0,51 |

Tabela 6.5: Resultados experimentais para o algoritmo SSS*-II (fonte: [KK84], p. 776).

Observe-se a boa eficiência deste algoritmo, mesmo com 8 processadores. Por outro lado, é interessante verificar a degradação de performance nas árvores de altura par, que provavelmente se deve à característica do algoritmo dual-SS* utilizado, de gerar todos os nós filhos de um nó de altura ímpar. Esta geração é feita tantas vezes, em uma árvore de altura par $n$, quantas em uma de altura $n + 1$, ocasionando aumento no tamanho da lista e, assim, adiamento de cortes.

## 6.4 Avaliação da Eficiência do Método de Redução de Janelas

No capítulo anterior fez-se referência à existência de dois métodos básicos, nos quais todos os algoritmos paralelos são baseados. Conforme foi dito, os métodos exploram duas propriedades

diferentes do processo de busca em árvores de jogo: a possibilidade de redução de janelas e o deslocamento da árvore solução para a esquerda (através da pré-ordenação dos movimentos).

A fim de verificar o poder de cada um dos métodos, realizaram-se simulações de algoritmos paralelos, executados sobre árvores geradas artificialmente. Para cada método escolheram-se dois algoritmos cuja diferença fosse, fundamentalmente, a propriedade na qual o método é baseado. Estes algoritmos foram então executados sobre árvores de diferentes tamanhos, geradas em dois modelos de distribuição diferentes, e os resultados médios comparados, em busca do ganho proporcionado por cada método.

Para a obtenção dos resultados experimentais foi utilizado um simulador de paralelismo desenvolvido pelo autor e pelo prof. Marcos D. Gubitoso (IME-USP), denominado *Máquina WORM*. Este simulador utiliza microcomputadores da linha IBM/PC/XT, e é bastante flexível, o que possibilitou à simulação da arquitetura de processadores em árvore. Uma melhor descrição de todo o sistema pode ser encontrada no apêndice A.

Por sua vez, as árvores de jogo foram produzidas inicialmente por geração aleatória de números inteiros entre -127 e 127, consistindo, dessa forma, em árvores baseadas no modelo de distribuição *aleatória* descrito no capítulo 4. Posteriormente foi feito um ordenamento nos nós terminais destas árvores, de forma que 85% das vezes o melhor movimento em um nó era o 1º à esquerda, constituindo-se, portanto, em uma versão simplificada do modelo de distribuição *fortemente ordenada*. Em ambos os casos as árvores foram geradas em três tamanhos, denominados Comprido, Largo e Grande a saber:

| | | | | |
|---|---|---|---|---|
| Comprido | : $d = 4$ | $h = 8$ | term.= | 65536 |
| Largo | : $d = 7$ | $h = 6$ | term.= | 117649 |
| Grande | : $d = 16$ | $h = 4$ | term.= | 65536 |

De cada tamanho foram geradas 5 árvores, em dois modelos de distribuição diferentes; maiores detalhes do método de geração são descritos também no apêndice A.

Esta seção tem como objetivo descrever a metodologia e os resultados obtidos no teste do método de paralelização por redução de janelas; a próxima seção analisa a paralelização por priorização do trabalho obrigatório.

Conforme já foi visto, a redução de janelas é baseada na propriedade descrita na proposição 5.1, estabelecida por Finkel e Fishburn. O algoritmo Tree-Splitting, dos mesmos autores, implementa-a por intermédio da interrupção Update: sempre que um processador recebe um valor de algum filho, a sua janela de busca é atualizada, e, se necessário, é feita também a correção de seus processadores escravos.

Uma maneira simples de avaliar a eficiência do método de redução de janelas é, portanto, comparar o algoritmo Tree-Splitting *com* e *sem* a interrupção Update. Para maior facilidade serão utilizadas, daqui em diante, as seguintes denominações:

| | | |
|---|---|---|
| algoritmo Tree | = | algoritmo Tree-Splitting sem a interrupção Update; |
| algoritmo TreeUp | = | algoritmo Tree-Splitting com a interrupção Update. |

É importante perceber que, como o algoritmo Tree-Splitting coloca nós de mesmo nível em processadores de mesmo nível, não há absolutamente nenhuma utilização do método de priorização do trabalho obrigatório. Mais que isso, não há influência importante de nenhum

outro fator, e portanto diferenças de desempenho entre Tree e TreeUp dever ser *exclusivamente creditadas* à redução de janelas.

Como método de comparação utilizou-se o número total de nós terminais examinados com $k$ processadores ($\sum_{i=1}^{k} NBP_i(k)$), e também o valor de eficiência média ($\overline{E}(k) = NBP_{\alpha-\beta} / \sum_{i=1}^{k} NBP_i(k)$), onde $k$ inclui todos os processadores da árvore e o algoritmo Alpha-Beta é tomado como referência. Não foi utilizado o cálculo de eficiência baseado no máximo número de nós terminais examinados por processador ($E(k)$, conforme descrito no início deste capítulo) porque o sistema de simulação apresenta deficiências de sincronização que comprometem significativamente a veracidade deste valor.

As cinco árvores de cada tamanho e modelo foram submetidas a diferentes configurações da árvore de processadores, e a média dos resultados das cinco árvores é mostrada na tabela 6.6. Foram utilizadas as seguintes configurações, onde $f$ é o grau e $n$ a altura da árvore de processadores:

$f = 2$ $n = 1$ total: 3 processadores
$f = 3$ $n = 1$ total: 4 processadores
$f = 5$ $n = 1$ total: 6 processadores
$f = 10$ $n = 1$ total: 11 processadores
$f = 3$ $n = 2$ total: 13 processadores
$f = 2$ $n = 3$ total: 15 processadores
$f = 3$ $n = 3$ total: 40 processadores

| tipo de árvore de jogo | | | configuração da árvore de processadores | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| tamanho | modelo | algoritmo | 2 × 1 | 3 × 1 | 5 × 1 | 10 × 1 | 3 × 2 | 2 × 3 | 3 × 3 |
| Comprido | dist. aleat. | Tree | 8027 | 9605 | 10367 | 10367 | 12431 | 11347 | 17318 |
| | | TreeUp | 7699 | 8823 | 9749 | 9743 | 10626 | 10455 | 14984 |
| | fort. orden. | Tree | 1199 | 1658 | 1864 | 1864 | 2631 | 2583 | 5112 |
| | | TreeUp | 1171 | 1471 | 1740 | 1729 | 2469 | 2441 | 4563 |
| Largo | dist. aleat. | Tree | 13266 | 14771 | 17514 | 20520 | 19844 | 17868 | 25763 |
| | | TreeUp | 12750 | 13325 | 15342 | 18236 | 17224 | 17059 | 22608 |
| | fort. orden. | Tree | 1950 | 2377 | 3319 | 4062 | 4099 | 3964 | 7713 |
| | | TreeUp | 1762 | 2031 | 2717 | 3508 | 3576 | 3652 | 6880 |
| Grande | dist. aleat. | Tree | 8791 | 10050 | 11824 | 15986 | 12860 | 11491 | 15527 |
| | | TreeUp | 8496 | 9266 | 10608 | 14469 | 12352 | 10674 | 14071 |
| | fort. orden. | Tree | 1486 | 1839 | 2526 | 4393 | 3338 | 2969 | 5346 |
| | | TreeUp | 1439 | 1680 | 2367 | 3719 | 3097 | 2809 | 4995 |

Tabela 6.6: Número total de nós terminais examinados pelos algoritmos Tree e TreeUp.

As mesmas árvores foram submetidas ao algoritmo seqüencial Alpha-Beta, obtendo-se o seguinte número médio de nós terminais examinados:

| | | | |
|---|---|---|---|
| Comprido | : | dist. aleat. | 6898 |
| | | fort. orden. | 898 |
| Largo | : | dist. aleat. | 10943 |
| | | fort. orden. | 1333 |
| Grande | : | dist. aleat. | 8020 |
| | | fort. orden. | 1179 |

Com base nestes dados e nos resultados da tabela 6.6 pôde-se calcular a eficiência média dos algoritmos Tree e TreeUp em cada caso. As figuras 6.1, 6.2 e 6.3 exibem, graficamente, os valores da eficiência média para árvores dos tamanhos Comprido, Largo e Grande, respectivamente.

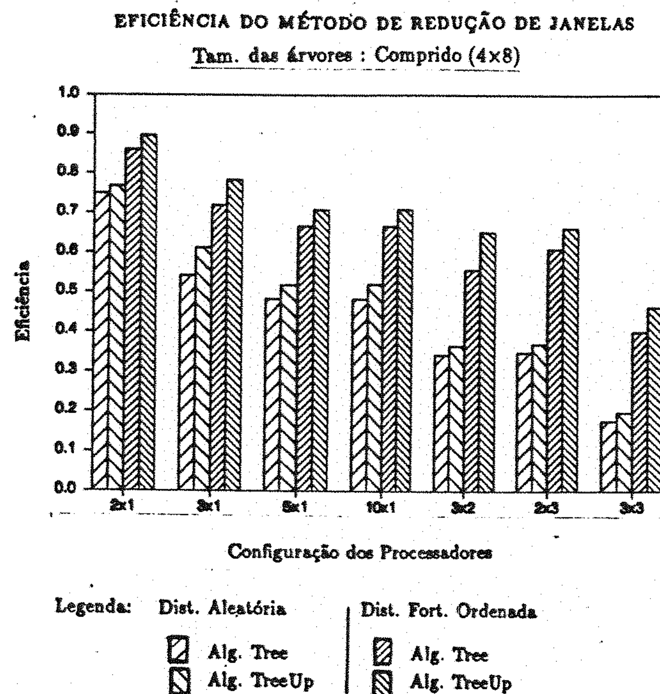



Figura 6.1: Eficiência dos algoritmos Tree e TreeUp na busca sobre árvores de tamanho Comprido.

Os gráficos das figuras 6.1, 6.2 e 6.3 apontam uma única conclusão, que o algoritmo TreeUp é sempre mais eficiente que o algoritmo Tree. Além disso, observam-se ganhos (entre 3% e 18%, tipicamente 5%) qualquer que seja o tamanho, modelo e configuração da árvore de processadores. Não há dúvida, pois, que o emprego do método de redução de janelas realmente melhora a performance de um algoritmo paralelo de busca em árvores de jogo.

Contudo, observa-se que este ganho nunca é muito grande. A causa deste fato é possivelmente a mesma que a observada quando do estudo da limitação de aceleração do algoritmo Aspiration Search: a redução da janela de busca não diminui *proporcionalmente* o tempo de busca. Mesmo com as sucessivas atualizações (através da interrupção Update), partes desnecessárias da busca continuam a ser feitas, devido ao *atraso* da chegada das mensagens (causado intrinsecamente pelo paralelismo).

Por outro lado, um exame mais cuidadoso do algoritmo Tree-Splitting (figura 5.6) mostra que o número de mensagens geradas é pequeno. Ou seja, com um baixo custo em termos de comunicação, consegue-se um relativo aumento na eficiência. Considerando-se ainda que

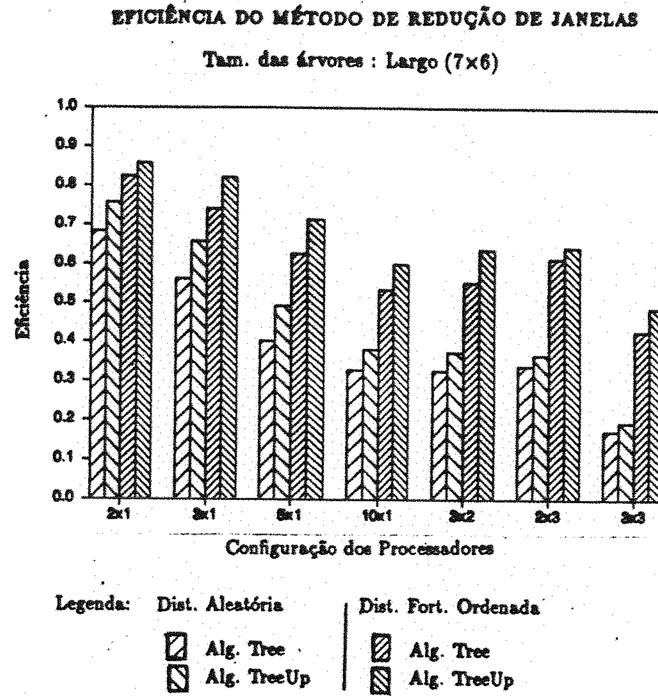



Figura 6.2: Eficiência dos algoritmos Tree e TreeUp na busca sobre árvores de tamanho Largo.

