

Gutierrez Soares Caexêta

*VisionDraughts – Um Sistema de  
Aprendizagem de Jogos de Damas Baseado  
em Redes Neurais, Diferenças Temporais,  
Algoritmos Eficientes de Busca em Árvores  
e Informações Perfeitas Contidas em  
Bases de Dados*

Uberlândia - MG

2008

UNIVERSIDADE FEDERAL DE UBERLÂNDIA

Autor: **Gutierrez Soares Caexêta**

Titulo: ***VisionDraughts*** – Um Sistema de Aprendizagem de Jogos de Damas Baseado em Redes Neurais, Diferenças Temporais, Algoritmos Eficientes de Busca em Árvores e Informações Perfeitas Contidas em Bases de Dados

Faculdade: **Faculdade de Computação**

Copyright 2008

Fica garantido à Universidade Federal de Uberlândia o direito de circulação e impressão deste material para fins não comerciais, bem como o direito de distribuição por solicitação de qualquer pessoa ou instituição

Gutierrez Soares Caexêta

*VisionDraughts – Um Sistema de  
Aprendizagem de Jogos de Damas Baseado  
em Redes Neurais, Diferenças Temporais,  
Algoritmos Eficientes de Busca em Árvores  
e Informações Perfeitas Contidas em  
Bases de Dados*

Dissertação apresentada à Coordenação do  
Mestrado em Ciência da Computação da  
Universidade Federal de Uberlândia para a  
obtenção do título de Mestre em Ciência da  
Computação.

Orientadora:  
Profa. Dra. Rita Maria da Silva Julia

MESTRADO EM CIÊNCIA DA COMPUTAÇÃO  
FACULDADE DE COMPUTAÇÃO  
UNIVERSIDADE FEDERAL DE UBERLÂNDIA

Uberlândia – MG

Julho / 2008

Dados Internacionais de Catalogação na Publicação (CIP)

---

C128v Caexêta, Gutierrez Soares, 1981-

VisionDraughts – um sistema de aprendizagem de jogos de damas baseado em redes neurais, diferenças temporais, algoritmos eficientes de busca em árvores e informações perfeitas contidas em bases de dados / Gutierrez Soares Caexêta. - 2008.  
140 f. : il.

Orientador: Rita Maria da Silva Julia.

Dissertação (mestrado) – Universidade Federal de Uberlândia, Programa de Pós-Graduação em Ciência da Computação.

Inclui bibliografia.

1. Aprendizado do computador - Teses. 2. Redes neurais (Computação) - Teses. 3. Teoria dos jogos - Teses. I. Julia, Rita Maria da Silva. II. Universidade Federal de Uberlândia. Programa de Pós-Graduação em Ciência da Computação. III. Título.

CDU: 681.3:007.52

---

Elaborado pelo Sistema de Bibliotecas da UFU / Setor de Catalogação e Classificação

Dissertação apresentada ao Programa de Pós-Graduação da Faculdade de Ciência da Computação da Universidade Federal de Uberlândia como requisito para obtenção do grau de Mestre em Ciência da Computação.

---

Profa. Dra. Rita Maria da Silva Julia  
Orientadora

---

Prof. Dr. Guilherme Bittencourt  
Universidade Federal de Santa Catarina/SC

---

Prof. Dr. Carlos Roberto Lopes  
Universidade Federal de Uberlândia UFU/MG

*Dedico esta dissertação a Deus por me conceder saúde,  
a minha família por me garantir estudo de qualidade  
em um país onde nem todos têm a mesma oportunidade e  
a minha querida namorada pelo incentivo.  
A todos eles pelo amor incondicional!*

# *Agradecimentos*

À minha mãe Eva por participar intensamente de toda a minha vida. Aliás, foi a melhor professora de matemática que já tive!

Ao meu pai José por participar intensamente de toda a minha vida. Aliás, nem lembro quantas manhãs me aprontou para a aula, com direito a café da manhã!

À minha irmã Giselle pelo seu enorme coração e por contribuir para a completude de nossa família!

À minha namorada Camila por ser tão especial. Aliás, com ela eu vislumbro um futuro seguro!

À professora Rita Maria da Silva Julia pelo sucesso com que conduziu seu trabalho de orientação científica, com presteza, eficiência, eficácia e gentileza!

Finalmente, a Deus por tudo!

*“Embora ninguém possa voltar atrás e fazer um novo começo,  
qualquer um pode começar agora e fazer um novo fim.”*

***Ayrton Senna***



# *Resumo*

O objetivo deste trabalho é propor um sistema de aprendizagem de damas, *VisionDraughts*, baseado nos trabalhos de Neto e Julia (*LS-Draughts*) e de Mark Lynch (*NeuroDraughts*). O *NeuroDraughts* é um bom jogador automático de damas que utiliza a técnica de aprendizagem por diferenças temporais para ajustar os pesos de uma rede neural artificial multi-camadas cujo papel é estimar o quanto um estado do tabuleiro do jogo, representado em sua camada de entrada através do mapeamento NET-FEATUREMAP, é favorável ao agente jogador. O conjunto de características do jogo é definido manualmente e a busca pela melhor ação a ser executada, a partir do estado corrente do tabuleiro, é realizada através do algoritmo minimax. O *LS-Draughts* expande o trabalho de Lynch por meio da técnica dos algoritmos genéticos, gerando, automaticamente, um conjunto mínimo e essencial de características do jogo de damas e otimizando, com grande sucesso, o treinamento do agente aprendiz. O *VisionDraughts* acrescenta dois módulos nas arquiteturas anteriores: um módulo de busca eficiente em árvores de jogos baseado no algoritmo alfa-beta, no aprofundamento iterativo e nas tabelas de transposição, que fornece ao agente jogador maior capacidade de analisar jogadas futuras (estados do tabuleiro mais distantes do estado corrente), um módulo para acessar bases de dados de finais de jogos que permite obter informações perfeitas para combinações de oito ou menos peças no tabuleiro. Foram realizados torneios entre os melhores jogadores obtidos por *NeuroDraughts*, *LS-Draughts* e *VisionDraughts*. Os resultados dos torneios, todos vencidos pelo *VisionDraughts*, evidenciam a importância dos dois novos módulos na construção de jogadores automáticos de damas: o tempo de execução para o treinamento do jogador foi drasticamente reduzido e seu desempenho significativamente melhorado. Aliás, o nome *VisionDraughts* foi escolhido, justamente, para destacar a importância da capacidade de analisar jogadas futuras para o sucesso do presente trabalho.

**Palavras-chave:** Aprendizagem Automática; Aprendizagem de Máquina; Aprendizagem Incremental; Aprendizagem por Reforço; Redes Neurais; Aprendizagem por Diferenças Temporais; Teoria dos Jogos; Damas; Busca em Árvores de Jogos; MiniMax; Alfa-Beta; Tabelas de Transposição; Aprofundamento Iterativo; Bases de Dados de Finais de Jogos; Chaves Zobrist; Tabela Hash; Tratamento de Colisões.

# *Abstract*

The objective of this work is to propose a draughts learning system, *VisionDraughts*, based on works of Neto and Julia (*LS-Draughts*) and Mark Lynch (*NeuroDraughts*). The *NeuroDraughts* is a good automatic draughts player which uses temporal difference learning to adjust the weights of an artificial neural network whose role is to estimate how much the board state represented in its input layer by NET-FEATUREMAP is favorable to the player agent. The set of features is manually defined. The search for the best action corresponding to a current board state is performed by minimax algorithm. The *LS-Draughts* expands the *NeuroDraughts*, through the genetic algorithms, generating automatically a set of minimal features which are necessary and essential to a game of draughts and optimizing, successfully, the training of the apprentice player. The *VisionDraughts* adds two modules to the former architectures: an efficient tree-search module with alfa-beta, iterative deepening and transposition table, providing the player agent larger capacity to analyse future moves (board states more distant from the current board) and a module to access endgame databases, allowing to acquire perfect information to positions with less than 8 pieces on the board. Some tournaments were promoted between the best players obtained by *NeuroDraughts*, *LS-Draughts* and *VisionDraughts*. The tournament's results, all won by the *VisionDraughts*, show the importance of the new two modules in the building of good automatic draughts players: the runtime required for training the new player was drastically reduced and its performance was significantly improved. Furthermore, the *VisionDraughts* name was just chosen to emphasize the great importance of analysing future moves in order to the success of this work.

**Keywords:** Automatic Learning; Machine Learning; Incremental Learning; Reinforcement Learning; Neural Network; Temporal Difference Learning; Game Theory; Draughts; Checkers; Minimax; Alfa-Beta; Transposition Table; Iterative Deepening; Endgame Databases; Zobrist Key; Hash Table; Collisions.

# *Sumário*

## Lista de Figuras

## Lista de Tabelas

|          |  |       |
|----------|--|-------|
| <b>1</b> | <b>Introdução</b>  | p. 16 |
| 1.1      | Introdução e Motivação . . . . .                             | p. 16 |
| 1.2      | Estrutura da dissertação . . . . .                           | p. 18 |
| <b>2</b> | <b>Referencial Teórico</b>                                   | p. 19 |
| 2.1      | Estratégias de Busca . . . . .                               | p. 19 |
| 2.1.1    | O Espaço de Estados . . . . .                                | p. 20 |
| 2.1.2    | Busca em Largura . . . . .                                   | p. 22 |
| 2.1.2.1  | Compleitude . . . . .  | p. 24 |
| 2.1.2.2  | Complexidade Temporal . . . . .                              | p. 25 |
| 2.1.2.3  | Complexidade Espacial . . . . .                              | p. 25 |
| 2.1.2.4  | Otimização . . . . .   | p. 26 |
| 2.1.3    | Busca em Profundidade . . . . .                              | p. 26 |
| 2.1.3.1  | Compleitude . . . . .  | p. 27 |
| 2.1.3.2  | Complexidade Temporal . . . . .                              | p. 28 |
| 2.1.3.3  | Complexidade Espacial . . . . .                              | p. 28 |
| 2.1.3.4  | Otimização . . . . .   | p. 29 |
| 2.1.4    | Busca em Profundidade com Aprofundamento Iterativo . . . . . | p. 29 |

|          |  |       |
|----------|--|-------|
| 2.1.5    | Busca em Profundidade com Aprofundamento Iterativo e Tabelas de Transposição . . . . . | p. 31 |
| 2.1.6    | Busca pela Melhor Escolha . . . . .  | p. 31 |
| 2.2      | Redes Neurais . . . . .  | p. 32 |
| <b>3</b> | <b>Estado da Arte</b>  | p. 36 |
| 3.1      | Redes Neurais como Recurso de Aproximação de Funções em Jogos . .                      | p. 36 |
| 3.2      | Aprendizagem por Reforço e Método das Diferenças Temporais . . . .                     | p. 37 |
| 3.2.1    | O sucesso do <i>TD-Gammon</i> para o Jogo de Gamão . . . . .                           | p. 38 |
| 3.2.2    | O sucesso do <i>Chinook</i> para o Jogo de Damas . . . . .                             | p. 39 |
| 3.2.3    | O <i>LS-Draughts</i> e os Algoritmos Genéticos . . . . .                               | p. 40 |
| 3.3      | Computação Evolutiva . . . . .   | p. 41 |
| 3.4      | Representação do Tabuleiro com <i>BitBoards</i> . . . . .                              | p. 41 |
| 3.5      | Busca em Profundidade com Aprofundamento Iterativo e Tabelas de Transposição . . . . . | p. 45 |
| 3.6      | Análise em Retrocesso e Bases de Dados . . . . .                                       | p. 46 |
| 3.6.1    | Damas . . . . .  | p. 50 |
| 3.6.1.1  | Chinook . . . . .  | p. 51 |
| 3.6.1.2  | KingsRow . . . . .   | p. 54 |
| 3.6.1.3  | Cake . . . . .   | p. 55 |
| 3.6.2    | <i>Awari</i> . . . . .   | p. 55 |
| 3.6.3    | Moinho . . . . .   | p. 57 |
| 3.6.4    | Xadrez . . . . .   | p. 58 |
| <b>4</b> | <b><i>VisionDraughts</i> – Um Sistema de Aprendizagem de Damas</b>                     | p. 60 |
| 4.1      | <i>NeuroDraughts</i> : O Jogador de Mark Lynch . . . . .                               | p. 61 |
| 4.1.1    | Representação do Tabuleiro e Mapeamento das Características .                          | p. 63 |
| 4.1.2    | Cálculo da Predição e Escolha da Melhor Ação . . . . .                                 | p. 64 |

|         |  |        |
|---------|--|--------|
| 4.1.3   | Reajuste de Pesos da Rede Neural Multi-Camadas . . . . .   | p. 66  |
| 4.1.4   | Estratégia de Treino por <i>Self-Play com Clonagem</i> . . . . .   | p. 69  |
| 4.2     | Fluxo de Aprendizagem do <i>VisionDraughts</i> . . . . .   | p. 70  |
| 4.3     | O Eficiente Mecanismo de Busca do <i>VisionDraughts</i> . . . . .  | p. 72  |
| 4.3.1   | O Algoritmo Alfa-Beta . . . . .  | p. 72  |
| 4.3.2   | A Tabela de Transposição . . . . .   | p. 82  |
| 4.3.2.1 | Transposição - Mais de uma Ocorrência do Mesmo Estado do Tabuleiro do Jogo . . . . .                               | p. 82  |
| 4.3.2.2 | Técnica de Zobrist - Criação de Chaves Hash para Indexação dos Estados do Tabuleiro do Jogo . . . . .              | p. 83  |
| 4.3.2.3 | Estrutura <i>ENTRY</i> - Dados Armazenados para um Determinado Estado do Tabuleiro do Jogo . . . . .               | p. 89  |
| 4.3.2.4 | Colisões - Conflitos de Endereços para Estados do Tabuleiro do Jogo . . . . .                                      | p. 91  |
| 4.3.2.5 | Estrutura <i>TTABLE</i> - Manipulação de Dados na Tabela de Transposição com Tratamento de Colisões . . . . .      | p. 95  |
| 4.3.3   | Integração entre o Algoritmo Alfa-Beta e a Tabela de Transposição . . . . .  | p. 97  |
| 4.3.3.1 | A Variante Fail-Soft do Algoritmo Alfa-Beta . . . . .  | p. 98  |
| 4.3.3.2 | Armazenar Estados do Tabuleiro na Tabela de Transposição a partir do Algoritmo Fail-Soft Alfa-Beta . . . . .       | p. 102 |
| 4.3.3.3 | Recuperação dos Estados do Tabuleiro da Tabela de Transposição a partir do Algoritmo Fail-Soft Alfa-Beta . . . . . | p. 103 |
| 4.3.3.4 | O Algoritmo Fail-Soft Alfa-Beta com Tabela de Transposição . . . . .   | p. 105 |
| 4.3.4   | O Aprofundamento Iterativo no <i>VisionDraughts</i> . . . . .  | p. 108 |
| 4.3.5   | O Algoritmo Alfa-Beta com Tabela de Transposição e Aprofundamento Iterativo . . . . .                              | p. 109 |
| 4.4     | O Algoritmo Alfa-Beta com Tabela de Transposição e Bases de Dados de Finais de Jogos . . . . .                     | p. 112 |

|          |  |        |
|----------|--|--------|
| <b>5</b> | <b>Resultados Experimentais e Técnicas Adicionais</b>  | p. 118 |
| 5.1      | Resultados Experimentais . . . . .   | p. 118 |
| 5.1.0.1  | Impacto do Módulo Eficiente de Busca em Árvores de<br>Jogos . . . . .                            | p. 118 |
| 5.1.0.2  | Impacto do Módulo de Acesso às Bases de Dados Finais   | p. 120 |
| 5.1.0.3  | Impacto do Módulo de Aprofundamento Iterativo . . .  | p. 122 |
| 5.2      | Técnicas Utilizadas Durante e Após o Treinamento do <i>VisionDraughts</i> .                      | p. 123 |
| 5.3      | Ferramenta Utilizada na Implementação do <i>VisionDraughts</i> . . . . .                         | p. 125 |
| 5.4      | Outras Técnicas Implementadas durante o Desenvolvimento do <i>Vision-<br/>Draughts</i> . . . . . | p. 125 |
| 5.4.1    | O Algoritmo <i>MTD-f</i> . . . . .   | p. 125 |
| 5.4.2    | Mapeamento Espacial do Tabuleiro . . . . .   | p. 128 |
| 5.4.3    | Mapeamento do Tabuleiro por Chave Hash . . . . .   | p. 128 |
| <b>6</b> | <b>Conclusões</b>  | p. 130 |
| 6.1      | Perspectiva de Trabalhos Futuros . . . . .   | p. 131 |
|          | <b>Referências</b>   | p. 134 |

# *Lista de Figuras*

|    |  |       |
|----|--|-------|
| 1  | Todas as divisas interestaduais possíveis para os estados MS, GO e MT  | p. 19 |
| 2  | Exemplo de uma expansão gerada por busca em largura. . . . .   | p. 22 |
| 3  | Exemplo de uma expansão gerada por busca em largura: o vértice 5 não precisa ser inserido na fila mais de uma vez. . . . .   | p. 24 |
| 4  | Exemplo de uma expansão gerada por uma busca em profundidade. . .  | p. 26 |
| 5  | Modelo de neurônio artificial. . . . .   | p. 33 |
| 6  | Modelo de função de ativação baseado na tangente hiperbólica. . . . .  | p. 34 |
| 7  | Esquerda: exemplo de rede acíclica. Direita: exemplo de rede cíclica. .  | p. 35 |
| 8  | Tabuleiro do jogo de Gamão . . . . .   | p. 39 |
| 9  | Esquerda: representação do tabuleiro para uso com as bases de dados das fases finais do jogo. Direita: representação antes das fases finais. . .                     | p. 44 |
| 10 | Representação do tabuleiro do jogo, através de <i>bitboards</i> , utilizada pelo jogador <i>KingsRow</i> . . . . .   | p. 44 |
| 11 | Espaço de estados para o jogo de damas: quantidade de estados possíveis de acordo com o número de peças sobre o tabuleiro (SCHAEFFER et al., 2007). . . . .          | p. 47 |
| 12 | Estatísticas para o final de jogo 3B1b3W. . . . .  | p. 51 |
| 13 | Estado do tabuleiro 3B1b(7)3W. . . . .   | p. 52 |
| 14 | O jogo <i>Awari</i> : representação do tabuleiro e exemplos de movimentos. Os números dentro dos círculos representam o número de pedras dentro dos buracos. . . . . | p. 56 |
| 15 | Abertura do jogo moinho. . . . .   | p. 57 |
| 16 | Fluxo de aprendizagem do <i>NeuroDraughts</i> : o sistema de Mark Lynch. .   | p. 61 |
| 17 | Fluxo do <i>NeuroDraughts</i> em partidas sem ajuste de pesos da rede neural.  | p. 63 |

|    |   |        |
|----|---|--------|
| 18 | Conjunto de características implementadas pelo <i>NeuroDraughts</i> . . . . .   | p. 65  |
| 19 | Rede Neural utilizada pelo <i>NeuroDraughts</i> . . . . .   | p. 66  |
| 20 | Fluxo de aprendizagem do <i>VisionDraughts</i> : um sistema de aprendizagem de jogos de damas. . . . .  | p. 70  |
| 21 | Evolução do jogo de damas com todos os movimentos possíveis a partir do tabuleiro raiz com dois níveis de profundidade. . . . .                         | p. 73  |
| 22 | Exemplo de árvore do jogo de damas criada pelo algoritmo minimax. . .   | p. 77  |
| 23 | Exemplo de árvore do jogo de damas criada pelo algoritmo alfa-beta. . .   | p. 79  |
| 24 | Exemplo de transposição em <i>c</i> e <i>f</i> : o mesmo estado do tabuleiro é alcançado por combinações diferentes de jogadas com peças simples. . . . | p. 83  |
| 25 | Exemplo de transposição em <i>a</i> e <i>c</i> : o mesmo estado do tabuleiro é alcançado por combinações diferentes de jogadas com reis. . . . .        | p. 83  |
| 26 | Vetor de 128 elementos inteiros aleatórios utilizados pelo <i>VisionDraughts</i> . .  | p. 86  |
| 27 | Um estado do tabuleiro do jogo de damas. . . . .  | p. 87  |
| 28 | Exemplo de movimento simples. . . . .   | p. 88  |
| 29 | Exemplo de árvore do jogo de damas criada pelo algoritmo alfa-beta em sua versão fail-soft. . . . .   | p. 99  |
| 30 | Exemplo de ordenação da árvore de busca com iterative deepening. . .  | p. 109 |
| 31 | Tempo de treinamento: 2 sessões de 10 jogos e <i>look-ahead</i> 8. . . . .  | p. 118 |
| 32 | Tempo de treinamento: 10 sessões de 200 jogos e <i>look-ahead</i> 8. . . . .  | p. 119 |
| 33 | O <i>problema do loop</i> e o uso de bases de dados. . . . .  | p. 121 |
| 34 | Mapeamento espacial utilizado por Fogel. . . . .  | p. 129 |



## *Lista de Tabelas*

- 1 Complexidade espacial e temporal da busca em largura. . . . . p. 25
- 2 Comparativo de requisitos de memória para os procedimentos de busca em largura e busca em profundidade. . . . . p. 29
- 3 Avaliação das estratégias de busca.  $b$  fator de ramificação;  $d$  profundidade do estado objetivo menos profundo;  $m$  profundidade máxima da árvore de busca; \* completa se fator de ramificação finito; \*\* ótima se os custos dos passos são idênticos. . . . . p. 30

# 1 *Introdução*

## 1.1 Introdução e Motivação

A escolha do jogo de damas como um domínio de aplicação se deve ao fato de que ele apresenta significativas semelhanças com inúmeros problemas práticos e, por outro lado, apresenta uma complexidade que demanda a utilização de poderosas técnicas de inteligência artificial. Como exemplos destes problemas práticos, podem-se citar os seguintes (NETO, 2007) (NETO; JULIA, 2007b) (NETO; JULIA, 2007a):

1. Problema de navegação em que os mapas são obtidos de maneira autônoma por um robô móvel: a tarefa de aprendizagem parte de um ponto de referência inicial, onde o robô deve aprender uma trajetória de navegação de modo a atingir um ponto alvo e ao mesmo tempo desviar dos obstáculos do ambiente (RIBEIRO; MONTEIRO, 2003);
2. Problema de interação com humanos por meio de um diálogo: cada vez mais, a vida moderna demanda agentes que dialogam com humanos (tais como os atendentes eletrônicos em empresas de prestação de serviços). Como exemplo de sistema que ataca esse problema, cita-se o sistema ELVIS (*Elvis Voice Interactive System*), de Walker (WALKER, 2000), que cria um agente que aprende a escolher uma ótima estratégia de diálogo por meio de suas experiências e interações com os usuários humanos;
3. Problema do controle de tráfego veicular urbano: o objetivo é criar um agente capaz de controlar o número médio de veículos sobre uma rede urbana de forma a minimizar os congestionamentos e o tempo de viagem (WIERING, 2000).

Percebendo o vasto campo de pesquisa proporcionado pelo domínio dos jogos de tabuleiro, Mark Lynch desenvolveu um bom jogador automático de damas (*NeuroDraughts*) que utiliza a técnica de aprendizagem por diferenças temporais para ajustar os pesos de uma rede neural artificial multi-camadas cujo papel é estimar o quanto um estado do

tabuleiro do jogo, representado em sua camada de entrada através de um conjunto de características específicas do próprio jogo, é favorável para o agente jogador (LYNCH, 1997) (LYNCH; GRIFFITH, 1997).

Como Lynch utilizou um conjunto de características definido manualmente, Neto e Julia criaram o *LS-Draughts*, um sistema que expande o trabalho de Lynch, por meio da técnica dos algoritmos genéticos, gerando automaticamente um conjunto mínimo e essencial de características e otimizando, com grande sucesso, o treinamento do jogador automático (NETO, 2007) (NETO; JULIA, 2007b) (NETO; JULIA, 2007a).

Os excelentes resultados obtidos pelo *LS-Draughts* mostraram o vasto potencial de melhorias possíveis em ambos os jogadores automáticos. Neste sentido, o *VisionDraughts*, utilizando redes neurais e aprendizagem por diferenças temporais  $TD(\lambda)$ , introduz dois novos módulos em relação aos jogadores *NeuroDraughts* e *LS-Draughts*:

1. Um módulo de busca eficiente em árvores de jogos que fornece ao agente jogador de damas maior capacidade de analisar jogadas futuras (estados do tabuleiro mais distantes do estado corrente). Este módulo conta com o algoritmo alfa-beta, tabelas de transposição e aprofundamento iterativo (PLAAT, 1996) (SCHAEFFER; PLAAT, 1996) (PLAAT et al., 1995);
2. Um módulo de acesso às bases de dados de finais de jogos que fornece ao agente jogador capacidade de anunciar, antes do final da partida, se um estado do tabuleiro, com até oito peças, representa vitória, derrota ou empate. Utilizando as informações presentes nas bases de dados, o agente jogador substitui informação heurística por conhecimento perfeito, consegue melhor ajuste em sua função de avaliação e torna-se mais eficiente. Este módulo conta com as bases de dados disponibilizadas pela equipe do jogador *Chinook* e com a biblioteca de funções de acesso a estas bases disponibilizada pelo autor do jogador *KingsRow* (SCHAEFFER et al., 1992) (SCHAEFFER, 1992) (SCHAEFFER et al., 2007).

Enquanto o *NeuroDraughts* e o *LS-Draughts* contam com um sistema básico de busca minimax que utiliza profundidade fixa de busca igual a quatro, o *VisionDraughts* conta com um módulo alfa-beta com tabelas de transposição e aprofundamento iterativo que lhe permite ajustar os pesos de sua rede neural de maneira muito mais precisa. Assim, foram realizados alguns torneios entre os três jogadores e os resultados, todos favoráveis ao *VisionDraughts*, indicam que seu nível de jogo supera o nível dos jogadores *NeuroDraughts* e *LS-Draughts*.

Outra motivação na construção do *VisionDraughts* foi a ocorrência do *problema do loop* nos jogadores *NeuroDraughts* e *LS-Draughts* (NETO, 2007). O *problema do loop* representa situação na qual o jogador automático, apesar de estar em vantagem em relação a seu adversário, não consegue pressioná-lo e entra em *loop* infinito (sem conseguir progredir rumo à “vitória óbvia”). Foram realizados alguns experimentos na etapa de treinamento dos três jogadores e os resultados indicam que, com o uso de bases de dados e aprofundamento iterativo, o número de ocorrências do *problema do loop* reduziu consideravelmente.

Os resultados do presente trabalho originaram um artigo aceito para ser publicado no *19th SBIA - Brazilian Symposium on Artificial Intelligence* (CAEXETA; JULIA, 2008) e um capítulo de livro a ser publicado pela *I-Tech Education and Publishing* (NETO; JULIA; CAEXETA, 2008).

## 1.2 Estrutura da dissertação

Os próximos capítulos estão organizados conforme disposto a seguir:

**Capítulo 2.** Referencial teórico para algumas das técnicas mais importantes utilizadas na construção do *VisionDraughts*: estratégias de busca, redes neurais, aprendizagem por reforço e análise em retrocesso na construção de bases de dados;

**Capítulo 3.** Estado da arte em programas que utilizam técnicas de inteligência artificial para fazer com que um determinado agente aprenda a jogar (principalmente jogos de tabuleiros);

**Capítulo 4.** Detalhes de projeto e desenvolvimento do *VisionDraughts*: um sistema de aprendizagem de jogos de damas baseado em redes neurais, diferenças temporais, algoritmos eficientes de busca em árvores e informações perfeitas contidas em bases de dados;

**Capítulo 5.** Resultados experimentais obtidos com o *VisionDraughts* e perspectivas de trabalhos futuros.

## 2 Referencial Teórico

### 2.1 Estratégias de Busca

Os agentes inteligentes devem maximizar sua medida de desempenho para a resolução de um determinado problema. A fim de descrever formalmente um problema e os passos necessários para sua resolução, será considerado o exemplo da escolha da melhor rota entre duas áreas geográficas distintas (RUSSELL; NORVIG, 2004).

Suponha que um determinado agente inteligente, denominado *SearchAgent*, se encontre em Minas Gerais e queira pesquisar um caminho para chegar até o estado do Amazonas, considerando as rotas disponíveis mostradas na figura 1.

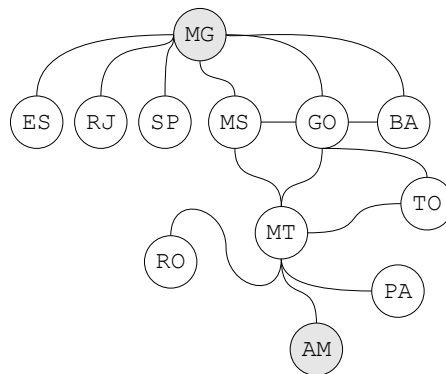


Figura 1: Todas as divisas interestaduais possíveis para os estados MS, GO e MT

A formalização de um problema para o exemplo apresentado pode ser definido por 4 componentes fundamentais:

1. Estado inicial: representa o ponto de partida para o problema, o estado em que o agente se encontra inicialmente. No exemplo, o estado inicial para o *SearchAgent* é Minas Gerais;
2. Ações possíveis: representam ações legais disponibilizadas para o agente a partir do estado inicial. No exemplo, o *SearchAgent* pode cruzar a fronteira de Minas Gerais

e chegar aos estados ES, RJ, SP, MS, GO e BA. As ações disponíveis definem o espaço de estados do problema, ou seja, o conjunto de todos os estados acessíveis a partir do estado inicial;

3. Teste objetivo: o teste objetivo determina se um dado estado, contido no espaço de estados, representa o objetivo do problema formulado. No exemplo, o estado do Amazonas representa o objetivo do *SearchAgent*;
4. Custo do caminho: para o *SearchAgent* alcançar o estado do Amazonas, várias rotas estão disponíveis (figura 1). Porém, percorrer uma rota como  $MG \rightarrow BA \rightarrow GO \rightarrow MS \rightarrow MT \rightarrow AM$  parece não fazer sentido considerando as disposições geográficas do território brasileiro (distâncias). Neste caso, a distância está representando o custo do caminho.

Depois de formular um problema usando os 4 componentes anteriores, parte-se para a solução do mesmo. Uma solução para o *SearchAgent* é encontrar alguma rota que ligue os estados de MG e AM. A solução ótima para o *SearchAgent* é encontrar o caminho de menor custo (distância) entre os estados de origem e destino.

### 2.1.1 O Espaço de Estados

O espaço de estados mostrado na figura 1 é representado por um grafo. Um grafo é uma estrutura matemática, visualizada através de diagramas, composta por um conjunto de vértices e um conjunto de arestas. Embora não seja necessário um estudo aprofundado de grafos, os seguintes conceitos são importantes para o entendimento do espaço de estados:

1. Grafo: pode ser definido como um par  $G = (V, A)$  no qual  $V$  é um conjunto cujos elementos são chamados vértices e  $A$  é um conjunto de pares ordenados de vértices chamados arestas ( $A \subseteq [V]^2$ ). Uma aresta do tipo  $a = (x, y)$  indica a existência de uma ligação entre os vértices  $x$  e  $y$  (DIESTEL, 2000);
2. Caminho: é um grafo não vazio  $P = (V, A)$  onde  $V = \{x_0, x_1, \dots, x_k\}$ ,  $A = \{x_0x_1, x_1x_2, \dots, x_{k-1}x_k\}$  e cada  $x_i$  representa um vértice distinto. Os vértices  $x_0$  e  $x_k$  estão ligados por  $P$  e são chamados *terminais* enquanto os vértices  $x_1, \dots, x_{k-1}$  são chamados vértices *internos* de  $P$ . O número de arestas em um caminho representa o comprimento do caminho e um caminho de comprimento  $k$  é denotado por  $P^k$  ( $k$  pode ser zero) (DIESTEL, 2000);

3. Grafo conectado: um grafo não vazio  $G$  é dito *conectado* se quaisquer dois de seus vértices estiverem ligados ao menos por um caminho em  $G$  (DIESTEL, 2000);
4. Árvore: uma árvore é um grafo acíclico (não contém ciclos) e conectado, ou seja, quaisquer dois de seus vértices são ligados por um único caminho (DIESTEL, 2000). Em teoria dos jogos, apesar de o espaço de estados ser normalmente um grafo (sentido amplo), devido ao sucesso de algoritmos recursivos baseados no *Alfa-Beta*, a área do conhecimento que trata de estratégias de busca no espaço de estados recebe o nome de “*game-tree search*” (PLAAT et al., 1996a). Conseqüentemente, de agora em diante, a palavra árvore poderá aparecer representando o conceito de grafo (mais amplo);
5. Fator de ramificação: o fator de ramificação de um grafo é um número calculado em razão da quantidade de vizinhos de cada vértice. Ele pode ser uniforme, quando todos os vértices possuem o mesmo número de vizinhos, ou médio, caso o número de vizinhos seja variável. Por exemplo, o fator de ramificação médio no jogo de xadrez é 35, significando que cada jogador tem, em média, 35 movimentos legais disponíveis em cada jogada (NETO, 2007).

No exemplo do agente inteligente *SearchAgent*, a solução do problema é construída através de uma busca pelo espaço de estados. A decisão de qual estratégia de busca escolher, para um determinado problema, deve ser realizada observando os 4 critérios seguintes (RUSSELL; NORVIG, 2004):

1. Completude: é garantido que a estratégia encontra a solução do problema quando ela existe?
2. Complexidade temporal: quanto tempo a estratégia de busca demora para encontrar a solução do problema?
3. Complexidade espacial: quanta memória a estratégia de busca necessita para encontrar a solução do problema?
4. Otimização: a estratégia encontra a melhor solução quando existem diferentes soluções?