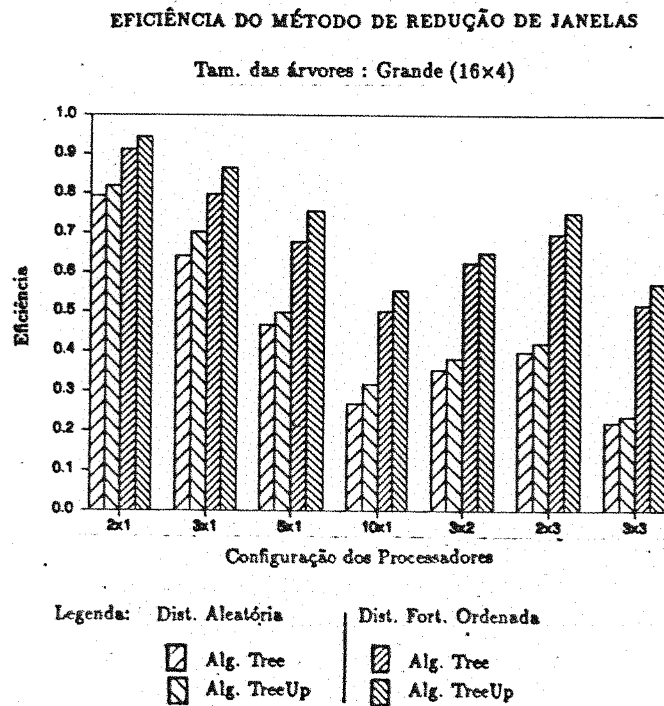



Figura 6.3: Eficiência dos algoritmos Tree e TreeUp na busca sobre árvores de tamanho Grande.

o método de redução de janelas pode ser utilizado sem a necessidade de sincronização, o uso de interrupções atualizadoras (do tipo da Update) não introduz nenhum atraso considerável no sistema. Também não há razão para supor que estes fenômenos não se repitam se a idéia de redução de janelas for utilizada em outros algoritmos paralelos, como, por exemplo, o PV-Splitting e o MWF-Tree (neste último, somente na 2ª fase).

Conclui-se, assim, que o método de paralelização por redução de janelas deve ser empregado sempre que a máquina paralela for suficientemente flexível para aceitá-lo, sem que, contudo, espere-se um aumento drástico na eficiência da busca.

## 6.5 Avaliação da Eficiência do Método de Priorização do Trabalho Obrigatório

A escolha de dois algoritmos para a avaliação do método de priorização do trabalho obrigatório é bem mais delicada que no caso anterior, no qual bastava a utilização discriminada da interrupção Update.

Após alguns estudos, decidiu-se comparar os algoritmos Tree-Splitting sem Update (algoritmo Tree, conforme denominação da seção anterior) e o algoritmo MWF-Tree. Este último apresenta algumas vantagens, em termos de estudos comparativos, sobre outros algoritmos que priorizam o trabalho obrigatório, como, por exemplo, o PV-Splitting. Como foi visto no capítulo anterior, o algoritmo MWF-Tree justapõe as raízes da árvore de jogo e da árvore de processadores, atribuindo ao primeiro escravo o primeiro filho, e aos escravos seguintes os netos mais à esquerda.

É portanto, mais semelhante ao algoritmo Tree do que o PV-Splitting. Além disso, quando não é encontrada refutação de uma árvore solução, há reexame de nós terminais; logo, MWF-Tree é um algoritmo bastante influenciado por pré-ordenação dos movimentos, o que possibilita maior clareza na avaliação da correlação existente entre essa técnica e o algoritmo utilizado.

Por outro lado, não se utiliza a interrupção Update no algoritmo Tree-Splitting para evitar a influência do método de redução de janelas (negativa a este estudo). Dessa forma a diferença entre os algoritmos comparados situa-se basicamente na forma pela qual os processadores são dispostos na árvore de jogo. A figura 6.4 ilustra as disposições iniciais dos dois algoritmos, e evidencia a tendência do MWF-Tree de buscar uma árvore-solução (e suas refutações) mais à esquerda.

Novamente foram usados como método de comparação o número total de nós examinados e a eficiência média com $k$ processadores. Os mesmos tamanhos, modelos e configurações foram utilizados, e os resultados obtidos são apresentados na tabela 6.7.

Com base nos dados para a busca com o algoritmo Alpha-Beta mostrados na seção anterior, calculou-se a eficiência média dos dois algoritmos em cada caso. As figuras 6.5 e 6.6 exibem, graficamente, os valores da eficiência média para árvores no tamanho Comprido com modelo de distribuição aleatória e fortemente ordenada, respectivamente; as figuras 6.7 e 6.8 fazem o mesmo para o tamanho Largo, e as figuras 6.9 e 6.10 fazem o mesmo para o tamanho Grande.

Examine-se inicialmente as figuras 6.5 e 6.6: para árvores de tamanho Comprido, o algoritmo Tree é melhor que o MWF-Tree se o modelo de distribuição for aleatório, e pior que este último no caso de distribuição fortemente ordenada. A situação se repete para o tamanho Largo (figuras 6.7 e 6.8), mas não ocorre para o tamanho Grande (figuras 6.9 e 6.10), onde, em três

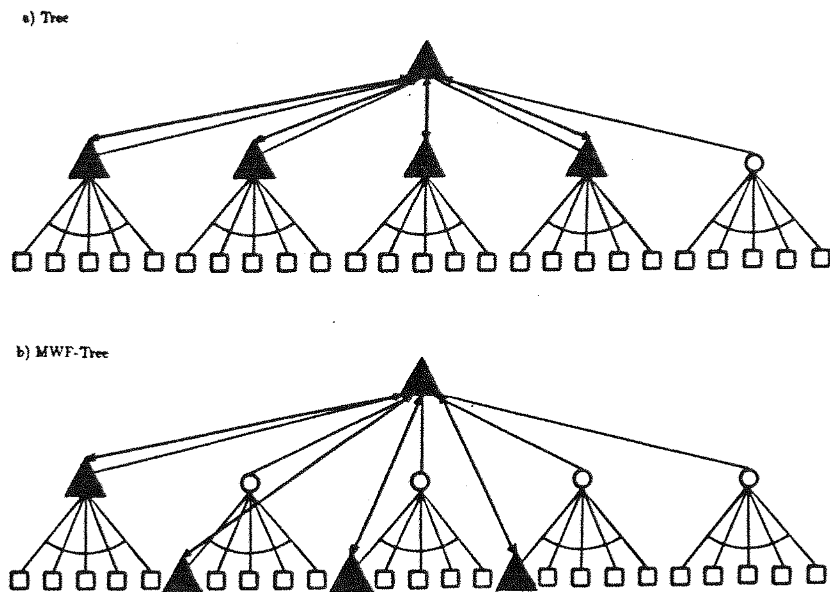



Figura 6.4: Comparação entre as disposições iniciais dos processadores nos algoritmos Tree e MWF-Tree.

| tipo de árvore de jogo | | | configuração da árvore de processadores | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| tamanho | modelo | algoritmo | 2 × 1 | 3 × 1 | 5 × 1 | 10 × 1 | 3 × 2 | 2 × 3 | 3 × 3 |
| | dist. | Tree | 8027 | 9605 | 10367 | 10367 | 12431 | 11347 | 17318 |
| comprido | aleat. | MWF-Tree | 11876 | 11877 | 11877 | 11877 | 19225 | 29406 | 30008 |
| | fort. | Tree | 1199 | 1658 | 1864 | 1864 | 2631 | 2583 | 5112 |
| | orden. | MWF-Tree | 1222 | 1222 | 1222 | 1222 | 1552 | 2069 | 2069 |
| | dist. | Tree | 13266 | 14771 | 17514 | 20520 | 19844 | 17868 | 25763 |
| largo | aleat. | MWF-Tree | 20785 | 20785 | 20785 | 20785 | 35583 | 57440 | 60415 |
| | fort. | Tree | 1950 | 2377 | 3319 | 4062 | 4099 | 3964 | 7713 |
| | orden. | MWF-Tree | 1948 | 1948 | 1948 | 1948 | 2869 | 3596 | 3828 |
| | dist. | Tree | 8791 | 10050 | 11824 | 15986 | 12860 | 11491 | 15527 |
| grande | aleat. | MWF-Tree | 18622 | 18839 | 18839 | 18839 | 32072 | 53617 | 55421 |
| | fort. | Tree | 1486 | 1839 | 2526 | 4393 | 3338 | 2969 | 5346 |
| | orden. | MWF-Tree | 2337 | 2371 | 2401 | 2401 | 3276 | 3686 | 3975 |

Tabela 6.7: Número total de nós terminais examinados pelo algoritmo Tree e MWF-Tree.

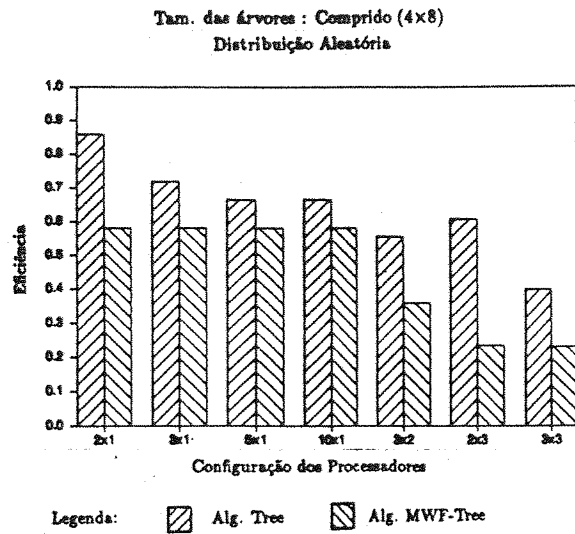



Figura 6.5: Eficiência dos algoritmos Tree e MWF-Tree na busca sobre árvores de tamanho Comprido com distribuição aleatória.

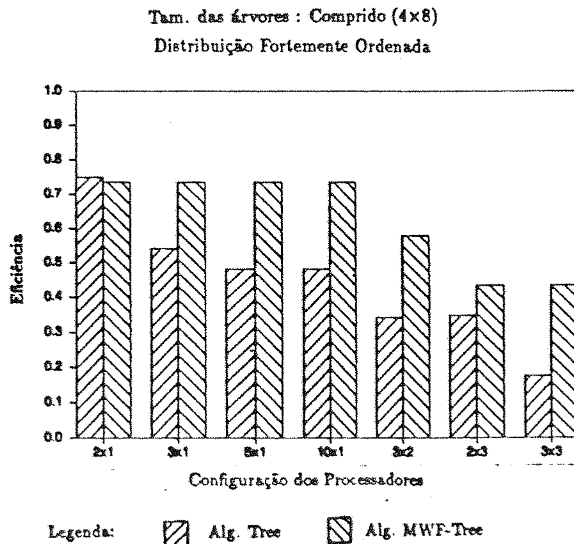



Figura 6.6: Eficiência dos algoritmos Tree e MWF-Tree na busca sobre árvores de tamanho Comprido com distribuição fortemente ordenada.