Assim, as próximas subseções abordarão as idéias básicas de algumas estratégias de busca, levando em consideração os 4 critérios recém-definidos.

## 2.1.2 Busca em Largura

A busca em largura (*breadth-first search*) é uma estratégia de busca cega ou busca não informada, pois não utiliza heurísticas. Um procedimento heurístico pode ser definido como um método de aproximação de soluções de problemas que se baseia em estimativas ou intuições.

O procedimento de busca em largura processa cada um dos estados que estão na vizinhança mais próxima do estado inicial, depois processa cada um dos estados que estão na vizinhança “nível 2”, depois “nível 3” e assim, sucessivamente, até que o estado objetivo do problema seja encontrado. Em outras palavras, o procedimento começa em um estado inicial  $S_0$ , de um determinado grafo, e explora todos os seus estados vizinhos:  $S_1, S_2, \dots, S_n$ . A seguir, passa para o estado  $S_1$  e explora todos os seus estados vizinhos ainda não explorados:  $S_{11}, S_{12}, \dots, S_{1m}$ . O processo se repete até que o estado objetivo do problema seja alcançado.

Veja, na figura 2, como a busca em largura é considerada uma busca “para fora” (PENTON, 2002): os estados vizinhos mais próximos do estado inicial são processados primeiro e os mais distantes processados por último. Na figura, os estados são processados de acordo com a ordem crescente de numeração.

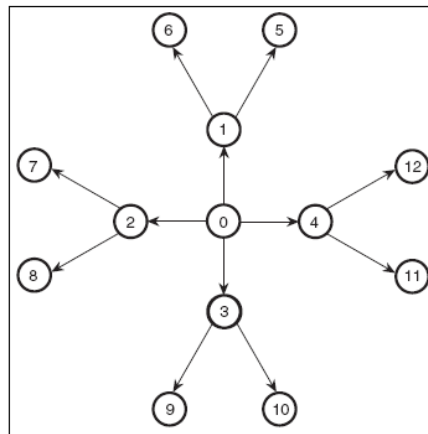


Figura 2: Exemplo de uma expansão gerada por busca em largura.

Visando formalizar o entendimento do procedimento de busca em largura, é apresentado o pseudo-código de um algoritmo (PENTON, 2002), juntamente com uma descrição sucinta de cada linha.

*Pseudo-código para busca em largura*

### 1. BreadthFirst(Node)



```
2.   Queue.Enqueue(Node)
3.   Mark(Node)
4.   While(Queue.IsNotEmpty)
5.       Process(Queue.Front)
6.       For Each Neighbor of Queue.Front
7.           if NotMarked(Neighbor)
8.               Queue.Enqueue(Neighbor)
9.               Mark(Neighbor)
10.          end if
11.      end For
12.      Queue.Dequeue()
13.  End While
14. End Function
```

**linha 1.** O algoritmo recebe o estado inicial *Node* do espaço de estados;

**linha 2.** O estado inicial *Node* é inserido em uma estrutura de dados do tipo fila (o primeiro elemento a ser inserido na estrutura é, também, o primeiro elemento a ser removido);

**linha 3.** O estado inicial *Node* recebe uma marcação de que já foi manipulado pelo algoritmo;

**linha 4.** O bloco de código entre as linhas 4 e 13 deve ser executado para todos os elementos da fila (enquanto a fila não estiver vazia);

**linha 5.** O primeiro elemento da fila (*Queue.Front*) deve ser removido e o estado representado por ele deve ser processado. Caso trate-se de um estado objetivo, ou seja, caso a solução do problema tenha sido encontrada, o algoritmo pode ser encerrado e o resultado retornado;

**linha 6.** Para cada um dos vizinhos de *Queue.Front*, realizam-se os procedimentos das linhas 7 a 10;

**linha 7.** A verificação presente nesta linha evita que um determinado estado seja processado mais de uma vez (a marcação indicando manipulação em cada um dos estados tem esta finalidade);

**linha 8.** O próximo vizinho *Neighbor* ainda não marcado é inserido no final da fila;

**linha 9.** Como na linha 3, assim que o estado *Neighbor* é manipulado pelo algoritmo, ele recebe uma marcação;

**linha 10.** Assim que todos os vizinhos mais próximos e não marcados do estado *Queue.Front* forem inseridos no final da fila, ele é removido do início da mesma e o algoritmo começa a se repetir (com o próximo estado presente no início da fila).

Na figura 3, após o primeiro passo do laço *while*, o vértice 0 terá sido inteiramente processado, os vértices 1, 2, 3, 4 e 5 terão recebido marcação e estarão inseridos, nesta ordem, na estrutura do tipo fila. No próximo passo do laço, o vértice 1 é processado e os vértices 6 e 7 são marcados e inseridos no final da fila. O processo se repete até que o primeiro elemento da fila seja o vértice 6: quando o vértice 6 estiver sendo processado, o vértice 5 não será inserido, novamente, no final da fila, pois já terá recebido uma marcação.

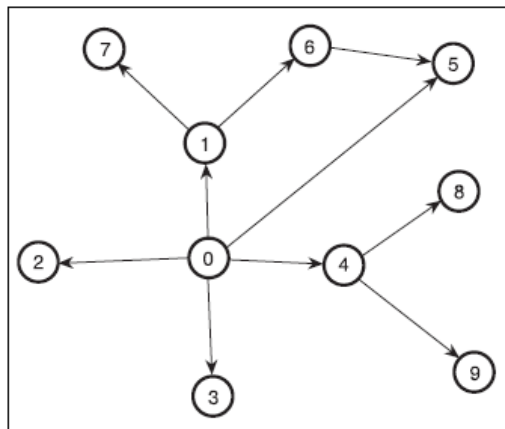


Figura 3: Exemplo de uma expansão gerada por busca em largura: o vértice 5 não precisa ser inserido na fila mais de uma vez.

As próximas subseções descreverão o comportamento da busca em largura de acordo com os 4 critérios definidos na seção 2.1.1.

### 2.1.2.1 Completude

A busca em largura é completa desde que o fator de ramificação seja finito. Isso significa que se existir uma solução, a busca em largura a encontrará para qualquer tipo de grafo com fator de ramificação finito.

### 2.1.2.2 Complexidade Temporal

Suponha um espaço de estados hipotético com fator de ramificação igual a  $b$ . O estado inicial terá  $b$  vizinhos no “nível 1”. Cada um deles terá  $b$  vizinhos, totalizando  $b^2$  vizinhos no “nível 2”. No “nível 3”, existirão  $b^3$  vizinhos e assim por diante. Agora, suponha que o estado objetivo esteja no “nível  $d$ ”. No pior caso, serão expandidos todos os vértices, exceto o último do nível  $d$ , pois o estado objetivo, propriamente dito, não é expandido. Serão gerados, portanto,  $b^{d+1} - b$  vértices no nível  $d + 1$ .

Portanto, tendo que, no pior caso, é necessário expandir  $(b^{d+1} - b)$  vértices no nível  $(d + 1)$  para que se encontre o estado objetivo do problema, presente no nível  $d$ , diz-se que a complexidade temporal do procedimento de busca em largura é da ordem de  $O(b^{d+1})$ .

### 2.1.2.3 Complexidade Espacial

Uma vez que todos os vértices de um determinado nível precisam ser armazenados em memória antes que os vértices do nível seguinte sejam gerados, a complexidade espacial é igual à complexidade temporal. Veja o impacto do crescimento exponencial de complexidade espacial e temporal na tabela mostrada nesta seção (RUSSELL; NORVIG, 2004). Ela lista o tempo e a memória exigidos para uma busca em largura com fator de ramificação  $b = 10$ , para diversos valores de profundidade  $d$ . A tabela pressupõe que 10.000 nós podem ser gerados por segundo e que um nó pode ser armazenado em 1.000 bytes.

| Profundidade | Vértices  | Tempo        | Memória       |
|--------------|-----------|--------------|---------------|
| 2            | 1100      | 0.11 segundo | 1 megabyte    |
| 4            | 111.100   | 11 segundos  | 106 megabytes |
| 6            | $10^7$    | 19 minutos   | 10 gigabytes  |
| 8            | $10^9$    | 31 horas     | 1 terabyte    |
| 10           | $10^{11}$ | 129 dias     | 101 terabytes |
| 12           | $10^{13}$ | 35 anos      | 10 petabytes  |
| 14           | $10^{15}$ | 3.523 anos   | 1 exabyte     |

Tabela 1: Complexidade espacial e temporal da busca em largura.

Analisando os dados da tabela, pode-se perceber que os requisitos de memória são o maior problema para uma busca em largura. Trinta e uma horas não seria tempo demais para se esperar pela solução de um problema importante de profundidade 8, mas poucos computadores têm memória principal da ordem de terabytes. Por outro lado, se um estado objetivo se encontrar em profundidade 12, serão necessários 35 anos para que a busca em largura o alcance.

Em geral, os problemas reais de busca com complexidade exponencial não podem ser resolvidos por métodos de busca cega, dados os requisitos de tempo e espaço.

#### 2.1.2.4 Otimização

Em um grafo cujos caminhos possuem custos diferentes, o estado objetivo mais próximo do estado inicial (em termos de níveis de vizinhança) não é, necessariamente, o estado de menor custo. Uma busca em largura sempre retorna o estado objetivo mais próximo do estado inicial. Assim, uma busca em largura será ótima no caso em que os custos dos caminhos forem idênticos.

### 2.1.3 Busca em Profundidade

A busca em profundidade (*depth-first search*) é uma estratégia de busca cega ou busca não informada, pois não utiliza heurísticas. O procedimento processa o estado inicial e, para cada um dos vizinhos mais próximos, chama a si mesmo, recursivamente, até que o estado objetivo seja encontrado. Veja, na figura 4, um exemplo de expansão em profundidade (observe que a ordem em que os vértices são visitados está definida pela ordem numérica crescente).

No exemplo da figura 4, a árvore é percorrida, sistematicamente, de cima para baixo e da esquerda para direita. Assim que um vértice terminal (sem vizinhos adiante) é encontrado, entra em funcionamento o mecanismo de *backtracking*: procedimento que faz com que o algoritmo retorne, pelo mesmo caminho percorrido, até o último vértice que possui outro caminho ainda não explorado.

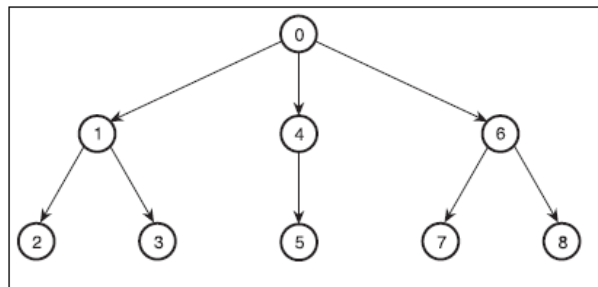


Figura 4: Exemplo de uma expansão gerada por uma busca em profundidade.

Para formalizar o procedimento de busca em profundidade, é apresentado o pseudo-código de um algoritmo (PENTON, 2002), juntamente com uma descrição sucinta de cada linha.

*Pseudo-código para uma busca em profundidade*

```
1. DepthFirst(Node)
2.   Process(Node)
3.   Mark(Node)
4.   For Every Neighbor of Node
5.     If NotMarked(Neighbor)
6.       DepthFirst(Neighbor)
7.     End If
8.   End For
9. End Function
```

**linha 1.** O algoritmo recebe o estado inicial *Node* do espaço de estados;

**linha 2.** O estado inicial é processado para ver se é um estado objetivo, ou seja, caso a solução do problema tenha sido encontrada, o algoritmo pode ser encerrado;

**linha 3.** O estado inicial *Node* recebe uma marcação de que já foi processado pelo algoritmo;

**linha 4.** O bloco de código entre as linhas 4 e 8 deve ser executado para todos os vizinhos de *Node*;

**linha 5.** A verificação presente nesta linha evita que um determinado estado seja processado mais de uma vez. A marcação em cada um dos estados (linha 3) tem esta finalidade;

**linha 6.** O algoritmo é chamado, recursivamente, com o novo estado inicial *Neighbor*. A condição de parada é testada pela linha 2 que processa cada um dos vértices do espaço de estados.

As próximas subseções descreverão o comportamento da busca em profundidade de acordo com os 4 critérios definidos na seção 2.1.1.

### 2.1.3.1 Completude

A busca em profundidade não é completa e isso pode ser comprovado com dois exemplos sobre a árvore da figura 4:

1. Imagine que a sub-árvore representada pelo vértice de número 1 e seus descendentes seja infinita. Neste caso, se o vértice de número 4 for um estado objetivo, a busca em profundidade nunca o encontrará;
2. Para que não ocorra o problema dos caminhos infinitos, imagine que o procedimento de busca seja realizado com uma profundidade limitada. Neste caso, se a profundidade for definida, por exemplo, como  $d = 1$  e o vértice de número 7 for um estado objetivo, uma busca em profundidade nunca o encontrará;

### 2.1.3.2 Complexidade Temporal

Suponha um espaço de estados hipotético com fator de ramificação igual a  $b$ . O estado inicial terá  $b$  vizinhos no “nível 1”. Cada um deles terá  $b$  vizinhos, totalizando  $b^2$  vértices no “nível 2”. No “nível 3”, existirão  $b^3$  vértices e assim por diante. O procedimento de busca em profundidade, considerando que o estado objetivo esteja no nível  $d$ , expandirá, no pior caso, todos os  $b^d$  vértices do nível  $d$  (o pior caso de uma busca em profundidade pode ser visualizado na árvore da figura 4, imaginando que o estado objetivo seja o vértice de número 8).

Portanto, como é necessário expandir  $b^d$  vértices no nível  $d$  para que se encontre, no pior caso, o estado objetivo do problema, diz-se que a complexidade temporal do procedimento de busca em profundidade é da ordem  $O(b^d)$ .

### 2.1.3.3 Complexidade Espacial

Uma busca em profundidade possui requisitos modestos de memória. Ela só precisa armazenar um único caminho do vértice inicial até cada um dos vértices mais profundos. Um estado pode ser retirado completamente da memória quando todos os seus descendentes tiverem sido explorados. Para um espaço de estados com fator de ramificação  $b$  e profundidade máxima  $m$ , a busca em profundidade exige o armazenamento de apenas  $bm + 1$  vértices. Usando as mesmas suposições da tabela da seção 2.1.2.3 e supondo que os vértices na mesma profundidade do vértice objetivo não têm sucessores, os requisitos espaciais de uma busca em profundidade podem ser vistos abaixo:

| Tipo                  | Profundidade | Memória       |
|-----------------------|--------------|---------------|
| Busca em profundidade | 12           | 118 kilobytes |
| Busca em largura      | 12           | 10 petabytes  |

Tabela 2: Comparativo de requisitos de memória para os procedimentos de busca em largura e busca em profundidade.

#### 2.1.3.4 Otimização

Uma busca em profundidade não é ótima e isso pode ser comprovado com um exemplo na árvore da figura 4. Caso os vértices de número 2 e 4 sejam estados objetivos, a busca em profundidade retornará o vértice de número 2 como solução do problema. Porém, considerando custos iguais para todas as arestas, a solução ótima seria o vértice de número 4, pois ele está mais próximo da raiz.

### 2.1.4 Busca em Profundidade com Aprofundamento Iterativo

Na seção 2.1.3, foi visto que a ocorrência de caminhos infinitos torna uma busca em profundidade não completa. Para evitá-los, normalmente, o procedimento de busca em profundidade é chamado com profundidade limitada. Porém, estabelecer o parâmetro que define qual deve ser a profundidade utilizada nem sempre é tarefa fácil (dependente da aplicação).

Uma busca em profundidade com aprofundamento iterativo (*iterative deepening*) pode ser utilizada para tentar encontrar o limite de profundidade mais adequado para um determinado problema, levando em consideração as restrições de recursos computacionais. Ela faz isso incrementando, gradualmente, o limite de profundidade até encontrar o estado objetivo mais raso.

A idéia básica do aprofundamento iterativo é realizar uma série de buscas, em profundidade, independentes, cada uma com um *look-ahead* acrescido de um nível. Assim, é garantido que o procedimento de busca iterativo encontra o caminho mais curto para a solução, justamente como a busca em largura encontraria. Porém, em comparação com a última estratégia, os recursos de memória são insignificantes. Em outras palavras, uma busca com aprofundamento iterativo “imita” uma busca em largura com uma série de buscas em profundidade (REINEFELD; MARSLAND, 1994). De fato, note que uma busca com aprofundamento iterativo em seu início, quando a profundidade de busca for  $d = 1$ , expande os mesmos nós que uma busca em largura expandiria no nível 1. Na segunda iteração, quando a profundidade de busca passa a ser  $d = 2$ , o procedimento expande

(seguindo a ordem de uma busca em profundidade de nível 2) os mesmos nós que uma busca em largura expandiria no nível 2.

O aprofundamento iterativo combina os benefícios dos procedimentos de busca em largura e busca em profundidade. Como na busca em largura, ela é completa quando o fator de ramificação é finito e ótima quando o custo do caminho é proporcional à profundidade. Assim, pode-se fazer o seguinte comparativo entre as estratégias de busca vistas até o momento (RUSSELL; NORVIG, 2004):

| <b>Critério</b>              | <b>Largura</b> | <b>Profundidade</b> | <b>Ap Iterativo</b> |
|------------------------------|----------------|---------------------|---------------------|
| <b>Completude</b>            | <i>Sim*</i>    | <i>Não</i>          | <i>Sim*</i>         |
| <b>Complexidade Temporal</b> | $O(b^{d+1})$   | $O(b^m)$            | $O(b^d)$            |
| <b>Complexidade Espacial</b> | $O(b^{d+1})$   | $O(bm)$             | $O(bd)$             |
| <b>Otimização</b>            | <i>Sim**</i>   | <i>Não</i>          | <i>Sim**</i>        |

Tabela 3: Avaliação das estratégias de busca.  $b$  fator de ramificação;  $d$  profundidade do estado objetivo menos profundo;  $m$  profundidade máxima da árvore de busca; \* completa se fator de ramificação finito; \*\* ótima se os custos dos passos são idênticos.

Conforme apresentado acima, a desvantagem do procedimento iterativo é que ele processa muitos estados repetidos antes de alcançar a profundidade do estado objetivo. Com uma análise mais detalhada, no entanto, percebe-se que o custo adicional não afeta a busca em árvores de crescimento exponencial de maneira significativa. A razão intuitiva é que quase todo o trabalho será realizado no nível mais profundo da árvore onde os vértices são expandidos apenas uma vez (os vértices do penúltimo nível são expandidos duas vezes, os do antepenúltimo, três vezes e assim por diante).

Korf (KORF, 1985), ao descrever sobre busca em profundidade com aprofundamento iterativo, elucida o entendimento do assunto com a seguinte definição e teorema:

**Definição 2.1.1** *Uma busca “brute-force” é um algoritmo de busca que não usa informações além do estado inicial, os operadores do espaço de estados e o teste objetivo da solução.*

**Teorema 2.1.1** *A busca em profundidade com aprofundamento iterativo é assintoticamente ótima, dentre as buscas “brute-force”, em termos de tempo, espaço e comprimento da solução.*

Além de combinar os benefícios de busca em largura e busca em profundidade, a qualidade geral do procedimento de busca em profundidade com aprofundamento iterativo cresce, consideravelmente, com o uso de tabelas de transposição, conforme seção 2.1.5.



### 2.1.5 Busca em Profundidade com Aprofundamento Iterativo e Tabelas de Transposição

A grande desvantagem do aprofundamento iterativo é o processamento repetido de estados em níveis mais rasos da árvore de busca que acontece antes de se encontrar a profundidade do estado objetivo do problema.

Tabelas de transposição são repositórios, em memória, de estados que foram previamente submetidos ao procedimento de busca (MILLINGTON, 2006). Com o uso de tabelas de transposição, o procedimento de busca em profundidade pode ser modificado de modo a verificar se um determinado estado encontra-se armazenado em memória antes de expandí-lo (os detalhes poderão ser observados, posteriormente, na seção 4.3.3.4). Deste modo, a grande desvantagem do aprofundamento iterativo (processamento repetido de estados em níveis mais rasos da árvore de busca) praticamente desaparece, uma vez que buscar informações em memória é um procedimento extremamente rápido.

Além disso, tabelas de transposição são utilizadas em conjunto com aprofundamento iterativo para produzir árvores de buscas parcialmente ordenadas. Dentre as informações armazenadas em uma tabela de transposição, para um determinado estado, está a melhor ação a ser executado a partir do mesmo. Quando o aprofundamento iterativo pesquisar um nível mais profundo da árvore de busca e revisitar um dado estado  $S_0$ , o filho  $S_m$  de  $S_0$ , originado pela execução da melhor ação eventualmente indicada na tabela de transposição para ser executada a partir de  $S_0$ , será pesquisado primeiro (ordenação parcial da árvore). Assumindo que uma busca mais rasa é uma boa aproximação para outra mais profunda, a melhor ação para um estado  $S_0$  na profundidade  $d$  será, possivelmente, a melhor ação para o estado  $S_0$  na profundidade  $d + 1$  (PLAAT et al., 1996b).

Com o uso de tabelas de transposição e aprofundamento iterativo, o procedimento de busca em profundidade é aprimorado para expandir árvores em uma sequência típica da estratégia de busca pela melhor escolha, apresentada na seção seguinte.

### 2.1.6 Busca pela Melhor Escolha

Os algoritmos de busca são categorizados de acordo com a estratégia de expansão utilizada. Nas seções 2.1.2 e 2.1.3 foram apresentadas as estratégias de busca em profundidade e busca em largura, respectivamente. Outra alternativa é a estratégia de busca pela melhor escolha (*best-first search*) que possibilita utilizar informações heurísticas selecionadas de acordo com as regras do domínio do problema. O uso de informações heurísticas

permite explorar, primeiro, as partes mais promissoras da árvore de busca. Assim, os algoritmos de busca pela melhor escolha tendem a ser mais eficientes que os algoritmos de busca em profundidade (PLAAT, 1996).

Embora os algoritmos de busca pela melhor escolha façam muito sucesso em outras áreas do conhecimento, a grande maioria dos programas de jogos de tabuleiro é baseada no algoritmo alfa-beta (busca em profundidade). Como no alfa-beta existe um grande intervalo entre os tamanhos das árvores mais ordenadas e das menos ordenadas, várias melhorias foram acrescentadas ao alfa-beta básico, dentre elas o aprofundamento iterativo, as tabelas de transposição e as janelas limitadas de busca (a janela de busca do procedimento alfa-beta é o intervalo entre os parâmetros alfa e beta).

A denominação busca pela melhor escolha é, normalmente, reservada para os algoritmos que se diferenciam da estratégia de expansão em profundidade da esquerda para a direita adotada pelo alfa-beta (PLAAT, 1996). Entretanto, com a adição de aprofundamento iterativo, tabelas de transposição e janelas limitadas de busca, poderia-se argumentar que o algoritmo alfa-beta acaba adotando a estratégia de busca pela melhor escolha (PLAAT, 1996).

Um algoritmo de grande sucesso que adota a estratégia de busca pela melhor escolha é o *MTD-f* (*Memory Enhanced Test Driver*) (PLAAT et al., 1995) (PLAAT, 1996). Plaata anunciou que possui uma implementação do *MTD-f* que supera sua melhor versão do alfa-beta. Segundo ele, o *MTD-f* expande menos nós e menos folhas, além de economizar tempo de execução (PLAAT et al., 1995).

As seções 5.4.1 e 5.1 apresentam mais detalhes teóricos e experimentais com o *MTD-f*.

## 2.2 Redes Neurais

Uma rede neural artificial é um modelo computacional, baseado em redes neurais biológicas, que consiste em uma rede de unidades básicas simples chamadas neurônios. Um mesmo algoritmo roda em cada um dos neurônios e cada um deles se comunica com um subconjunto qualquer de outros neurônios da mesma rede (MILLINGTON, 2006).

O primeiro modelo matemático de um neurônio artificial foi proposto em 1943, por McCulloch e Pitts (MCCULLOCH; PITTS, 1943) e pode ser visto na figura 5. Neste modelo, tem-se:

1. O neurônio é referenciado pela letra  $j$ ;

2. Os vínculos de entrada são representados por  $a_0, a_1$  até  $a_n$ . Excetuando a entrada  $a_0$  que está fixa e vale -1, as demais entradas do neurônio  $j$  representam saídas de outros neurônios da rede;
3. Os pesos são referenciados pela letra  $w$ . O peso  $w_{1j}$ , por exemplo, define o grau de importância que o vínculo de entrada  $a_1$  possui em relação ao neurônio  $j$ ;
4. A *função de soma* acumula os estímulos recebidos pelos vínculos de entrada a fim de que a *função de ativação* possa processá-los. A função de ativação é dada por  $a_j = g(in_j) = g(\sum_{i=0}^n w_{ij}.a_i)$ , onde  $a_i$  é a ativação de saída do neurônio  $i$  conectado a  $j$  e  $w_{ij}$  é o peso na ligação entre os neurônios  $i$  e  $j$ ;
5. O peso  $w_{0j}$ , conectado à entrada fixa  $a_0 = -1$ , define o limite real para o neurônio  $j$ , no sentido de que o neurônio  $j$  será ativado quando a soma ponderada das entradas reais  $\sum_{i=1}^n w_{ij}.a_i$  exceder  $w_{0j}.a_0$ .

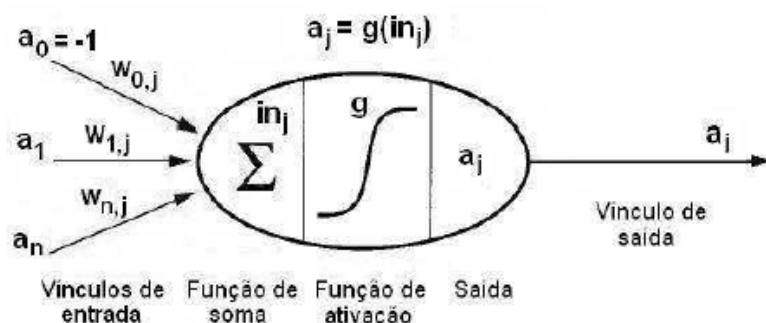


Figura 5: Modelo de neurônio artificial.

Existem vários tipos de função de ativação, por exemplo, a tangente hiperbólica mostrada na figura 6. Uma função de de ativação é projetada para atender a duas aspirações (NETO, 2007):

1. O neurônio deverá ser ativado quando as entradas recebidas forem “*corretas*” e inativado quando as entradas recebidas forem “*incorretas*”;
2. A ativação precisa ser não-linear, caso contrário a rede neural inteira entrará em colapso, tornando-se uma função linear simples.

As redes neurais se classificam, quanto à estrutura, em cíclicas ou acíclicas (NETO, 2007) (XING; PHAM, 1995):

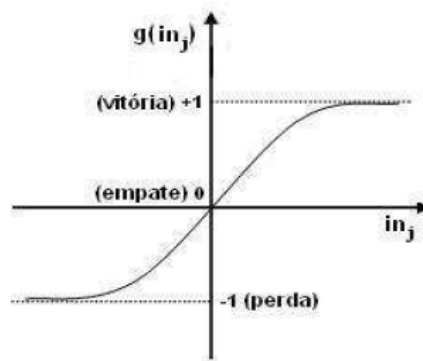


Figura 6: Modelo de função de ativação baseado na tangente hiperbólica.

1. Redes acíclicas ou redes com alimentação direta (*feed-forward*): a propagação do processamento neural é feita em camadas sucessivas, ou seja, neurônios dispostos em camadas terão seus sinais propagados, sequencialmente, da primeira à última camada, de forma unidirecional. A rede à esquerda da figura 7 é acíclica;
2. Redes cíclicas ou recorrentes: as saídas de um (ou todos) os neurônios podem ser redirecionadas aos neurônios das camadas precedentes. A rede à direita da figura 7 é cíclica;

As redes neurais se classificam, quanto a forma de treinamento, em redes com treinamento supervisionado, com treinamento não supervisionado e com aprendizagem por reforço (NETO, 2007) (XING; PHAM, 1995):

1. Redes com treinamento supervisionado: uma sequência de padrões de entrada associados a padrões de saída é apresentada à rede, daí ela utiliza comparações entre a classificação para o padrão de entrada e a classificação correta do padrão de saída para ajustar seus pesos;
2. Redes com treinamento não supervisionado: não existe a apresentação de mapeamentos entrada-saída para a rede. Para este tipo de treinamento não se usa um conjunto de exemplos previamente conhecidos. Uma medida da qualidade da representação do ambiente é estabelecida e os pesos da rede são modificados de modo a otimizar tal medida;
3. Redes com aprendizagem por reforço: utiliza-se alguma função heurística (definida a priori) para descrever o quanto uma resposta da rede para uma determinada entrada é boa. Não é fornecido à rede o mapeamento direto entrada-saída, mas sim uma

recompensa (ou penalização) decorrente da saída gerada pela rede em consequência da entrada apresentada. Tal reforço é utilizado no ajuste dos pesos da rede.

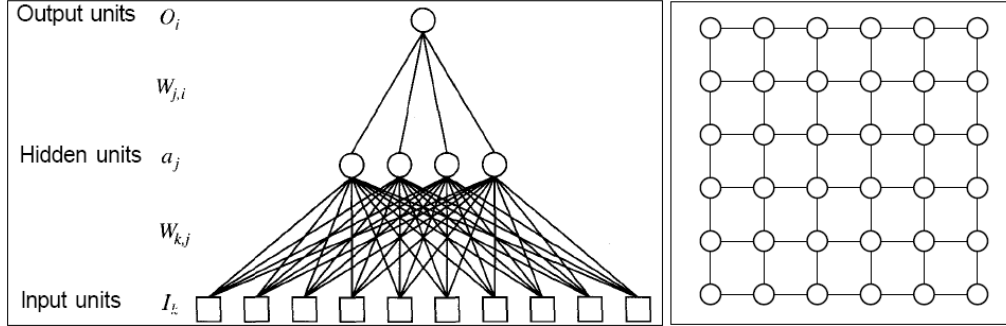


Figura 7: Esquerda: exemplo de rede acíclica. Direita: exemplo de rede cíclica.

O conhecimento adquirido e mantido por uma rede neural reside em seus pesos, pois nenhum neurônio guarda em si todo o conhecimento, mas faz parte de uma malha que retém a informação graças a todos os seus neurônios (NETO, 2007). Ajustar os pesos de uma rede neural significa treiná-la com algum tipo de treinamento (supervisionado, não supervisionado ou com aprendizagem por reforço). Para cada tipo de treinamento, existem algoritmos específicos que podem ser utilizados. Particularmente, a aprendizagem por reforço é essencial para o entendimento do sistema apresentado no capítulo 4 e é, portanto, a única técnica de ajuste de pesos detalhada neste trabalho (seção 3.2).

## 3 *Estado da Arte*

Cientistas de todo o mundo utilizam jogos de natureza recreativa e competitiva com o intuito de aprimorar e desenvolver técnicas poderosas de inteligência artificial. Neste capítulo, são apresentados alguns dos melhores projetos científicos na área de jogos de tabuleiro que foram construídos utilizando as técnicas descritas no referencial teórico.

### 3.1 **Redes Neurais como Recurso de Aproximação de Funções em Jogos**

Um dos principais requisitos para se construir um jogador automático com alto nível de jogo é possuir um modelo eficiente de função de avaliação. Dado um estado legal possível, a função de avaliação deve orientar o jogador automático a executar uma ação que traga resultados positivos para si próprio. Quanto mais complexo for um jogo, maior a quantidade de estados distintos possíveis e mais complexa se torna a tarefa de construir um bom modelo de função de avaliação.

Representar, através de funções, todas as descontinuidades produzidas pelos diversos estados legais disponíveis em um jogo é quase impossível. Então, a utilização de redes neurais se tornou bastante popular como recurso de aproximação de funções (HAYKIN, 2001). Como exemplo, pode-se citar o *TD-GAMMON* de Tesauro (TESAURO, 1995) o *NeuroDraughts* de Lynch (LYNCH, 1997) (LYNCH; GRIFFITH, 1997) e o *LS-Draughts* de Neto e Julia (NETO, 2007) (NETO; JULIA, 2007b) (NETO; JULIA, 2007a). Todos eles utilizam redes neurais como funções de avaliação para treinarem seus jogadores automáticos.

## 3.2 Aprendizagem por Reforço e Método das Diferenças Temporais

Um dos paradigmas utilizados no ajuste dos pesos de uma rede neural é o treinamento por reforço. Treinar por reforço significa aprender a jogar de forma a poder, incrementalmente, testar e refinar uma função de avaliação (NETO, 2007).

O paradigma da aprendizagem por reforço tem sido de grande interesse na área da aprendizagem automática, uma vez que dispensa um “professor” inteligente para o fornecimento de exemplos de treino, fato que o torna particularmente adequado a domínios complexos em que a obtenção destes exemplos seja difícil ou até mesmo impossível (RUSSELL; NORVIG, 2004).

Na aprendizagem por reforço, um indivíduo deve aprender a partir de sua interação com o ambiente onde se encontra e do conhecimento dos estados e ações possíveis (SUTTON; BARTO, 1998). O agente atua sobre o ambiente e aguarda pelo valor de reforço que o mesmo deve informar como resposta à ação executada. Assim, as próximas tomadas de decisões levarão em conta o aprendizado obtido pela interação (NETO, 2007).