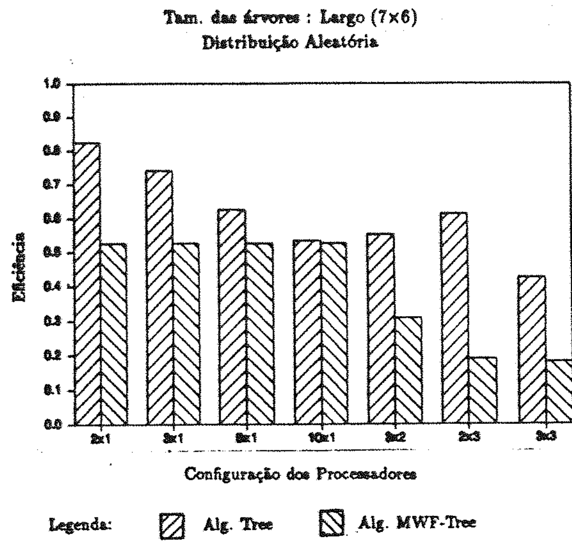



Figura 6.7: Eficiência dos algoritmos Tree e MWF-Tree na busca sobre árvores de tamanho Largo com distribuição aleatória.

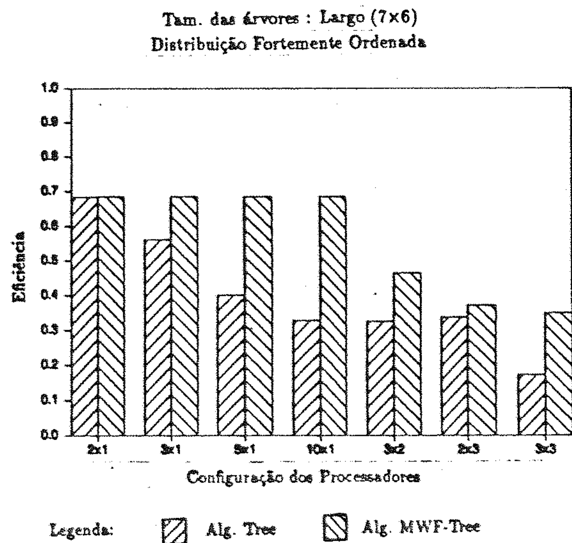



Figura 6.8: Eficiência dos algoritmos Tree e MWF-Tree na busca sobre árvores de tamanho Largo com distribuição fortemente ordenada.

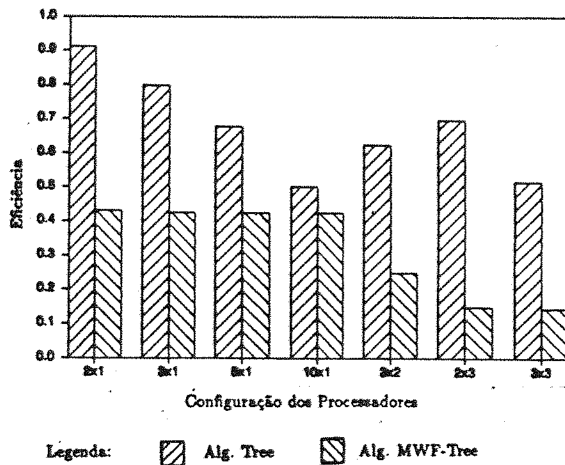



Figura 6.9: Eficiência dos algoritmos Tree e MWF-Tree na busca sobre árvores de tamanho Grande com distribuição aleatória.

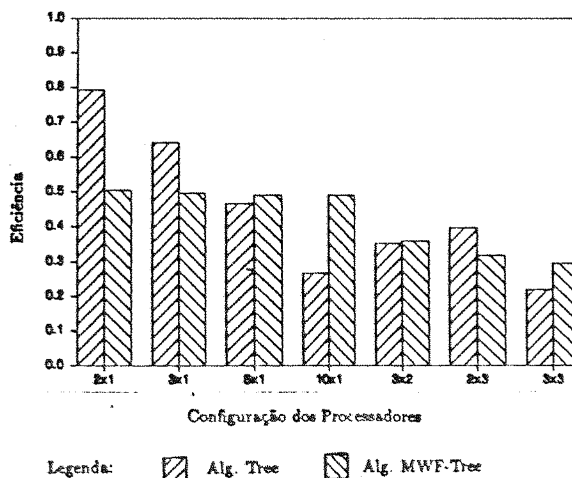



Figura 6.10: Eficiência dos algoritmos Tree e MWF-Tree na busca sobre árvores de tamanho Grande com distribuição fortemente ordenada.

dos sete casos, o algoritmo Tree é mais eficiente.

Há razões para crer que este último é um resultado anômalo, pois a altura das árvores de tamanho Grande, $h$=4, não permite a utilização de toda a potência do algoritmo MWF-Tree. Por exemplo, quando é utilizada uma árvore de processadores de altura $n$=2, vários processadores são colocados diretamente sobre nós terminais da árvore de jogo. Isto faz com que os valores retornados sejam de pouco valor para a determinação de refutações, pois são *demasiadamente aleatórios*, causando um número excessivo de reexames. Esta anomalia também ocorre para árvores de processadores de alturas $n$=1 e $n$=3, conforme pode ser visto nos gráficos, e, compreensivelmente, é mais acentuada no caso de distribuição aleatória.

Dado, pois, que a situação do tamanho Grande é fruto de condições perturbantes, é mais prudente considerar-se, na avaliação de eficiência, somente os tamanhos Comprido e Largo. Nestes observa-se que, no modelo aleatório, a eficiência do algoritmo Tree é maior que a do algoritmo MWF-Tree, com o ganho variando entre 16% e 165%. Contudo, para o modelo fortemente ordenado a situação se inverte drasticamente, com o MWF-Tree possibilitando ganhos de até 138% sobre o algoritmo Tree.

Esta inversão deixa evidente que algoritmos baseados em priorização do trabalho obrigatório são bem mais competitivos quando utilizados, em testes, sobre árvores fortemente ordenadas, ou equivalentemente, em jogos reais, quando há possibilidade de uma razoável pré-ordenação de movimentos. Embora essa característica seja especialmente evidente no algoritmo MWF-Tree (pois ele, no pior caso, faz *reexame* de nós terminais), é de se esperar que seja comum a todos os algoritmos que usam o método: a priorização do trabalho obrigatório é conseguida sempre graças a algum *custo extra*, e, nos casos em que o trabalho obrigatório não é suficiente para resolver a busca, este custo extra é suficiente para degradar a eficiência a níveis abaixo da conseguida por algoritmos que não façam priorização. Contudo, se a árvore de jogo possui boa ordenação, este custo extra torna-se desprezível se comparado à quantidade de trabalho inútil feito por uma paralelização simples.

É particularmente importante se perceber que, mesmo utilizando um algoritmo demasiadamente sensível ao desordenamento, como é o MWF-Tree, ainda assim é suficiente um índice de cerca de 85% de ordem para se conseguir, por vezes, dobrar a eficiência. É provável que algoritmos melhores, como o PV-Splitting e o MWF, consigam aumentar ainda mais a eficiência, e diminuindo também a degradação.

Comparem-se estes números com os da seção anterior: com redução de janelas consegue-se, invariavelmente, algum ganho, embora limitado, enquanto que com a priorização do trabalho obrigatório o ganho, embora dependente da possibilidade de pré-ordenação das árvores, chega facilmente a valores consideráveis.

Extraem-se, assim, duas observações importantes sobre os dois métodos de paralelização:

- o método de redução de janelas é bastante estável, mas produz ganhos pequenos;
- o método de priorização do trabalho obrigatório é capaz de produzir alto ganho na eficiência, sendo, contudo, instável em relação à ordenação da árvore de jogo.

No próximo capítulo, conclusão deste trabalho, são levantadas algumas hipóteses para explicar estes fenômenos, e também algumas de suas prováveis conseqüências.

## 6.6 Observações sobre Algoritmos para Processadores com Memória Compartilhada

De todos os algoritmos paralelos apresentados no capítulo anterior, restaram dois que não foram comentados até agora: o MWF e o Pool-SSS*. Ambos os algoritmos possuem uma estrutura semelhante, quer seja, uma lista de mensagens entre nós da árvore, da qual é extraída o topo, sempre que um processador fica livre. Esta mensagem é então tratada por um programa interno ao processador (igual para todos), e, ao final, um novo conjunto de mensagens é gerado e apropriadamente inserido na lista. No caso do MWF, a ordem da lista é função do nó da árvore ao qual a mensagem se refere (nós mais à esquerda com maior prioridade); no Pool-SSS*, as mensagens são decrescentemente ordenadas pelo valor.

É importante notar que na arquitetura de memória compartilhada o problema de comunicação é substituído pelo de colisão no acesso à memória. Assim, é de se esperar que haja perda de eficiência devido a atrasos decorrentes de acesso múltiplo a dados, especialmente no que se refere à lista de mensagens, que deve ser constantemente examinada e atualizada. No algoritmo Pool-SSS*, este problema é agravado, em certos casos, pela necessidade de retirada de todos os descendentes de um nó, exigindo o exame quase que completo da lista de mensagens.

Contudo, é provável que o exame da aceleração e da eficiência pelo número de nós terminais examinados revele um valor bem elevado. Isto já aconteceu para os resultados experimentais do algoritmo KNM, e é explicado pela distribuição proporcional de serviço entre os nós e pela ordem da lista de mensagens. No entanto, é preciso ter em mente que a medida de eficiência pelos nós terminais esconde todos os atrasos decorrentes de colisões, que devem, neste particular caso, consistir no fator principal de degradação desses algoritmos.

Em outras palavras, simulações e idealização de condições de trabalho são fatores perturbadores muito fortes em algoritmos que utilizam arquitetura de memória compartilhada. Além disso, no caso específico dos dois algoritmos em consideração, o tempo de exame e ordenamento da lista não é desprezível; se forem utilizadas árvores de jogo geradas artificialmente se estará em uma condição mais crítica ainda, pois o tempo de exame de um nó terminal é mínimo, e todos os fenômenos de colisão serão provenientes exclusivamente do trabalho de ordenação.

Um último fator deve ainda ser considerado: conforme notam Akl, Barnard e Doran em [ABD82], à medida que aumenta o número de processadores, mais mensagens são geradas simultaneamente, aumentando também a lista de mensagens e, conseqüentemente, seu tempo de exame e manutenção. No algoritmo MWF, caso o número de processadores seja muito grande, poderá ser gerada, para uma árvore de grau $d$ e altura $h$, uma lista de $((d - \frac{3}{4})^{\frac{1}{2}} + \frac{1}{2})^{h}$ mensagens; para $k$ pequeno, este tamanho é , tipicamente, $O(k.h)$.

No algoritmo Pool-SSS*, a lista pode chegar a $O(k.d^{\frac{h}{2}})$, o que, com certeza, consome um tempo considerável no trabalho de ordenação, exame e eliminação de mensagens.

Em virtude de todos esses fatores negativos, não foi considerada apropriada a realização de simulações para os dois algoritmos: para tanto seria necessário desenvolver um simulador mais sofisticado que o existente, bem como programar um jogo real e parametrizar uma avaliação de seu desempenho, o que está bastante além do escopo deste trabalho.

# Capítulo 7

# Conclusão

Após a análise e avaliação feita no capítulo anterior, cabe a esta conclusão o aprofundamento e a extensão das observações feitas sobre algoritmos paralelos para busca em árvores de jogo. Relembrando os objetivos propostos na introdução deste trabalho (capítulo 1), vê-se que é necessário uma melhor determinação das propriedades nas quais os algoritmos paralelos são baseados, o que é o objetivo das seções iniciais desta conclusão.

Também é preciso, a partir do exame do caso de algoritmos para busca em árvores de jogo, extrair propriedades e problemas típicos dos processos de paralelização, o que é abordado nas seções intermediárias. É importante ter em mente, contudo, que estas conclusões serão só em parte baseadas em fatos concretos ou em demonstrações: devido a sua generalidade, as principais idéias podem somente ser apresentadas, juntamente com argumentos razoáveis levantados a seu favor. Não pode ser esperado, considerando as limitações de uma dissertação de mestrado, que conceitos profundos sejam defendidos rigorosamente.

## 7.1 A Componente Serial do Algoritmo Alpha-Beta

Conforme foi visto na seção 5.2, uma característica fundamental do algoritmo Alpha-Beta é aproveitar os resultados da busca já realizada na diminuição do trabalho a realizar. Viu-se, também, que a divisão de trabalho entre vários processadores afeta drasticamente esse aproveitamento de informações, e que boa parte do problema de paralelização, neste caso específico, consiste precisamente em diminuir a perda de informação.

Pode-se dizer, então, que o algoritmo Alpha-Beta apresenta uma forte *componente serial* (assim como os demais algoritmos seriais para busca em árvores de jogo, à exceção do Minimax), ou seja, uma parte considerável de sua eficiência é proveniente da seqüência na qual as operações são realizadas. Algoritmos com forte componente serial seriam, desse modo, algoritmos cujo funcionamento é bastante baseado na utilização de informação previamente obtida; por exemplo, o algoritmo Alpha-Beta tem uma componente serial mais forte que o algoritmo Bound, porque este último utiliza somente o valor $\beta$ proveniente de nós irmãos, enquanto que o valor $\alpha$ pode depender de nós de parentesco mais distante (relembre-se as noções de cortes rasos e profundos, seção 3.5).

É claro que quanto maior a componente serial de um algoritmo, maiores as dificuldades esperadas em sua paralelização. No caso do Alpha-Beta, o problema é agravado por outra característica indesejável dessa componente serial: sua dependência em relação à instância

particular da árvore de jogo que está sendo examinada. Conforme o arranjo da AND-estratégia defensiva ótima ao longo da árvore de jogo, mais à esquerda ou mais à direita, a informação prévia tem maior ou menor peso, respectivamente. Além disso, a relação de dependência entre os nós não é constante: se no início de qualquer algoritmo toda uma AND-árvore solução pode ser examinada ao mesmo tempo (portanto, $d^{\frac{h}{2}}$ nós terminais), no final do processo quase sempre um único nó pode ser analisado de cada vez. Esta relação deve ser entendida, neste caso, como a necessidade que um nó tem de possuir informações para realizar um corte, informações estas que são provenientes do exame de outros nós terminais, estabelecendo-se assim uma dependência de exame prévio destes últimos para a determinação do primeiro.

Estas dificuldades se traduzem na moderada eficiência alcançada pelos algoritmos e no esforço computacional (memória e comunicação) requeridos para superar o atraso de chegada de informações. É particularmente ilustrativo observar os resultados da análise teórica do algoritmo Tree-Splitting (seção 6.2), sem redução de janelas, para árvores de jogo de grau $d$ e altura $h$, e $k$ processadores: para árvores com a melhor ordem possível, o algoritmo seqüencial examina $2d^{\frac{h}{2}}-1$ nós terminais, mas consegue-se somente uma aceleração de $\sqrt{k}$; no caso oposto, em que a árvore está ordenada da pior forma possível, examinam-se $d^h$ nós terminais, mas com uma aceleração de $k$.

Observe-se que, no caso de melhor ordem da árvore de jogo, consegue-se examinar serialmente o mínimo possível de nós terminais. Contudo, como essa performance é fruto da avaliação seqüencial ótima, tem-se o mínimo de eficiência deste algoritmo paralelo. Ao contrário, no caso de pior ordem, a informação prévia de nada auxilia a busca, o que permite que nenhum trabalho desnecessário seja feito, e assim a aceleração atinge o máximo teórico.

Tais características tornam o problema de busca em árvores de jogo particularmente interessante do ponto de vista teórico, na medida em que as maiores dificuldades no desenvolvimento de algoritmos paralelos devam ser esperadas em problemas cujas melhores soluções conhecidas sejam também fortemente seriais. O desafio, em computação paralela, traduz-se sempre na manutenção da eficiência do algoritmo seqüencial, mas "repartida" igualmente pelos múltiplos processadores.

O algoritmo Alpha-Beta — e seus semelhantes — pertence a esse grupo muito importante de algoritmos de forte componente serial, apresentando, desse modo, dificuldades consideráveis na obtenção de alta e uniforme eficiência de suas versões paralelas.

## 7.2 A Dependência entre o Método de Paralelização e o Tipo de Corte Obtido

Um ponto interessante levantado pelas simulações realizadas pelo autor consistiu em verificar que o método de paralelização por redução de janelas produz um ganho inferior à priorização do trabalho obrigatório, desconsiderando-se as questões de instabilidade deste último. No primeiro observou-se um ganho típico de 5% a 10%, enquanto que no segundo a melhora variou entre 50% e 140% (para o caso do modelo de distribuição fortemente ordenada).

Estes números devem ser atribuídos à existência de uma dependência entre o método de paralelização empregado e o tipo de corte obtido: redução de janelas viabiliza, majoritariamente, a ocorrência de cortes profundos, enquanto que priorização do trabalho obrigatório realiza somente cortes rasos. Relembrando a conjectura de Knuth e Moore — cortes profundos têm

somente um efeito de segunda ordem na eficiência do algoritmo Alpha-Beta — apresentada na seção 4.5, (e posteriormente verificada por J. Pearl), pode-se concluir que os efeitos esperados quando do emprego de redução de janelas sejam também de segunda ordem, enquanto que o aumento na execução de cortes rasos deve produzir ganhos decisivos na eficiência.

É necessário, contudo, entender-se o motivo pelo qual o método de redução de janelas trabalha basicamente com cortes profundos. Examine-se um algoritmo típico, o Tree-Splitting, na situação diagramalizada pela figura 7.1. Nesta, uma árvore de jogo é avaliada por uma árvore de processadores de grau 4 e altura 1.

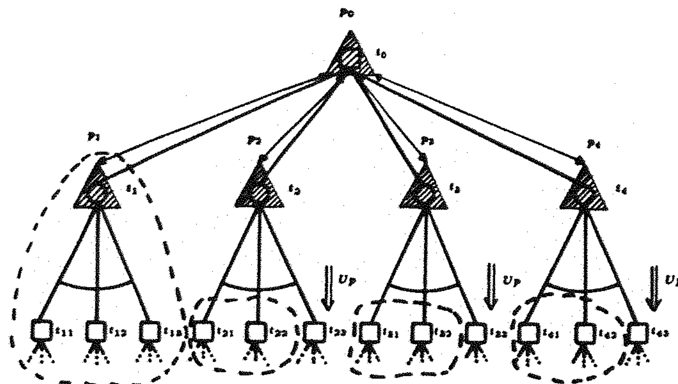


Figura 7.1: Situação típica de chamada da interrupção Update pelo algoritmo Tree-Splitting.

Se for suposto que todos os processadores examinam suas sub-árvores exatamente no mesmo tempo, então todos terminam a busca juntos e, portanto, não há efeito algum de uma eventual chamada da interrupção Update. A figura 7.1 retrata uma situação mais provável, na qual um processador ($p_1$) termina o exame de sua sub-árvore (de raiz $t_1$) antes dos demais. Nestas condições, a mensagem ascendente para $p_0$ provoca a execução das interrupções Update indicadas na figura. Contudo, devido ao sincronismo de tarefas, os demais processadores já realizaram boa parte da busca em suas sub-árvores (assinaladas na figura), de maneira *desinformada*.

Por esse motivo, tanto cortes rasos como profundos deixam de ser realizados na hora correta, os primeiros pela insuficiência do valor $\beta$ inicial dos nós $t_2$, $t_3$ e $t_4$, os segundos porque este mesmo $\beta$ corrigido não atuou nos níveis inferiores da árvore. O efeito da interrupção Update é, dessa forma, tardio, e tem somente a função de acelerar o *pouco* trabalho que resta a ser feito.

Esta aceleração, mesmo reduzida, é obtida de duas maneiras: pela comunicação de um valor $\alpha$ à busca realizada nas sub-árvores restantes, caso em que se produz um corte profundo; e também pela execução tardia de cortes rasos, causados pela propagação de um valor $\beta$ suficientemente baixo para cessar a busca em em todos os níveis. No entanto, este último não se caracteriza completamente como um corte raso, porque boa parte da busca na sub-árvore foi realizada *antes* de sua ocorrência.

É mais preciso, conseqüentemente, afirmar que a redução de janelas realiza cortes profundos e rasos, mas que, na prática, estes últimos não são feitos em sua plenitude, configurando-se, para o cômputo da eficiência, com o mesmo valor de cortes profundos.

Por outro lado, a afirmação de que o método de priorização do trabalho obrigatório realiza somente cortes rasos é bem mais fácil de verificar. Basta lembrar-se que o método busca precisamente encontrar uma estratégia defensiva ótima e suas refutações, e que são estas últimas que

caracterizam a ocorrência de cortes rasos. Além disso, os algoritmos que utilizam priorização do trabalho obrigatório não fazem, de um modo geral, propagação de valores $\alpha$, em parte para evitar o aumento no trabalho "administrativo" de manipulação da árvore ou na memória ocupada, em parte porque o efeito da redução de janelas não se faz sentir, comparativamente ao da priorização.

Foi visto, então, que a escolha do método de paralelização consiste, basicamente, na escolha entre realizar cortes rasos ou profundos. Mais do que uma opção, esta propriedade deve indicar que os dois métodos não se sobrepõem, mas sim que *ambos devem ser empregados complementarmente*, sendo a priorização do trabalho obrigatório responsável pelo "trabalho pesado" da diminuição do tempo de busca, e a redução de janelas pelo "polimento indispensável" alcançado pelo sistema de janelas Alpha-Beta.

## 7.3 A Influência do Modelo de Árvore na Escolha do Algoritmo Paralelo

A seção 6.5 do capítulo anterior mostrou que, apesar de eficiente, o método de paralelização por priorização do trabalho obrigatório produz um desempenho bastante instável relativamente à disposição da estratégia defensiva ótima dentro da árvore de jogo. Através dos resultados das simulações, foi visto que para árvores "artificiais" de modelo aleatório, o algoritmo MWF-Tree era bem menos eficiente que o Tree-Splitting, e vice-versa quando se utilizava o modelo de distribuição fortemente ordenada.

Antes de mais nada, é necessário compreender as causas desse efeito, que estão intimamente ligadas à forte componente serial do algoritmo Alpha-Beta. Conforme já foi explicado no início deste capítulo, o algoritmo Alpha-Beta (e sua versão fraca, Bound), tem a característica de realizar inicialmente, o exame de um grande número de nós terminais desinformadamente, e, à medida que a busca vai sendo feita, a interdependência entre os nós aumenta, "estrangulando", de certa forma, o paralelismo.

Ora, priorizar o trabalho obrigatório nada mais é do que fazer com que os diversos processadores explorem sempre nós independentes, através do direcionamento do poder computacional para o seu exame. Quando a árvore está bem ordenada, ou seja, a estratégia defensiva ótima está à esquerda, a ordem de prioridades se encaixa quase que perfeitamente à estrutura da árvore, e assim os nós corretos são examinados nos momentos certos.