Um sistema típico de aprendizagem por reforço constitui-se, basicamente, de um agente interagindo em um ambiente via percepção e ação. O agente percebe as situações dadas no ambiente (pelo menos parcialmente) e seleciona uma ação a ser executada em consequência de sua percepção. A ação executada muda de alguma forma o ambiente e as mudanças são comunicadas ao agente através de um sinal de reforço (NETO, 2007).

Formalmente, o modelo de um sistema de aprendizagem por reforço consiste em (SUTTON; BARTO, 1998):

1. um conjunto de variáveis de estado percebidas por um agente. As combinações de valores destas variáveis formam o conjunto de estados discretos do agente ( $S$ );
2. um conjunto de ações discretas, que escolhidas pelo agente mudam o estado do ambiente ( $A(s)$ , onde  $s \in S$ );
3. um conjunto de valores das transições de estados (reforços tipicamente entre  $[0,1]$ ).

O objetivo do método de aprendizagem por reforço é fazer com que o agente escolha uma sequência de ações que aumente a soma dos valores das transições de estados, ou seja, é encontrar uma política, definida como um mapeamento de estados em ações, que maximize as medidas de reforço acumuladas ao longo do tempo.

Dentre todos os algoritmos existentes para solucionar o problema da aprendizagem por reforço, este trabalho enfoca o algoritmo de diferenças temporais  $TD(\lambda)$  de Sutton (SUTTON; BARTO, 1998).

As diferenças temporais são capazes de utilizar o conhecimento prévio de ambientes parcialmente conhecidos para prever o comportamento futuro (por exemplo, prever se uma determinada disposição de peças no tabuleiro de damas conduzirá à vitória). Assim, o aprendizado do agente pelo método  $TD(\lambda)$  é extraído de forma incremental, diretamente da experiência deste sobre o domínio de atuação, atualizando as estimativas a cada passo, sem a necessidade de ter que alcançar o estado final de um episódio (um episódio pode ser definido como sendo um único estado ou uma sequência de estados de um domínio) (NETO, 2007).

Em jogos, algumas das aplicações mais conhecidas que utilizam o método das diferenças temporais são: damas (SCHAEFFER et al., 2001) (LYNCH, 1997) (LYNCH; GRIFFITH, 1997) (NETO, 2007) (NETO; JULIA, 2007b) (NETO; JULIA, 2007a) (SAMUEL, 1959) (SAMUEL, 1967), xadrez (THRUN, 1995), Go (SCHRAUDOLPH; DAYAN; SEJNOWSKI, 2001), Gamão (TESAURO, 1994) e Othello (LEUSKI, 1995).

### 3.2.1 O sucesso do *TD-Gammon* para o Jogo de Gamão

Gamão é um jogo praticado em um tabuleiro com 24 casas, 15 peças para o jogador preto e 15 peças para o jogador vermelho (todas as peças são do mesmo tipo). Os jogadores se movimentam arremessando 2 dados e seguindo o fluxo mostrado na figura 8 (PTKFGS, 2007). Gamão é considerado complexo, pois seu espaço de estados é da ordem de  $10^{20}$  e seu fator de ramificação  $\pm 420$  (NETO, 2007).

Devido à complexidade do jogo de Gamão, suas regras não serão apresentadas neste trabalho, porém, o objetivo final de cada jogador é levar todas as suas peças para as seis casas finais (seguindo o fluxo mostrado pelas setas da figura 8).

Uma pequena revolução no campo da aprendizagem por reforço ocorreu quando Tesauro apresentou os primeiros resultados com o seu jogador de Gamão. O *TD-GAMMON*, utilizando pouco conhecimento específico sobre o jogo de Gamão, conseguiu atingir resultados em nível dos maiores jogadores mundiais (TESAURO, 1992) (TESAURO, 1994) (TESAURO, 1995). O algoritmo de aprendizagem utilizado no *TD-GAMMON* é uma combinação do algoritmo  $TD(\lambda)$  com uma função de aproximação não-linear baseada em uma rede neural multi-camadas. Para mapear o tabuleiro do jogo na entrada da rede neural,



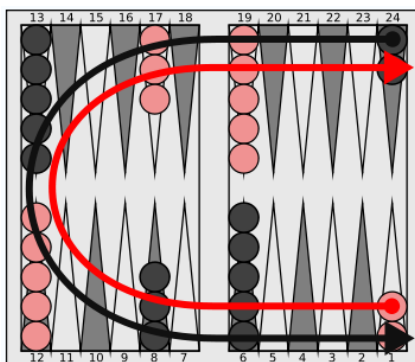


Figura 8: Tabuleiro do jogo de Gamão

Tesauro utilizou um conjunto de características do domínio do jogo de Gamão. Mark Lynch (LYNCH, 1997) (LYNCH; GRIFFITH, 1997) e Neto e Julia (NETO, 2007) (NETO; JULIA, 2007b) (NETO; JULIA, 2007a) também utilizaram conjuntos de características para seus jogadores.

Pollack e Blair (POLLACK; BLAIR, 1998) levantaram a hipótese de que o sucesso do *TD-GAMMON* não está ligado ao método de diferenças temporais, mas sim a uma predisposição inerente à própria dinâmica do jogo de gamão. Por outro lado, Sutton demonstrou a convergência do algoritmo  $TD(\lambda)$  para sistemas nos quais a probabilidade de evolução para um determinado estado  $S_{t+1}$  só depende do estado atual  $S_t$  e da ação  $A_t$  selecionada a partir de  $S_t$  (SUTTON, 1988).

### 3.2.2 O sucesso do *Chinook* para o Jogo de Damas

O *Chinook* é o mais famoso e mais forte jogador de damas do mundo (SCHAEFFER et al., 1992), (SCHAEFFER et al., 1996), (SCHAEFFER et al., 2001), (SCHAEFFER et al., 2007). Ele é o campeão mundial homem-máquina para o jogo de damas e usa uma função de avaliação ajustada manualmente para estimar quanto um determinado estado do tabuleiro é favorável para o jogador automático. Além disso, ele tem acesso à coleções de jogadas utilizadas por grandes mestres na fase inicial do jogo (*opening books*) e um conjunto de bases de dados para as fases finais do jogo: 39 trilhões de estados do tabuleiro (todos os estados com  $\leq 10$  peças) com valor teórico provado (vitória, derrota ou empate). Para escolher a melhor ação a ser executada, o *Chinook* utiliza um procedimento de busca alfa-beta com aprofundamento iterativo e tabelas de transposição.

O *Chinook* conseguiu o título de campeão mundial de damas em agosto de 1994 ao empatar 6 jogos com o Dr. Marion Tinsley que até então defendia seu título mundial a

mais de 40 anos. Para o *Chinook*, o jogo é dividido em 4 fases e cada uma delas possui 21 características que são ajustadas manualmente para totalizar os 84 parâmetros de sua função de avaliação. Os parâmetros foram ajustados ao longo de 5 anos, através de testes extensivos em jogos contra si mesmo e em jogos contra os melhores jogadores humanos.

Diante de tamanho esforço para ajustar a função de avaliação do *Chinook*, Schaeffer e sua equipe realizaram um estudo detalhado de comparação entre uma função de avaliação treinada, manualmente, por peritos e uma função ajustada por diferenças temporais (SCHAEFFER et al., 2001). A primeira abordagem do estudo consistiu em treinar os pesos jogando contra o próprio *Chinook* para determinar a eficácia da aprendizagem face ao benefício de jogar contra um oponente de alto desempenho. O segundo conjunto de experiências envolveu o jogo contra si próprio (estratégia de treino por *self-play*) a fim de verificar se a aprendizagem poderia alcançar um alto nível de desempenho sem ter o privilégio de treinar jogando contra um oponente forte (em ambos os casos, o treino foi realizado utilizando *look-ahead* de 5, 9 e até 13 níveis de profundidade).

Os resultados do treino por *self-play*, obtidos por Schaeffer, evidenciaram que não é necessário um bom professor para que o programa ajuste um conjunto de pesos de uma função de avaliação de forma a alcançar o nível de jogo de um campeão mundial (SCHAEFFER et al., 2001).

### 3.2.3 O *LS-Draughts* e os Algoritmos Genéticos

Dentro do paradigma de aprendizagem por reforço, vários jogadores automáticos utilizam conjuntos de características, específicas dos respectivos domínios, como parte fundamental do processo de ajuste de suas funções de avaliação. Em damas, particularmente, Samuel foi o pioneiro e utilizou 26 características (SAMUEL, 1959) (SAMUEL, 1967). Lynch (LYNCH, 1997) (LYNCH; GRIFFITH, 1997) escolheu manualmente 12 características, dentre aquelas utilizadas por Samuel, para implementar o *NeuroDraughts* (figura 18).

Pensando em automatizar o processo de escolha de características, Neto e Julia desenvolveram o *LS-Draughts*: um sistema que gera automaticamente, por meio da técnica de algoritmos genéticos, um conjunto de características mínimas necessárias e essenciais para o jogo de damas, de forma a otimizar o treino de um jogador automático (NETO, 2007) (NETO; JULIA, 2007b) (NETO; JULIA, 2007a).

O *LS-Draughts* aproxima uma rede neural multi-camadas através do método  $TD(\lambda)$ , aliado com busca *minimax* e a técnica *self-play com clonagem*. O mapeamento do tabu-

leiro na camada de entrada da rede neural é feito utilizando o conjunto de características escolhidas pelos algoritmos genéticos. Com o processo de escolha de características automatizado, o *LS-Draughts* supera o nível de jogo alcançado pelo *NeuroDraughts*, conforme pode ser visto nos resultados experimentais mostrados nos trabalhos de Neto e Julia (NETO, 2007) (NETO; JULIA, 2007b) (NETO; JULIA, 2007a).

### 3.3 Computação Evolutiva

Um paradigma alternativo ao processo de treinamento de uma rede neural para obter bons jogadores automáticos é a computação evolutiva. Nela, as melhores soluções tendem a evoluir com o passar das gerações e as piores soluções tendem a desaparecer. Neste sentido, Fogel (CHELLAPILLA; FOGEL, 2000) (CHELLAPILLA; FOGEL, 2001) (FOGEL; CHELLAPILLA, 2002) utilizou um processo co-evolutivo para implementar um bom jogador de damas que aprende a jogar sem utilizar perícia humana, na forma de características específicas do domínio do jogo.

O melhor jogador de Fogel, chamado *Anaconda*, foi resultado da evolução de 30 redes neurais multi-camadas ao longo de 840 gerações. Cada geração tinha em torno de 150 jogos de treinamento (5 jogos para cada um dos 30 indivíduos da população). Assim, foram necessários 126.000 jogos de treinamento e 6 meses de execução para obtê-lo. Em um torneio de 10 jogos contra uma versão baixo nível do *Chinook*, o *Anaconda* obteve 2 vitórias, 4 empates e 4 derrotas, o que permitiu classificá-lo como “*expert*”.

Atualmente, escolher um método co-evolutivo ou de aprendizagem por diferenças temporais para treinar uma rede neural pode ser uma tarefa árdua. Devido ao sucesso de ambos em alguns domínios específicos como gamão (DARWEN, 2001) e xadrez (FOGEL et al., 2004), a conclusão é que ainda existe muito a ser explorado nesta área do conhecimento humano.

### 3.4 Representação do Tabuleiro com *BitBoards*

A escolha de uma estrutura de dados para representar o tabuleiro em um jogo é fundamental para a eficiência de um jogador automático e implicará quais informações serão, computacionalmente, mais difíceis ou fáceis de serem obtidas. A escolha adequada de uma estrutura de dados também afetará, consideravelmente, a velocidade de execução do algoritmo alfa-beta (presente nos melhores projetos da história dos jogos de tabuleiro).

As operações básicas do algoritmo são (RASMUSSEN, 2004):

1. Geração de movimentos legais a partir de um determinado estado;
2. Transformação de um estado pai em um estado filho e vice-versa (realização dos movimentos gerados anteriormente);
3. Avaliação de um determinado estado.

Uma maneira simples e direta de representar o tabuleiro do jogo de damas, por exemplo, pode ser vista na estrutura abaixo:

*O tabuleiro do jogo de damas representado de maneira simples e direta*

```
BOARDVALUES {
    EMPTY = 0,
    BLACKMAN = 1,
    REDMAN = 2,
    BLACKKING = 3,
    REDKING = 4
}
```

```
BOARD {
    BOARDVALUES p[32]
}
```

O tabuleiro *BOARD* é representado por um vetor de 32 elementos do tipo *BOARDVALUES*. Cada elemento do vetor *BOARD* representa uma casa do tabuleiro e cada casa do tabuleiro possui um dos valores presentes em *BOARDVALUES*.

Outra maneira de representar o tabuleiro do jogo de damas é através de uma estrutura de dados conhecida como *BitBoard*. Um *BitBoard* para o jogo de damas, normalmente um *unsigned int* de 32 bits, representa o tabuleiro de forma que cada bit corresponda a uma das casas do tabuleiro. Como o jogo de damas possui 2 tipos de peças (peças simples e reis), 2 cores de peças (pretas e brancas) e 32 casas no tabuleiro, os três *BitBoards* seguintes são suficientes para a representação completa do tabuleiro:

1. White Pieces (WP): Sempre que existir uma peça qualquer, de cor branca, em uma casa *i* do tabuleiro, o *i*-ésimo bit do *BitBoard* WP deve ser configurado como 1. Caso contrário, o *i*-ésimo bit de WP deve ser 0;

2. Black Pieces (BP): Sempre que existir uma peça qualquer, de cor preta, em uma casa  $i$  do tabuleiro, o  $i$ -ésimo bit do *BitBoard* BP deve ser configurado como 1. Caso contrário, o  $i$ -ésimo bit de BP deve ser 0;
3. Kings (K): Sempre que existir um rei, de qualquer cor, em uma casa  $i$  do tabuleiro, o  $i$ -ésimo bit do *BitBoard* K deve ser configurado como 1. Caso contrário, o  $i$ -ésimo bit de K deve ser 0;

Utilizando os *BitBoards*, o uso das operações lógicas AND, OR, XOR, NOT, SHIFTING LEFT e SHIFTING RIGHT, simbolizadas, respectivamente, por  $\wedge$ ,  $\vee$ ,  $\oplus$ ,  $\neg$ ,  $\gg$  e  $\ll$ , passa a ser fundamental para o fornecimento de informações importantes sobre o tabuleiro. A fim de exemplificar, veja os 3 casos abaixo:

1.  $WP \wedge K$ : informações relativas aos reis brancos presentes no tabuleiro;
2.  $WP \wedge \neg K$ : informações relativas às peças brancas simples presentes no tabuleiro;
3.  $\neg(WP \vee BP)$ : informações relativas às casas não ocupadas do tabuleiro.

Para o jogo de damas, o uso dos *BitBoards* para armazenar informações relativas ao tabuleiro pode ser observado desde o trabalho de Samuel (SAMUEL, 1959), publicado no *IBM Journal of Research and Development*. Além de Samuel, os principais e mais eficientes softwares jogadores de damas da atualidade também fazem uso dos *BitBoards* (a razão aparente para o uso de *BitBoards* é que eles agilizam a geração de movimentos e a avaliação dos estados do tabuleiro, operações básicas do algoritmo alfa-beta). Os jogadores automáticos que influenciaram na construção do presente trabalho e que utilizam *BitBoards* são:

1. *Chinook* (SCHAEFFER et al., 2008): utiliza dois tipos distintos de *BitBoards* para representar o tabuleiro. A representação, mostrada à direita da figura 9, é utilizada durante todo o jogo, com exceção do momento de consultar as bases de dados das fases finais. Antes de consultar um determinado estado junto às bases de dados finais, a representação mostrada à direita deve ser mapeada na representação mostrada à esquerda da figura 9;
2. *Cake* (FIERZ, 2008): utiliza a mesma representação que o *Chinook* utiliza para consultar as bases de dados das fases finais, isto é, a representação mostrada à esquerda da figura 9;

```

/*
 * Table to map between DB board representation and Chinook's
 * Note: the colours are on opposite sides
 *
 *      Database Board:      Chinook's Board:
 *      WHITE                BLACK
 *      28 29 30 31          7 15 23 31
 *      24 25 26 27          3 11 19 27
 *      20 21 22 23          6 14 22 30
 *      16 17 18 19          2 10 18 26
 *      12 13 14 15          5 13 21 29
 *      8  9 10 11           1  9 17 25
 *      4  5  6  7           4 12 20 28
 *      0  1  2  3           0  8 16 24
 *      BLACK                WHITE
 */
/* Note that if your board representation is already the left one, you */
/* do not need the code to convert it to the right one.                */

```

Figura 9: Esquerda: representação do tabuleiro para uso com as bases de dados das fases finais do jogo. Direita: representação antes das fases finais.

```

Database Board:
      WHITE
      31 30 29 28
      27 26 25 24
      23 22 21 20
      19 18 17 16
      15 14 13 12
      11 10 09 08
      03 02 01 00
      BLACK

```

Figura 10: Representação do tabuleiro do jogo, através de *bitboards*, utilizada pelo jogador *KingsRow*.

3. *GuiCheckers* (KREUZER, 2008): mesma representação utilizada pelo *Cake*;
4. *KingsRow* (GILBERT, 2008a): utiliza a representação mostrada na figura 10. Apesar do jogador *NeuroDraughts* (LYNCH, 1997) (LYNCH; GRIFFITH, 1997) não representar o tabuleiro utilizando *BitBoards*, ao menos, o esquema de numeração do *KingsRow* coincide com os índices do vetor de representação do tabuleiro do *NeuroDraughts*.

Outro bom exemplo do uso dos *BitBoards* foi documentado pelos autores do programa de xadrez *Chess*, da *U.S. Northwestern University*, no livro *Chess Skill in Man and Machine* (FREY, 1979).

## 3.5 Busca em Profundidade com Aprofundamento Iterativo e Tabelas de Transposição

Muitos programas jogadores utilizam aprofundamento iterativo (FREY, 1979), (SCHAEFFER, 1989), (SCHAEFFER et al., 1991), (SCHAEFFER et al., 1992), (SCHAEFFER et al., 1993), (REINEFELD; MARSLAND, 1994), (PLAAT, 1996), (PLAAT et al., 1996b). O uso do aprofundamento iterativo baseia-se na hipótese de que uma busca mais rasa, em jogos de tabuleiro, é uma boa aproximação para uma busca mais profunda.

O procedimento começa pesquisando com profundidade igual a um e termina quase imediatamente. Depois, a profundidade de busca é incrementada passo a passo e o procedimento de busca é realizado para cada um deles. Devido ao crescimento exponencial da árvore, o tempo necessário para expandir seu nível mais profundo é muito superior ao tempo necessário para cada um dos níveis mais rasos. Os benefícios mais evidentes do uso do aprofundamento iterativo para os programas jogadores são: obtenção de um mecanismo eficiente de controle de tempo e obtenção de árvores de buscas parcialmente ordenadas (PLAAT et al., 1996b).

As tabelas de transposição são utilizadas em conjunto com o aprofundamento iterativo para alcançar árvores de buscas parcialmente ordenadas. O valor da predição de cada estado já visitado pelo procedimento de busca e o melhor movimento a ser executado a partir dele são armazenados em uma tabela de transposição. Quando o aprofundamento iterativo pesquisar um nível mais profundo da árvore e revisitar estados, o movimento sugerido pela tabela de transposição (caso disponível) será pesquisado primeiro. Assumindo que uma busca mais rasa é uma boa aproximação para outra mais profunda, o melhor movimento da profundidade  $d$  será, possivelmente, o melhor movimento para a profundidade  $d + 1$  (PLAAT et al., 1996b).

O *Chinook* utiliza busca em profundidade com aprofundamento iterativo (SCHAEFFER, 2002), (SCHAEFFER et al., 2005), (SCHAEFFER et al., 2007) iterando  $2\text{-ply}$  por vez (um *ply* é definido como um movimento de um jogador).

No *Chinook*, antes de iniciar o procedimento de busca, o programa realiza uma busca alfa-beta com profundidade  $3\text{-ply}$  e larga janela ( $\alpha = -\infty$  e  $\beta = +\infty$ ) para conseguir uma primeira impressão da qualidade de cada um dos movimentos possíveis a partir do estado corrente. O resultado da busca realizada pode sugerir que o movimento  $M_0$  é bem melhor que os outros. Se o melhor movimento conseguir manter seus status pelas iterações seguintes, o programa cancelará a última iteração numa tentativa de realizar

$M_0$  o mais rápido possível (uma questão de economia de tempo, mas também importante na construção de programas que jogam contra humanos, pois eles não gostam de ficar esperando o adversário realizar jogadas óbvias).

A busca regular começa com profundidade *5-ply* e itera *2-ply* por vez (alcançando profundidades médias entre *17-ply* e *23-ply*). A decisão de iterar nas profundidades ímpares foi proposital, pois o *Chinook* tende a ser mais otimista (agressivo) quando os nós folhas estão a uma profundidade ímpar, partindo da raiz.

## 3.6 Análise em Retrocesso e Bases de Dados

A complexidade temporal de uma busca em profundidade, considerando um fator de ramificação  $b$  e uma profundidade máxima de busca  $m$ , é igual a  $O(b^m)$  (seção 2.1.3). Isso significa que o espaço de estados cresce, exponencialmente, com o aumento da profundidade. Em inteligência artificial, a técnica *análise em retrocesso* (*retrograde analysis*) é utilizada, com frequência, para auxiliar a exploração de espaços de estados com crescimento exponencial.

Em damas, por exemplo, existem aproximadamente  $5 \times 10^{20}$  estados distintos possíveis para o tabuleiro do jogo (figura 11). Mesmo considerando os mais modernos recursos computacionais, é impraticável percorrer tal espaço de estados com um algoritmo baseado na técnica de busca em profundidade. Com a análise em retrocesso, a solução do problema de encontrar um caminho, que permita a um determinado jogador automático alcançar uma vitória, é construída partindo da própria solução do problema, isto é, realizando uma “busca para trás”: com as informações disponibilizadas pelos estados terminais (vitória, derrota ou empate), tenta-se classificar os estados predecessores.

Análise em retrocesso tem sido aplicada, com sucesso, na construção de bases de dados para vários jogos (LAKE; SCHAEFFER; LU, 1994), (SCHAEFFER et al., 2007), (GASSER, 1990), (GASSER, 1996), (ROMEIN; BAL, 2003), (ROMEIN; BAL, 2002). Como a construção de bases de dados para jogos exige o tratamento de grandes quantidades de memória, tempo de execução, entrada e saída (I/O) e capacidade de armazenamento, as técnicas empregadas neste domínio podem ser utilizadas na resolução de diversos problemas, em matemática e ciências correlatas, que exigem que uma solução ótima seja encontrada em um espaço de estados excessivamente grande (análise combinatória).

Suponha, então, que se deseja construir bases de dados de  $n$  peças para o jogo de damas. O espaço de estados a ser pesquisado é um grafo, possivelmente cíclico. Uma



| Pieces     | Number of positions         |
|------------|-----------------------------|
| 1          | 120                         |
| 2          | 6,972                       |
| 3          | 261,224                     |
| 4          | 7,092,774                   |
| 5          | 148,688,232                 |
| 6          | 2,503,611,964               |
| 7          | 34,779,531,480              |
| 8          | 406,309,208,481             |
| 9          | 4,048,627,642,976           |
| 10         | 34,778,882,769,216          |
| Total 1–10 | 39,271,258,813,439          |
| 11         | 259,669,578,902,016         |
| 12         | 1,695,618,078,654,976       |
| 13         | 9,726,900,031,328,256       |
| 14         | 49,134,911,067,979,776      |
| 15         | 218,511,510,918,189,056     |
| 16         | 852,888,183,557,922,816     |
| 17         | 2,905,162,728,973,680,640   |
| 18         | 8,568,043,414,939,516,928   |
| 19         | 21,661,954,506,100,113,408  |
| 20         | 46,352,957,062,510,379,008  |
| 21         | 82,459,728,874,435,248,128  |
| 22         | 118,435,747,136,817,856,512 |
| 23         | 129,406,908,049,181,900,800 |
| 24         | 90,072,726,844,888,186,880  |
| Total 1–24 | 500,995,484,682,338,672,639 |

Figura 11: Espaço de estados para o jogo de damas: quantidade de estados possíveis de acordo com o número de peças sobre o tabuleiro (SCHAEFFER et al., 2007).

aresta de um vértice  $P$  para outro vértice  $Q$  representa um movimento de  $P$  para  $Q$ .  $P$  é chamado *pai* de  $Q$  e  $Q$  é chamado *filho* de  $P$ . Um filho pode ter vários pais e um pai, normalmente, tem vários filhos.

As bases de dados são construídas considerando o número de peças no tabuleiro (os estados do tabuleiro são representados pelos vértices do grafo). Por exemplo, as bases de  $n$  peças são calculadas usando um algoritmo iterativo que utiliza os resultados das bases de dados anteriormente calculadas de  $1, 2, \dots, (n - 1)$  peças.

A base de dados de 1 peça deve ser construída primeiro. Ora, mas se existe apenas uma peça sobre o tabuleiro, o jogador proprietário da peça é o campeão, conforme regra do jogo. O algoritmo deve enumerar os 120 estados distintos possíveis com uma única peça sobre o tabuleiro (estados *terminais*) e, depois, classificá-los como *win* ou *loss*.

A base de dados de 2 peças (6.972 estados distintos) deve ser construída em seguida. Depois, a base de dados de 3 peças (7.092.774 estados distintos) e assim sucessivamente. Para tanto, o algoritmo deverá enumerar todas as combinações possíveis de cada uma das bases e adotar os procedimentos mostrados no seguinte pseudo-código (LAKE; SCHAEFFER;

LU, 1994):

*Algoritmo básico iterativo para associar valores win, loss ou draw a estados do tabuleiro do jogo de damas*

1. Set all positions to UNKNOWN.
2. Iterate and resolve all capture positions.
3. Iterate and resolve non-capture positions.
4. Go to step 3 if any non-capture position was resolved.
5. Set all remaining UNKNOWNs to DRAWs.

Inicialmente, todos os estados do tabuleiro são classificados como *unknown*. A partir daí, as seguintes considerações devem ser observadas:

1. Alguns estados podem ser classificados como *win* ou *loss*, de acordo com as regras do jogo. Por exemplo, um jogador sem peças no tabuleiro ou sem movimento legal disponível encontra-se em um *estado terminal* de derrota;
2. Se um determinado jogador estiver em um estado  $S_0$  e tiver a chance de executar um movimento legal que resulte em um estado  $S_1$ , anteriormente calculado como *win*, então o estado  $S_0$  deve ser, também, definido como *win*;
3. Para cada estado não terminal, sua classificação pode ser calculada no momento em que todos os seus filhos já tenham sido classificados ou, antes mesmo disso, caso algum dos filhos tenha sido classificado como *win*;
4. Quando inexistir informações suficientes para alterar a classificação de qualquer estado do tabuleiro, todos os estados ainda não classificados serão considerados *draw* (nenhum jogador pode mais forçar uma vitória).

A execução do primeiro bloco iterativo (linha 2) considera uma importante regra do jogo de damas: a captura obrigatória. Como resultado, o primeiro bloco iterativo determina o valor de todos os estados de captura e adia os restantes para os passos seguintes. Uma vez que uma captura leva para um estado com  $n - 1$  peças (ou menos), cada estado de captura com  $n$  peças é resolvido consultando os valores das bases de dados anteriormente calculadas. Aproximadamente metade dos estados em uma base de dados para o jogo de damas é de captura (LAKE; SCHAEFFER; LU, 1994).

A execução do segundo bloco iterativo (linha 3) resolve somente os estados que não possuem capturas. Para cada estado considerado, todos os movimentos legais são gerados.

Cada movimento é executado e o valor associado ao estado resultante é consultado na base de dados. O valor *unknown* somente é trocado quando um dos movimentos legais resultar em um estado *win* ou todos os movimentos legais tiverem sido resolvidos. O procedimento continua os blocos de iteração até que nenhum estado do tabuleiro possa ser resolvido. Neste ponto, todos os demais estados são considerados *draw*.

Existem, de fato, duas abordagens opostas para resolver estados do tabuleiro *unknown*:

- Abordagem “*para frente*”: geram-se os estados sucessores para cada estado  $S_0$  não resolvido do tabuleiro e, assim, tenta-se determinar o valor de  $S_0$ ;
- Abordagem “*para trás*”: geram-se os estados antecessores para cada estado resolvido do tabuleiro e, assim, verifica-se se existe informação suficiente para determinar o valor de algum dos antecessores.

A melhor escolha depende da proporção de estados resolvidos e não resolvidos em uma iteração. No jogador *Chinook* (seção 3.2.2), uma combinação das duas abordagens mostrou-se superior (LAKE; SCHAEFFER; LU, 1994).

Em teoria, a técnica apresentada é suficiente para criar qualquer base de dados, porém os problemas reais começam a surgir quando o número de peças aumenta em um espaço de estados de complexidade exponencial. Espaço de armazenamento e tempo de execução tornam-se pontos críticos. Por exemplo, se a análise em retrocesso for utilizada para construir uma base de dados de 8 peças para o jogo de damas, serão necessários mais de 100 gigabytes de memória, considerando apenas 2 bits por estado. Ainda em damas, se uma estação de trabalho com processador de 1.5 GHz for utilizada para construir as bases de dados de 2 até 9 peças e o subconjunto da base de 10 peças, no qual cada jogador possui 5 peças, serão necessários, aproximadamente, 15 anos de processamento (SCHAEFFER et al., 2003).

Técnicas para resolver os problemas de memória, tempo de execução, entrada e saída (I/O) e capacidade de armazenamento existem e são detalhadas pelos autores do *Chinook* (LAKE; SCHAEFFER; LU, 1994), porém, estão fora do escopo do presente trabalho. Somente a título informativo:

1. O problema de memória pode ser tratado decompondo o espaço de estados em pedaços menores para serem tratados individualmente;
2. O problema de tempo de execução pode ser tratado utilizando uma rede distribuída heterogênea de estações de trabalho;

3. O problema de entrada e saída (I/O) pode ser tratado dividindo a computação em fases distintas e eliminando redundâncias;
4. O problema de armazenamento pode ser tratado por um algoritmo de compressão, dependente da aplicação, que permita acesso em tempo real;

Projetar soluções para problemas complexos com recursos limitados é desafiador e importante, pois sempre existirão problemas impossíveis de serem resolvidos com os recursos tecnológicos existentes ou disponíveis para os pesquisadores.

O impacto do uso das base de dados em cada tipo de jogo difere bastante e alguns foram completamente resolvidos. Existem 3 níveis de resolução de um determinado jogo (SCHAEFFER et al., 2007):

1. *ultraweakly solved*: representa o nível mais fraco de resolução de um jogo. O resultado perfeito do jogo é conhecido, porém não é conhecida a estratégia para se conseguir o feito;
2. *weakly solved*: representa o nível intermediário de resolução de um jogo. O resultado perfeito do jogo é conhecido e existe uma estratégia, a partir do estado inicial, para se conseguir o feito;
3. *strongly solved*: representa o nível mais forte de resolução de um jogo. O resultado perfeito do jogo é conhecido e existem estratégias para se jogar, perfeitamente, a partir de qualquer estado possível (não somente o estado inicial).

### 3.6.1 Damas

Em damas, a utilização de bases de dados de finais de jogos tem papel fundamental para a construção de agentes jogadores de alto desempenho. Isso acontece pelos seguintes motivos:

1. Todas as partidas terminam em um conjunto limitado de classes finais, pois só existem peças simples e reis (uma classe representa os finais de jogos com mesmo número e tipo de peças);
2. A regra de captura forçada faz com que o número de peças no tabuleiro reduza, bastante, com o progresso do jogo, implicando árvores de busca mais profundas. Buscas mais profundas garantem substituição das avaliações heurísticas por conhecimento perfeito das bases de dados.

### 3.6.1.1 Chinook

As bases de dados de 8 peças tiveram papel fundamental para o sucesso do *Chinook* contra os melhores jogadores humanos. Elas começaram a ser construídas em 1989 e a construção somente se completou em 1993. Foram mais de 400 bilhões de estados possíveis do tabuleiro comprimidos em mais de 5 gigabytes de dados. Tais números eram impressionantes para os padrões do início da década de 90 quando uma excelente estação de trabalho possuía um Intel 486, 32 megabytes de memória e 1 gigabyte de disco (LAKE; SCHAEFFER; LU, 1994) (SCHAEFFER et al., 1993).

Para se ter uma idéia do impacto das bases de dados na eficiência do jogador *Chinook*, serão apresentados alguns números dos finais de jogos agrupados pelo nome *3B1b3W*, ou seja, finais de jogos que contém 3 reis pretos e uma peça preta simples contra 3 reis brancos. A tabela mostrada na figura 12 (LAKE; SCHAEFFER; TRELOAR, 1998) expõe estatísticas detalhadas para os finais dos jogos *3B1b3W*, considerando que o jogador preto é o próximo a se movimentar e tendo como parâmetro a linha do tabuleiro que se encontra a peça preta simples.

| Endgame   | Positions<br>(millions) | No Capture<br>(millions) | Wins<br>(millions,%) | Losses<br>(millions,%) | Draws<br>(millions,%) |
|-----------|-------------------------|--------------------------|----------------------|------------------------|-----------------------|
| 3B1b(7)3W | 58,902,480              | 33,151,340<br>100%       | 30,193,121<br>91.08% | 37,869<br>0.11%        | 2,920,350<br>8.81%    |
| 3B1b(6)3W | 58902480                | 30,202,582<br>100%       | 27,803,491<br>92.06% | 11,480<br>0.03%        | 2,387,611<br>7.91%    |
| 3B1b(5)3W | 58902480                | 30,054,286<br>100%       | 27,570,210<br>91.73% | 17,291<br>0.06%        | 2,466,785<br>8.21%    |
| 3B1b(4)3W | 58902480                | 29,975,252<br>100%       | 27,500,244<br>91.74% | 21,196<br>0.07%        | 2,453,812<br>8.19%    |
| 3B1b(3)3W | 58902480                | 29,985,968<br>100%       | 27,933,663<br>93.16% | 14,709<br>0.05%        | 2,037,596<br>6.79%    |
| 3B1b(2)3W | 58902480                | 29,199,398<br>100%       | 27,009,062<br>92.50% | 20,839<br>0.07%        | 2,169,497<br>7.43%    |
| 3B1b(1)3W | 58902480                | 28,388,086<br>100%       | 27,757,098<br>97.77% | 1562<br>0.01%          | 629,426<br>2.22%      |

Figura 12: Estatísticas para o final de jogo 3B1b3W.