Se, do contrário, a árvore apresenta pouca ordenação, os efeitos da priorização não são sentidos, pois a ordem de exame dos nós não consegue encontrar refutações imediatas. Por outro lado, à medida que se caminha para a direita da árvores, o fenômeno de estrangulamento da dependência entre nós, descrito acima, começa a ser sentido: a partir de um certo momento, os nós estão quase que seqüencialmente dependentes uns dos outros, e deixa de haver sentido em falar em trabalho obrigatório. Nestas condições, os algoritmos normalmente começam a atribuir prioridades espúrias aos nós, bem como a realizar tarefas "precipitadamente", causando a degradação de performance observada nas simulações.

Considerando que, no algoritmo MWF-Tree, a atribuição de uma prioridade inadequada causa o reexame de nós, e que, conforme explicado acima, não existe o sentido de prioridade no exame de uma árvore com modelo aleatório, então ficam evidentes as causas da degradação de desempenho deste algoritmo na simulação. Observe-se que apenas 85% de ordenamento já

é suficiente para garantir que haja um razoável sentido de prioridade entre os nós, e portanto, que os efeitos do método de priorização do trabalho obrigatório se manifestem.

A conseqüência lógica destas observações é a recomendação de uso de algoritmos que façam priorização do trabalho obrigatório sempre que for possível a *pré-ordenação dos movimentos* com um certo grau de precisão. Nestes casos se pode esperar que a eficiência seja alta, especialmente se forem utilizados algoritmos que não façam reexame de nós terminais como o MWF e o PV-Splitting. Opostamente, é de se esperar que, em jogos nos quais seja díficil estabelecer uma precedência *a priori* entre os movimentos, sejam mais eficientes algoritmos mais simples, como, por exemplo, o Tree-Splitting e o Aspiration Search.

É interessante notar que esta dependência entre o modelo de árvore de jogo e o algoritmo utilizado já existia no caso serial. Nas seções 4.5 e 4.6 desta dissertação foi visto que, em árvore aleatórias, o algoritmo SCOUT tem desempenho inferior ao Alpha-Beta, e este em relação ao SSS* (a nível de constantes, pois o fator de ramificação é o mesmo); entretanto, nos testes realizados sobre árvores com distribuição fortemente ordenada, estes três algoritmos seqüenciais praticamente se equivalem.

De certa forma, a paralelização parece reforçar esta dependência entre a instância do problema e o algoritmo utilizado: a relação de ganho entre SCOUT e Alpha-Beta, que se atenua em árvores bem ordenadas no caso serial, passa por um processo de *radicalização* quando se comparam os dois algoritmos paralelos derivados destes, MWF-Tree e Tree-Splitting, respectivamente. O que antes era uma moderada redução de ganho passa a ser uma drástica e considerável inversão de desempenhos.

A explicação deste fenômeno de radicalização pode ser, tentativamente, atribuída ao fato que, quando se tenta adaptar a árvore de processadores à uma árvore de jogo, está sendo feita uma "moldagem" mais efetiva entre o problema e o mecanismo solucionador (computador + algoritmo). Na medida em que os algoritmos passam a refletir mais acentuadamente o problema, não como um todo — o que é impossível — mas como uma instância particular, é de se esperar que variações de instâncias do problema produzam impactos mais fortes sobre a eficiência dos algoritmos.

Os algoritmos paralelos para busca em árvores de jogo parecem, de algum modo, ser mais especializados às características das árvores de jogo em exame do que seus equivalentes seriais.

## 7.4 A Relação Tempo × Comunicação

Na seção 6.4 do capítulo anterior, foi analisada a eficiência do método de redução de janelas, obtendo-se experimentalmente um ganho entre 3% e 18%, para o caso de algoritmo Tree-Splitting. É importante lembrar que a implementação utilizada não possuía qualquer espécie de sincronização, não havendo, dessa forma, perda de eficiência por espera de mensagens.

De um modo geral, dado o baixo número de mensagens geradas pelo método de redução de janelas, pode-se dizer que o tempo gasto no envio e tratamento dessas mensagens (no caso, interrupções Update) é desprezível se comparado com o tempo de processamento da árvore de jogo. Assim, é de se esperar que, mesmo que se meça o tempo real do algoritmo (e não conforme foi feito, o número de nós terminais examinados), o ganho na eficiência se mantenha nos mesmos limites, concluindo-se daí que a utilização de comunicação de resultados parciais entre processadores diminui o tempo de execução.

A relação inversamente proporcional entre tempo de execução e memória ocupada por um algoritmo é amplamente conhecida na ciência da computação. Os resultados das simulações realizadas nesta pesquisa parecem apontar também para outra direção, sugerindo uma nova relação, entre *tempo de execução e quantidade de comunicação*, e de natureza também inversa: quanto maior a comunicação, menor o tempo, e vice-versa.

É interessante notar que, como no caso da memória ocupada, o custo associado à redução do tempo é relativo a características de *hardware*, neste caso concernentes à rede de comunicação entre processadores necessária para a plena transmissão de mensagens. Contudo, há necessidade que a topologia da rede de processadores seja adequada ao roteamento de mensagens exigido pelo algoritmo, introduzindo-se assim uma componente de caráter qualitativo na avaliação da quantidade de comunicação exigida entre os processadores.

Esta última propriedade provavelmente se constitui em um importante fator de aumento de dificuldade na elaboração de uma demonstração matemática, análoga a do caso tempo × memória, que estabeleça a relação precisa entre tempo de execução e comunicação.

Longe disso, este trabalho propõe-se a destacar a provável ocorrência dessa relação inversa entre tempo de execução e comunicação entre processadores, que se acredita ser uma característica genérica de soluções baseadas em paralelismo para problemas computacionais.

Embora evidenciada nesta dissertação somente por alguns resultados experimentais, tal relação deve ser responsável, em última análise, pela eficiência apresentada pelo método de paralelização de algoritmos para busca em árvores de jogo baseado em redução de janelas.

## 7.5 A Relação Espaço × Comunicação

Conforme assinalado no capítulo 6, seção 6.6, quando é empregada uma arquitetura de múltiplos processadores com memória compartilhada, ocorre um fenômeno de "ocultamento" da comunicação entre os processadores, que é substituída pelo acesso a estruturas de dados comuns. A substituição, nestas condições, é capaz de provocar todos os problemas típicos de linhas de comunicação, como atrasos, *deadlock*, etc.

Os algoritmos paralelos para busca em árvores de jogo confirmam a ocorrência desse fenômeno, considerando-se os exemplos já fornecidos do MWF, do KNM e do Pool-SSS*, nos quais todo o intercâmbio de informações ocorre por intermédio da lista de mensagens. Particularmente nestes casos, o dispêndio de memória é bastante alto, conforme pode ser visto pelas seções 5.6 e 6.6, o que parece sugerir que o preço "pago" pela inexistência de intercomunicação é o aumento na área de memória ocupada pelas estruturas de dados.

Novamente, a topologia dos processadores influi decisivamente no espaço ocupado; por exemplo, para uma árvore de processadores fracamente conectada, seria necessário manter, em cada memória privativa, praticamente toda a lista de mensagens, recrudescendo-se, assim, as necessidades de memória. Por outro lado, se a comunicação for muito eficiente, pode-se dividir a lista (e mesmo otimizá-la) entre os diversos processadores.

Embora menos evidente no problema estudado, a relação inversamente proporcional entre *quantidade de comunicação* e *espaço de memória ocupado* deve, a exemplo da anterior, ser uma característica genérica de algoritmo paralelos. De certa forma, as diferentes arquiteturas mostradas na seção 5.1 já refletem essa relação, pois caracterizam soluções extremas e opostas para a configuração do *hardware*. É importante perceber que, no cômputo de qualquer comparação, é

indispensável considerar os algoritmos mais adequados à estrutura da máquina, e que só nesses casos se deve esperar que a relação seja satisfeita.

A plena caracterização matemática da dependência inversamente proporcional entre memória ocupada e quantidade de comunicação deve ser tão ou mais difícil que a anterior. Particularmente para algoritmos paralelos para busca em árvores de jogo, a quase inexistência de dados analíticos ou experimentais não contribui significativamente para a elucidação do problema, cabendo, entretanto, ressaltar que os algoritmos MWF, KNM e Pool-SSS* provêm algumas evidências comprobatórias da veracidade dessa relação.

## 7.6 Problemas em Aberto

As seções anteriores deste capítulo mostraram algumas conclusões sobre o problema de paralelização de algoritmos para busca em árvores de jogo. Como em toda pesquisa, um papel importante das afirmações feitas é sugerir novos problemas a serem investigados, impulsionando à frente a espiral do conhecimento humano.

Após este trabalho, as principais questões que se colocam dizem respeito à *generalização das observações para outros problemas de paralelização.* Os resultados e suas análises tiveram sempre como objeto o problema de busca em árvores de jogo, mas, de um modo geral, os mesmos parecem ser "reflexos" de propriedades mais genéricas, relativas à classe das máquinas e dos algoritmos paralelos. Coloca-se, então, a necessidade de realização de maiores investigações e pesquisas nos temas listados a seguir:

**Definição de Componente Serial**: todo problema de paralelização de algoritmos seqüenciais envolve necessariamente a questão de quanto trabalho serial o algoritmo intrinsecamente contém. Parece não haver dúvida de que há algoritmos com componente serial mais forte do que outros — a comparação Bound×Alpha-Beta é um flagrante exemplo. A pergunta que se coloca é como definir e calcular a componente serial de um algoritmo seqüencial, de forma que o resultado possa ser útil na escolha de candidatos à paralelização.
É importante notar que há outras evidentes aplicações para um tal quantificador, como, por exemplo, a Lei de Amdahl [1], que pode prover um limite superior de aceleração. Também é provável que um estudo mais rigoroso sobre componente seriais ajude uma análise da estrutura subjacente aos algoritmos paralelos, essencial para um melhor entendimento dos mesmos e, conseqüentemente, de seu processo de desenvolvimento.

**Verificação da Ocorrência de Radicalização da Dependência entre Algoritmo e Instância do Problema**: o fenômeno de aumento da relação entre o algoritmo de busca e o modelo de distribuição da árvore de jogo utilizados já foi devidamente comentado em seções anteriores deste capítulo. Conforme foi dito, uma possível explicação é o aumento de similaridade entre a instância e o algoritmo, gerada pela "moldagem" da máquina paralela ao problema considerado. Se esta última hipótese for verdadeira, é de se esperar que algoritmos paralelos, de um modo geral, sejam mais "sensíveis" a diferentes extremos da configuração do problema

[1] Conforme [Qui87], p. 19, a Lei de Amdahl afirma que a aceleração $S_p$ de um algoritmo paralelo $p$ com uma fração $f$ de operações seqüenciais, em uma máquina de $k$ processadores é limitada, $S_p \leq \frac{1}{f+(1-f)/k}$ .

do que seus análogos seriais. Mais do que objetivar uma definição precisa da propriedade, deve-se, neste caso, inicialmente colecionar evidências e diferentes ocorrências, para posteriormente tentar-se o enunciamento de leis e princípios gerais.
Caso seja verdade, o estabelecimento do príncipio de que máquinas paralelas radicalizam a dependência entre algoritmo, computador e problema terá profundas conseqüencias no sentido e objetivo da análise de algoritmos, aumentando-se em muito o grau de dificuldade envolvido no estudo e escolha de um algoritmo. Uma das possíveis conseqüências poderia ser a necessidade de pré-avaliar as características de uma instância de um problema e, com base nesses dados, realizar a escolha do algoritmo — e mesmo da máquina — adequado à tarefa. A desejável automatização deste processo de decisão também é, por si só, um potencial campo de pesquisa.

**Verificação da Relação Espaço × Tempo × Comunicação**: os algoritmos para busca em árvores de jogo parecem sugerir que a clássica troca entre espaço ocupado e tempo de execução deverá dar lugar a uma tripla relação, envolvendo também a quantidade de comunicação. Contudo, conforme foi alertado nas seções anteriores deste capítulo, a demonstração matemática de tal fato envolve, necessariamente, uma melhor definição do termo *comunicação*, relacionando-o com a topologia necessária ao algoritmo e com os problemas de sincronização. Esta definição, se feita hoje, talvez fosse prematura, pela falta de modelos mais concretos de máquinas paralelas, e também pela inexperiência e incipiência em sua utilização.
Contudo, é evidente que a investigação teórica da relação é fundamental, de forma a possibilitar a comparação entre algoritmos que apresentem diferenças marcantes entre os três fatores. É provável que, inclusive, modelos muito simplificados do fenômeno de comunicação entre processadores sejam inadequados, e que, de alguma forma, as características arquiteturais do *hardware* tenham de ser incorporadas ao cálculo.

**Desenvolvimento dos Métodos de Análise de Algoritmos Paralelos**: durante os 12 meses de estudo e pesquisa gastos na elaboração desta dissertação, ficou patente a precariedade dos métodos tradicionais de análise frente às necessidades de comparação de algoritmos paralelos. É fundamental a busca de modelos analíticos que incluam as características típicas de arquitetura com múltiplos processadores, a fim de permitir a análise baseada em referências concretas. Um bom exemplo de tentativa nesta direção é o trabalho de Edsger Dijkstra, exposto sucintamente em [Dij75], que, mais do que apresentar soluções viáveis, expõe de forma brilhante as dificuldades inerentes ao problema, e a complexidade de seu tratamento matemático.
Adentrando o terreno da pura especulação, talvez seja o início de uma nova era da análise de algoritmos, com utilização crescente de ferramental matemático mais sofisticado, com o emprego, por exemplo, de conceitos e técnicas de Topologia, de Teoria de Grupos e de Lógica Matemática.

**Adequação da Paralelização como Método de Desenvolvimento de Algoritmos Paralelos**: é importante observar que para o problema de busca em árvores de jogo, todos os algoritmos paralelos propostos são paralelizações de algoritmos seqüenciais. A pergunta natural que surge é se existe algoritmo paralelo não baseado em idéias seqüenciais. Em diversos outros problemas, soluções paralelas originais foram encontradas, muitas vezes com eficiência bastante superior às paralelizações. Contudo, não há motivos para crer que o estudo das versões seqüenciais e sua adaptação para múltiplos processadores seja, *a priori*, inadequado,

justificando-se assim uma investigação mais profunda dos métodos e idéias de paralelização, conforme foi feito neste trabalho.
Vale ainda ressaltar a necessidade de maior pesquisa sobre tais métodos, visando uma melhor compreensão do processo, passo fundamental para estabelecimento de uma metodologia mais definida para o desenvolvimento de algoritmos para computadores com múltiplos processadores.

Os problemas acima enumerados são, na opinião do autor, os mais importantes a serem futuramente investigados. É claro que a própria dissertação suscita, em diversos trechos, outras linhas de pesquisa interessantes, de natureza mais técnica, e facilmente identificáveis por uma leitura mais atenta e criteriosa.

## 7.7 Observações Finais

Como fecho a esta dissertação, será feito um resumo das principais conclusões obtidas, seguido de alguns comentários sobre o processo e a metodologia de pesquisa empregados.

### Resumo das Conclusões

- O problema de busca em árvores de jogo é intrinsecamente exponencial na altura da árvore, apresentando uma variação da ordem de raiz quadrada entre o melhor e o pior caso seqüencial.

- Os principais algoritmos seqüenciais são, em ordem crescente de dominância, o Bound, o Alpha-Beta e o SSS*, sendo que este último necessita de muito mais memória que os anteriores.

- Os algoritmos paralelos para busca em árvores de jogo podem ser vistos como paralelizações de algoritmos seriais, baseados em dois métodos distintos e complementares: redução de janelas e priorização do trabalho obrigatório.

- O método de redução de janelas, no qual se baseiam os algoritmos Tree-Splitting, Aspiration Search e SSS*-I, tem um poder limitado de aumentar a eficiência, e pode ser utilizado independentemente das características do jogo, embora necessite de uma boa rede de comunicação entre os processadores.

- O método de priorização do trabalho obrigatório, empregado no PV-Splitting, MWF-Tree, MWF, Parallel SCOUT, KNM, SSS*-II e Pool-SSS* (este, original deste trabalho), permite ganhos substanciais na eficiência, desde que seja possível realizar algum tipo satisfatório de pré-ordenação dos movimentos do jogo analisado.

- Os algoritmos MWF, KNM e Pool-SSS* devem atingir um alto grau de eficiência, às custas de grande dispêndio de memória e de um sistema eficiente de gerenciamento da memória compartilhada.

- A paralelização do algoritmo Alpha-Beta é dificultada pela forte componente serial do mesmo.

- Os métodos de análise de algoritmos paralelos necessitam ser revistos e ampliados, a fim de se obterem conclusões úteis dos seus resultados.

Na opinião do autor, algumas idéias foram fundamentais para o bom andamento da pesquisa. Cite-se, por exemplo, o formalismo de descrição dos algoritmos, desenvolvido especialmente para o trabalho, e que teve o mérito de apresentar claramente as semelhanças e diferenças entre os algoritmos. Como exemplo de sua utilidade, registre-se que uma dificuldade inicial séria foi o estabelecimento de relações entre os algoritmos Alpha-Beta e SSS*, superada em uma primeira fase com o conceito de árvore solução (que é mais elucidativo que o de valor minimax de uma árvore de jogo), e depois apresentado na forma — mais simplificada — de descrição por mensagens.

O formalismo foi ainda mais útil na descrição dos algoritmos paralelos, pois estes eram descritos, nos artigos originais, em linguagens absolutamente díspares; o formalismo adotado possibilitou uma muito melhor compreensão de suas estruturas, e garantiu também que a implementação não fosse afetada por problemas específicos de linguagens paralelas.

Outro ponto considerado fundamental foi a determinação dos dois métodos básicos de paralelização, que se constituiu no pilar essencial para o entendimento e análise dos algoritmos. De certa forma, o autor acredita que tal determinação é elucidativa do âmago do problema, no sentido de esclarecer as semelhanças e diferenças dos algoritmos, assim como ela estabelece uma base segura sobre a qual podem ser feitas previsões de desempenho. É sempre oportuna a descoberta das características básicas de um problema, e o autor crê que, no caso de busca em árvores de jogo, a chave para a solução está precisamente na identificação desses dois métodos básicos de paralelização.

Finalmente, devem ser registradas as dificuldades encontradas na programação e, principalmente, na depuração de algoritmos paralelos. É necessária uma estrita disciplina de programação, pois os erros, em máquinas ou simuladores paralelos, são mais sutis e inesperados do que nos computadores convencionais. É razoável esperar-se que a implementação de algoritmos paralelos requeira uma evolução significativa nos métodos e ferramentas de desenvolvimento, programação e depuração.

Esta dissertação de mestrado reflete, de acordo com os conhecimentos do autor, o estado da arte do problema de algoritmos paralelos para busca em árvores de jogo. Crê, ainda, que este estudo, além de agrupar e comparar diversas fontes e conceitos, estende significativamente o conhecimento sobre a *natureza* das soluções para o problema, bem como contribui para o estudo de conceitos e métodos genéricos de desenvolvimento de algoritmos para máquinas com múltiplos processadores.

# Apêndice A

# Metodogia Experimental

Este apêndice tem por objetivo descrever os aspectos principais da metologia experimental adotada na obtenção dos resultados descritos nas tabelas 6.6 e 6.7 do capítulo 6.

O trabalho de simulação consistiu na programação e teste dos seguintes algoritmos:

- Alpha-Beta (seqüencial)
- Tree (Tree-Splitting **sem** Update)
- TreeUp (Tree-Splitting **com** Update)
- MWF-Tree

O algoritmo Alpha-Beta foi programado considerando-se o formalismo de descrição por mensagens, conforme a figura 3.6, e executado normalmente. Os três algoritmos paralelos tiveram como referência de programação também a sua descrição por mensagens, de acordo com as figuras 5.5, 5.7 e 5.10, e foram executados com o auxílio de um simulador de paralelismo desenvolvido simultaneamente com esta pesquisa, denominado *Máquina WORM* (em conjunto com o prof. Marcos D. Gubitoso, MAC/IME/USP).

A simulação consistiu na execução dos 4 algoritmos sobre 30 árvores de jogo geradas artificialmente, sendo que cada algoritmo paralelo foi testado para 7 configurações diferentes da árvore de processadores, totalizando assim o exame de 660 árvores de jogo, em um tempo de processamento de cerca de 24 horas em microcomputadores da linha IBM/PC/XT, a 4,7MHz.

Os programas foram escritos em linguagem *C*, e compilados com *C Compiler 5.0* da Microsoft, inibindo-se a otimização automática a fim de dar maior confiabilidade aos resultados. Os fontes do programa não foram anexados à dissertação devido ao seu tamanho — cerca de 60 páginas — e encontram-se disponíveis para consulta (ou mesmo cópia) com o autor.

## A.1 Geração das Árvores

Em virtude da limitação de memória de trabalho dos microcomputadores utilizados, limitou-se a simulação a 3 tamanhos de árvores de jogo uniformes, listados a seguir (onde $d$ é o grau e $h$ é a altura):

| | | | | | |
|---|---|---|---|---|---|
| Comprido | : | $d = 4$ | $h = 8$ | nós terminais= | 65536 |
| Largo | : | $d = 7$ | $h = 6$ | nós terminais= | 117649 |
| Grande | : | $d = 16$ | $h = 4$ | nós terminais= | 65536 |

Para garantir maior compactação, os valores dos nós terminais foram atribuídos no intervalo discreto $[-127, +127]$, de forma que pudessem ser armazenados em um único *byte*.

De cada tamanho foram construídas 5 árvores diferentes, com modelo de distribuição aleatória: os valores dos nós terminais foram gerados aleatoriamente, da esquerda para a direita, através da função da biblioteca do *C*:

```
int rand (void);
```

e com semente, para a primeira árvore, de valor pré-fixado 1, através da função:

```
void srand (unsigned seed);
```

É claro que a função `rand ()` é somente pseudo-aleatória; no entanto, nenhum efeito ou característica indesejável, proveniente de sua periodicidade, foi detectado. Pelo contrário, foi verificado experimentalmente o fenômeno, mencionado na subseção 4.5.4, de convergência do valor minimax da raiz para um valor dependente do grau, da altura e do intervalo considerados: os valores efetivos das árvores diferiam no máximo em 3 unidades do valor teórico previsto, o que é uma excelente aproximação para um intervalo discreto de 256 valores diferentes.

Dessa forma, foram obtidas 15 árvores de jogo cujos nós terminais obedeciam ao modelo aleatório. Para a geração das árvores do modelo de distribuição fortemente ordenada, foram re-utilizadas as árvores do modelo aleatório, submetendo-as ao seguinte método de ordenação:

- para cada nó, gera-se um número aleatório entre 0 e 1;
- caso o valor gerado esteja entre 0 e 0,85, determina-se o filho mais à esquerda que tenha o mesmo mérito de seu pai, e troca-se este filho com o primeiro;
- os filhos são recursivamente percorridos por este método, da esquerda para a direita: sempre que a busca chega a um nó terminal, seu valor é gerado;
- os valores gerados são consecutivamente atribuídos aos nós terminais, da esquerda para a direita.

Observa-se que este método produz uma variação do modelo de distribuição fortemente ordenada, conforme definido na subseção 4.1.4, e de espírito bastante semelhante: para os nós de qualquer nível, e na maioria das vezes (85%), o melhor nó é o filho mais à esquerda.

Devido ao longo tempo necessário à sua produção, as árvores foram geradas uma única vez, e armazenadas em arquivos em disco, totalizando uma massa de dados de cerca de 2,5 *Mbytes*.