A entrada *3B1b(4)3W*, por exemplo, considera a peça preta simples na quarta linha

do tabuleiro. O número total de 58.902.480 estados distintos é dado na coluna 2. A coluna 3 subtrai os estados em que o próximo jogador a se movimentar possui captura imediata, restando 29.975.252 estados distintos. Deles, 91,74% são estados de vitória para o jogador preto, 0,07% são estados de derrota e 8,19% são estados de empate, conforme colunas 4, 5 e 6 (LAKE; SCHAEFFER; TRELOAR, 1998).

As bases de dados do *Chinook* foram construídas de maneira sistemática, tentando provar as posições de vitória ou derrota em um movimento, dois movimentos, três movimentos e assim por diante. Quando acabaram os estados de vitória ou derrota, os estados restantes foram declarados empates. As bases de dados do *Chinook*, considerando restrições de tempo de execução e espaço de armazenamento, não contam com a informação de quantos movimentos são necessários para que o jogador alcance uma vitória ou derrota (LAKE; SCHAEFFER; LU, 1994).

Um estado do tabuleiro é considerado *convertido* quando alguma mudança irreversível acontece. Os dois movimentos irreversíveis óbvios, em damas, são capturas e avanços de peças simples. Sempre que algum deles acontecer, é impossível repetir o estado prévio do tabuleiro. Com intuito de estabelecer o grau de dificuldade de um determinado estado, o *Chinook* calcula o número de movimentos necessários para que uma conversão aconteça. Por exemplo, o estado do tabuleiro presente na figura 13, contém uma das 4 posições mais difíceis de serem vencidas pelo jogador preto, quando o mesmo possui uma peça simples na sétima linha do tabuleiro. Neste caso, o jogador preto necessita de, no mínimo, 68 movimentos para conseguir uma conversão (LAKE; SCHAEFFER; TRELOAR, 1998).

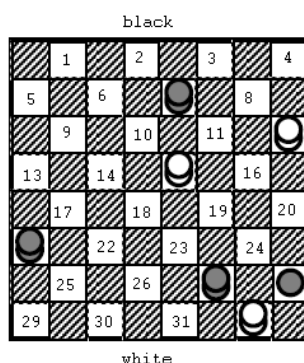


Figura 13: Estado do tabuleiro 3B1b(7)3W.

Conseqüentemente, assumindo que o jogador branco se defenda adequadamente, o jogador preto não consegue converter o tabuleiro em menos de 68 movimentos, ou seja, não consegue passar para um estado presente nas bases de dados de 6 peças nem forçar a promoção da peça simples da casa 28 em menos de 68 movimentos. Quantos jogadores

humanos conseguem encontrar tal linha de vitória? Além disso, se o jogador preto quiser garantir a vitória, ele só tem a opção de mover seu rei preto da casa 27 para a casa 24 (qualquer outro movimento não implica vitória garantida) (LAKE; SCHAEFFER; TRELOAR, 1998).

Assumindo a eficiência do jogador *Chinook*, as bases de dados de 9 e 10 peças foram iniciadas em 2001 com o intuito de resolver o jogo de damas. O *Chinook* já possuía o título de campeão mundial e não necessitava de mais bases de dados para ser reconhecido como o melhor jogador do mundo. Para calcular o valor teórico do jogo de damas, basta que todas as folhas presentes na árvore, montada pelo procedimento de busca em profundidade, estejam presentes nas bases de dados. Na prática, um jogo em tempo real possui limitações de tempo e espaço que impedem estender a busca até o ponto em que todas as folhas estejam presentes nas bases. Para cada folha não presente nas bases, uma função heurística é utilizada para avaliá-la. Cada aplicação da função heurística introduz possibilidades de erros na avaliação de um estado. Uma combinação de folhas presentes nas bases de dados com folhas não presentes é a situação mais comum. A precisão dos resultados do procedimento de busca aumenta com o aumento do percentual de folhas presentes nas bases de dados (SCHAEFFER et al., 2003).

Considerando todas as combinações com 10 peças ou menos sobre o tabuleiro, existem mais de 39 trilhões de estados distintos. Diante de números tão expressivos, o código do *Chinook* teve de ser transformado para utilizar índices de 64 bits de endereçamento. Todos os estados com  $\leq 10$  peças no tabuleiro tiveram seus valores teóricos (vitória, derrota ou empate) calculados e foram comprimidos em 237 gigabytes (154 posições por byte). As bases de dados são indispensáveis para a solução do jogo de damas e um eficiente algoritmo de compressão é utilizado para permitir rápida localização e descompressão em tempo real (SCHAEFFER et al., 2003).

Em 1994, o *Chinook* utilizou um subconjunto das bases de 8 peças ou menos na partida contra Marion Tinsley, considerado o maior jogador humano de todos os tempos. Ele nunca perdeu um campeonato que disputou e perdeu somente 9 partidas em seus 45 anos de carreira (SCHAEFFER et al., 1993) (SCHAEFFER et al., 1996) (SCHAEFFER; LAKE, 1996).

Historicamente, as primeiras bases de dados foram construídas em 1989 e continham posições com 4 peças ou menos. Em 1996, as bases de dados de 8 peças estavam completas. Em 2001, a capacidade dos computadores cresceu bastante e os esforços foram retomados: as bases de dados originais de 8 peças foram construídas em 7 anos (1989 a 1996) enquanto,

em 2001, foram recalculadas em apenas um mês. Em 2005, as bases de dados de 10 peças estavam completas. A partir daí, todos os recursos computacionais foram direcionados para o procedimento de busca em profundidade (SCHAEFFER et al., 2007).

Por fim, em 2007, a equipe do *Chinook* anunciou que o jogo de damas está fracamente resolvido (*weakly solved*): a partir da posição inicial do jogo, existe uma prova computacional de que o jogo é um empate. A prova consiste em uma estratégia explícita com a qual o programa nunca perde, isto é, o programa pode alcançar o empate contra qualquer oponente jogando tanto com peças pretas quanto brancas (SCHAEFFER et al., 2007).

A maior contribuição de aplicar técnicas de inteligência artificial na construção de programas jogadores, segundo a equipe do *Chinook*, é a concretização do fato de que a abordagem de busca intensiva (força bruta) pode gerar altíssima eficiência, usando o mínimo de conhecimento dependente de domínio específico (SCHAEFFER et al., 2007).

#### 3.6.1.2 KingsRow

O autor do *KingsRow* (GILBERT, 2008a) construiu 2 tipos de bases de dados com até 10 peças sobre o tabuleiro, considerando um máximo de 5 peças para cada um dos dois jogadores:

1. WLD Databases (*Win, Loss, Draw*): bases de dados com informação de vitória, derrota ou empate para cada um dos estados do tabuleiro;
2. MTC Databases (*Moves To Conversion*): bases de dados com informação de quantos movimentos são necessários para que se consiga uma *conversão* (movimento de peças simples ou capturas). O espaço necessário para armazenar bases de dados MTC é muito maior que bases WLD, então, somente os estados do tabuleiro com conversão acima de *10-ply* são armazenados (os estados com menos de *10-ply* para conversão são tratados com o procedimento alfa-beta).

Quando as bases de dados WLD estão sendo utilizadas, existe informação se um determinado estado do tabuleiro representa vitória, derrota ou empate. Não existe informação de qual o melhor movimento a ser realizado a partir daquele estado. Quando o *KingsRow* está em um estado de vitória, ele precisa escolher o movimento adequado que lhe permita continuar em um estado vitorioso e, também, progredir rumo à vitória (GILBERT, 2008b). Normalmente, o progresso é garantido por uma função de avaliação que encoraja as conversões. Porém, existem alguns finais de jogos que necessitam de longas seqüências de



movimentos para que uma conversão se realize. Quando tal seqüência de movimentos é maior que o *look-ahead* do *KingsRow*, existe a possibilidade de que o jogador automático não encontre o caminho rumo à conclusão do jogo. As bases de dados MTC são utilizadas para anular a possibilidade do *KingsRow* não encontrar o caminho rumo à conclusão do jogo.

O *KingsRow* mostrou uma posição na base de dados de 10 peças que necessita de uma seqüência com *279-ply* para demonstrar uma vitória forçada. Este é um limite conservativo, pois o comprimento de vitória ainda não foi computado para as posições mais difíceis das bases de dados (SCHAEFFER et al., 2007). Ed Gilbert verificou, também, as bases WLD do *Chinook*, com até 9 peças, e nenhum valor divergente foi encontrado (as bases WLD de 10 peças ainda estão sendo verificadas).

### 3.6.1.3 Cake

O *Cake* é um dos mais fortes agentes jogadores de damas do mundo, construído sobre plataforma PC. Ele joga melhor que qualquer ser humano e é muito mais forte que a famosa versão do *Chinook* que venceu o campeonato mundial contra Marion Tinsley (FIERZ, 2008).

O *Cake* utiliza bases de dados de finais de jogos com até 8 peças sobre o tabuleiro, livros de aberturas de jogos que possuem entre 93 mil e 2 milhões de movimentos, além de um sofisticado mecanismo de busca que garante a um computador moderno a capacidade de analisar, aproximadamente, 2 milhões de posições por segundo (FIERZ, 2008).

Tanto o *Cake* quanto o *KingsRow* podem utilizar a interface *CheckerBoard*: a mais completa interface livre para programas jogadores de damas. Ela suporta o formato de arquivos *PDN* (*Portable Draughts Notation*) que é o padrão para o jogo de damas, além de possuir inúmeras outras características. Com o formato *PDN* e a interface *CheckerBoard*, podem ser utilizadas bases de dados com centenas de jogos previamente descritos, inclusive jogos de Marion Tinsley.

### 3.6.2 Awari

O jogo *Awari* foi resolvido, utilizando análise em retrocesso, por Romein (ROMEIN; BAL, 2002) (ROMEIN; BAL, 2003): a partida termina em empate quando disputada de maneira perfeita pelos dois oponentes.

*Awari* é um jogo de tabuleiro disputado por dois jogadores. Cada jogador tem 6

*buracos* normais e um *buraco* adicional para armazenar *pedras* capturadas. Por exemplo, na figura 14, um dos jogadores se posiciona do lado *South* e o outro do lado *North*.

Inicialmente, 4 pedras são distribuídas em cada um dos 12 buracos normais, totalizando 24 pedras para cada jogador. O jogador a se mover remove todas as pedras de um de seus buracos e as *semeia*, no sentido anti-horário, sobre os outros buracos (uma pedra em cada um dos próximos buracos). Caso o ciclo se complete, isto é, caso existam mais de 11 peças a semear, o buraco original deve ser pulado. Na figura 14-*a*, por exemplo, o jogador posicionado no lado *South* remove 3 pedras do buraco *B* e as semeia da maneira mostrada na figura 14-*b*.

Após a semeadura, deve-se verificar a possibilidade de captura de pedras. A captura acontece quando o número total de pedras dentro do buraco inimigo, após a semeadura, é igual a 2 ou 3. Veja o movimento entre os tabuleiros 14-*b* e 14-*c*: as duas pedras do buraco *F* são removidas e colocadas nos buracos *a* e *b*. Como o número total de pedras no buraco *a* passa para 2 e o número total de pedras no buraco *b* passa para 3, todas elas são removidas para o buraco de pedras capturadas do jogador *South*. As pedras no buraco de pedras capturadas permanecem lá até o final da partida e o jogador que capturar mais pedras vence o jogo.

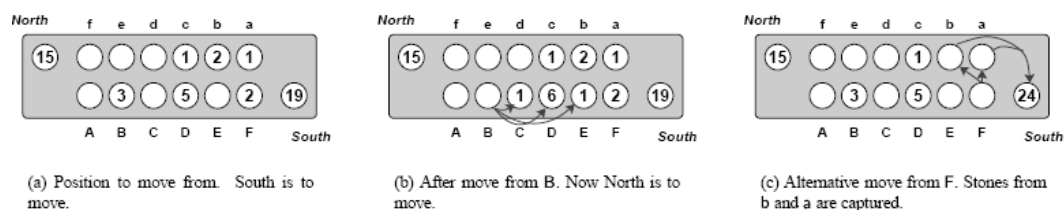


Figura 14: O jogo *Awari*: representação do tabuleiro e exemplos de movimentos. Os números dentro dos círculos representam o número de pedras dentro dos buracos.

Romein provou o valor teórico do jogo *Awari* buscando completamente o espaço de estados em um sistema paralelo com 144 processadores. Com a análise em retrocesso, os melhores movimentos e os resultados exatos para todos os 889.063.398.406 estados alcançáveis foram determinados, começando nas posições finais rumo às posições iniciais. Nenhum computador simples tem capacidade de processamento e requisitos de memória suficientes para buscar tal espaço de estados, porém, mesmo para um moderno computador paralelo, o problema é extremamente desafiador.

As bases de dados foram construídas separadamente, considerando o número de pedras no tabuleiro. A base de dados mais trabalhosa foi, claro, a de 48 pedras. Ela contém mais de 200 bilhões de estados distintos, tamanho maior que qualquer outra base construída,

indivisivelmente, até o presente momento (ROMEIN; BAL, 2002) (ROMEIN; BAL, 2003). Outros pesquisadores tentaram resolver o jogo criando bases de dados com até 40 peças e realizando “*busca para frente*”. Infelizmente, a “*busca para frente*” não alcançava as bases de dados finais no tempo necessário.

Resumindo, o jogo *Awari* é considerado *strongly solved*, ou seja, todos os seus estados distintos possuem valor teórico definido e existe, sempre, uma estratégia conhecida para alcançar tal valor teórico. Na resolução do jogo, 144 processadores Pentium III de 1.0 GHz, 72 GB de memória principal distribuída e 1.4 TB de espaço de armazenamento foram utilizados como suporte para a técnica de análise em retrocesso (ROMEIN; BAL, 2002) (ROMEIN; BAL, 2003).

### 3.6.3 Moinho

O jogo moinho foi resolvido, utilizando análise em retrocesso, por Gasser (GASSER, 1990) (GASSER, 1996): a partida termina em empate quando disputada de maneira perfeita pelos dois oponentes.

O moinho é um jogo de estratégia disputado em um tabuleiro com 24 *pontos*, nos quais *pedras* podem ser colocadas. Inicialmente, o tabuleiro fica vazio e cada um dos dois jogadores seguram 9 pedras. O jogador com pedras brancas começa o jogo. Durante a abertura, os jogadores, alternadamente, colocam suas *pedras* em cada um dos *pontos* vazios (veja um exemplo de abertura do jogo na figura 15).

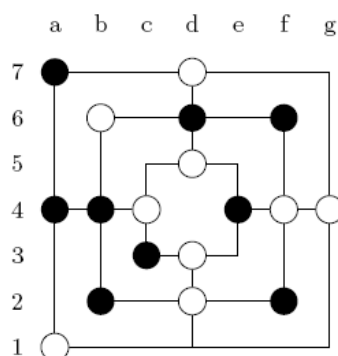


Figura 15: Abertura do jogo moinho.

Durante a fase de abertura do jogo, todas as peças são colocadas no tabuleiro. Depois, cada jogador passa a deslizar suas pedras para um ponto adjacente desocupado (fase de meio do jogo). A qualquer tempo, se um jogador conseguir colocar 3 *pedras* em uma linha, situação conhecida como *mill*, ele remove qualquer pedra do oponente que não faça parte

de outro *mill*.

A fase de fim do jogo começa quando um dos jogadores tem somente 3 pedras no tabuleiro. Nela, o jogador com 3 pedras pode mover uma delas para qualquer ponto vazio do tabuleiro, diferente das outras fases em que o movimento só era permitido para pontos adjacentes. O final de uma partida é determinado da seguinte maneira:

1. Um jogador com menos de 3 pedras perde o jogo;
2. Um jogador que não possui movimentos legais perde o jogo;
3. Se uma posição das fases de meio ou fim do jogo se repetir, o jogo é considerado empate.

Resumindo, o jogo moinho é considerado *strongly solved*, ou seja, todos os seus estados distintos possuem valor teórico definido e existe, sempre, uma estratégia conhecida para alcançar tal valor teórico. Na resolução do jogo, um algoritmo de análise em retrocesso computou 28 bases de dados finais contendo, aproximadamente,  $10^{10}$  estados distintos e um algoritmo alfa-beta, com profundidade *18-ply*, utilizou tais bases para provar que o valor teórico da posição inicial do jogo (tabuleiro vazio) é um empate (GASSER, 1990) (GASSER, 1996).

### 3.6.4 Xadrez

A utilização de bases de dados para as fases finais do jogo de xadrez não apresenta um impacto tão grande quanto para damas, *awari* e moinho. Algumas razões para o pequeno impacto do uso de bases de dados no xadrez são:

1. A maior variedade de peças no xadrez resulta em diferentes classes de finais de jogos, algumas raramente vistas em uma partida real;
2. Os resultados de muitas partidas são, comumente, decididos bem antes das fases contidas nas bases de dados finais;
3. São necessários recursos computacionais relevantes para se construir bases de dados de finais de jogos para xadrez, impossibilitando que muitos programadores possam construir suas próprias bases.

Difícilmente, as bases de dados de 6 peças para o xadrez serão completadas em um futuro próximo, pois poucos pesquisadores e desenvolvedores têm conhecimento, motivação, paciência e, principalmente, recursos computacionais disponíveis para construir bases de dados para um espaço de estados tão complexo (SCHAEFFER et al., 2003).

## 4 *VisionDraughts – Um Sistema de Aprendizagem de Damas*

Neste capítulo, é apresentado o *VisionDraughts*, um sistema de aprendizagem de jogos de damas que tem como principal objetivo construir um agente automático capaz de aprender a jogar com alto nível de desempenho e sem auxílio de especialistas humanos.

Para conseguir alto nível de desempenho, o *VisionDraughts* introduz dois novos módulos em relação aos jogadores *NeuroDraughts* (LYNCH, 1997), (LYNCH; GRIFFITH, 1997) e *LS-Draughts* (NETO, 2007), (NETO; JULIA, 2007b), (NETO; JULIA, 2007a):

1. Um módulo de busca eficiente em árvores de jogos que fornece ao agente jogador de damas maior capacidade de analisar jogadas futuras (estados do tabuleiro mais distantes do estado corrente). O nome *VisionDraughts* foi escolhido para destacar a importância do módulo de busca na arquitetura geral do presente trabalho;
2. Um módulo de acesso às bases de dados de finais de jogos (SCHAEFFER; LAKE, 1996) que fornece ao agente jogador capacidade de anunciar, antes do final da partida, se um estado do tabuleiro, com até 8 peças, representa vitória, derrota ou empate. Utilizando as informações presentes nas bases de dados, o agente jogador substitui informação heurística por conhecimento perfeito, consegue melhor ajuste em sua função de avaliação e torna-se um jogador mais eficiente.

As próximas seções abordam, em detalhe, o desenvolvimento do *VisionDraughts*. Como a compreensão do *NeuroDraughts* é fundamental para o entendimento do presente sistema, inicialmente será apresentado o *NeuroDraughts* de Mark Lynch (LYNCH, 1997) (LYNCH; GRIFFITH, 1997).



Lynch utiliza o mapeamento NET-FEATUREMAP, ou seja, o mapeamento é feito com base em funções que descrevem as próprias características do jogo de damas (LYNCH, 1997) (LYNCH; GRIFFITH, 1997);

3. **Módulo Mapeamento** → avaliação nós folhas → **Rede Neural Artificial**: o módulo de mapeamento processa as folhas da árvore mapeando-as na camada de entrada da rede neural multi-camadas;
4. **Rede Neural Artificial** → valores nós folhas → **Módulo Minimax**: a rede neural multi-camadas recebe um estado do tabuleiro mapeado em sua camada de entrada e retorna, em seu único neurônio da camada de saída, um valor entre -1.0 e +1.0, representando a avaliação do estado de entrada sob a ótica do jogador automático. Tal valor, denominado predição do estado de entrada, é retornado para o módulo de busca que o utilizará com o propósito de descobrir a melhor ação a ser executada;
5. **Módulo Minimax** → melhor ação → **Estado Corrente**: após montar a árvore de busca e utilizar os módulos de mapeamento e rede neural para avaliar as folhas da árvore, o algoritmo minimax propaga, de baixo para cima, a melhor ação a ser executada pelo agente jogador;
6. **Estado Corrente** → movimento → **Próximo Estado**: de posse da melhor ação a ser executada, o estado do tabuleiro é modificado com a realização de uma ação concreta para um próximo estado;
7. **Próximo Estado** → percepção → **Módulo Mapeamento**: este novo estado do tabuleiro do jogo é enviado para o módulo de mapeamento, como anteriormente;
8. **Módulo Mapeamento** → percepção → **Rede Neural Artificial**: o módulo de mapeamento, agora, mapeia este novo estado direto na entrada da rede neural multi-camadas. Note que, agora, não se usa o módulo de busca;
9. **Rede Neural Artificial** → predição melhor ação → **Aprendizagem DT**: assim que o novo estado é mapeado na camada de entrada da rede neural, um novo valor (predição) é obtido no seu único neurônio de saída;
10. **Aprendizagem DT** → ajuste de pesos → **Rede Neural Artificial**: esta nova predição, recém calculada, é utilizada juntamente com a última predição, anteriormente calculada no neurônio de saída, para atualizar todos os pesos da rede neural multi-camadas;



11. **Rede Neural Artificial**  $\rightarrow$  próximo passo  $\rightarrow$  **Estado Corrente**: a partir de agora, com os pesos da rede atualizados, o fluxo retorna ao estágio inicial e o procedimento começa a se repetir, até o fim de uma partida de treinamento.

O jogador de Mark Lynch disputa dois tipos de partidas:

1. Partidas de treinamento: os pesos da rede neural são ajustados pelo método das diferenças temporais. A figura 16 ilustra o fluxo do *NeuroDraughts* para as partidas de treinamento;
2. Partidas sem treinamento: os pesos da rede neural são fixos (invariáveis). A figura 17 ilustra o fluxo do *NeuroDraughts* para as partidas sem treinamento. Cada um dos passos do fluxo da figura 17 tem o mesmo significado do respectivo passo na figura 16;

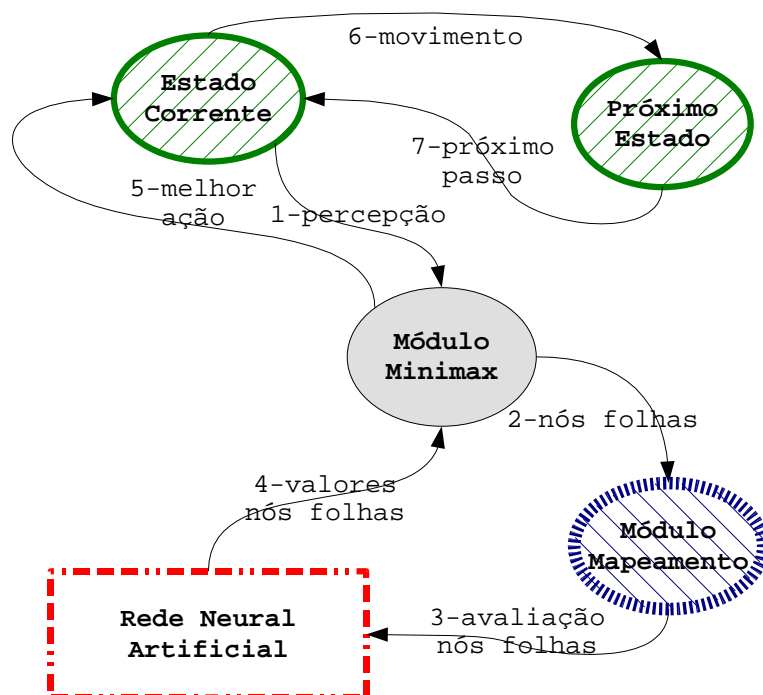


Figura 17: Fluxo do *NeuroDraughts* em partidas sem ajuste de pesos da rede neural.

#### 4.1.1 Representação do Tabuleiro e Mapeamento das Características

Existem diversas maneiras de representar internamente o tabuleiro do jogo de damas. A representação interna do *NeuroDraughts* utiliza a maneira mais simples apresentada na

seção 3.4. O tabuleiro é uma estrutura do tipo *BOARD* implementada como um vetor de 32 elementos do tipo *BOARDVALUES*: cada elemento do vetor representa uma casa do tabuleiro e cada casa do tabuleiro possui um dos valores presentes em *BOARDVALUES* (*EMPTY*, *BLACKMAN*, *REDMAN*, *BLACKKING*, *REDKING*).

O módulo de mapeamento do *NeuroDraughts* recebe a representação interna do tabuleiro como entrada e utiliza um conjunto de características do domínio de damas para produzir sua saída. A utilização de um conjunto de características para treinar um jogador de damas foi proposta por Samuel (SAMUEL, 1959), (SAMUEL, 1967) com intuito de prover medidas numéricas para melhor representar as diversas propriedades das posições das peças sobre um tabuleiro. Samuel implementou 26 características e várias delas resultaram de análises de jogos de especialistas humanos.

Baseando-se em Samuel, Lynch implementou 12 características para mapear o tabuleiro do jogo no *NeuroDraughts* (LYNCH, 1997) (LYNCH; GRIFFITH, 1997). Cada característica tem um valor absoluto que representa a sua medida analítica sobre um determinado estado do tabuleiro. Tal valor absoluto é convertido (conversão numérica entre as bases decimal e binária) em bits significativos que, em conjunto com os demais bits das outras características presentes no mapeamento, constituem a sequência binária de saída do módulo de mapeamento e entrada na rede neural. As características implementadas no *NeuroDraughts*, juntamente com as quantidades de bits significativos, podem ser vistas na figura 18.

#### 4.1.2 Cálculo da Predição e Escolha da Melhor Ação

O cálculo das predições é efetuado através de uma rede neural artificial acíclica com 3 camadas. O número de neurônios na camada de entrada da rede varia de acordo com o número de características e bits significativos presentes no módulo de mapeamento. A camada oculta é formada por 20 neurônios fixos e a camada de saída por um único neurônio. A arquitetura geral da rede neural é mostrada na figura 19: cada um dos neurônios da rede está conectado a todos os outros das camadas subsequentes, um termo *bias* é utilizado para todos os neurônios da camada oculta e outro para o neurônio da camada de saída.

Uma predição  $P_t$  associada a um estado do tabuleiro  $S_t$  é calculada da seguinte maneira:

1. A representação interna do tabuleiro  $S_t$  é mapeada na entrada da rede neural através

| CARACTERÍSTICAS          | DESCRIÇÃO FUNCIONAL  | BITS |
|--------------------------|--|------|
| <i>PieceAdvantage</i>    | Contagem de peças em vantagem para o jogador preto.  | 4    |
| <i>PieceDisadvantage</i> | Contagem de peças em desvantagem para o jogador preto.   | 4    |
| <i>PieceThreat</i>       | Total de peças pretas que estão sob ameaça.  | 3    |
| <i>PieceTake</i>         | Total de peças vermelhas que estão sob ameaça de peças pretas.   | 3    |
| <i>Advancement</i>       | Total de peças pretas que estão na 5ª e 6ª linha do tabuleiro menos as peças que estão na 3ª e 4ª linha. | 3    |
| <i>DoubleDiagonal</i>    | Total de peças pretas que estão na diagonal dupla do tabuleiro.  | 4    |
| <i>Backrowbridge</i>     | Se existe peças pretas nos quadrados 1 e 3 e se não existem reis vermelhos no tabuleiro.                 | 1    |
| <i>Centrecontrol</i>     | Total de peças pretas no centro do tabuleiro.  | 3    |
| <i>XCentrecontrol</i>    | Total de quadrados no centro do tabuleiro onde tem peças vermelhas ou que elas podem mover.              | 3    |
| <i>TotalMobility</i>     | Total de quadrados vazios para onde as peças vermelhas podem mover.                                      | 4    |
| <i>Exposure</i>          | Total de peças pretas que são rodeadas por quadrados vazios em diagonal.                                 | 3    |
| <i>KingCentreControl</i> | Total de reis pretos no centro do tabuleiro.   | 3    |

Figura 18: Conjunto de características implementadas pelo *NeuroDraughts*.

do módulo de mapeamento (figura 16);

2. Calcula-se o campo local induzido  $in_j^{(l)}$  para o neurônio  $j$  (valor de entrada para o neurônio  $j$ ) na camada  $l$ , para  $1 \leq l \leq 2$ , da seguinte forma:

$$in_j^{(l)} = \begin{cases} \sum_{i=0}^{m_{(l-1)}} w_{ij}^{(l-1)} \cdot a_i^{(l-1)}, & \text{para neurônio } j \text{ na camada } l=1 \\ \sum_{i=0}^{m_{(l-1)}} w_{ij}^{(l-1)} \cdot a_i^{(l-1)} + \sum_{i=0}^{m_{(l-2)}} w_{ij}^{(l-2)} \cdot a_i^{(l-2)}, & \text{para neurônio } j \text{ na camada } l=2 \end{cases}$$

onde  $m_l$  representa o número de neurônios na camada  $l$ ;  $a_i^l$  é o sinal de saída do neurônio  $i$  na camada  $l$ ; e  $w_{ij}^l$  é o peso sináptico da conexão de um neurônio  $i$  da camada  $l$  com o neurônio  $j$  das camadas posteriores à camada  $l$ . Para as camadas ocultas ( $l = 1$ ) e de saída ( $l = 2$ ) sendo  $i = 0$ , tem-se que  $a_0^{(l-1)} = +1$  e  $w_{0j}^{(l-1)}$  é o peso do bias aplicado ao neurônio  $j$  na camada  $l$ ;

3. Obtido o campo local induzido, o sinal de saída do neurônio  $j$  na camada  $l$ , para  $1 \leq l \leq 2$ , é dado por  $a_j^{(l)} = g_j(in_j^{(l)})$ , onde  $g_j(x)$  é a função de ativação tangente hiperbólica definida por  $g(x) = \frac{2}{(1+e^{(-2x)})} - 1$ ;

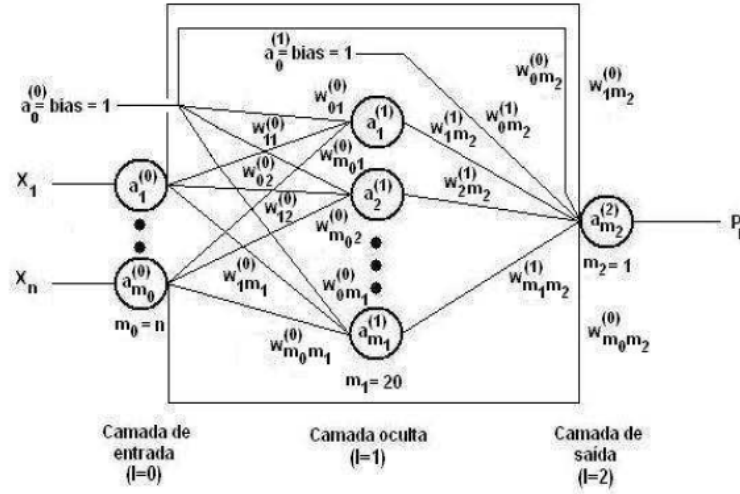


Figura 19: Rede Neural utilizada pelo *NeuroDraughts*.

4. A predição retornada pela rede neural para o estado  $S_t$  é  $a_j^{(2)} = a_{m_2}^{(2)} = P_t$ .

Com o uso da função tangente hiperbólica, os estados do tabuleiro receberão predições no intervalo  $(-1, +1)$ . Assim, os estados que receberem predições próximas de  $+1$  serão considerados convergentes para vitória. Da mesma forma, os estados do tabuleiro que receberem predições próximas de  $-1$  serão considerados convergentes para derrota.

O problema de escolher a melhor ação a ser executada pelo jogador *NeuroDraughts* é solucionado com a utilização de uma rede neural artificial através dos cálculos das predições e do algoritmo *minimax*. O *minimax* recebe como parâmetro de entrada o estado corrente do tabuleiro do jogo e a profundidade máxima de busca, expande uma árvore de busca e envia suas folhas para o módulo de mapeamento que gera a seqüência binária de entrada na rede neural. O pseudo-código do algoritmo *minimax* será apresentado na seção 4.3.1 e seu funcionamento será detalhado com o exemplo da figura 22.

### 4.1.3 Reajuste de Pesos da Rede Neural Multi-Camadas

Os pesos da rede neural são reajustados durante a fase do treinamento por *self-play com clonagem* (seção 4.1.4), através da técnica de aprendizagem por diferenças temporais  $TD(\lambda)$ . O agente jogador seleciona a melhor ação  $M_t$  a ser executada, a partir de um estado  $S_t$ , com o auxílio do procedimento *minimax* e dos pesos atuais da rede neural. O estado  $S_{t+1}$  resulta da ação  $M_t$  sobre o estado  $S_t$ .