É importante ressaltar que, mesmo considerando-se a baixa potência do microcomputador empregado na simulação, as árvores utilizadas encontram-se na mesma faixa de tamanho das examinadas por simulações realizadas por outros autores, conforme descrições na seção 6.3 desta dissertação.

## A.2 A Máquina WORM

Para a execução dos algoritmos em paralelo, foi utilizado o simulador de paralelismo denominado *Máquina WORM* [1], desenvolvido e programado pelo autor e pelo prof. M.D. Gubitoso.

Alguns príncipios básicos nortearam o desenvolvimento do WORM:

- deveria rodar em um microcomputador da linha IBM/PC/XT, sem qualquer alteração de *hardware*;
- deveria ser acessível através de programas escritos em *C*, utilizando compiladores comerciais, sem restrições ao uso de funções de biblioteca;
- deveria ser flexível o suficiente para ser adaptado a qualquer arquitetura paralela, assíncrona ou sistólica;
- deveria ser simples de programar e depurar, devido ao pouco tempo disponível de desenvolvimento.

Tendo em vista esses objetivos, foi elaborado um sistema multitarefa simples, com capacidade para até 50 processos, cujo chaveamento é feito pela interrupção do relógio ou por um comando específico (modelos assíncrono e sistólico, respectivamente). Os processos são fornecidos como rotinas escritas em *C*, cujos endereços iniciais são passados em um vetor, que é, por sua vez, parâmetro da rotina que gerencia todo o chaveamento, do início ao fim da execução.

As rotinas podem ser diferentes ou iguais (estas, desde que o código seja reentrante) e o seu término automaticamente as exclui do chaveamento. O sistema multitarefa pára quando todas as rotinas acabam, ou através de um comando especial que, executado por qualquer um dos processos, cessa os demais.

Um meio simples é utilizado para evitar colisões entre processos quando do acesso a memórias compartilhadas ou a funções que usam variáveis globais: nestas situações, é possível a um processo suspender a execução dos demais, e depois, proceder a retomada do chaveamento. A fim de garantir uma melhor distribuição do tempo entre as tarefas, registra-se o número de interrupções do relógio chamadas durante a suspensão do chaveamento; na retomada, o processo seguinte ao que suspendeu é automaticamente chamado, ficando o primeiro tantos ciclos inativo quantos foram gastos durante a suspensão.

Este sistema de débitos de tempo não garante perfeito sincronismo das tarefas; contudo, foi o meio mais simples encontrado para garantir que, *em média*, todos os processos gastassem aproximadamente a mesma quantidade de tempo. Esta última afirmação é particularmente válida para simulações longas como as que foram feitas, nas quais chegou-se, em certos casos, ao extremo de gastar 1 hora para processar uma única árvore de jogo.

A máquina WORM possui, além destas, outras interessantes propriedades, que lhe garantem maior flexibilidade e adaptabilidade. São estas aqui omitidas por não terem sido necessárias na simulação dos algoritmos paralelos para árvores de jogo; oportunamente, deverá ser publicado um relatório técnico dedicado integralmente ao assunto.

[1] Do original, em português, *Máquina LESMA*.

## A.3 A Árvore de Processadores

A máquina WORM, como foi visto, faz apenas o trabalho de chavear processos definidos como funções escritas em linguagem *C*. Assim, para se conseguir executar os algoritmos Tree-Splitting (com e sem Update) e MWF-Tree, é necessário a elaboração de uma estrutura que simule uma árvore de processadores, com seus respectivos canais de comunicação.

Os algoritmos mencionados acima foram implementados utilizando-se uma estrutura de dados que emula o sistema de gerenciamento de comunicações. A figura A.1 mostra, esquematicamente, a constituição de cada processador, e a figura A.2 mostra a correspondente declaração da estrutura de cada processador em *C*.

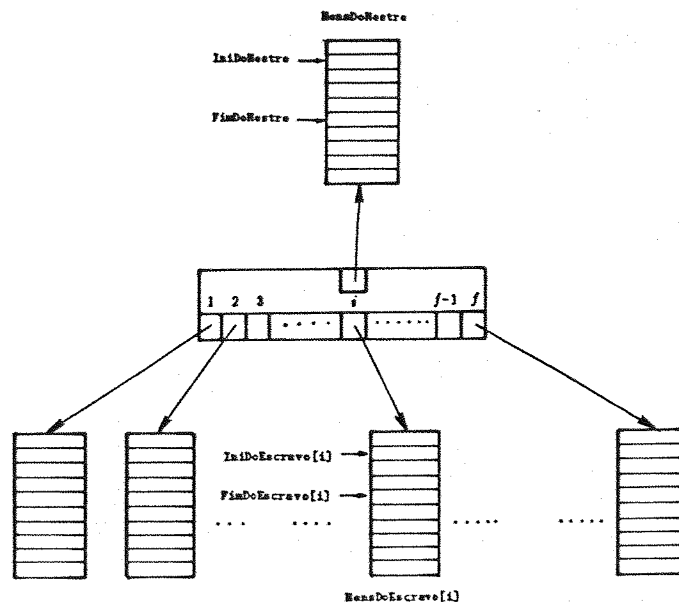



Figura A.1: Diagrama esquemático do processador.

Para acesso a essa estrutura de dados, foram definidas quatro rotinas: duas rotinas para envio de mensagens (para o processador mestre e para os escravos) e duas para consulta ao início da fila de mensagens (uma para as provenientes do mestre e outra para as provenientes de cada um dos escravos). As rotinas são reentrantes, e utilizam o efeito suspensivo da máquina WORM quando da utilização de dados comuns.

Complementarmente foram escritas rotinas que implementam os algoritmos paralelos, também reentrantes, e para cada processo da máquina WORM foi feita a associação bijetora entre o número do processo e o de uma estrutura do tipo `Processador` do vetor `Arvore[]`. Dessa forma, cada processo funciona como um "processador" independente, que fica continuamente examinando suas filas de mensagens: quando alguma é detectada, é retirada da fila correspondente e interpretada conforme expresso na descrição por mensagens do algoritmo, gerando, se necessário, novas mensagens para outros processadores.

A fim de possibilitar um estudo mais abrangente, foram feitas simulações em árvores de processadores de diversos graus ($f$) e alturas ($n$), a saber:

```
/***********************************************************************/
/*          DEFINICAO DA ESTRUTURA DA ARVORE DE PROCESSADORES          */
/***********************************************************************/

#define NPROC .....                 /* Numero total de processadores.    */
#define GRAUF .....                 /* Grau da arvore de processadores.  */

typedef struct { ..... } Mens ;     /* Structure que define o formato das */
                                    /*   mensagens trocadas entre os     */
                                    /*   processadores.                  */

typedef Mens FilaMens [10];         /* FilaMens e' uma fila de 10 mens.  */

typedef struct
   { int Proc;                      /* Numero do processador.            */
     int Mestre;                    /* Numero do processador mestre.     */
     int Escravos [GRAUF];          /* Numero de cada processador escra- */
                                    /*   vo.                             */

     FilaMens MensDoMestre;         /* Fila de mensagens enviadas pelo   */
                                    /*   mestre do processador.          */
     int IniDoMestre;               /* Inicio da fila MensDoMestre.      */
     int FimDoMestre;               /* Fim da fila MensDoMestre.         */

     FilaMens MensDoEscravo [GRAUF]; /* Fila de mensages enviadas por    */
                                    /*   cada escravo deste processador. */
     int IniDoEscravo [GRAUF];      /* Inicio da fila MensDoEscravo.     */
     int FimDoEscravo [GRAUF];      /* Fim da fila MensDoEscravo.        */

   } Processador;                   /* Tipo que define um processador.   */

Processador Arvore [NPROC];         /* Declaracao de todos os processa-  */
                                    /*   dores, armazenados na forma de  */
                                    /*   um vetor.                       */

/***********************************************************************/
```

Figura A.2: Listagem em *C* da definição da estrutura de dados que emula uma árvore de processadores.

| | | | | |
|---|---|---|---|---|
| $f = 2$ | $n = 1$ | total: | 3 | processadores |
| $f = 3$ | $n = 1$ | total: | 4 | processadores |
| $f = 5$ | $n = 1$ | total: | 6 | processadores |
| $f = 10$ | $n = 1$ | total: | 11 | processadores |
| $f = 3$ | $n = 2$ | total: | 13 | processadores |
| $f = 2$ | $n = 3$ | total: | 15 | processadores |
| $f = 3$ | $n = 3$ | total: | 40 | processadores |

## A.4 A Confecção das Tabelas

As tabelas 6.6 e 6.7 foram geradas submetendo-se as árvores de jogo, de 3 tamanhos e 2 modelos diferentes, a cada um dos 3 algoritmos paralelos, com as 7 configurações da árvore de processadores. Para a obtenção dos dados reportados nas tabelas, determinou-se a *média aritmética* do número de nós terminais examinados nas 5 árvores de jogo.

Os gráficos de eficiência foram facilmente obtidos a partir destes dados, considerando-se que se dispunha dos resultados médios das diversas árvores no caso de busca seqüencial Alpha-Beta.

Reporte-se ainda que 5 árvores é um número baixo, e que o desvio padrão encontrado foi razoavelmente grande. Todavia, o considerável tempo de processamento gasto pelos algoritmos inviabilizaram a simulação com maior número de árvores, embora o autor não tenha dúvidas quanto à adequação dos resultados.

# Bibliografia

[ABD80] AKL, S. G.; BARNARD, D. T.; DORAN, R. J. Simulation and analysis in deriving time and storage requirements for a parallel alpha-beta algorithm. In: *Proceedings 1980 International Conference on Parallel Processing.* S.l., Dept. of Electrical and Computer Engineering/Ohio State University & IEEE Computer Society. Aug. 26–29, 1980. p. 231–234.

[ABD82] —— Design, analysis and implementation of a parallel tree search algorithm. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 4(2):192–203, Mar. 1982.

[AD83] AKL, S. G.& DORAN, R. J. A comparison of parallel implementations of the alpha-beta and SCOUT tree search algorithms using the game of checkers. In: BRAMER, M.A., ed., *Computer Game-Playing: Theory and Practice.* Chichester, Ellis Horwood. 1983. p. 290–303.

[Bau78a] BAUDET, Gérard M. *The Design and Analysis of Algorithms for Asynchronous Multiprocessors.* Pittsburgh, Department of Computer Science/Carnegie-Mellon University. Apr. 1978. (PhD. thesis).

[Bau78b] —— On the branching factor of the alpha-beta pruning algorithm. In: *Artificial Intelligence*, 10(2):173–199, 1978.

[Bru63] BRUDNO, A. L. Bounds and valuations for shortening the scanning of variations. In: *Problemy Kibernet*, (10):141–150, 1963. (Original em russo). ⋆. [1]

[CKS81] CHANDRA, Ashok K.; KOZEN, Dexter C.; STOCKMEYER, Larry J. Alternation. *Journal of the ACM*, 28(1):114–133, Jan. 1981.

[CM83] CAMPBELL, Murray S. & MARSLAND, T. A. A comparison of minimax tree search algorithms. In: *Artificial Intelligence*, 20(4):347–367, 1983.

[Dij75] DIJKSTRA, Edsger W. Guarded commands, nondeterminacy and formal derivation of programs. In: *Communications of the ACM*, 18(8):453–457, Aug. 1975.

[FF80] FISHBURN, J. P. & FINKEL, R. A. *Parallel Alpha Beta Search on Arachne.* Madison, Computer Sciences Department/University of Wiscosin-Madison. July 1980. (Computer Sciences Technical Report 394).

[1] O símbolo ⋆ indica que o autor desta dissertação não teve acesso direto à referência.