A partir de então, o estado  $S_{t+1}$  é mapeado na entrada da rede neural e tem sua predição  $P_{t+1}$  calculada (a predição é o valor de saída no neurônio da última camada da

rede neural). Os pesos da rede neural são reajustados com base na diferença entre  $P_{t+1}$  e a predição  $P_t$ , calculada anteriormente para o estado  $S_t$ .

Após o fim de cada partida de treino, um reforço final é fornecido pelo ambiente informando o resultado obtido pelo agente jogador em função da seqüência de ações que executou (+1 para vitória, -1 para derrota e um valor próximo de 0 para empate).

Formalmente, o cálculo do reajuste dos pesos é definido pela equação do método TD( $\lambda$ ) de Sutton (SUTTON, 1988):

$$\begin{aligned} w_{ij}^{(l)}(t) &= w_{ij}^{(l)}(t-1) + \Delta w_{ij}^{(l)}(t) \\ &= w_{ij}^{(l)}(t-1) + \alpha^{(l)} \cdot (P_{t+1} - P_t) \cdot \sum_{k=1}^t \lambda^{t-k} \nabla_w P_k \\ &= w_{ij}^{(l)}(t-1) + \alpha^{(l)} \cdot (P_{t+1} - P_t) \cdot \text{elig}_{ij}^{(l)}(t), \end{aligned} \quad (4.1)$$

- $\alpha^{(l)}$  é o parâmetro da taxa de aprendizagem na camada  $l$  (Lynch (LYNCH, 1997) (LYNCH; GRIFFITH, 1997) utilizou uma mesma taxa de aprendizagem para todas as conexões sinápticas de uma mesma camada  $l$ );
- $w_{ij}^{(l)}(t)$  representa o peso sináptico da conexão entre a saída do neurônio  $i$  da camada  $l$  e a entrada do neurônio  $j$  da camada  $(l+1)$ , no instante temporal  $t$ . A correção aplicada a este peso no instante temporal  $t$  é representada por  $\Delta w_{ij}^{(l)}(t)$ ;
- O termo  $\text{elig}_{ij}^{(l)}(t)$  é único para cada peso sináptico  $w_{ij}^{(l)}(t)$  da rede neural e representa o traço de elegibilidade das predições calculadas pela rede para os estados resultantes de ações executadas pelo agente desde o instante temporal 1 do jogo até o instante temporal  $t$ ;
- $\nabla_w P_k$  representa a derivada parcial de  $P_k$  em relação aos pesos da rede no instante  $k$ . Cada predição  $P_k$  é uma função dependente do vetor de entrada  $\overrightarrow{X(k)}$  e do vetor de pesos  $\overrightarrow{W(k)}$  da rede neural no instante temporal  $k$ ;
- O termo  $\lambda^{t-k}$ , para  $0 \leq \lambda \leq 1$ , tem o papel de dar uma “pesagem exponencial” para a taxa de variação das predições calculadas a  $k$  passos anteriores de  $t$ . Quando maior for  $\lambda$ , maior o impacto dos reajustes anteriores ao instante temporal  $t$  sobre a atualização dos pesos  $w_{ij}^{(l)}(t)$ .

Neto e Julia (NETO, 2007) (NETO; JULIA, 2007b) (NETO; JULIA, 2007a) descrevem o processo de reajuste de pesos por diferenças temporais TD( $\lambda$ ) através das seguintes etapas:

1. O vetor inicial  $\overrightarrow{W(k)}$  de pesos é gerado aleatoriamente;
2. As eligibilidades associadas aos pesos da rede são inicialmente nulas;
3. Dada duas predições sucessivas  $P_t$  e  $P_{t+1}$ , referentes a dois estados consecutivos  $S_t$  e  $S_{t+1}$ , calculadas em consequência de ações executadas pelo agente durante o jogo, define-se o sinal de erro através da equação:

$$e(t) = (\gamma P_{t+1} - P_t)$$

onde o parâmetro  $\gamma$  é uma constante de compensação da predição  $P_{t+1}$  em relação a predição  $P_t$ ;

4. Cada eligibilidade  $elig_{ij}^{(l)}(t)$  está vinculada a um peso sináptico  $w_{ij}^{(l)}(t)$  correspondente. Assim, as eligibilidades vinculadas aos pesos da camada  $l$ , para  $0 \leq l \leq 1$ , no instante temporal  $t$  ( $elig_{ij}^{(l)}(t)$ ), são calculadas observando as equações dispostas a seguir:

- Para os pesos associados as ligações diretas entre as camadas de entrada ( $l = 0$ ) e saída ( $l = 2$ ):

$$elig_{ij}^{(l)}(t) = \lambda \cdot elig_{ij}^{(l)}(t-1) + g'(P_t) \cdot a_i^{(l)}$$

onde  $\lambda$  tem o papel de dar uma “pesagem exponencial” para a taxa de variação das predições calculadas a  $k$  passos anteriores de  $t$ ;  $a_i^{(l)}$  é o sinal de saída do neurônio  $i$  na camada  $l$ ;  $g'(x) = (1 - x^2)$  representa a derivada da função de ativação (tangente hiperbólica) utilizada por Lynch (LYNCH, 1997) (LYNCH; GRIFFITH, 1997).

- Para os pesos associados as ligações entre as camadas de entrada ( $l = 0$ ) e oculta ( $l = 1$ ):

$$elig_{ij}^{(l)}(t) = \lambda \cdot elig_{ij}^{(l)}(t-1) + g'(P_t) \cdot w_{ij}^{(l)}(t) \cdot g'(a_j^{(l+1)}) \cdot a_i^{(l)}$$

onde  $a_j^{(l+1)}$  é o sinal de saída do neurônio  $j$  na camada oculta ( $l + 1$ ).

- Para os pesos associados as ligações entre as camadas oculta ( $l = 1$ ) e de saída ( $l = 2$ ):

$$elig_{ij}^{(l)}(t) = \lambda \cdot elig_{ij}^{(l)}(t-1) + g'(P_t) \cdot a_i^{(l)}$$

5. Calculados as eligibilidades, a correção dos pesos  $w_{ij}^{(l)}(t)$  da camada  $l$ , para  $0 \leq l \leq 1$ , é efetuada através da seguinte equação:

$$\Delta w_{ij}^{(l)}(t) = \alpha^{(l)} \cdot e(t) \cdot elig_{ij}^{(l)}(t), \quad (4.2)$$

onde o parâmetro de aprendizagem  $\alpha^{(l)}$  é definido por Lynch (LYNCH, 1997) (LYNCH; GRIFFITH, 1997) como:

$$\alpha^{(l)} = \begin{cases} \frac{1}{n}, & \text{para } l=0 \\ \frac{1}{20}, & \text{para } l=1 \end{cases}$$

onde  $n$  representa o número de neurônios na camada de entrada da rede neural.

6. Existe um problema típico associado ao uso de redes neurais no qual a convergência não é garantida para o melhor valor (NETO, 2007). Lynch (LYNCH, 1997) (LYNCH; GRIFFITH, 1997) utilizou o termo momento  $\mu$  para tentar solucionar este tipo de problema. Para isso, ele empregou uma checagem de direção na equação 4.1, ou seja, o termo momento  $\mu$  é aplicado somente quando a correção do peso atual  $\Delta w_{ij}^{(l)}(t)$  e a correção anterior  $\Delta w_{ij}^{(l)}(t-1)$  estiverem na mesma direção. Portanto, a equação final TD( $\lambda$ ) utilizada por Lynch para calcular o reajuste dos pesos da rede neural na camada  $l$ , para  $0 \leq l \leq 1$ , é definida por:

$$w_{ij}^{(l)}(t) = w_{ij}^{(l)}(t-1) + \Delta w_{ij}^{(l)}(t); \quad (4.3)$$

onde  $\Delta w_{ij}^{(l)}(t)$  é obtido nas seguintes etapas:

- Calcule  $\Delta w_{ij}^{(l)}(t)$  através da equação 4.1;
- Se  $(\Delta w_{ij}^{(l)}(t) > 0 \text{ e } \Delta w_{ij}^{(l)}(t-1) > 0)$  ou  $(\Delta w_{ij}^{(l)}(t) < 0 \text{ e } \Delta w_{ij}^{(l)}(t-1) < 0)$  então faça:

$$\Delta w_{ij}^{(l)}(t) = \Delta w_{ij}^{(l)}(t) + \mu \cdot \Delta w_{ij}^{(l)}(t-1);$$

#### 4.1.4 Estratégia de Treino por *Self-Play com Clonagem*

A idéia básica do treinamento por *self-play com clonagem* é treinar um jogador através de vários jogos contra uma cópia de si próprio. A partir de um certo momento, o oponente com menor nível de desempenho é descartado e o oponente com maior nível de desempenho é clonado para que outros jogos sejam realizados e o novo jogador, com o maior nível de desempenho, seja selecionado para clonagem. O processo se repete até que um jogador com alto nível de desempenho seja obtido, conforme as seguintes etapas:

1. Os pesos de uma rede neural *opp1* são gerados aleatoriamente;
2. A rede *opp1* é clonada para gerar a rede *opp1-clone*;

3. As redes começam uma sequência de  $n$  jogos de treinamento. Somente os pesos da rede *opp1* são ajustados durante os  $n$  jogos.
4. Ao final dos  $n$  jogos de treinamento, um torneio de 2 jogos é realizado para verificar qual das redes é melhor. Caso o nível de desempenho da rede *opp1* supere o nível da rede *opp1-clone*, uma cópia dos pesos da rede *opp1* para a rede *opp1-clone* é realizada. Caso contrário, não se copia os pesos.
5. Vá para etapa 3 e execute uma nova sessão de  $n$  jogos de treinamento entre a rede *opp1* e o seu último clone *opp1-clone*. Repita o processo até que um número máximo de jogos de treinamento (parâmetro) seja alcançado.

Em cada um dos dois jogos do torneio, efetuado com objetivo de identificar qual a rede neural possui maior nível de desempenho, *opp1* representa um dos jogadores possíveis (jogador preto no primeiro jogo e vermelho no segundo).

## 4.2 Fluxo de Aprendizagem do *VisionDraughts*

O fluxo de aprendizagem do *VisionDraughts* é apresentado na figura 20.

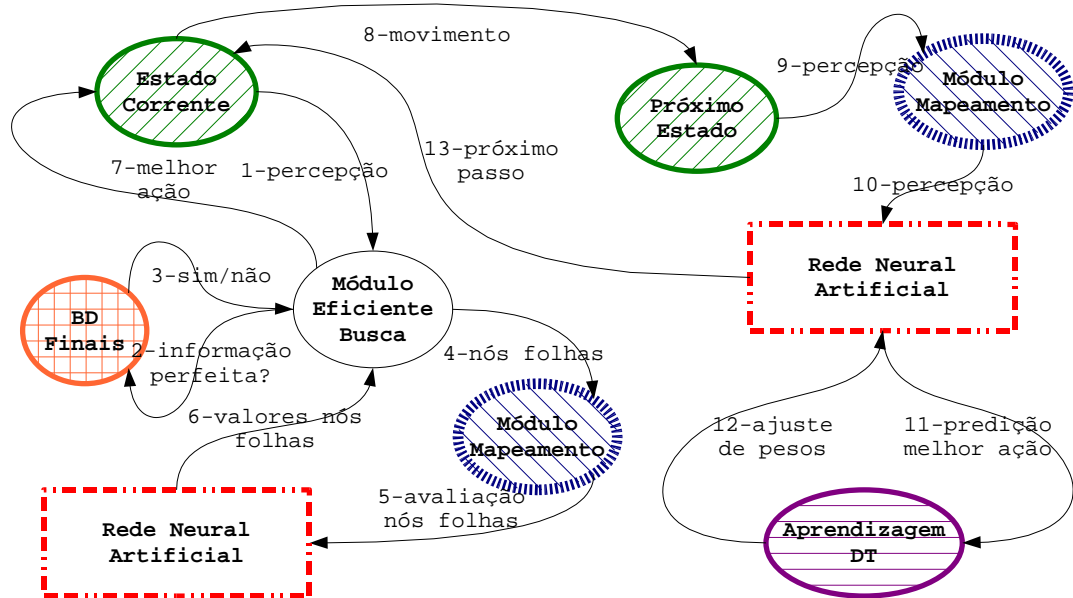


Figura 20: Fluxo de aprendizagem do *VisionDraughts*: um sistema de aprendizagem de jogos de damas.

Veja que os módulos são bastante parecidos com os do sistema *NeuroDraughts*. As exceções são as bases de dados de finais de jogos e o módulo de busca. Enquanto o *NeuroDraughts* utiliza o módulo simples de busca minimax, o *VisionDraughts* conta com



um eficiente mecanismo de busca alfa-beta com tabelas de transposição e aprofundamento iterativo (explicado com detalhes nas seções seguintes), além de utilizar as informações perfeitas (vitória, derrota ou empate) constantes das bases de dados finais. Os módulos do *VisionDraughts* operam conforme descrito a seguir:

1. **Rede Neural Artificial:** o jogador automático de damas é representado por uma rede neural multi-camadas. A rede neural recebe, em sua camada de entrada, um estado do tabuleiro do jogo e devolve, em seu único neurônio na camada de saída, um valor real compreendido entre -1.0 e +1.0. Tal valor representa como o estado do tabuleiro presente na camada de entrada da rede é avaliado do ponto de vista do jogador automático;
2. **Módulo Mapeamento:** O tabuleiro do jogo de damas é implementado como um vetor de 32 elementos, sendo que cada elemento do vetor representa uma casa do tabuleiro (representação simples e direta mostrada na seção 3.4). O mapeamento utilizado pelo sistema *VisionDraughts* é o NET-FEATUREMAP (representação por características de Lynch). Este módulo é responsável por fazer o mapeamento do vetor de 32 elementos na camada de entrada da rede neural, através de características próprias do domínio de damas;
3. **Módulo Eficiente Busca:** a seleção da melhor ação a ser executada pelo agente em função do estado corrente do tabuleiro é feita através de um eficiente mecanismo de busca em árvores de jogos. Este módulo expande uma árvore em busca da melhor ação e os nós presentes na camada mais profunda (folhas) são mapeados, através do módulo 2, na entrada da rede neural do módulo 1;
4. **Aprendizagem DT:** o ajuste dos pesos da rede neural é feito através do método de diferenças temporais (aprendizagem por reforço). Este processo é o mesmo do sistema *NeuroDraughts* e é responsável pela aquisição de conhecimento do sistema;
5. **BD Finais:** o uso de bases de dados de finais de jogos (SCHAEFFER; LAKE, 1996), (LAKE; SCHAEFFER; LU, 1994) reduz o número de movimentos necessários, a partir da jogada inicial, para se alcançar posições com valor teórico definido (vitória, derrota ou empate). O *VisionDraughts* tem acesso às bases de dados disponibilizadas em (SCHAEFFER et al., 2008) e à biblioteca com funções de acesso disponibilizada em (GILBERT, 2008a).

Quase todos os programas para jogos de tabuleiro possuem a mesma arquitetura básica (PATIST; WIERING, 2004). Suas diferentes partes precisam gerar os próximos movimentos

legais, avaliá-los e retornar o melhor movimento a ser executado pelo agente jogador. Assim, o *VisionDraughts* foi projetado adicionando os módulos 3 e 5 ao *NeuroDraughts*, de maneira a otimizar seu procedimento de busca e sua função de avaliação.

Note que, para o problema de treinar uma rede neural para jogar damas utilizando o método TD( $\lambda$ ) e algum tipo de mapeamento do tabuleiro, a escolha da melhor ação está vinculada a dois fatores fundamentais:

1. **Profundidade de busca:** indica a capacidade que o agente jogador possui de analisar combinações futuras de jogadas. Assim sendo, o *VisionDraughts* cria um mecanismo eficiente de busca em árvores de jogos de modo a otimizar o processo de escolha da melhor ação. Em outras palavras, o *VisionDraughts* utiliza um módulo que reduz o número de estados avaliados do tabuleiro, ignorando ramificações da árvore que não contribuem mais para o resultado final, e permite aumentar, **substancialmente**, a profundidade de busca utilizada.
2. **Função de avaliação:** avalia uma determinada posição do tabuleiro do jogo. Neto e Julia (NETO, 2007) (NETO; JULIA, 2007b) (NETO; JULIA, 2007a) observaram a ocorrência do *problema do loop* de final de jogo no *NeuroDraughts* e, tal fato, indica certa imprecisão da função de avaliação. Assim sendo, o *VisionDraughts* utiliza bases de dados de finais de jogos para reduzir a ocorrência do *problema do loop* e aprimorar a função de avaliação.

## 4.3 O Eficiente Mecanismo de Busca do *VisionDraughts*

### 4.3.1 O Algoritmo Alfa-Beta

O jogo de damas pode ser pensado como uma árvore de possíveis estados futuros do tabuleiro do jogo, isto é, uma árvore representando uma evolução dos próximos movimentos disponíveis de acordo com as regras do jogo. Por exemplo, na figura 21, o estado corrente do jogo é mostrado no tabuleiro presente na raiz da árvore e passará a ser chamado de estado  $S_0$ . Em geral, a raiz da árvore tem vários filhos, representando todas as possíveis jogadas permitidas a partir do estado corrente. Por exemplo, o estado  $S_0$  possui três filhos que passarão a ser chamados de  $S_1$ ,  $S_2$  e  $S_3$ . O primeiro filho  $S_1$  do tabuleiro raiz, por sua vez, possui outros três filhos que se chamarão  $S_{11}$ ,  $S_{12}$  e  $S_{13}$ . Da mesma forma, o segundo filho  $S_2$  do tabuleiro raiz possui um único filho que se chamará  $S_{21}$  e o terceiro filho  $S_3$  do tabuleiro raiz possui outros três filhos que se chamarão  $S_{31}$ ,  $S_{32}$  e  $S_{33}$ .

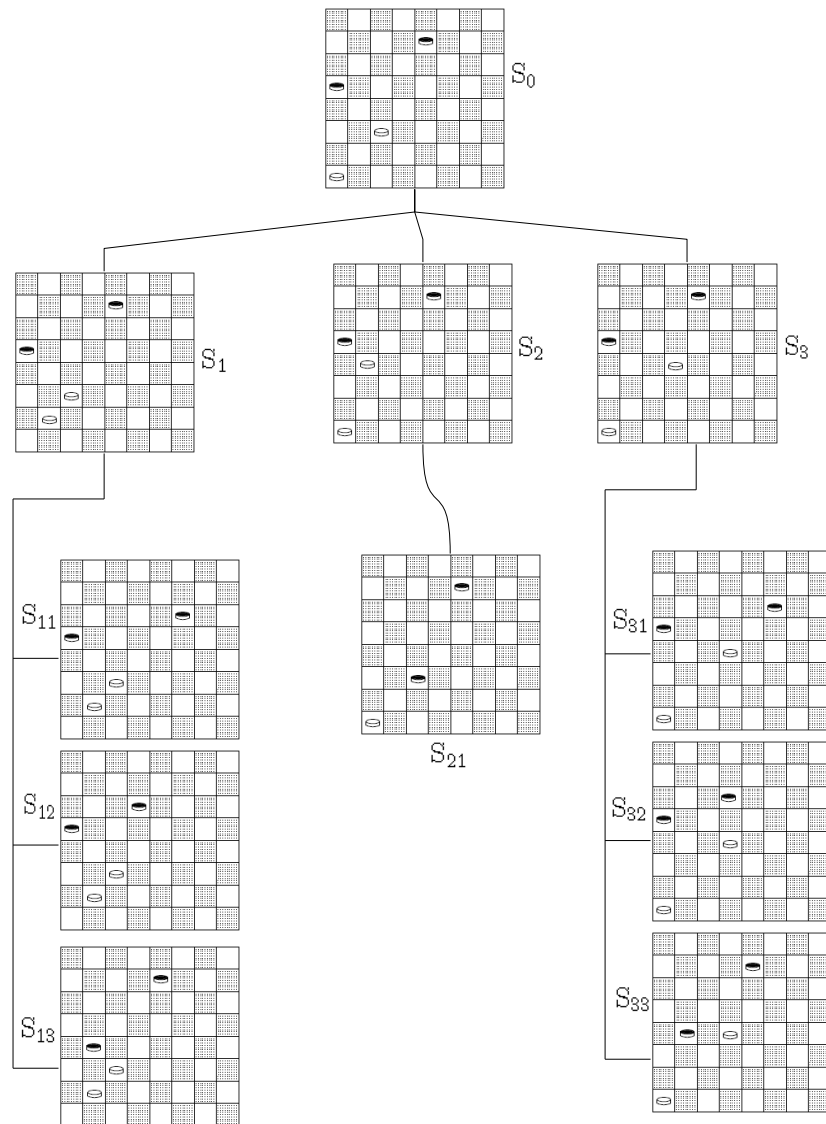


Figura 21: Evolução do jogo de damas com todos os movimentos possíveis a partir do tabuleiro raiz com dois níveis de profundidade.

Veja que a evolução do jogo de damas mostrada na figura 21 está limitada a uma profundidade de 2 níveis. A profundidade máxima de evolução, conhecida também como profundidade máxima de busca, é determinada baseando-se na quantidade de recursos computacionais disponíveis, ou seja, caso os recursos computacionais disponíveis fossem ilimitados, poder-se-ia iniciar uma busca a partir da posição inicial do jogo e varrer todas as combinações possíveis de jogadas disponíveis até o final da partida. Infelizmente, este não é o caso e, portanto, uma profundidade máxima de busca precisa ser estabelecida. Os estados do tabuleiro presentes na camada mais profunda da árvore de busca são denominados estados folhas ou, simplesmente, folhas da árvore de busca (para simplificar a representação da árvore de busca, cada estado do tabuleiro é representado, de agora em diante, por um simples círculo, chamado nó da árvore de busca).

O *VisionDraughts* utiliza o algoritmo alfa-beta para escolher a melhor ação a ser executada de acordo com o estado corrente do tabuleiro do jogo. O algoritmo alfa-beta pode ser resumido como um procedimento recursivo que escolhe a melhor jogada a ser executada fazendo uma busca em profundidade, da esquerda para a direita, na árvore do jogo (SCHAEFFER; PLAAT, 1996).

Visando simplificar o entendimento do algoritmo alfa-beta, é apresentado o pseudo-código do algoritmo minimax, juntamente com uma descrição sucinta de cada linha. Em seguida, será adicionado o mecanismo de poda alfa-beta no contexto do procedimento minimax.

*Pseudo-código do algoritmo minimax*

```
1. fun minimax(n:node,depth:int,bestmove:move):float =
2.     if leaf(n) or depth=0 return evaluate(n)
3.     if n is a max node
4.         besteval := - infinity
5.         for each child of n
6.             v := minimax (child,d-1,bestmove)
7.             if v > besteval
8.                 besteval:= v
9.                 thebest = bestmove
10.        bestmove = thebest
11.        return besteval
12.    if n is a min node
13.        besteval := + infinity
14.        for each child of n
15.            v := minimax (child,d-1,bestmove)
16.            if v < besteval
17.                besteval:= v
18.                thebest = bestmove
19.        bestmove = thebest
20.        return besteval
```

**linha 1.** O algoritmo minimax recebe um estado do tabuleiro do jogo de damas  $n$ , uma profundidade de busca  $d$  e um parâmetro de saída  $bestmove$  (para armazenar a melhor jogada a ser executada a partir do estado do tabuleiro  $n$ ). Ele retorna a

predição associada ao estado  $n$  na forma de um número real, isto é, retorna qual a avaliação do estado  $n$  do ponto de vista do agente jogador;

**linha 2.** O algoritmo minimax é um procedimento recursivo que escolhe a melhor jogada a ser executada (armazena a melhor jogada no parâmetro de saída *bestmove*) fazendo uma busca em profundidade, da esquerda para a direita, na árvore do jogo. Por se tratar de um procedimento recursivo, exige-se uma condição de parada. A condição de parada do algoritmo verifica se o estado do tabuleiro  $n$  é uma folha da árvore ( $n$  não possui filhos). Nos dois casos, a função *evaluate*( $n$ ) é retornada indicando a predição dada pela rede neural para o estado do tabuleiro  $n$ ;

**linha 3.** O estado do tabuleiro presente na raiz da árvore do jogo montada pelo algoritmo minimax é um nó maximizador e a predição associada a um nó maximizador será igual à maior predição de seus filhos. Os filhos do estado do tabuleiro presente na raiz da árvore do jogo são nós minimizadores e a predição associada a um nó minimizador será igual a menor predição de seus filhos. Assim, os níveis da árvore do jogo alternam entre nós maximizadores e minimizadores e o pedaço de código delimitado por esta linha é executado para os nós maximizadores;

**linha 4.** *besteval* representa a melhor avaliação encontrada para o nó  $n$  até o presente momento. Como  $n$  representa um nó maximizador, inicialmente, o valor de *besteval* é configurado como o maior valor negativo possível, por exemplo, - *infinity*. Note que *besteval* recebe - *infinity* e será incrementado até o máximo valor das predições associadas aos filhos de  $n$ ;

**linha 5.** Para cada um dos filhos de  $n$ , realiza-se os procedimentos das linhas 6 a 9;

**linha 6.** Para cada um dos filhos *child*, do estado do tabuleiro  $n$ , o algoritmo minimax é chamado, recursivamente, com profundidade  $d - 1$ . O valor da predição associada ao filho *child* de  $n$  é armazenado na variável  $v$ ;

**linha 7.** Caso a predição armazenada na variável  $v$  seja maior que *besteval* (melhor avaliação encontrada até o presente momento para  $n$ ), o algoritmo atualiza o valor de *besteval* com  $v$  e o valor de *thebest* (melhor movimento encontrado até o presente momento para  $n$ ) com *bestmove* (BITTENCOURT, 2006);

**linha 10.** Assim que o algoritmo sair do laço da linha 5, *thebest* conterá o melhor movimento a ser executado a partir do estado  $n$ . Então, *bestmove* é atualizado com o valor de *thebest* (BITTENCOURT, 2006);

- linha 11.** Assim que o algoritmo sair do laço da linha 5, *besteval* conterà a maior predição de todos os filhos do estado do tabuleiro  $n$  ( $n$  é maximizador). Então, *besteval* será propagado, ou seja, retornado como valor da predição associada ao estado do tabuleiro  $n$ ;
- linha 12.** O pseudo-código delimitado por esta linha é similar ao delimitado pela linha 3, porém, tratam-se, agora, de nós minimizadores.
- linha 13.** *besteval* representa a melhor avaliação encontrada para o nó  $n$  até o presente momento. Como  $n$  representa um nó minimizador, inicialmente, o valor de *besteval* é configurado como o maior valor positivo possível, por exemplo,  $+infinity$ . Note que *besteval* recebe  $+infinity$  e será decrementado até o mínimo valor das predições associadas aos filhos de  $n$ ;
- linha 14.** Para cada um dos filhos de  $n$ , realizam-se os procedimentos das linhas 15 a 18;
- linha 15.** Para cada um dos filhos *child*, do estado do tabuleiro  $n$ , o algoritmo minimax é chamado, recursivamente, com a profundidade de busca decrementada de uma unidade. O valor da predição associada ao filho *child* de  $n$  é armazenada na variável  $v$ .
- linha 16.** Caso a predição armazenada na variável  $v$  seja menor que *besteval* (melhor avaliação encontrada até o presente momento para  $n$ ), o algoritmo atualiza o valor de *besteval* com  $v$  e o valor de *thebest* (melhor movimento encontrado até o presente momento para  $n$ ) com *bestmove* (BITTENCOURT, 2006);
- linha 19.** Assim que o algoritmo sair do laço da linha 14, *thebest* conterà o melhor movimento a ser executado a partir do estado  $n$ . Então, *bestmove* é atualizado com o valor de *thebest* (BITTENCOURT, 2006);
- linha 20.** Assim que o algoritmo sair do laço da linha 14, *besteval* conterà a menor predição de todos os filhos do estado do tabuleiro  $n$  ( $n$  é minimizador). Então, *besteval* será propagado, ou seja, retornado como valor da predição associada ao estado do tabuleiro  $n$ .

Considerando o exemplo de árvore do jogo de damas mostrado na figura 22 e o pseudo-código do algoritmo minimax, veja os seguintes casos:

1. Os estados  $S_{11}$ ,  $S_{12}$ ,  $S_{13}$ ,  $S_{21}$ ,  $S_{31}$ ,  $S_{32}$  e  $S_{33}$  são apresentados diretamente à rede neural, através da função  $evaluate(n)$ , pois são estados folhas.

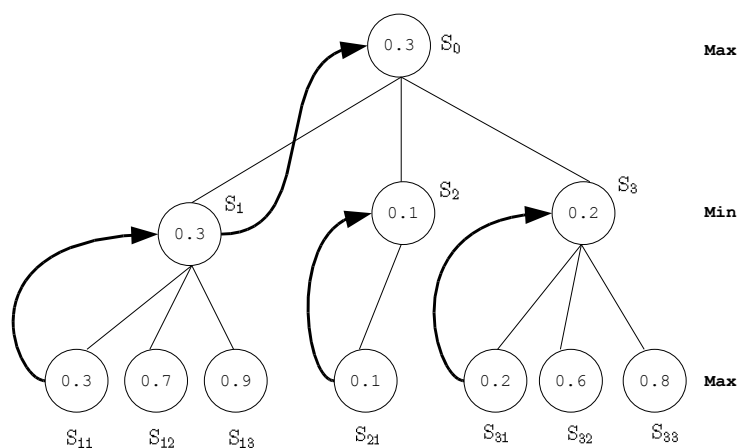


Figura 22: Exemplo de árvore do jogo de damas criada pelo algoritmo minimax.

2. O estado  $S_1$  está presente em um nível de minimização. Portanto, recebe o valor de predição associada ao estado  $S_{11}$  que possui a menor predição dentre os seus filhos.
3. O estado  $S_2$  recebe a predição associada à  $S_{21}$  que é seu único filho.
4. O estado  $S_3$ , também está presente em um nível de minimização. Portanto, recebe o valor de predição associada ao estado  $S_{31}$  que possui a menor predição dentre os seus filhos.
5. O estado  $S_0$  está presente em um nível de maximização. Portanto, recebe o valor de predição associada ao estado  $S_1$  que possui a maior predição dentre os seus filhos.

O algoritmo minimax examina mais estados do tabuleiro que o necessário e, neste sentido, o mecanismo de poda alfa-beta elimina seções da árvore de busca que, definitivamente, não podem conter a melhor ação a ser executada pelo agente jogador. Abaixo, é apresentado o pseudo-código do algoritmo alfa-beta seguido de uma explanação dos pontos em que ele difere do minimax.

#### *Pseudo-código do algoritmo alfa-beta*

1. `fun alfaBeta(n:node,depth:int,min:int,max:int,bestmove:move):float =`
2.   `if leaf(n) or depth=0 return evaluate(n)`
3.   `if n is a max node`
4.     `besteval := min`
5.     `for each child of n`
6.       `v := alfaBeta(child,d-1,besteval,max,bestmove)`
7.       `if v > besteval`

```

8.         besteval := v
9.         thebest = bestmove
10.        if besteval >= max then return max
11.        bestmove = thebest
12.        return besteval
13.  if n is a min node
14.    besteval := max
15.    for each child of n
16.      v := alfaBeta(child,d-1,min,besteval,bestmove)
17.      if v < besteval
18.        besteval := v
19.        thebest = bestmove
20.      if besteval <= min then return min
21.    bestmove = thebest
22.    return besteval

```

**linha 1.** Enquanto o algoritmo minimax recebe um estado do tabuleiro do jogo de damas,  $n$ , uma profundidade de busca,  $d$ , e um parâmetro de saída para armazenar a melhor ação a ser executada,  $bestmove$ , a partir de  $n$ , o algoritmo alfa-beta recebe, ainda, um intervalo de busca delimitado pelos parâmetros  $min$  (representando o limite inferior do intervalo de busca, sendo também conhecido como alfa) e  $max$  (representando o limite superior do intervalo de busca, sendo também conhecido como beta). A predição de retorno do algoritmo alfa-beta para o estado do tabuleiro presente na raiz da árvore de busca é exatamente igual à predição retornada pelo algoritmo minimax.

**linha 6.** Para cada um dos filhos  $child$ , do estado do tabuleiro  $n$ , o algoritmo alfa-beta é chamado, recursivamente, com profundidade  $d - 1$ . O valor da predição associada ao filho  $child$  de  $n$  é armazenado na variável  $v$ . Note, porém, que um novo intervalo de busca será utilizado para a chamada recursiva: em vez de utilizar o intervalo  $[min, max]$ , será utilizado o intervalo  $[besteval, max]$ . Tal fato significa que, no nível de maximização, sempre que ocorrer atualização de um dos limites do intervalo de busca, ela será um incremento no limite inferior (isso acontece sempre que a predição  $v$  calculada para  $child$  superar o valor de  $besteval$  na linha 7).

**linha 10.** Se acontecer de a predição armazenada na variável  $besteval$  ultrapassar o limite superior do intervalo de busca ( $besteval \geq max$ ), o algoritmo retornará, imediata-



mente, o limite superior do intervalo de busca ( $max$ ) como predição associada ao estado do tabuleiro  $n$ . Tal fato expressa a idéia de que o intervalo de busca deve ser respeitado, ou seja, a predição associada ao estado do tabuleiro  $n$  deve estar contida dentro do intervalo de busca passado como parâmetro.

**linha 13.** Para cada um dos filhos *child*, do estado do tabuleiro  $n$ , o algoritmo minimax é chamado, recursivamente, com profundidade  $d - 1$ . O valor da predição associada ao filho *child* de  $n$  é armazenado na variável  $v$ . Note, porém, que um novo intervalo de busca será utilizado para a chamada recursiva: em vez de utilizar o intervalo  $[min, max]$ , será utilizado o intervalo  $[min, besteval]$ . Tal fato significa que, no nível de minimização, sempre que ocorrer atualização de um dos limites do intervalo de busca, ela será um decremento no limite superior (isso acontece sempre que a predição  $v$  calculada para *child* for inferior ao valor de *besteval* na linha 17).

**linha 20.** Se acontecer de a predição armazenada na variável *besteval* tornar-se menor que o limite inferior do intervalo de busca ( $besteval \leq min$ ), o algoritmo retornará, imediatamente, o limite inferior do intervalo de busca ( $min$ ) como predição associada ao estado do tabuleiro  $n$ . Tal fato expressa a idéia de que o intervalo de busca deve ser respeitado, ou seja, a predição associada ao estado do tabuleiro  $n$  deve estar contida dentro do intervalo de busca passado como parâmetro.

Considerando o exemplo mostrado na figura 23 e o pseudo-código do algoritmo alfa-beta, veja os seguintes casos:

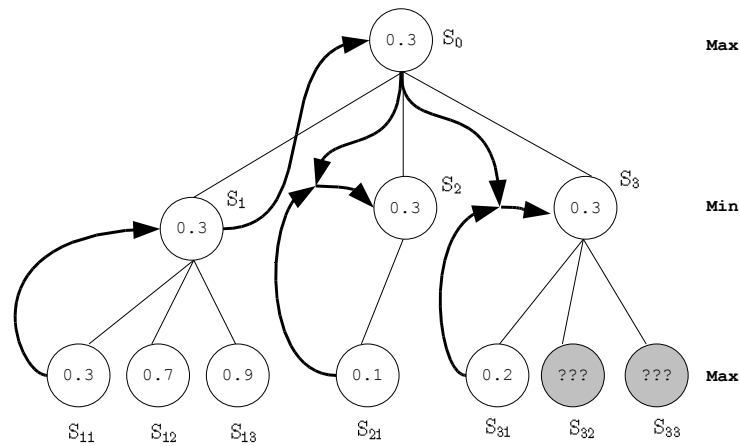


Figura 23: Exemplo de árvore do jogo de damas criada pelo algoritmo alfa-beta.

1. O estado do tabuleiro presente na raiz da árvore do jogo  $S_0$ , é apresentado para o algoritmo alfa-beta com um intervalo de busca configurado, inicialmente, para  $[-\infty, +\infty]$ ;

2. O estado do tabuleiro  $S_1$ , primeiro filho de  $S_0$ , é apresentado para o algoritmo alfa-beta de maneira recursiva e com um intervalo de busca, também, configurado para  $[-\infty, +\infty]$ ;
3. O estado do tabuleiro  $S_{11}$ , primeiro filho de  $S_1$ , é apresentado para o algoritmo alfa-beta de maneira recursiva e com um intervalo de busca, também, configurado para  $[-\infty, +\infty]$ ;
4. A rede neural retorna, através da função  $evaluate(n)$ , uma predição de 0.3 para o estado  $S_{11}$ . Neste momento, o intervalo de busca do estado do tabuleiro  $S_1$  é alterado para o intervalo  $[-\infty, +0.3]$  (note que o estado  $S_1$  é representado por um nó minimizador e, logo, as alterações de intervalo acontecem no parâmetro do limite superior);
5. Ao pesquisar os demais filhos do estado  $S_1$ , seu intervalo de busca não é alterado. Uma vez que todos os filhos de  $S_1$  foram pesquisados pelo alfa-beta, sua predição é definida como 0.3;
6. O valor 0.3 é propagado do estado  $S_1$  para o estado  $S_0$ . Neste momento, o intervalo de busca do estado do tabuleiro  $S_0$  é alterado para o intervalo  $[0.3, +\infty]$  (note que o estado  $S_0$  é representado por um nó maximizador e, logo, as alterações de intervalo acontecem no parâmetro do limite inferior);
7. O estado do tabuleiro  $S_2$ , segundo filho de  $S_0$ , é apresentado para o algoritmo alfa-beta de maneira recursiva e com um intervalo de busca configurado para  $[0.3, +\infty]$ ;
8. O estado do tabuleiro  $S_{21}$ , primeiro filho de  $S_2$ , é apresentado para o algoritmo alfa-beta de maneira recursiva e com um intervalo de busca, também, configurado para  $[0.3, +\infty]$ ;
9. A rede neural retorna, através da função  $evaluate(n)$ , uma predição de 0.1 para o estado  $S_{21}$ . No entanto, o valor 0.1 encontra-se abaixo do valor mínimo passado como parâmetro para o intervalo de busca do alfa-beta ( $[0.3, +\infty]$ ). De acordo com o pseudo-código apresentado para o algoritmo alfa-beta, o intervalo de busca deve ser respeitado e, neste caso, o valor 0.3 deve ser associado como predição do estado  $S_2$ ;
10. O estado do tabuleiro  $S_3$ , terceiro filho de  $S_0$ , é apresentado para o algoritmo alfa-beta de maneira recursiva e com um intervalo de busca configurado para  $[0.3, +\infty]$ ;

11. O estado do tabuleiro  $S_{31}$ , primeiro filho de  $S_3$ , é apresentado para o algoritmo alfa-beta de maneira recursiva e com um intervalo de busca, também, configurado para  $[0.3, +\infty]$ ;
12. A rede neural retorna, através da função  $evaluate(n)$ , uma predição de 0.2 para o estado  $S_{31}$ . No entanto, o valor 0.2 encontra-se abaixo do valor mínimo passado como parâmetro para o intervalo de busca do alfa-beta ( $[0.3, +\infty]$ ). De acordo com o pseudo-código apresentado para o algoritmo alfa-beta, o intervalo de busca deve ser respeitado e, neste caso, o valor 0.3 deve ser associado como predição do estado  $S_2$ ;
13. A linha 15 do pseudo-código alfa-beta garante o retorno imediato do algoritmo alfa-beta com o valor 0.3 associado como predição do estado  $S_3$ . Assim, os outros dois filhos  $S_{32}$  e  $S_{33}$  de  $S_3$  não precisam ser analisados (poda alfa). Logo que o valor 0.2 é encontrado para o primeiro filho de  $S_3$ , a variável *besteval*, na linha 14, é configurada como 0.2. O valor de *min*, na linha 15, é igual a 0.3. Portanto, como  $0.2 \leq 0.3$ , retorna-se o valor 0.3.

Portanto, com base no exemplo mostrado na figura 23 e no pseudo-código do algoritmo alfa-beta, é possível abstrair a seguinte idéia para o mecanismo de poda: A avaliação dos filhos de um nó de minimização pode ser interrompida tão logo a predição calculada para um de seus filhos seja menor que o parâmetro alfa (poda alfa). Similarmente, a avaliação dos filhos de um nó de maximização pode ser interrompida tão logo a predição calculada para um de seus filhos seja maior que o parâmetro beta (poda beta).

O primeiro grande avanço do *VisionDraughts* em relação ao *NeuroDraughts* é conseguido com o algoritmo alfa-beta. Conforme demonstrado na seção 5.1, o tempo de execução necessário para que o *VisionDraughts* encontre a melhor ação a ser executada é inferior à 10% do tempo exigido pelo algoritmo minimax do *NeuroDraughts*.

A fim de proporcionar ainda mais eficiência ao mecanismo de busca, o *VisionDraughts* utiliza o algoritmo alfa-beta em conjunto com tabelas de transposição. As subseções seguintes introduzem o conceito de transposição e descrevem o funcionamento de uma tabela de transposição. Em seguida, será demonstrado como o *VisionDraughts* realiza a integração de uma tabela de transposição com o algoritmo alfa-beta.

### 4.3.2 A Tabela de Transposição

O algoritmo alfa-beta, apresentado na seção 4.3.1, não mantém um histórico dos estados da árvore de jogo procurados anteriormente. Assim, se um estado  $S_0$  do tabuleiro for apresentado 2 vezes para o algoritmo alfa-beta, a mesma rotina será executada 2 vezes a fim de encontrar a predição associada ao estado  $S_0$ . Para evitar redundância de trabalho, ou seja, evitar que o algoritmo alfa-beta seja executado duas vezes para encontrar a predição do mesmo estado  $S_0$ , pode-se associá-lo com uma tabela de transposição. Uma tabela de transposição (MILLINGTON, 2006) é um repositório de predições passadas associadas aos estados do tabuleiro do jogo que já foram submetidos ao procedimento de busca. Os detalhes da tabela de transposição utilizada pelo *VisionDraughts* serão descritos a seguir.

#### 4.3.2.1 Transposição - Mais de uma Ocorrência do Mesmo Estado do Tabuleiro do Jogo

No jogo de damas, dentro de uma mesma partida, pode-se chegar a um mesmo estado do tabuleiro várias vezes e, quando isso ocorre, diz-se que houve uma transposição (MILLINGTON, 2006). As transposições ocorrem, em damas, de duas maneiras básicas:

1. Diferentes combinações de jogadas com peças simples: as peças simples não se movem para trás. Apesar disso, elas podem desencadear uma transposição, conforme mostrado na figura 24. Nesse caso, os estados do tabuleiro mostrados em *a* e *d* são idênticos, assim como os estados mostrados em *c* e *f*. Assumindo *a* como estado inicial, é possível alcançar *c* passando por *b*. Assumindo *d* como estado inicial, é possível alcançar *f* passando por *e*. Então, os únicos estados diferentes são *b* e *e*. No caso da sequência de movimentos *a*, *b* e *c*, o jogador preto move-se primeiro para a direita e, em seguida, para a esquerda, enquanto na sequência de movimentos *d*, *e* e *f*, o jogador preto move-se primeiro para a esquerda e, em seguida, para a direita.
2. Diferentes combinações de jogadas com reis: os reis se movem em qualquer direção, gerando transposições facilmente, conforme mostrado na figura 25. Partindo do estado *a*, avançando o rei, é possível alcançar o estado *b* e, em seguida, recuando o rei, é possível alcançar o estado *c*, idêntico ao *a*.

As próximas subseções abordarão, em detalhe, como o *VisionDraughts* utiliza esse repositório de estados, chamado tabela de transposição. Como a compreensão da técnica

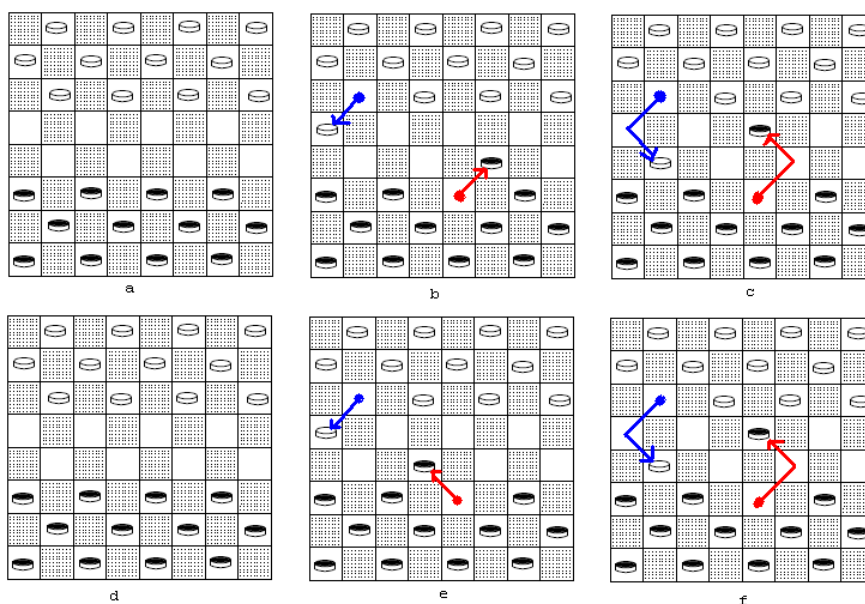


Figura 24: Exemplo de transposição em *c* e *f*: o mesmo estado do tabuleiro é alcançado por combinações diferentes de jogadas com peças simples.

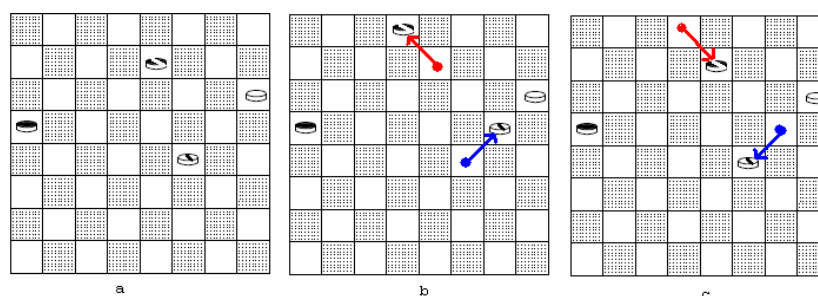


Figura 25: Exemplo de transposição em *a* e *c*: o mesmo estado do tabuleiro é alcançado por combinações diferentes de jogadas com reis.

de Zobrist é fundamental para a construção da tabela de transposição, inicialmente é apresentada a técnica de Zobrist. Na sequência, são apresentados a estrutura da tabela de transposição e como tratar possíveis colisões de estados do tabuleiro dentro da tabela. Assim, as seções se sucedem de acordo com o disposto a seguir: i) Técnica de Zobrist - Criação de Chaves Hash para Indexação dos Estados do Tabuleiro do Jogo; ii) Estrutura *ENTRY* - Dados Armazenados para um Determinado Estado do Tabuleiro do Jogo; e, iii) Colisões - Conflitos de Endereços para Estados do Tabuleiro do Jogo.

#### 4.3.2.2 Técnica de Zobrist - Criação de Chaves Hash para Indexação dos Estados do Tabuleiro do Jogo

A tabela de transposição utilizada pelo *VisionDraughts*, ou seja, o repositório de predições anteriormente calculadas e associadas aos estados do tabuleiro do jogo que

já foram submetidos ao procedimento de busca, foi implementada como uma tabela hash. Uma tabela hash é uma estrutura de dados que associa *chaves* a *valores* (RUSSELL; NORVIG, 2004). Cada *chave* representa um estado do tabuleiro do jogo de damas e é associada a informações relevantes obtidas, a partir do algoritmo alfa-beta, para àquele estado. A representação de um determinado estado do tabuleiro do jogo, na forma de uma chave hash, é feita utilizando a técnica descrita por Zobrist (ZOBRIST, 1969) e apresentada nesta seção.

Quando um programa de computador armazena um item  $I_1$  em uma tabela muito grande  $T_1$ , há duas formas de se tentar localizar este item na tabela: uma, executando-se um procedimento de busca em  $T_1$  (o que pode ser muito ineficiente devido ao tamanho da tabela); outra, dispondo-se de um método de cálculo do endereço de  $I_1$ , na tabela  $T_1$ , a partir do próprio item  $I_1$ . Uma função que converte itens em endereços é chamada função hash e a tabela resultante é chamada tabela hash.

O método descrito por Zobrist utiliza o operador XOR (ou exclusivo), simbolizado matematicamente por  $\oplus$ . Logicamente, o XOR é um tipo de disjunção lógica entre dois operandos que resulta em “verdadeiro” se, e somente se, exatamente um dos operandos tiver o valor “verdadeiro”. Computacionalmente, o operador XOR pode ser aplicado sobre dois operandos numéricos. Por exemplo:

1. Operandos numéricos na base binária: o XOR aplicado sobre dois bits quaisquer resulta em “1” se, e somente se, exatamente um dos operandos tiver o valor “1”. Assim, considere  $Seq_1 = b_1, b_2, \dots, b_n$  uma sequência binária de  $n$  bits. Além disso, considere  $Seq_2 = r_1, r_2, \dots, r_n$  outra sequência binária, também, de  $n$  bits. Para calcular  $Seq_3 = Seq_1 \oplus Seq_2$ , basta aplicar o operador XOR sobre os bits das posições correspondentes de  $Seq_1$  e  $Seq_2$ , isto é, basta fazer  $Seq_3 = b_1 \oplus r_1, b_2 \oplus r_2, \dots, b_n \oplus r_n$ .
2. Operandos numéricos na base decimal: a operação XOR sobre dois inteiros decimais segue o mesmo procedimento mostrado para operandos numéricos na base binária, exceto que, os dois argumentos inteiros decimais devem ser, antes de tudo, convertidos para a base binária. A conversão de inteiros decimais para binários é transparente em C++. Isso significa que dois operandos inteiros decimais podem ser passados como argumentos para o operador XOR (a conversão é feita implicitamente).

Assuma as seguintes propriedades, descritas em (ZOBRIST, 1969), para o operador XOR aplicado sobre sequências aleatórias ( $r$ ) de inteiros decimais de  $n$  bits:

1.  $r_i \oplus (r_j \oplus r_k) = (r_i \oplus r_j) \oplus r_k$ ;
2.  $r_i \oplus r_j = r_j \oplus r_i$ ;
3.  $r_i \oplus r_i = 0$ ;
4. se  $s_i = r_1 \oplus r_2 \oplus \dots \oplus r_i$  então  $s_i$  é uma seqüência aleatória de  $n$  bits;
5.  $s_i$  é uniformemente distribuída (uma variável é dita uniformemente distribuída quando assume qualquer um dos seus valores possíveis com a mesma probabilidade);

Suponha que exista um conjunto finito  $S$  qualquer e que se deseje criar chaves hash para os subconjuntos de  $S$ . Um método simples seria associar inteiros aleatórios de  $n$  bits aos elementos de  $S$  e, a partir daí, definir a chave hash de um subconjunto  $S_0$  de  $S$  como sendo o resultado da operação  $\oplus$  sobre os inteiros associados aos elementos de  $S_0$ . Pelas propriedades 1 e 2, a chave hash é única e, pelas propriedades 4 e 5, a chave hash é aleatória e uniformemente distribuída. Se qualquer elemento for adicionado ou retirado do subconjunto  $S_0$ , a chave hash muda pelo inteiro que corresponde àquele elemento.

No caso do *VisionDraughts*, existem 2 tipos distintos de peças (peça simples e rei), 2 cores distintas de peças (peça preta e branca) e 32 casas no tabuleiro do jogo. Então, existem, no máximo, 128 possibilidades distintas ( $2 \times 2 \times 32$ ) de colocar alguma peça em alguma casa do tabuleiro. Assim, foi criado um vetor de 128 elementos inteiros aleatórios para representar os estados possíveis do tabuleiro (cada elemento representa uma possibilidade de se ocupar uma das 32 casas do tabuleiro com alguma das 4 peças inerentes ao jogo). A chave hash para representar cada estado do tabuleiro é o resultado da operação XOR realizada entre todos os elementos do vetor associados às casas não vazias do tabuleiro. Veja o seguinte exemplo:

1. Considere um vetor  $V$  de 128 elementos inteiros aleatórios de 64 bits como sendo o mostrado na figura 26. O vetor  $V$  foi utilizado pelo sistema *VisionDraughts* para implementação da tabela de transposição. A maioria dos geradores de números aleatórios, principalmente geradores baseados em softwares, não gera seqüências verdadeiramente aleatórias e sim seqüências que possuem algumas das propriedades dos números aleatórios (uma cuidadosa análise matemática é necessária para assegurar que a geração de números seja suficientemente aleatória). Números verdadeiramente aleatórios são impossíveis de serem gerados com *máquinas de estado finito* (computadores atuais). Neste caso, cientistas são obrigados a usar geradores

baseados em caras e especializadas arquiteturas de hardware ou, mais frequentemente, contentarem-se com soluções sub-ótimas (como números pseudo-aleatórios gerados por software). Assim, para garantir a qualidade dos números aleatórios utilizados pelo *VisionDraughts*, os inteiros aleatórios do vetor  $V$  foram gerados a partir do sítio (STEVANOVIC, 2008), utilizando a técnica descrita em (STIPCEVIC; ROGINA, 2007), que garante a aleatoriedade da sequência gerada baseando-se na aleatoriedade intrínseca de processos físicos em que fótons são detectados ao acaso.

| RANDOM INT64         | PIECE      | SQUARE | RANDOM INT64         | PIECE      | SQUARE |
|----------------------|------------|--------|----------------------|------------|--------|
| 14787540466645868636 | black man  | 1      | ...                  |            | ...    |
| 2120251484556677534  | white man  |        | ...                  |            |        |
| 584882445155849028   | black king |        | ...                  |            |        |
| 3760951787791404667  | white king |        | ...                  |            |        |
| 17903615704209920410 | black man  | 2      | 8978665553187022367  | black man  | 25     |
| 5781218707178284009  | white man  |        | 6792129980026176469  | white man  |        |
| 7894141919871615785  | black king |        | 11106003084864057887 | black king |        |
| 3578131985066232389  | white king |        | 5684749757081299935  | white king |        |
| 1817657397089932766  | black man  | 3      | 3967728617316940461  | black man  | 26     |
| 9537396155164801519  | white man  |        | 16232032669744814011 | white man  |        |
| 5808583100557493539  | black king |        | 13546780321862426801 | black king |        |
| 3651659200175719294  | white king |        | 3009792841844867034  | white king |        |
| 11250323712845617096 | black man  | 4      | 13422590923753360614 | black man  | 27     |
| 15592542546949822810 | white man  |        | 10221763887329211198 | white man  |        |
| 16204138130260099375 | black king |        | 5616157223557226974  | black king |        |
| 9585321403807695269  | white king |        | 2865046354894257591  | white king |        |
| 15915542026527195059 | black man  | 5      | 14642594631129895935 | black man  | 28     |
| 16248679709773236148 | white man  |        | 8381146724961928037  | white man  |        |
| 6685379756495787903  | black king |        | 3023307655632321181  | black king |        |
| 6977407078633077238  | white king |        | 8375086150794650026  | white king |        |
| 1729081295984380347  | black man  | 6      | 11810041679881260088 | black man  | 29     |
| 6892212846999406827  | white man  |        | 1213308520865758682  | white man  |        |
| 632708781781195948   | black king |        | 9734715559513728574  | black king |        |
| 8082145037705841596  | white king |        | 12184937488032720561 | white king |        |
| 11740811010298599996 | black man  | 7      | 4993510297519374450  | black man  | 30     |
| 348921443543585631   | white man  |        | 12124137870041646186 | white man  |        |
| 14579749940077582302 | black king |        | 2664161134633443445  | black king |        |
| 6486449913624012919  | white king |        | 327774891080306970   | white king |        |
| 3466492341137833191  | black man  | 8      | 14888968537176605210 | black man  | 31     |
| 471079928059731524   | white man  |        | 6271745259985944523  | white man  |        |
| 12658037930106435315 | black king |        | 14507257672045050736 | black king |        |
| 11963310641682407293 | white king |        | 8740695389947450601  | white king |        |
| ...                  |            | ...    | 9487810991141940225  | black man  | 32     |
| ...                  |            |        | 14639527447367762922 | white man  |        |
| ...                  |            |        | 8795549574004575914  | black king |        |
| ...                  |            |        | 18030604617695974466 | white king |        |

Figura 26: Vetor de 128 elementos inteiros aleatórios utilizados pelo *VisionDraughts*.

2. Considere o estado  $S_0$  do tabuleiro do jogo de damas mostrado na figura 27.
3. Para conseguir o número aleatório associado à peça preta simples, localizada na casa 2 do tabuleiro  $S_0$ , basta fazer uma consulta ao vetor  $V$  e encontrar o valor  $I_2 = 17903615704209920410$ .
4. Para conseguir o número aleatório associado ao rei preto, localizado na casa 31






|    |    |   |    |   |    |    |    |    |
|----|----|---|----|---|----|----|----|----|
|    |    | 32  |    |  |    | 30 |    | 29 |
| 28 |    |   | 27 |   | 26 |    | 25 |    |
|    | 24 |   |    | 23  |    | 22 |    | 21 |
| 20 |    |   | 19 |   | 18 |    | 17 |    |
|    | 16 |   |    | 15  |    | 14 |    | 13 |
| 12 |    |   | 11 |   | 10 |    | 9  |    |
|    | 8  |   |    | 7   |    | 6  |    | 5  |
| 4  |    |  |    |  |    |    | 1  |    |

Figura 27: Um estado do tabuleiro do jogo de damas.

do tabuleiro  $S_0$ , basta fazer uma consulta ao vetor  $V$  e encontrar o valor  $I_{31} = 14507257672045050736$ .

- Para conseguir o número aleatório associado ao rei branco, localizado na casa 3 do tabuleiro  $S_0$ , basta fazer uma consulta ao vetor  $V$  e encontrar o valor  $I_3 = 3651659200175719294$ .
- Assim, para conseguir a chave hash associada ao estado do tabuleiro  $S_0$ , basta considerar  $C_0 = I_2 \oplus I_{31} \oplus I_3$ . Logo,  $C_0 = 17903615704209920410 \oplus 14507257672045050736 \oplus 3651659200175719294$ , ou seja,  $C_0 = 256431317426989460$ .

A técnica de Zobrist é, provavelmente, o método mais rápido disponível para calcular uma chave hash associada a um estado do tabuleiro do jogo de damas (ZOBTRIST, 1969). Os fundamentos dessa informação são a velocidade com que a operação XOR é executada por uma CPU e a possibilidade de atualização incremental. Para entender como ocorre a atualização incremental das chaves hash, associadas aos estados do tabuleiro, considere as movimentações possíveis no jogo de damas:

- Movimento simples: um movimento simples pode ser tratado como a remoção de uma peça simples da casa de origem do movimento e sua inserção na casa de destino do movimento.
- Promoção: uma promoção acontece quando uma peça simples se torna rei. Pode ser tratada como sendo a remoção de um tipo peça e a inserção de outro tipo de peça na mesma casa do tabuleiro.
- Captura: uma captura pode ser tratada como sendo a remoção da peça capturada, a remoção da peça capturadora da casa de origem do movimento e a inserção da peça capturadora na casa de destino do movimento.

Considere, ainda, o exemplo de movimento simples da figura 28 e o vetor de números aleatórios, utilizado pelo *VisionDraughts*, mostrado na figura 26.

|    |    |    |    |    |    |    |    |    |
|----|----|----|----|----|----|----|----|----|
|    |    | 32 |    |    |    | 30 |    | 29 |
| 28 |    |    | 27 |    | 26 |    | 25 |    |
|    | 24 |    | 23 |    |    |    |    |    |
| 20 |    | 19 |    | 18 |    |    |    |    |
|    | 16 |    | 15 |    | 14 |    | 13 |    |
| 12 |    | 11 |    | 10 |    | 9  |    |    |
|    | 8  |    | 7  |    | 6  |    | 5  |    |
| 4  |    | 3  |    | 2  |    | 1  |    |    |

$S_0$

|    |    |    |    |    |    |    |    |    |
|----|----|----|----|----|----|----|----|----|
|    |    | 32 |    |    |    | 30 |    | 29 |
| 28 |    |    | 27 |    | 26 |    |    |    |
|    | 24 |    | 23 |    |    |    |    | 21 |
| 20 |    | 19 |    | 18 |    |    |    |    |
|    | 16 |    | 15 |    | 14 |    | 13 |    |
| 12 |    | 11 |    | 10 |    | 9  |    |    |
|    | 8  |    | 7  |    | 6  |    | 5  |    |
| 4  |    | 3  |    | 2  |    | 1  |    |    |

$S_1$

Figura 28: Exemplo de movimento simples.

1. Para conseguir o número aleatório associado à peça preta simples, localizada na casa 21 do tabuleiro  $S_0$ , basta fazer uma consulta ao vetor  $V$  e encontrar o valor  $I_{21} = 1171196056361380757$ ;
2. Para conseguir o número aleatório associado à peça preta simples, localizada na casa 22 do tabuleiro  $S_0$ , basta fazer uma consulta ao vetor  $V$  e encontrar o valor  $I_{22} = 9204715365712158256$ ;
3. Para conseguir o número aleatório associado à peça branca simples, localizada na casa 31 do tabuleiro  $S_0$ , basta fazer uma consulta ao vetor  $V$  e encontrar o valor  $I_{31} = 6271745259985944523$ ;
4. Para conseguir a chave hash  $C_0$ , associada ao estado do tabuleiro  $S_0$ , basta considerar  $C_0 = I_{21} \oplus I_{22} \oplus I_{31}$ . Logo,  $C_0 = 1171196056361380757 \oplus 9204715365712158256 \oplus 6271745259985944523$ , ou seja,  $C_0 = 4104165011015584366$ ;
5. Para conseguir a chave hash  $C_1$ , associada ao estado do tabuleiro  $S_1$ , não é necessário repetir os passos anteriores; basta atualizar, incrementalmente, o valor da chave  $C_0$ , conforme passos 6 e 7;
6.  $C_1 = C_0 \oplus I_{21}$ . Indica a remoção da peça preta simples da casa 21 do estado  $S_0$  (veja a propriedade número 3). Logo,  $C_1 = 4104165011015584366 \oplus 1171196056361380757$ , ou seja,  $C_1 = 2932970144385327611$ ;
7.  $C_1 = C_1 \oplus I_{25}$ . Isso indica a inserção da peça preta simples na casa 25 do tabuleiro, gerando o estado  $S_1$  (veja a propriedade número 1). Logo,  $C_1 = 2932970144385327611 \oplus 8978665553187022367$ , ou seja,  $C_1 = 8988798927364941823$ .

#### 4.3.2.3 Estrutura *ENTRY* - Dados Armazenados para um Determinado Estado do Tabuleiro do Jogo

A tabela de transposição utilizada pelo *VisionDraughts*, implementada como uma tabela hash a fim de que se consiga máxima velocidade de manipulação, mantém um registro de estados analisados do tabuleiro do jogo, com respectivos valores obtidos para os mesmos através do algoritmo alfa-beta. A partir de então, sempre que um estado  $S_0$  do tabuleiro for apresentado como entrada do procedimento alfa-beta, primeiro o algoritmo verificará se  $S_0$  encontra-se na tabela de transposição e, caso afirmativo, utilizará os valores armazenados em memória, o que abreviará todo o procedimento de busca.

O *VisionDraughts* utiliza uma estrutura chamada *ENTRY* (tipo de dado definido pelo usuário) para armazenar os dados de entrada para a tabela de transposição. Veja:

```
struct ENTRY{
    int64    hashvalue;
    int      bestvalue;
    MOVE     bestmove;
    int      depth;
    int      scoretype;
    int      checksum;
}
```

O campo *hashvalue* armazena a chave hash mostrada na seção 4.3.2.2. Logo, dado um estado  $S_0$  do tabuleiro do jogo, gera-se a chave hash  $C_0$  associada à  $S_0$  e armazena-se a mesma em *hashvalue*.

Os campos *bestvalue* e *bestmove* armazenam, respectivamente, com relação ao algoritmo alfa-beta, o valor da predição e a melhor jogada sugerida. Logo, dado um estado  $S_0$  do tabuleiro, aplica-se o algoritmo alfa-beta, com o intuito de se descobrir qual é a predição associada a ele e qual a melhor ação a ser executada a partir dele, isto é, qual a avaliação do estado  $S_0$  do ponto de vista do agente jogador e qual a melhor ação a ser executada, pelo agente jogador, a partir de  $S_0$ . Por exemplo, na figura 23, supondo que o valor da chave hash associada ao estado  $S_0$  seja *hv*, a estrutura *ENTRY* ficará:

```
struct ENTRY{
    int64    hashvalue = hv;
    int      bestvalue = 0.3;
```

```

    MOVE    bestmove = S1;
}

```

O campo *depth* armazena a profundidade de busca utilizada pelo algoritmo alfa-beta. O algoritmo alfa-beta recebe, como parâmetros de entrada, um estado  $S_0$  do tabuleiro do jogo e uma profundidade de busca *depth*. No exemplo mostrado na figura 23, o estado  $S_0$  foi submetido ao algoritmo alfa-beta com profundidade *depth* = 2. Os estados  $S_1$ ,  $S_2$  e  $S_3$  foram pesquisados com profundidade *depth* = 1 e os estados  $S_{11}$ ,  $S_{12}$ ,  $S_{13}$ ,  $S_{21}$  e  $S_{31}$  com profundidade *depth* = 0. Assim, no exemplo da figura 23, a estrutura *ENTRY* ficará:

```

struct ENTRY{
    int64    hashvalue = hv;
    int      bestvalue = 0.3;
    MOVE     bestmove = S1;
    int      depth = 2;
}

```

Note que o estado  $S_0$  do tabuleiro, identificado pela chave hash  $C_0$  e mostrado na figura 23, possui predição *bestvalue* = 0.3, no caso em que foi pesquisado com profundidade de busca *depth* = 2. Caso fosse necessário conhecer a predição associada ao estado  $S_0$ , com profundidade de busca maior ou igual a 3, o valor armazenado na estrutura *ENTRY* não poderia ser utilizado, pois não teria a precisão (look-ahead) exigida.

O campo *scoretype* armazena um flag que indica o real significado da predição contida em *bestvalue*. Por exemplo, na figura 23, o estado  $S_0$  foi submetido ao algoritmo alfa-beta com profundidade *depth* = 2. A predição de retorno do algoritmo foi *bestvalue* = 0.3. Nesse caso, como todos os filhos de  $S_0$  foram analisados, é possível afirmar que 0.3 significa a real predição do estado  $S_0$ . Assim, no exemplo da figura 23, a estrutura *ENTRY* ficará:

```

struct ENTRY{
    int64    hashvalue = hv;
    int      bestvalue = 0.3;
    MOVE     bestmove = S1;
    int      depth = 2;
    int      scoretype = hashExact;
}

```

Como todos os filhos de  $S_0$  foram analisados na figura 23, *scoretype* recebeu o valor *hashExact*. Para os estados em que nem todos os filhos são analisados, *scoretype* pode assumir os valores *hashAtMost* ou *hashAtLeast*, conforme demonstrado na seção 4.3.3.2.