[FF82] FINKEL, R. A. & FISHBURN, J. P. Parallelism in alpha beta search. In: *Artificial Intelligence*, 19(1):89–106, Sep. 1982.

[FF83] —— Improved speedup bounds for parallel alpha-beta search. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 5(1):89–92, Jan. 1983.

[FGG73] FULLER, S. H.; GASCHING, J. G.; GILLOGLY, J. J. *Analysis of the Alpha-Beta Pruning Algorithm.* Pittsburgh, Dept. of Computer Science/Carnegie-Mellon University. July 1973. (Technical Report). ⋆.

[Fly66] FLYNN, M. J. Very high-speed computing systems. In: *Proceedings of IEEE*, 54(12):1901–1909, Dec. 1966. ⋆.

[HE61] HART, T. P. & EDWARDS, D. J. *The Tree Prune (TP) Algorithm.* Cambridge, Mass. , Massachusetts Institute of Technology. Dec. 1961. 6p. (Technical Report, MIT Artificial Intelligence Project Memo #30). ⋆.

[KK83] KUMAR, V. & KANAL, L. N. A general branch-and-bound formulation for understanding and synthetisizing AND/OR tree search procedures. In: *Artificial Intelligence*, 21(1):179–198, 1983.

[KK84] —— Parallel branch-and-bound formulations for AND/OR tree search. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 6(6):768–778, Nov. 1984.

[KM75] KNUTH, D. E. & MOORE, R. W. An analysis of alpha-beta pruning. In: *Artificial Intelligence*, 6(4):293–326, 1975.

[Lin83] LINDSTROM, G. *The Key-Node Method: A Highly-Parallel Alpha-Beta Algorithm.* Salt Lake City, Dept. Comput. Sci./University of Utah. Mar. 1983. (Tech. Rep. UUCS 83-101 101).

[MC82] MARSLAND, T. A. & CAMPBELL, M. Parallel search of strongly ordered game trees. In: *Computing Surveys*, 14(4):533–551, Dec. 1982.

[MV87] MONIEN, B. & VORNBERGER, O. Parallel Processing of Combinatorial Search Trees. In: *Lecture Notes in Computer Science*, (269):60–69, 1987.

[New77] NEWBORN, M. The efficiency of the alpha-beta search on trees with branch-dependent terminal node scores. In: *Artificial Intelligence*, 8(2):137–153, 1977.

[Nil82] NILSSON, Nils J. *Principles of Artificial Intelligence.* Palo Alto, Tioga, 1982. 475 p.

[NSS58] NEWELL, A.; SHAW, J. C.; SIMON, H. A. Chess-playing programs and the problem of complexity. In: *IBM Journal of Research and Development*, (2):320–335, 1958.

[Pea80] PEARL, J. Asymptotic properties of minimax trees and game-searching procedures. In: *Artificial Intelligence*, 14(2):113–138, Sep. 1980.

[Pea82] —— The solution for the branching factor of the alpha-beta pruning algorithm and its optimality. In: *Communications of the ACM*, 25(8):559–564 , Aug. 1982.

[Qui87] QUINN, Michael J. *Designing Efficient Algorithms for Parallel Computers.* S.l.p., McGraw-Hill, 1987. 367 p.

[RP83] ROIZEN, I. & PEARL, J. A minimax algorithm better than alpha-beta? Yes and no. In: *Artificial Intelligence*, 21(1/2):199–220, Mar. 1983.

[Sam59] SAMUEL, A. L. Some studies in machine learning using the game of checkers. In: *IBM Journal of Research and Development*, (3):211–229, 1959.

[SB68] SLAGLE, J. R. & BURSKY, P. Experiments with a multipurpose, theorem-proving heuristic program. In: *Journal of the ACM*, 15(2):85–99, 1968.

[SC79] STOCKMEYER, Larry J. & CHANDRA, Ashok K. Provably difficult combinatorial games. In: *SIAM Journal of Computing*, 8(2):151–173, May 1979.

[SD69] SLAGLE, J. R. & DIXON, J. K. Experiments with some programs that search game trees. In: *Journal of the ACM*, 16(2):189–207, 1969.

[SK83] STOCKMAN, G. C. & KANAL, L. N. Problem reduction representation for the linguistic analysis of waveforms. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 5(3):287–298, May 1983.

[SL71] SLAGLE, J. R. & LEE, R. C. T. Application of game tree searching techniques of sequential pattern recognition. In: *Communications of the ACM*, 14(2):103–110, Feb. 1971.

[Sto79] STOCKMAN, G. C. A minimax algorithm better than alpha-beta? In: *Artificial Intelligence*, 12(2):179–196, Aug. 1979.

[Tar83] TARSI, M. Optimal searching of some game trees. In: *Journal of the ACM*, 30(3):389–396, July 1983.

# Lista de Figuras


2.1 Árvore AND/OR. . . . . 7
2.2 Árvore de jogo parcial com valores nos nós terminais. . . . . 8
2.3 Árvore de jogo. . . . . 10
2.4 a)Árvore AND/OR com posição terminal OR; b)Árvore AND/OR equivalente. . . . . 10
2.5 Algumas AND-árvores solução. . . . . 10
2.6 Algumas OR-árvores solução. . . . . 12
2.7 AND-árvore solução $T$ e uma refutação $\overline{T}^t$ de $T$. . . . . 15
2.8 Exemplo da notação utilizada. . . . . 16
2.9 Árvore solução $V$ e refutação $\overline{T}_k^{j+1}$. . . . . 16
2.10 Árvores solução $T$ e $T'$. . . . . 17
2.11 Árvore $T$ e as refutações necessárias . . . . . 18
2.12 Cálculo dos valores $H$ e $I$ de dois nós. . . . . 20

3.1 Descrição por mensagens de um algoritmo para percurso em pré-ordem de uma árvore. 23
3.2 Algoritmo Minimax. . . . . 24
3.3 Árvore de jogo examinada pelo algoritmo Minimax. . . . . 25
3.4 Algoritmo Bound. . . . . 26
3.5 Árvore de jogo examinada pelo algoritmo Bound. . . . . 28
3.6 Algoritmo Alpha-Beta. . . . . 29
3.7 Árvore de jogo examinada pelo algoritmo Alpha-Beta. . . . . 31
3.8 Corte raso (a) e corte profundo (b). . . . . 32
3.9 Algoritmo Bound em situações de corte raso e profundo. . . . . 33
3.10 Algoritmo Alpha-Beta em situações de corte raso e profundo. . . . . 33
3.11 Algoritmo SSS*. . . . . 35
3.12 Árvore de jogo examinada pelo algoritmo SSS*. . . . . 37
3.13 Exemplos de árvores cujo mérito é $\pm\infty$. . . . . 37
3.14 Equivalência básica de um teste sobre uma árvore de jogo $G$ com uma árvore de jogo $G'$ do tipo $m(G')=\pm\infty$. . . . . 38
3.15 Algoritmo $\text{TEST}_{\text{AND}}$ . . . . . 38
3.16 Algoritmo SCOUT. . . . . 40
3.17 Árvore de jogo examinada pelo algoritmo SCOUT. . . . . 41

4.1 Contra-exemplo da dominação de SSS* por Alpha-Beta. . . . . 50
4.2 Contra-exemplo da dominação de SCOUT por Alpha-Beta. . . . . 50
4.3 Contra-exemplo da dominação de Alpha-Beta por SCOUT. . . . . 51

4.4 Probabilidade de vitória do 1º jogador em árvores maniqueístas binárias de distribuição $P_0$ (fonte: [Pea80], p. 116). . . . 52
4.5 Algoritmo Falphabeta . . . . 59

5.1 Arquitetura de processadores em árvore de grau $f$=3 e altura $n$=2. . . . 61
5.2 Arquitetura de processadores com memória compartilhada. . . . 61
5.3 Árvore de processadores sobreposta a uma árvore de jogo. . . . 64
5.4 Exemplo de melhor caso de busca em uma árvore de jogo por um algoritmo direcional. 65
5.5 Configuração inicial do algoritmo Tree-Splitting. . . . 66
5.6 Algoritmo Tree-Splitting. . . . 66
5.7 Interrupção Update de atualização. . . . 67
5.8 Movimento da árvore de processadores ao longo da árvore de jogo no algoritmo PV-Splitting. . . . 68
5.9 Algoritmo PV-Splitting. . . . 69
5.10 Fases 1 e 2 do algoritmo MWF-Tree. . . . 71
5.11 Algoritmo MWF-Tree. . . . 72
5.12 a)Partição simples; b)Partição recursiva. . . . 73
5.13 Algoritmo Aspiration Search. . . . 74
5.14 Fases 1 e 2 do algoritmo MWF, utilizando-se 3 processadores. . . . 75
5.15 Algoritmo MWF. . . . 77
5.16 Atribuição de prioridades aos nós de uma árvore de jogo. . . . 78
5.17 Algoritmo Parallel SCOUT. . . . 79
5.18 Árvores mwf e mwf-deslocada. . . . 80
5.19 Algoritmo KNM. . . . 82
5.20 Exemplo de condições de utilização de valores temporários. . . . 84

6.1 Eficiência dos algoritmos Tree e TreeUp na busca sobre árvores de tamanho Comprido. 99
6.2 Eficiência dos algoritmos Tree e TreeUp na busca sobre árvores de tamanho Largo. . . 100
6.3 Eficiência dos algoritmos Tree e TreeUp na busca sobre árvores de tamanho Grande. . 100
6.4 Comparação entre as disposições iniciais dos processadores nos algoritmos Tree e MWF-Tree. . . . 102
6.5 Eficiência dos algoritmos Tree e MWF-Tree na busca sobre árvores de tamanho Comprido com distribuição aleatória. . . . 103
6.6 Eficiência dos algoritmos Tree e MWF-Tree na busca sobre árvores de tamanho Comprido com distribuição fortemente ordenada. . . . 103
6.7 Eficiência dos algoritmos Tree e MWF-Tree na busca sobre árvores de tamanho Largo com distribuição aleatória. . . . 104
6.8 Eficiência dos algoritmos Tree e MWF-Tree na busca sobre árvores de tamanho Largo com distribuição fortemente ordenada. . . . 104
6.9 Eficiência dos algoritmos Tree e MWF-Tree na busca sobre árvores de tamanho Grande com distribuição aleatória. . . . 105
6.10 Eficiência dos algoritmos Tree e MWF-Tree na busca sobre árvores de tamanho Grande com distribuição fortemente ordenada. . . . 105

7.1 Situação típica de chamada da interrupção Update pelo algoritmo Tree-Splitting. . . . 110

A.1 Diagrama esquemático do processador. . . . . . . . . . . . . . . . . . . . . . . . . . 121
A.2 Listagem em *C* da definição da estrutura de dados que emula uma árvore de processadores. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 122

# Lista de Tabelas


4.1 Simulação dos algoritmos ótimos sobre dois tipos de árvores aleatórias de grau $d = 24$ (fonte: [CM83], p. 10 e 12). . . . . . . . . . 57

6.1 Custo e aceleração teóricos para o algoritmo Aspiration Search. . . . . . . . 90
6.2 Resultados experimentais para o algoritmo Tree-Splitting (fonte: [FF82], p. 96 e 97). . 93
6.3 Resultados comparativos entre os algoritmos Tree-Splitting e PV-Splitting (fonte: [MC82], p. 550). . . . . . . . . . 94
6.4 Resultados experimentais para o algoritmo KNM (fonte: [Lin83], p. 15). . . . . . . . 95
6.5 Resultados experimentais para o algoritmo SSS*-II (fonte: [KK84], p. 776). . . . . . . . 96
6.6 Número total de nós terminais examinados pelos algoritmos Tree e TreeUp. . . . . . . 98
6.7 Número total de nós terminais examinados pelo algoritmo Tree e MWF-Tree. . . . . . 102