O campo *checksum* armazena outra chave hash, gerada de forma idêntica à chave *hashvalue*, porém, partindo de números aleatórios diferentes. A chave *checksum* utiliza números inteiros de 32 bits enquanto *hashvalue* utiliza inteiros de 64 bits (quanto maior o número de bits, maior a precisão da chave hash e maior o espaço de memória necessário para armazená-la). A seção 4.3.2.4 trata dos possíveis problemas de colisão de registros na tabela de transposição e utiliza esta segunda chave hash, *checksum*, para o mesmo estado do tabuleiro, como um dos instrumentos para detecção e tratamento de colisões. Supondo que o valor da segunda chave hash associada ao estado  $S_0$  da figura 23 seja *ck*, a estrutura *ENTRY* ficará:

```
struct ENTRY{
    int64    hashvalue = hv;
    int      bestvalue = 0.3;
    MOVE     bestmove = S1;
    int      depth = 2;
    int      scoretype = hashExact;
    int      checksum = ck;
}
```

A estrutura *ENTRY* é a unidade básica de entrada para a tabela de transposição utilizada pelo *VisionDraughts*. A tabela de transposição é uma estrutura do tipo *TTABLE* que contém dois vetores, *e1* e *e2*, ambos com elementos do tipo *ENTRY*, um método para armazenar novas entradas e um método para ler as entradas já existentes na tabela de transposição. A estrutura *TTABLE* será detalhada na seção 4.3.2.5, porém, é necessário entender, primeiro, o mecanismo de tratamento de colisões utilizado pelo *VisionDraughts*.

#### 4.3.2.4 Colisões - Conflitos de Endereços para Estados do Tabuleiro do Jogo

A tabela de transposição do *VisionDraughts* trata dois tipos de erros identificados por Zobrist (ZOBRIST, 1969). O primeiro tipo de erro, chamado erro *tipo 1* ou *clash*, ocorre quando dois estados distintos do tabuleiro do jogo de damas são mapeados na mesma chave hash. Caso o *clash* não seja detectado, pode acontecer de predições incorretas serem retornadas pela rotina de busca alfa-beta com tabela de transposição.

Para controlar os erros *tipo 1*, são usadas as duas chaves hash da estrutura *ENTRY*: *hashvalue* e *checksum*. Cada uma das chaves é gerada utilizando números aleatórios independentes. Se dois estados diferentes do tabuleiro produzirem a mesma chave hash, *hashvalue*, é improvável que produzam, também, o mesmo valor para *checksum* (fórmula 4.4). A segunda chave hash não precisa ser, necessariamente, do mesmo tamanho da primeira. Porém, quanto maior o número de bits presentes nas chaves hash, menor a probabilidade de ocorrência dos erros *tipo 1*. A primeira chave hash, *hashvalue*, contém 64 bits e a segunda, *checksum*, possui 32 bits.

Foi possível constatar, experimentalmente, que utilizando as duas chaves hash, *hashvalue* de 64 bits e *checksum* de 32 bits, os erros *tipo 1* foram totalmente controlados no *VisionDraughts*. Matematicamente, Zobrist descreveu a fórmula 4.4 para estimar a probabilidade de não ocorrência de erros *tipo 1*, baseando-se no tamanho da tabela de transposição e na quantidade de bits presentes nas chaves.

$$p = \left((1/e)^2\right)^{-m}; \quad (4.4)$$

onde  $m$  é igual ao número de bits presentes na chave hash menos o logaritmo na base 2 do número de entradas na tabela de transposição.

Por exemplo, a tabela de transposição do *VisionDraughts*, *TTABLE*, utiliza tanta memória quanto disponível, porém, exige uma quantidade mínima de 1 GB de RAM, o que possibilita trabalhar com  $2^n$  posições ( $n = 23$ ), ou seja, *TTABLE* armazena 8.388.608 entradas do tipo *ENTRY*. A menor chave hash da estrutura *ENTRY*, *checksum*, possui  $k = 32$  bits. Considerando  $m = (k - n) = (32 - 23) = 9$ , a probabilidade de não ocorrer erros *tipo 1*, no *VisionDraughts*, é de **99,805%**, conforme a fórmula acima.

O erro *tipo 2* ocorre em decorrência dos recursos finitos de memória existentes, quando dois estados distintos do tabuleiro, apesar de serem mapeados em chaves hash diferentes, são direcionados para o mesmo endereço na tabela de transposição. A sequência abaixo explica como é feito o endereçamento dos estados na tabela de transposição do *VisionDraughts*. Em seguida, é apresentada a estratégia utilizada pelo *VisionDraughts* para resolver o dilema dos erros *tipo 2*.

1. A *TTABLE* armazena 8.388.608 entradas do tipo *ENTRY* e os endereços são enumerados na tabela de transposição de 0 a 8.388.607. Assim, pode-se definir seu tamanho como  $size = 8.388.608$ ;
2. Assuma  $C_1$  como sendo o valor da primeira chave hash associada a uma estrutura

- $E_1$  que deverá ser armazenada em  $TTABLE$ ;
3. Assuma que a operação *mod* (divisão modular) retorna o resto da divisão inteira entre um dividendo e um divisor;
  4. Assuma  $L_1$  como sendo o valor resultante da aplicação do operação modular (*mod*) sobre os argumentos  $C_1$  e *size*, ou seja,  $L_1 = C_1 \text{ mod } size$ ;
  5.  $L_1$  será o endereço em  $TTABLE$  em que a estrutura  $E_1$  deverá ser armazenada;
  6. Assuma  $C_2$  como sendo o valor da primeira chave hash associada a uma estrutura  $E_2$  que deverá ser armazenada em  $TTABLE$ ;
  7. Assuma  $L_2$  como sendo o valor resultante da aplicação do operação modular (*mod*) sobre os argumentos  $C_2$  e *size*, ou seja,  $L_2 = C_2 \text{ mod } size$ ;
  8. Então, mesmo que os dividendos  $C_1$  e  $C_2$  sejam diferentes, o que indica estados de tabuleiro distintos, pode acontecer de os restos  $L_1$  e  $L_2$  serem iguais;

O último item da sequência acima ilustra um caso de colisão. Tal fato acontece várias vezes durante uma mesma partida do jogo de damas. Deve-se decidir, neste caso, entre manter a entrada já presente na tabela ou substituí-la, usando algum esquema de substituição (*replacement scheme*). Neste contexto, Breuker comparou a eficiência de sete esquemas de substituição para o problema da colisão em tabelas de transposição e concluiu que uma tabela de transposição com dois níveis, ou seja, uma tabela que possua, em um mesmo endereço, espaços reservados para armazenar informações relativas a dois estados do tabuleiro, tem melhor desempenho que uma tabela com um único nível e o dobro de tamanho (BREUKER; UITERWIJK; HERIK, 1994), (BREUKER; UITERWIJK; HERIK, 1997).

O *VisionDraughts* utiliza dois esquemas de substituição, um chamado *Deep* e outro chamado *New*, dentre os sete estudados por Breuker. A escolha dos dois esquemas baseou-se no jogador *Chinook*. O *Chinook* utiliza uma tabela de transposição de dois níveis em que cada entrada da tabela pode conter informações relativas a até 2 estados do tabuleiro (um estado em cada nível). Experimentos no *Chinook* mostram que tal estrutura para a tabela de transposição reduz o tamanho da árvore de busca em até 10% (SCHAEFFER, 2002).

Caso um endereço da tabela de transposição armazene informações sobre o mesmo estado do tabuleiro  $S_0$  em seus dois níveis, isso indica que as informações sobre  $S_0$  do

primeiro nível terão sido obtidas a partir de uma busca mais profunda que aquela a partir da qual foram obtidas as informações sobre  $S_0$  armazenadas no segundo nível (ver item 2 da descrição da operação *StoreEntry* 4.3.2.5). Caso um endereço da tabela de transposição armazene em seus dois níveis informações sobre dois estados distintos do tabuleiro, isso indica que o segundo nível foi utilizado para resolver um problema de colisão (ver item 3 da operação *StoreEntry*). Em termos práticos, o primeiro nível armazena os dados de maior precisão, enquanto, o segundo, age como um “cache” temporal.

Então, a tabela *TTABLE* contém 2 vetores  $e1$  e  $e2$ , do tipo *ENTRY*, representando os 2 níveis da tabela de transposição, e cada esquema de substituição está associado a um dos dois vetores de *TTABLE*, conforme mostrado a seguir:

1. **Deep**: o esquema de substituição *Deep* é tradicional e baseado nas profundidades das sub-árvores examinadas para as posições envolvidas. Em uma colisão, a posição com a mais profunda sub-árvore é preservada na tabela. O conceito por trás deste esquema é que uma sub-árvore mais profunda, normalmente, contém mais nós do que uma sub-árvore mais rasa e, tal fato, faz com que o algoritmo alfa-beta economize um tempo maior quando é poupado de pesquisar uma sub-árvore mais profunda. Além disso, a predição calculada para um estado do tabuleiro com uma sub-árvore mais profunda possui um maior “look ahead” e tem uma probabilidade maior de representar uma predição mais acertada. Assim sendo, armazenar a posição mais profunda na tabela de transposição, potencialmente, salva mais trabalho do que armazenar uma posição menos profundamente investigada. O primeiro vetor da tabela de transposição *TTABLE* ( $ENTRY^* e1$ ) utiliza esse esquema de substituição.
2. **New**: o esquema de substituição *New* substitui o conteúdo de uma posição na tabela quando uma colisão ocorre. Tal conceito é baseado na observação de que a maioria das transposições ocorrem localmente, dentro de pequenas sub-árvores da árvore de busca global. O segundo vetor da tabela de transposição *TTABLE* ( $ENTRY^* e2$ ) utiliza esse esquema de substituição.

Foi possível constatar, experimentalmente, que utilizando os dois esquemas de substituição (*deep* e *new*), os erros *tipo 2* foram totalmente controlados no *VisionDraughts*.



#### 4.3.2.5 Estrutura *TTABLE* - Manipulação de Dados na Tabela de Transposição com Tratamento de Colisões

Conforme dito anteriormente, a tabela de transposição utilizada pelo *VisionDraughts* é uma estrutura do tipo *TTABLE* que contém:

1. Dois vetores, *e1* e *e2*, com elementos do tipo *ENTRY*;
2. Um método para armazenar novas entradas na tabela;
3. Um método para ler as entradas já existentes na tabela;

Logo, escrevendo em termos de pseudo-código, pode-se estruturar *TTABLE* da seguinte maneira:

```
class TTABLE{  
    int          tableSize;  
    ENTRY*       e1;  
    ENTRY*       e2;  
  
    StoreEntry (...);  
    GetEntry    (...);  
}
```

O campo *tableSize* passa a idéia do espaço alocado na tabela de transposição para manipulação de registros do tipo *ENTRY*. *TTABLE* trabalha com um tamanho mínimo de 1 GB de RAM, espaço suficiente para armazenar uma quantidade máxima de 8.388.608 entradas do tipo *ENTRY*. Como são dois vetores do tipo *ENTRY* em cada linha da tabela de transposição, a quantidade máxima de linhas em uma *TTABLE* de 1 GB de RAM é de 4.194.304. Importante notar que *TTABLE* utiliza tanta memória quanto disponível e, claro, quanto mais memória, melhor o desempenho da tabela de transposição.

Cada par de vetores *e1* e *e2* é formado por elementos do tipo *ENTRY* e disponibiliza, em memória, os dados relevantes de até dois estados do tabuleiro do jogo de damas. Tais dados são obtidos durante o procedimento de busca e os detalhes de como são armazenados e recuperados são definidos, respectivamente, pelos métodos *GetEntry(...)* e *StoreEntry(...)* apresentados a seguir.

$$GetEntry(n, hashvalue, checksum, pdepth); \quad (4.5)$$

O método *GetEntry(...)* da expressão 4.5 visa a recuperar informações sobre um dado estado  $S_0$  do tabuleiro eventualmente armazenado na tabela de transposição. Para tanto, ele recebe 4 parâmetros de entrada associados ao estado  $S_0$ . Os parâmetros são:

1.  $n$  representando o próprio estado  $S_0$ ;
2. *hashvalue* representando a primeira chave hash associada a  $S_0$ ;
3. *checksum* representando a segunda chave hash associada a  $S_0$ ;
4. *pDepth* especificando a profundidade mínima de busca desejada a partir da qual, eventualmente, tenham sido calculadas e armazenadas em *TTABLE* as informações relativas ao estado  $S_0$ .

A partir daí, *GetEntry* localiza o endereço  $E_1$  associado ao parâmetro *hashvalue* na tabela de transposição e verifica se existe algum elemento do tipo *ENTRY* gravado no primeiro vetor  $e1$  de  $E_1$ . Caso exista, verifica se a primeira chave hash associada ao elemento de  $e1$  é igual a *hashvalue*. Caso afirmativo, verifica se a segunda chave hash associada ao elemento de  $e1$  é igual a *checksum*. Caso seja, verifica se a profundidade de busca associada ao elemento de  $e1$  é maior ou igual a *pDepth*. Caso isso também se confirme, o método *GetEntry* terá obtido sucesso em sua busca e terá encontrado em *TTABLE* as informações desejadas para o tabuleiro  $n$ . Assim, o algoritmo alfa-beta pode utilizar os dados armazenados em memória em vez de continuar expandindo a árvore de busca do jogo. Caso não se tenha obtido êxito com o primeiro vetor  $e1$ , o mesmo processo é realizado, novamente, para o segundo vetor  $e2$  de  $E_1$ .

$$StoreEntry(newEntry); \quad (4.6)$$

A expressão 4.6 representa o método de armazenamento de informações sobre um dado estado do tabuleiro na tabela de transposição. Para tanto, o método recebe um elemento *newEntry* do tipo *ENTRY* contendo tais informações e tenta armazená-las em *TTABLE*. O método localiza o endereço  $E_1$  em *TTABLE* associado ao parâmetro *hashvalue* de *newEntry*. Considerando o endereço  $E_1$ , três situações podem ocorrer:

1.  $E_1$  encontra-se vazio: como inexistente informação gravada no endereço  $E_1$ , basta armazenar *newEntry* no primeiro vetor  $e1$  do endereço  $E_1$ ;

2.  $E_1$  possui o mesmo elemento passado como parâmetro: o endereço  $E_1$  já possui informação e o valor da primeira chave do primeiro vetor  $e1$  de  $E_1$  é exatamente igual ao valor *hashvalue* de *newEntry*. Assim, caso a profundidade de busca do elemento *newEntry* seja maior que a profundidade de busca do elemento  $e1$  presente em  $E_1$ , transfere-se o elemento presente no vetor  $e1$  para a mesma posição no vetor  $e2$  e sobrescreve-se a informação presente em  $e1$  com a informação de *newEntry*. Caso a profundidade de busca do elemento *newEntry* seja menor, mantém-se a informação presente em  $e1$  e perde-se as informações de *newEntry*;
3.  $E_1$  possui elemento diferente: o endereço  $E_1$  já possui informação e o valor da primeira chave do primeiro vetor  $e1$  de  $E_1$  é diferente do valor *hashvalue* de *newEntry*. Neste caso, *StoreEntry* resolve a colisão mantendo a informação de  $e1$  e, independentemente do conteúdo do segundo vetor  $e2$ , gravando o valor de *newEntry* em  $e2$ ;

Portanto, havendo uma tabela de transposição como a *TTABLE*, sempre que um determinado estado do tabuleiro for apresentado ao algoritmo alfa-beta, ele verificará, primeiro, a tabela de transposição para ver se aquele estado do tabuleiro já foi analisado. Caso afirmativo, a informação armazenada na memória será utilizada diretamente (respeitadas as restrições de profundidade). Caso contrário, a árvore do jogo será expandida pela rotina de busca e uma nova entrada será gravada na tabela de transposição.

### 4.3.3 Integração entre o Algoritmo Alfa-Beta e a Tabela de Transposição

A primeira tentativa do *VisionDraughts* para integrar o algoritmo alfa-beta com uma tabela de transposição não obteve êxito. A tentativa baseou-se no pseudo-código da seção 4.3.1 e na tabela de transposição da seção 4.3.2.5. A introdução da estrutura *TTABLE* dentro do procedimento de busca fez surgir erros difíceis de serem rastreados, impedindo a escolha apropriada da melhor ação a ser executada pelo agente jogador (a resposta obtida, em alguns casos, não coincidia com a apontada pelo minimax na mesma situação).

O problema foi solucionado com a utilização de uma variante do algoritmo alfa-beta chamada *fail-soft alfa-beta* (SHAMS; KAINDL; HORACEK, 1991), (PLAAT, 1996), detalhada na seção seguinte.

#### 4.3.3.1 A Variante Fail-Soft do Algoritmo Alfa-Beta

O pseudo-código da seção 4.3.1 é referenciado como variante *hard-soft alfa-beta*, fazendo reforçar a idéia de que o intervalo de busca imposto pelos parâmetros alfa e beta deve sempre ser respeitado. A variante *fail-soft alfa-beta* é bastante parecida com a *hard-soft*. De fato, basta realizar alterações mínimas na versão *hard-soft* para que se consiga a versão *fail-soft*. Tais alterações, no entanto, provocam grandes mudanças nas predições retornadas pelo algoritmo e permitem a integração com as tabelas de transposição.

A seguir é apresentado o pseudo-código da variante *fail-soft alfa-beta*, no qual são enfatizados os pontos em que ele difere do pseudo-código da variante *hard-soft alfa-beta*.

*Pseudo-código da variante fail-soft alfa-beta*

```

1. fun alfaBeta(n:node,depth:int,min:int,max:int,bestmove:move):float =
2.   if leaf(n) or depth=0 return evaluate(n)
3.   if n is a max node
4.     besteval := min
5.     for each child of n
6.       v := alfaBeta(child,d-1,besteval,max,bestmove)
7.       if v > besteval
8.         besteval:= v
9.         thebest = bestmove
10.    if besteval >= max then return besteval
11.    bestmove = thebest
12.    return besteval
13.  if n is a min node
14.    besteval := max
15.    for each child of n
16.      v := alfaBeta(child,d-1,min,besteval,bestmove)
17.      if v < besteval
18.        besteval:= v
19.        thebest = bestmove
20.    if besteval <= min then return besteval
21.    bestmove = thebest
22.    return besteval

```

**linha 10.** Se acontecer de a predição armazenada na variável *besteval* ultrapassar o limite

superior do intervalo de busca ( $besteval \geq max$ ), o algoritmo retornará, imediatamente, o valor constante da variável *besteval*, em vez do limite superior do intervalo de busca ( $max$ ), como predição associada ao estado do tabuleiro  $n$ . Tal fato expressa a idéia de que a predição associada ao estado do tabuleiro  $n$  é, no mínimo, *besteval*;

**linha 20.** Se acontecer de a predição armazenada na variável *besteval* for menor que o limite inferior do intervalo de busca ( $besteval \leq min$ ), o algoritmo retornará, imediatamente, o valor constante da variável *besteval*, em vez do limite inferior do intervalo de busca ( $min$ ), como predição associada ao estado do tabuleiro  $n$ . Tal fato expressa a idéia de que a predição associada ao estado do tabuleiro  $n$  é, no máximo, *besteval*;

Considerando o exemplo de árvore do jogo de damas mostrado na figura 29 e o pseudo-código do algoritmo *fail-soft alfa-beta*, veja os seguintes casos:

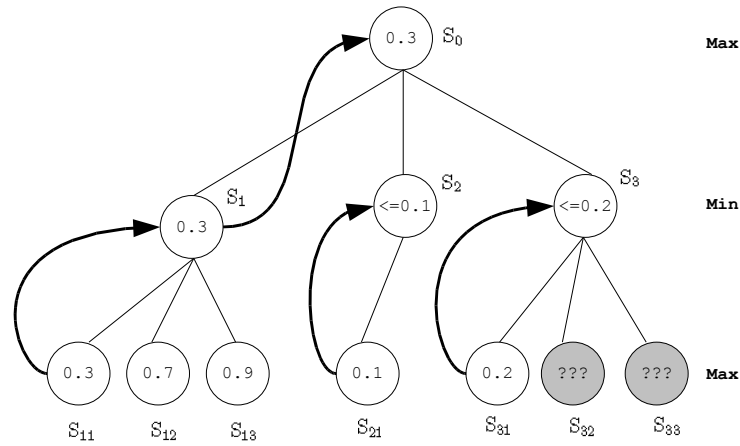


Figura 29: Exemplo de árvore do jogo de damas criada pelo algoritmo alfa-beta em sua versão fail-soft.

1. O estado do tabuleiro  $S_0$  presente na raiz da árvore do jogo é apresentado para o algoritmo alfa-beta com um intervalo de busca configurado, inicialmente, para  $[-\infty, +\infty]$ ;
2. O estado do tabuleiro  $S_1$ , primeiro filho de  $S_0$ , é apresentado para o algoritmo alfa-beta de maneira recursiva e com um intervalo de busca, também, configurado para  $[-\infty, +\infty]$ ;
3. O estado do tabuleiro  $S_{11}$ , primeiro filho de  $S_1$ , é apresentado para o algoritmo alfa-beta de maneira recursiva e com um intervalo de busca, também, configurado para  $[-\infty, +\infty]$ ;

4. A rede neural retorna, através da função  $evaluate(n)$ , uma predição de 0.3 para o estado  $S_{11}$ . Neste momento, o intervalo de busca do estado do tabuleiro  $S_1$  é alterado para o intervalo  $[-\infty, +0.3]$  (note que o estado  $S_1$  é representado por um nó minimizador e, logo, as alterações de intervalo acontecem no parâmetro do limite superior);
5. Ao pesquisar os demais filhos do estado  $S_1$ , seu intervalo de busca não é alterado. Uma vez que todos os filhos de  $S_1$  foram pesquisados pelo alfa-beta, sua predição é definida como 0.3;
6. O valor 0.3 é propagado do estado  $S_1$  para o estado  $S_0$ . Neste momento, o intervalo de busca do estado do tabuleiro  $S_0$  é alterado para o intervalo  $[0.3, +\infty]$  indicando que o estado  $S_0$  tem predição associada de, no mínimo, 0.3 (note que o estado  $S_0$  é representado por um nó maximizador e, logo, as alterações de intervalo acontecem no parâmetro do limite inferior);
7. O estado do tabuleiro  $S_2$ , segundo filho de  $S_0$ , é apresentado para o algoritmo alfa-beta de maneira recursiva e com um intervalo de busca configurado para  $[0.3, +\infty]$ ;
8. O estado do tabuleiro  $S_{21}$ , primeiro filho de  $S_2$ , é apresentado para o algoritmo alfa-beta de maneira recursiva e com um intervalo de busca, também, configurado para  $[0.3, +\infty]$ ;
9. A rede neural retorna, através da função  $evaluate(n)$ , uma predição de 0.1 para o estado  $S_{21}$ . No entanto, o valor 0.1 encontra-se abaixo do valor mínimo passado como parâmetro para o intervalo de busca do alfa-beta ( $[0.3, +\infty]$ ). De acordo com o pseudo-código apresentado para a variante *fail-soft*, o valor 0.1 deve ser associado como predição máxima do estado  $S_2$ , ou seja,  $S_2$  terá um valor de predição máxima igual a 0.1;
10. O estado do tabuleiro  $S_3$ , terceiro filho de  $S_0$ , é apresentado para o algoritmo alfa-beta de maneira recursiva e com um intervalo de busca configurado para  $[0.3, +\infty]$ ;
11. O estado do tabuleiro  $S_{31}$ , primeiro filho de  $S_3$ , é apresentado para o algoritmo alfa-beta de maneira recursiva e com um intervalo de busca, também, configurado para  $[0.3, +\infty]$ ;
12. A rede neural retorna, através da função  $evaluate(n)$ , uma predição de 0.2 para o estado  $S_{31}$ . No entanto, o valor 0.2 encontra-se abaixo do valor mínimo passado como parâmetro para o intervalo de busca do alfa-beta ( $[0.3, +\infty]$ ). De acordo com

o pseudo-código apresentado para a variante *fail-soft*, o valor 0.2 deve ser associado como predição máxima do estado  $S_3$ , ou seja,  $S_3$  terá um valor de predição máxima igual a 0.2;

13. A linha 15 do pseudo-código *fail-soft* alfa-beta garante o retorno imediato do algoritmo com o valor 0.2 associado como predição máxima do estado  $S_3$ . Assim, os outros dois filhos  $S_{32}$  e  $S_{33}$  de  $S_3$  não precisam ser analisados (o estado  $S_0$  já possui predição mínima de 0.3). Logo que o valor 0.2 é encontrado para o primeiro filho de  $S_3$ , a variável *besteval*, na linha 14, é configurada como 0.2. O valor de *min*, na linha 15, é igual a 0.3. Portanto, como  $0.2 \leq 0.3$ , retorna-se o valor 0.2.

Assim, com base na explicação detalhada do algoritmo *fail-soft alfa-beta*, mostrada acima, é possível abstrair a seguinte idéia:

1. **Poda alfa:** a avaliação dos filhos de um nó  $n$  de minimização pode ser interrompida tão logo uma predição  $P_1$ , calculada para um dos filhos de  $n$ , seja menor que o parâmetro alfa. Neste caso, a valor  $P_1$  indica que o nó  $n$  possui predição igual a, no máximo,  $P_1$ .
2. **Poda beta:** a avaliação dos filhos de um nó  $n$  de maximização pode ser interrompida tão logo uma predição  $P_1$ , calculada para um dos filhos de  $n$ , seja maior que o parâmetro beta. Neste caso, a valor  $P_1$  indica que o nó  $n$  possui predição igual a, no mínimo,  $P_1$ .

Tanto a versão *hard-soft* quanto a *fail-soft* retornam a mesma predição 0.3 para o estado  $S_0$  na figura 29. Ambas incluem na tabela de transposição o estado  $S_{31}$  com predição 0.2. Ambas efetuam as mesmas podas. Porém, apesar de  $S_3$  ser minimizador, na versão *hard-soft* o valor coerente 0.2 de um de seus filhos foi preterido em relação ao valor 0.3 (imposto pelo maximizador  $S_0$ ). Assim, na versão *hard-soft* o valor 0.3 é armazenado na tabela de transposição junto com o estado  $S_3$ , enquanto na *fail-soft* o valor armazenado é 0.2. Portanto, durante a construção do *VisionDraughts*, foi possível concluir que a versão *hard-soft* é incompatível com o uso de tabelas de transposição: caso, com a evolução do jogo, o estado  $S_3$  precise ser avaliado novamente, pode acontecer do valor indevido 0.3 ser resgatado da tabela de transposição.

A seguir, serão analisados os métodos utilizados para ler e escrever dados na tabela de transposição a partir do algoritmo *fail-soft alfa-beta*.

#### 4.3.3.2 Armazenar Estados do Tabuleiro na Tabela de Transposição a partir do Algoritmo Fail-Soft Alfa-Beta

O algoritmo alfa-beta não mantém um histórico dos estados da árvore de jogo procurados anteriormente. Então, se um estado  $S_0$  do tabuleiro for apresentado 2 vezes para o algoritmo alfa-beta, a mesma rotina será executada 2 vezes a fim de encontrar a predição associada ao estado  $S_0$ . O repositório de dados utilizado pelo *VisionDraughts*, tabela de transposição, evita a referida pesquisa em duplicidade, armazenando e recuperando dados em memória.

A estrutura *ENTRY*, mostrada na seção 4.3.2.3, é a unidade básica de entrada para a tabela de transposição utilizada pelo *VisionDraughts*. Um método para armazenar estados do tabuleiro na tabela de transposição precisa criar uma nova estrutura do tipo *ENTRY* e armazená-la no endereço de memória correto. Então, para que seja possível criar uma estrutura do tipo *ENTRY*, o método precisa ter a seguinte assinatura:

$$\text{store}(n, \text{besteval}, \text{bestmove}, \text{depth}, \text{scoreType}); \quad (4.7)$$

onde  $n$  representa um estado qualquer do tabuleiro do jogo, *besteval* representa a predição associada ao estado  $n$ , *bestmove* corresponde ao melhor movimento para o jogador a partir de  $n$ , *depth* representa a profundidade de busca associada a  $n$  e *scoreType* indica se a predição do estado do tabuleiro  $n$  é exatamente igual a *besteval*, no máximo igual a *besteval*, ou, no mínimo igual a *besteval*. Considerando que todo e qualquer estado do tabuleiro do jogo possui duas chaves hash associadas (seção 4.3.2.2), o método acima já possui todas as informações para escrever na tabela de transposição, bastando obedecer o esquema de substituição descrito na seção 4.3.2.5. Para tanto, o método *store* invoca o método *StoreEntry* (4.6).

O detalhe mais importante do método *store* é o parâmetro *scoreType*. Ele pode assumir os valores *hashExact*, *hashAtLeast* ou *hashAtMost* de acordo com as seguintes regras:

1. **Ausência de poda:** caso o estado do tabuleiro  $n$  seja uma folha da árvore de busca ou caso todos os filhos de  $n$  tenham sido analisados (nenhuma poda), significa que a predição associada a  $n$  é exatamente igual a *besteval*. Então, *scoreType* deve receber o valor *hashExact*;
2. **Poda alfa:** caso algum dos filhos de  $n$  tenha sido descartado do procedimento de busca através de uma poda alfa, significa que  $n$  é um minimizador e sua predição



é, no máximo, igual a *besteval*. Então, *scoreType* deve receber o valor *hashAtMost*;

3. **Poda beta:** caso algum dos filhos de  $n$  tenha sido descartado do procedimento de busca através de uma poda beta, significa que  $n$  é um maximizador e sua predição é, no mínimo, igual a *besteval*. Então, *scoreType* deve receber o valor *hashAtLeast*;

#### 4.3.3.3 Recuperação dos Estados do Tabuleiro da Tabela de Transposição a partir do Algoritmo Fail-Soft Alfa-Beta

O *VisionDraughts* tenta utilizar os dados armazenados em memória em vez de expandir uma árvore de busca do jogo. Um método para verificar se um determinado estado do tabuleiro do jogo encontra-se armazenado na tabela de transposição precisa ter a seguinte assinatura:

$$\text{retrieve}(n, \text{besteval}, \text{bestmove}, \text{depth}, \text{nodeType}); \quad (4.8)$$

O estado do tabuleiro do jogo que está sendo procurado na tabela de transposição é representado por  $n$ , *depth* representa a profundidade de busca associada a  $n$ , *nodeType* indica se o estado pai de  $n$  é minimizador ou maximizador, *besteval* e *bestmove* são parâmetros de saída que indicarão, caso ocorra sucesso no procedimento de recuperação do estado  $n$  na tabela de transposição, a predição e a melhor jogada associadas ao estado  $n$ , respectivamente.

Assim sendo, inicialmente, o método *retrieve* aciona o procedimento de leitura *GetEntry* (4.5) para checar se o estado  $n$  está armazenado na tabela de transposição. Caso obtenha sucesso e consiga recuperar uma entrada  $E_1$  na tabela, tal entrada precisa ser tratada, de acordo com o parâmetro *nodeType*, a fim de se verificar se os valores constantes da entrada  $E_1$  podem ser utilizados para preencher os parâmetros de saída *besteval* e *bestmove*.

O estado do tabuleiro  $n$  no método *retrieve* representa, sempre, um nó filho. O tratamento baseado no parâmetro *nodeType* é feito levando-se em conta o fato de o pai de  $n$  ser maximizador ou minimizador. Caso o pai de  $n$  seja representado por um nó maximizador (*nodeType* = *parentIsMaxNode*), os seguintes fatos devem ser considerados:

1. Cada irmão de  $n$  que tiver sido analisado pelo algoritmo alfa-beta, até o início da avaliação de  $n$ , terá contribuído para o ajuste do limite inferior do intervalo de busca até aquele ponto;

2. Considerando o limite inferior do intervalo que está sendo ajustado, então, há 3 análises possíveis para os valores encontrados na estrutura  $E_1$ , do tipo *ENTRY*, retornada pelo método *GetEntry* (4.5):

- $E_1$  possui campo *scoretype* = *hashExact*: como *hashExact* indica predição exata, o valor de *besteval* presente em  $E_1$  pode ser utilizado, desde que a profundidade de busca armazenada em  $E_1$  seja maior ou igual à profundidade corrente;
- $E_1$  possui campo *scoretype* = *hashAtLeast*: como *hashAtLeast* indica predição “no mínimo de”, caso o valor de *besteval* armazenado em  $E_1$  seja maior ou igual ao limite inferior ajustado até o presente momento, o valor *besteval* pode ser utilizado, desde que a profundidade de busca armazenada em  $E_1$  seja maior ou igual à profundidade corrente;
- $E_1$  possui campo *scoretype* = *hashAtMost*: como *hashAtMost* indica predição “no máximo de”, caso o valor de *besteval* armazenado em  $E_1$  seja menor do que o limite inferior ajustado até o presente momento, o valor *besteval* **não** pode ser utilizado;

Por outro lado, caso o pai de  $n$  seja representado por um nó minimizador (*nodeType* = *parentIsMinNode*), os seguintes fatos devem ser considerados:

1. Cada irmão de  $n$  que tiver sido analisado pelo algoritmo alfa-beta, até o início da avaliação de  $n$ , terá contribuído para o ajuste do limite superior do intervalo de busca até aquele ponto;
2. Considerando o limite superior do intervalo de busca que está sendo ajustado, então, há 3 análises possíveis para os valores encontrados na estrutura  $E_1$ , do tipo *ENTRY*, retornada pelo método *GetEntry* (4.5):
  - $E_1$  possui campo *scoretype* = *hashExact*: como *hashExact* indica predição exata, o valor de *besteval* presente em  $E_1$  pode ser utilizado, desde que a profundidade de busca armazenada em  $E_1$  seja maior ou igual à profundidade corrente;
  - $E_1$  possui campo *scoretype* = *hashAtLeast*: como *hashAtLeast* indica predição “no mínimo de”, caso o valor de *besteval* armazenado em  $E_1$  seja menor ou igual ao limite superior ajustado até o presente momento, o valor *besteval* pode

ser utilizado, desde que a profundidade de busca armazenada em  $E_1$  seja maior ou igual à profundidade corrente;

- $E_1$  possui campo *scoretype* = *hashAtMost*: como *hashAtMost* indica predição “no máximo de”, caso o valor de *besteval* armazenado em  $E_1$  seja maior que o limite superior ajustado até o presente momento, o valor *besteval* **não** pode ser utilizado;

#### 4.3.3.4 O Algoritmo Fail-Soft Alfa-Beta com Tabela de Transposição

Todos os pré-requisitos para a integração entre algoritmo alfa-beta e tabelas de transposição, no jogo de damas, foram detalhados nas seções anteriores. Para esclarecer o assunto, definitivamente, é apresentado o pseudo-código do algoritmo utilizado pelo *VisionDraughts* e detalhadas as linhas mais importantes (as linhas que diferem do pseudo-código da variante *fail-soft alfa-beta* sem tabela de transposição, mostrado em 4.3.3.1).

*Pseudo-código do algoritmo fail-soft alfa-beta com tabela de transposição*

```

1. fun alfaBeta(n:node,depth:int,min:int,max:int,bestmove:move):float =
2.   if leaf(n) or depth=0 then
3.     besteval := evaluate(n)
4.     store(n, besteval,bestmove,depth,hashExact)
5.     return besteval
6.   if n is a max node
7.     besteval := min
8.     for each child of n
9.       if retrieve(child,besteval,bestmove,depth-1,parentIsMaxNode)
10.        then v := besteval
11.        else v := alfabeta(child,depth-1,besteval,max,bestmove)
12.       if v > besteval
13.         besteval:= v
14.         thebest = bestmove
15.       if besteval >= max then
16.         store(child,besteval,bestmove,depth,hashAtLeast)
17.         return besteval
18.   bestmove = thebest
19.   store(n,besteval,bestmove,depth,hashExact)
20.   return besteval

```

```

21.  if n is a min node
22.    besteval := max
23.    for each child of n
24.      if retrieve(child,besteval,bestmove,depth-1,parentIsMinNode)
25.        then v := besteval
26.        else v := alfabeta(child,depth-1,besteval,max,bestmove)
27.      if v < besteval
28.        besteval:= v
29.        thebest = bestmove
30.      if besteval <= min then
31.        store(child,besteval,bestmove,depth,hashAtMost)
32.      return besteval
33.  bestmove = thebest
34.  store(n, besteval,bestmove,depth,hashExact)
35.  return besteval

```

**linha 4.** Quando o algoritmo de busca chamar a rede neural na linha 3, com a rotina *evaluate(n)*, para conhecer a predição associada ao estado folha *n*, o valor da predição recém calculada deve ser armazenado na tabela de transposição. Assim, o algoritmo invoca o procedimento *store*, descrito em 4.7. Como *n* é um nó folha, o valor da predição deve ser armazenado com *scoretype* = *hashExact*, ou seja, predição exatamente igual a *besteval*;

**linha 9.** Para cada um dos filhos de um estado maximizador *n*, antes de chamar a rotina de busca recursivamente, o repositório de dados em memória (tabela de transposição) deve ser consultado. Então o algoritmo invoca o procedimento *retrieve*, descrito em 4.8, com *nodeType* = *parentIsMaxNode*. Outro detalhe muito importante é que, nesta linha, a profundidade de busca deve ser igual a *depth* – 1, que corresponde à profundidade de *n*;

**linha 10.** Se o filho do estado *n* estiver armazenado na tabela de transposição e o método *retrieve* obtiver sucesso no tratamento das informações, o valor da predição, retornado por *retrieve*, deve ser utilizado, em vez de chamar o algoritmo alfa-beta recursivamente (veja que o método *retrieve* é chamado com um flag indicando que o estado do tabuleiro *n* é do tipo maximizador);

**linha 16.** Detectada a ocorrência de uma *poda beta*, quando o algoritmo estiver analisando os filhos de um nó  $n$  do tipo maximizador, significa que os demais filhos de  $n$ , caso existam, não precisam mais ser analisados. A variável *besteval* conterá, nesta linha, o valor mínimo aceitável para o estado do tabuleiro representado pelo nó  $n$ . Assim, a predição presente na variável *besteval* deve ser armazenada na tabela de transposição pelo método *store* (4.7), com o flag *hashAtLeast* indicando que a predição associada ao estado do tabuleiro  $n$  é, no mínimo, igual a *besteval*;

**linha 19.** Quando o algoritmo tiver analisado todos os filhos de um nó  $n$  do tipo maximizador, a variável *besteval* conterá exatamente o valor da predição para o estado do tabuleiro representado pelo nó  $n$ . Assim, a predição presente na variável *besteval* deve ser armazenada na tabela de transposição pelo método *store* (4.7), com o flag *hashExact* indicando que a predição associada ao estado do tabuleiro  $n$  é exatamente igual a *besteval*;

**linha 24.** Se o filho do estado  $n$  estiver armazenado na tabela de transposição e o método *retrieve* obtiver sucesso no tratamento das informações, o valor da predição retornado por *retrieve* deve ser utilizado, em vez de chamar o algoritmo alfa-beta recursivamente (veja que o método *retrieve* é chamado com um flag indicando que o estado do tabuleiro  $n$  é do tipo minimizador);

**linha 31.** Detectada a ocorrência de uma *poda alfa*, quando o algoritmo estiver analisando os filhos de um nó  $n$  do tipo minimizador, significa que os demais filhos de  $n$ , caso existam, não precisam mais ser analisados. A variável *besteval* conterá, nesta linha, o valor máximo aceitável para o estado do tabuleiro representado pelo nó  $n$ . Assim, a predição presente na variável *besteval* deve ser armazenada na tabela de transposição pelo método *store* (4.7), com o flag *hashAtMost* indicando que a predição associada ao estado do tabuleiro  $n$  é, no máximo, igual a *besteval*;

**linha 34.** Quando o algoritmo analisar todos os filhos de um nó  $n$ , do tipo minimizador, a variável *besteval* conterá exatamente o valor da predição para o estado do tabuleiro representado pelo nó  $n$ . Assim, a predição presente na variável *besteval* deve ser armazenada na tabela de transposição pelo método *store* (4.7), com o flag *hashExact* indicando que a predição associada ao estado do tabuleiro  $n$  é exatamente igual a *besteval*;

O segundo grande avanço do *VisionDraughts* em relação ao sistema *NeuroDraughts* (LYNCH, 1997) (LYNCH; GRIFFITH, 1997) é conseguido com o uso das tabelas de trans-

posição em conjunto com a variante *fail-soft* do algoritmo alfa-beta. Conforme demonstrado na seção 5.1, o uso de tabelas de transposição reduz o tempo de execução, necessário para encontrar a melhor ação a ser executada pelo agente jogador, para menos de 40% do tempo exigido pelo algoritmo alfa-beta puro.

#### 4.3.4 O Aprofundamento Iterativo no VisionDraughts

A qualidade de um programa jogador que utiliza o algoritmo de busca alfa-beta depende muito do número de jogadas que ele pode olhar adiante (look-ahead). Para jogos com um fator de ramificação grande, o jogador pode levar muito tempo para pesquisar poucos níveis adiante.

Em damas, a maioria dos programas jogadores utiliza mecanismos para delimitar o tempo máximo permitido de busca. Como o algoritmo alfa-beta realiza uma busca com profundidade fixa, não existe garantia que a busca irá se completar antes do tempo máximo estabelecido. Para evitar que o tempo se esgote e o programa jogador não possua nenhuma informação de qual a melhor jogada a ser executada, profundidade fixa não pode ser utilizada (MARSLAND, 1986).

Larry Atkin (FREY, 1979) introduziu a técnica de aprofundamento iterativo como um mecanismo de controle do tempo de execução, durante a expansão da árvore de busca. Seguindo a idéia de Atkin, o *VisionDraughts* utiliza a técnica de aprofundamento iterativo da seguinte forma: inicialmente, o algoritmo alfa-beta pesquisa com profundidade 2, depois com profundidade 4, depois com profundidade 6 e assim, sucessivamente, até que o tempo máximo de busca se esgote.

No *VisionDraughts*, o algoritmo alfa-beta é chamado com profundidades  $depth = 2, 4, 6, \dots, max$ , até que o tempo de busca se esgote em uma profundidade qualquer  $depth = d$ , tal que  $4 \leq d \leq max$ . Nesta situação, os resultados encontrados para a profundidade  $depth = d - 2$  são utilizados (se o tempo de busca não se esgotar, os resultados encontrados para a profundidade  $depth = max$  são utilizados).

O outro grande benefício do uso da rotina de aprofundamento iterativo é a ordenação parcial das árvores de buscas. No *VisionDraughts*, a ordenação parcial é conseguida ordenando os estados filhos da raiz, como mostrado na figura 30. Veja que a iteração com profundidade  $d$  retornou o filho  $B$  como melhor movimento a ser realizado a partir da raiz ( $0.20 > 0.15 > 0.13$ ). Assumindo que o resultado obtido pela iteração  $d$  contém uma boa aproximação do melhor movimento a ser realizado na iteração  $d+1$ , a árvore de busca

é ordenada de forma que o filho *B* fique mais à esquerda da mesma. No exemplo, após a execução da iteração  $d+1$ , o filho *B* mostrou-se, realmente, a melhor opção de movimento com uma predição igual a 0.30.

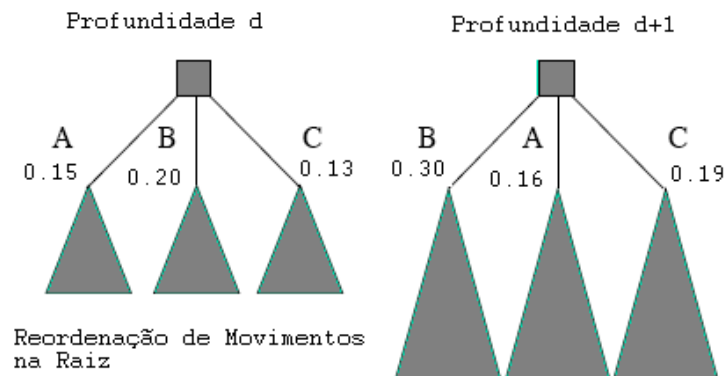


Figura 30: Exemplo de ordenação da árvore de busca com iterative deepening.

### 4.3.5 O Algoritmo Alfa-Beta com Tabela de Transposição e Aprofundamento Iterativo

Além de proporcionar um bom controle do tempo, a busca com aprofundamento iterativo é, normalmente, mais eficiente que uma busca direta com profundidade fixa. A visualização da eficiência do procedimento iterativo não é óbvia e, de fato, o procedimento não é eficiente se não for utilizado em conjunto com tabelas de transposição uma vez que elas armazenam os melhores movimentos das sub-árvores pesquisadas nas iterações anteriores. Neste caso, vários experimentos mostraram que uma busca com aprofundamento iterativo é mais eficiente que a mesma busca com profundidade fixa (MARSLAND, 1986).

O pseudo-código para o procedimento de busca alfa-beta com tabela de transposição e aprofundamento iterativo é apresentado a seguir. As linhas 6, 7 e 8 são as únicas que diferem do pseudo-código da seção 4.3.3.4.

*Pseudo-código do algoritmo fail-soft alfa-beta com tabela de transposição e aprofundamento iterativo*

```

1. fun alfaBeta(n:node,depth:int,min:int,max:int,bestmove:move):float =
2.   if leaf(n) or depth=0 then
3.     besteval := evaluate(n)
4.     store(n, besteval,bestmove,depth,hashExact)

```

```

5.     return besteval
6.   if root(n) then
7.     if not isEmpty(bestmove) then
8.       setChildrenOrder(n,bestmove)
9.   if n is a max node
10.    ...
11.    ...
12.    ...
13.  if n is a min node
14.    ...
15.    ...
16.    ...

```

**linha 6.** A ordenação da árvore de busca acontecerá em seu primeiro nível, ou seja, somente os filhos do estado do tabuleiro presente na raiz da árvore serão ordenados;

**linha 7.** A variável *bestmove* estará vazia no primeiro passo da iteração, quando o procedimento *alfaBeta* tiver sido chamado com profundidade  $depth = 2$ . Nos demais passos ( $depth \geq 4$ ), *bestmove* conterá o melhor movimento da iteração anterior;

**linha 8.** Utilizando o melhor movimento *bestmove* da iteração anterior, a árvore de busca é ordenada em seu primeiro nível da maneira mostrada na figura 30;

Para executar o procedimento *alfaBeta* repetidas vezes é necessário uma rotina de iteração. O pseudo-código dela pode ser visualizado abaixo:

*Rotina de iteração*

```

1. fun iterative(n:node,depth:int,min:int,max:int,bestmove:move):float =
2.   besteval = alfaBeta(n, 2, min, max, bestmove)
3.   start = clock()
4.   for (d=4; d<=MAX_SEARCH_DEPTH; d=d+2)
5.     besteval = alfaBeta(n, d, min, max, bestmove)
6.     stop = clock()
7.     if ((stop - start) > MAX_EXECUTION_TIME)
8.       return besteval
9.   return besteval

```



- linha 1.** A rotina de iteração possui os mesmos parâmetros utilizados no procedimento de busca alfa-beta com tabela de transposição;
- linha 2.** O primeiro passo de iteração é realizado com profundidade igual a 2. O melhor movimento a ser realizado é armazenado em *bestmove* e a predição em *besteval*;
- linha 3.** A variável *start* representa o início da contagem de tempo disponível para que o agente jogador encontre o melhor movimento a ser realizado;
- linha 4.** O procedimento iterativo se repetirá até que o limite de tempo ou profundidade de busca máxima seja alcançada;
- linha 5.** Representa os passos de iteração com profundidade  $depth \geq 4$ ;
- linha 6.** A variável *stop* representa o fim cronológico de um passo de iteração;
- linha 8.** Se o tempo máximo estabelecido para que o agente jogador encontre o melhor movimento a ser realizado se esgotar, o procedimento iterativo deve ser encerrado com retorno do melhor movimento *besteval* encontrado até o presente momento;
- linha 9.** Retorna o melhor movimento *besteval* após a profundidade máxima de busca ter sido atingida.

Como o algoritmo alfa-beta do *VisionDraughts* utiliza tabela de transposição, o procedimento de busca não perde eficiência, apesar de montar várias árvores de busca (uma árvore de busca para cada nível de profundidade), além de obter os benefícios oriundos da utilização de busca com profundidade variável: mecanismos de controle do tempo de execução e de ordenação dos estados filhos da raiz da árvore do jogo.

Com a tabela de transposição, existiria uma maneira ainda mais eficiente do que a mostrada acima para implementar o aprofundamento iterativo, conforme explicado a seguir. A tabela de transposição armazena, dentre outras informações, o valor da predição *besteval* e o melhor movimento *bestmove* a ser executado a partir de um estado do tabuleiro  $S_0$ . Caso ocorra uma transposição e o estado  $S_0$  seja revisitado, as informações da tabela de transposição podem ser utilizadas para ordenar a árvore de busca. Assim, a ordenação não seria executada apenas na raiz da árvore (como faz o *VisionDraughts*), mas sempre que existir tal possibilidade. No entanto, a ordenação na raiz da árvore de busca já é suficiente para os objetivos propostos para o *VisionDraughts*: mecanismo de controle de tempo e busca em profundidade variável.

Portanto, o terceiro grande avanço do *VisionDraughts* em relação ao sistema *NeuroDraughts* é conseguido com o uso da rotina de aprofundamento iterativo. Tal rotina contribui parcialmente para a diminuição da ocorrência do *problema do loop*, identificado por Neto e Julia (NETO, 2007) (NETO; JULIA, 2007b) (NETO; JULIA, 2007a) no trabalho de Lynch (LYNCH, 1997) (LYNCH; GRIFFITH, 1997). O *problema do loop* representa situação na qual o agente jogador de damas, apesar de estar em vantagem com relação a seu adversário, não consegue pressioná-lo e entra em *loop* infinito. O aprofundamento iterativo contribui parcialmente com o *problema do loop*, pois em algumas situações o jogador consegue visualizar jogadas diferentes com um *look-ahead* maior e variável.

## 4.4 O Algoritmo Alfa-Beta com Tabela de Transposição e Bases de Dados de Finais de Jogos

O uso de bases de dados para as fases finais do jogo de damas tem papel fundamental na construção de agentes jogadores de alto nível, pois embora o estado do tabuleiro que está sendo avaliado (raiz da árvore) possa estar longe do fim do jogo, alguns dos estados representados pelas folhas já podem estar nas bases de dados (LAKE; SCHAEFFER; LU, 1994).

Essencialmente, a construção de bases de dados para fases finais do jogo é realizada através de uma “*busca para trás*”, partindo da solução do problema. Tal técnica, combinada com o algoritmo alfa-beta que realiza uma “*busca para frente*”, permite que se encontre a solução ótima para o problema de escolha da melhor ação em menos tempo. O grande sucesso do programa *Chinook*, por exemplo, se baseia principalmente em suas bases de dados de finais de jogos (LAKE; SCHAEFFER; LU, 1994).

O espaço de busca do jogo de damas possui aproximadamente  $5 \times 10^{20}$  estados distintos do tabuleiro. Sendo assim, a construção de bases de dados para as fases finais do jogo é uma árdua tarefa. No projeto *Chinook*, por exemplo, os esforços para construção das bases de dados se iniciaram em 1989 e, desde então, quase continuamente, dezenas de computadores trabalham exclusivamente com este intuito (SCHAEFFER et al., 2007). Em 1992, no pico dos trabalhos, existiam mais de 200 computadores trabalhando simultaneamente na construção das bases de dados (SCHAEFFER, 2002).

Considerando, portanto, a dificuldade de construção de bases de dados próprias, o *VisionDraughts* utiliza as amostras disponibilizadas para download pela equipe do *Chinook* (SCHAEFFER et al., 2008). Tais amostras possuem mais 5.0 GB de tamanho e informação

perfeita (vitória, derrota ou empate) para os seguintes estados do tabuleiro, envolvendo 8 peças ou menos:

1. Todos os estados do tabuleiro com 6 ou menos peças;
2. Os estados do tabuleiro formados pela combinação 4 peças  $\times$  3 peças;
3. Os estados do tabuleiro formados pela combinação 4 peças  $\times$  4 peças;

Atualmente, o *Chinook* tem acesso às bases de dados contendo todas as combinações de estados do tabuleiro com 10 peças ou menos (SCHAEFFER et al., 2007). Mesmo considerando que o *VisionDraughts* utiliza apenas uma amostra das bases de dados de 8 peças ou menos, seu desempenho é melhorado, consideravelmente, em dois sentidos:

1. Quando um estado do tabuleiro é encontrado em uma base de dados, o *VisionDraughts* utiliza a predição exata para o mesmo (vitória, empate ou derrota), encontrada na base de dados, em vez de utilizar a função de avaliação heurística.
2. Quando um estado do tabuleiro é encontrado em uma base de dados, o *VisionDraughts* não precisa pesquisar nenhum dos descendentes do mesmo na árvore de busca (grandes porções da árvore de busca podem ser eliminadas, ou seja, grandes porções da árvore de busca deixam de ser pesquisadas pelo algoritmo alfa-beta).

O efeito combinado dos benefícios acima resulta em um mecanismo de busca que expande uma menor árvore de busca e alcança resultados mais precisos (SCHAEFFER, 2002), uma vez que mais estados do tabuleiro são associados com predições exatas (vitória, derrota ou empate) extraídas das bases de dados.

Sendo assim, de posse das bases de dados, resta detalhar os procedimentos utilizados pelo *VisionDraughts* para consultar se um determinado estado do tabuleiro encontra-se presente nas bases e como utilizá-las em conjunto com o algoritmo fail-soft alfa-beta com tabelas de transposição, mostrado na seção 4.3.3.4.

Felizmente, Ed Gilbert, autor do *KingsRow*, disponibilizou em seu site uma biblioteca de funções para acessar as bases de dados de seu próprio jogador e dos jogadores *Cake* e *Chinook*. A criação da biblioteca de funções por Ed Gilbert teve como objetivo facilitar, para os programadores de agentes jogadores de damas, o uso de qualquer uma das bases de dados disponibilizadas publicamente e encapsular as dificuldades inerentes aos complexos

algoritmos utilizados para acessar, de maneira eficiente, às bases de dados, principalmente quanto às técnicas de indexação e “caching”.

Para pesquisar estados do tabuleiro nas bases de dados, a biblioteca de funções fornecida pelo autor do *KingsRow* precisa ser utilizada da seguinte maneira:

1. As bases de dados precisam ser abertas através de uma função, no caso, chamada *egdb\_open(...)*;
2. A função *egdb\_open(...)* precisa ser testada a fim de detectar se houve algum tipo de erro durante a abertura das bases de dados. A função *egdb\_open(...)* pode ser chamada por vários processos sem que tal fato implique conflito (“safe for multi-threading”);
3. O primeiro argumento da função *egdb\_open(EGDB\_NORMAL, ...)* indica que tipo de representação do tabuleiro do jogo será utilizado para consultas, isto é, que tipo de *BitBoards* (seção 3.4) está sendo usado pelo agente jogador. A biblioteca de funções está preparada para representações do tabuleiro nos formatos definidos para os jogadores *Cake* ou *KingsRow* (as representações do tabuleiro no *Cake* e *Chinook* são idênticas). Como a numeração utilizada para as casas do tabuleiro são iguais nos jogadores *KingsRow* e *VisionDraughts*, é utilizado o argumento *EGDB\_NORMAL* para indicar que o *VisionDraughts* utiliza o esquema de representação do *KingsRow*;
4. O segundo argumento da função *egdb\_open(EGDB\_NORMAL, 8, ...)* representa o número máximo de peças permitidas no tabuleiro para que as bases de dados sejam consultadas. No caso do *VisionDraughts*, é utilizado o argumento 8 pois este é o número máximo de peças permitidas para as bases de dados disponibilizadas pela equipe do *Chinook*. Se um número menor que 8 for utilizado, menos memória RAM será necessária e a abertura das bases de dados será mais rápida. Se um número maior que a capacidade das bases de dados for utilizado, a função retornará com falha;
5. O terceiro argumento da função *egdb\_open(EGDB\_NORMAL, 8, 1024, ...)* representa a quantidade de memória em MB que será utilizada para as bases de dados. No *VisionDraughts*, a quantidade mínima é 1 GB de RAM.
6. Os dois últimos argumentos da função *egdb\_open(EGDB\_NORMAL, 8, 1024, db-dir, msg\_fn)* representam a localização das bases de dados e a mensagem que será retornada durante o procedimento;

7. Com as bases de dados abertas, a função *lookup(...)* é utilizada para realizar a busca de um determinado estado do tabuleiro;
8. Os quatro argumentos da função *lookup(db, pos, color, onlyMemory)* indicam, respectivamente, as bases de dados abertas pela função *egdb\_open(...)*, o estado do tabuleiro que se deseja consultar, a cor do jogador que realizará o próximo movimento e, finalmente, se a consulta acontecerá só em memória ou, também, em disco;

As bases de dados dos jogadores *Chinook*, *Cake* ou *KingsRow* estão preparadas para trabalhar com *BitBoards* (seção 3.4). O *VisionDraughts* precisa converter sua representação interna do tabuleiro em *BitBoards* antes de consultá-las, pois o parâmetro *pos* da função *lookup*, mostrada no item 8, indica o estado do tabuleiro representado por *BitBoards*.

As bases de dados disponibilizadas pela equipe do *Chinook* armazenam, para cada estado do tabuleiro, a informação de que ele representa vitória, derrota ou empate. Por isso, elas são chamadas bases *WLD* (*win, loss or draw*). As bases de dados *WLD* não retornarão resultados válidos para estados nos quais o próximo movimento será de captura ou seria de captura caso fosse a vez do oponente jogar (limitação das bases *WLD* que utilizam apenas 2 bits por estado do tabuleiro). O *VisionDraughts*, neste casos, deve se certificar das restrições de captura antes de consultar as bases *WLD*.

Para fins de integração com o procedimento alfa-beta, as seguintes linhas são acrescentadas ao algoritmo mostrado na seção 4.3.3.4:

*Pseudo-código do algoritmo fail-soft alfa-beta com tabela de transposição e bases de dados de finais de jogos*

```

1. fun alfaBeta(n:node,depth:int,min:int,max:int,bestmove:move):float =
2.   if ((not isRoot(n)) and (isLookupBoard(n)))
3.     getBitBoards(n, BP,WP,K)
4.     db_value = lookup_positions(..., BP, WP, K)
5.     if (db_value==1) and (n is a min node)
6.       return -1.0
7.     if (db_value==1) and (n is a max node)
8.       return +1.0
9.     if (db_value==2) and (n is a min node)
10.      return +1.0

```

```

11.      if (db_value==2) and (n is a max node)
12.          return -1.0
13.      if (db_value==3)
14.          return 0.0
15.  if leaf(n) or depth=0 then
16.      ...
17.      ...
18.  if n is a max node
19.      ...
20.      ...
21.  if n is a min node
22.      ...
23.      ...

```

**linha 2.** A função *isLookupBoard* é utilizada para garantir que o tabuleiro do jogo cumpre os pré-requisitos necessários para que possa ser consultado junto às bases de dados (possui o número de peças adequado e respeita as restrições de captura). A função *isRoot* é utilizada para garantir que os estados consultados nas bases de dados sempre terão, no mínimo, um antecessor na árvore de busca (todos os estados da árvore com exceção de sua raiz);

**linha 3.** A função *getBitBoards* pega um estado do tabuleiro e gera os *BitBoards* (BP, WP, K) necessários para integração com as bases de dados.

**linha 4.** A função *lookup\_positions* realiza o procedimento de consulta junto às bases de dados;

**linha 5.** O resultado  $db\_value = 1$  indica que o estado do tabuleiro  $n$  representa vitória para o próximo jogador a se mover. Logo, o estado do tabuleiro  $n$  representa derrota para o estado pai de  $n$ . Caso  $n$  seja um minimizador, é possível concluir que o valor  $-1.0$  deve ser retornado na linha 6 para indicar que o pai de  $n$  (maximizador) encontra-se derrotado na partida de acordo com as bases de dados;

**linha 7.** O resultado  $db\_value = 1$  indica que o estado do tabuleiro  $n$  representa vitória para o próximo jogador a se mover. Logo, o estado do tabuleiro  $n$  representa derrota para o estado pai de  $n$ . Caso  $n$  seja um maximizador, é possível concluir que o valor  $+1.0$  deve ser retornado na linha 8 para indicar que o pai de  $n$  (minimizador) encontra-se derrotado na partida de acordo com as bases de dados;

- linha 9.** O resultado  $db\_value = 2$  indica que o estado do tabuleiro  $n$  representa derrota para o próximo jogador a se mover. Logo, o estado do tabuleiro  $n$  representa vitória para o estado pai de  $n$ . Caso  $n$  seja um minimizador, é possível concluir que o valor  $+1.0$  deve ser retornado na linha 10 para indicar que o pai de  $n$  (maximizador) encontra-se vitorioso na partida de acordo com as bases de dados;
- linha 11.** O resultado  $db\_value = 2$  indica que o estado do tabuleiro  $n$  representa derrota para o próximo jogador a se mover. Logo, o estado do tabuleiro  $n$  representa vitória para o estado pai de  $n$ . Caso  $n$  seja um maximizador, é possível concluir que o valor  $-1.0$  deve ser retornado na linha 12 para indicar que o pai de  $n$  (minimizador) encontra-se vitorioso na partida de acordo com as bases de dados;
- linha 13.** O resultado  $db\_value = 3$  indica que o estado do tabuleiro  $n$  representa empate para o próximo jogador a se mover. Logo, o estado do tabuleiro  $n$  representa empate para o estado pai de  $n$ . Portanto, o valor  $0.0$  deve ser retornado na linha 14 para indicar que o pai de  $n$  encontra-se empatado na partida de acordo com as bases de dados.

## 5 *Resultados Experimentais e Técnicas Adicionais*

### 5.1 Resultados Experimentais

#### 5.1.0.1 Impacto do Módulo Eficiente de Busca em Árvores de Jogos

O módulo de busca eficiente em árvores de jogos (figura 20) foi desenvolvido no *VisionDraughts* para fornecer ao jogador automático maior capacidade de analisar jogadas futuras (estados do tabuleiro mais distantes do estado corrente).

A figura 31 mostra o tempo de execução necessário para realizar 2 sessões com 10 jogos de treinamento utilizando as estratégias minimax do *NeuroDraughts*, alfa-beta e alfa-beta com tabela de transposição do *VisionDraughts* (durante uma partida de treinamento, os pesos de uma das redes neurais são ajustados pelo método  $TD(\lambda)$ ).

| Algoritmo                 | Tempo (minutos) | (%) MiniMax | (%) Alpha-Beta |
|---------------------------|-----------------|-------------|----------------|
| MiniMax                   | 441,27          | 100,00%     | 1715,67%       |
| Alpha-Beta                | 25,72           | 5,83%       | 100,00%        |
| Alpha-Beta com Tab Transp | 10,03           | 2,27%       | 39,00%         |

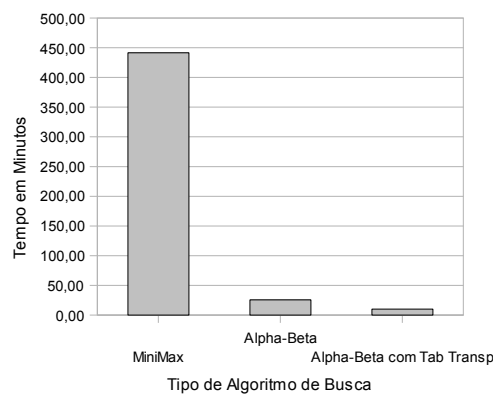


Figura 31: Tempo de treinamento: 2 sessões de 10 jogos e *look-ahead* 8.



Para se ter um bom parâmetro de comparação, os módulos de aprofundamento iterativo e acesso às bases de dados finais foram desativados no *VisionDraughts* e todos os algoritmos rodaram com profundidade de busca fixa e igual a 8.

Note que o algoritmo minimax necessita de 441,27 minutos para completar as duas sessões, enquanto o algoritmo alfa-beta básico necessita de 25,72 minutos (5,83%) e o algoritmo alfa-beta com tabela de transposição necessita de 10,03 minutos (2,27%).

No experimento da figura 31, o treinamento foi realizado apenas com 2 sessões de 10 jogos, devido ao tempo impraticável que levaria o algoritmo minimax para completar, com look-ahead igual a 8, a execução de mais sessões de jogos.

Deixando de lado o algoritmo minimax (tempo de execução impraticável), a figura 32 mostra o tempo de execução necessário para realizar 10 sessões com 200 jogos de treinamento utilizando as estratégias alfa-beta e alfa-beta com tabela de transposição.

| Algoritmo                 | Tempo (minutos) | (%) Alpha-Beta |
|---------------------------|-----------------|----------------|
| Alpha-Beta                | 46,52           | 100,00%        |
| Alpha-Beta com Tab Transp | 16,79           | 36,09%         |

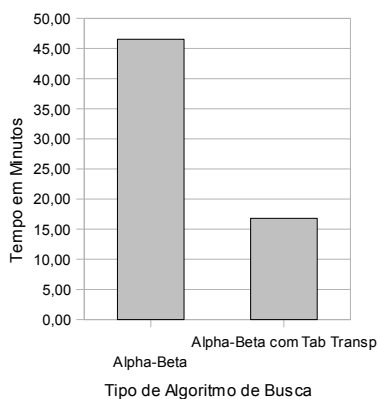


Figura 32: Tempo de treinamento: 10 sessões de 200 jogos e *look-ahead* 8.

Da mesma forma que o experimento anterior, os módulos de aprofundamento iterativo e acesso às bases de dados finais foram desativados no *VisionDraughts* e a profundidade de busca foi fixada em 8. Assim, foi possível comprovar a eficiência do uso de tabelas de transposição em 2.000 jogos de treinamento.

Note que o algoritmo alfa-beta básico necessita de 46,52 minutos enquanto o algoritmo alfa-beta com tabela de transposição necessita de apenas 16,79 minutos (36,09%).

Para verificar o impacto do módulo de busca eficiente no desempenho (nível de jogo) do *VisionDraughts*, 2 torneios com 14 jogos foram executados com os jogadores *VisionDraughts*, *NeuroDraughts* e *LS-Draughts*. Os módulos de aprofundamento iterativo e acesso às bases de dados finais foram desativados no *VisionDraughts* e a profundidade de busca foi fixada em 8. O *NeuroDraughts* e o *LS-Draughts* utilizaram seus próprios módulos de busca (minimax com profundidade fixa igual a 4). Os resultados dos torneios comprovaram a eficiência da utilização de um maior *look-ahead*:

1. *VisionDraughts* x *NeuroDraughts*: 5 vitórias para o *VisionDraughts*, 8 empates e 1 derrota;
2. *VisionDraughts* x *LS-Draughts*: 4 vitórias para o *VisionDraughts*, 8 empates e 2 derrotas;

Com os resultados apresentados nesta seção, um artigo chamado “*A Draughts Learning System Based on Neural Networks and Temporal Differences: The Impact of an Efficient Tree-Search Algorithm*” foi escrito e aceito para publicação (CAEXETA; JULIA, 2008).

#### 5.1.0.2 Impacto do Módulo de Acesso às Bases de Dados Finais

O módulo de acesso às bases de dados de finais de jogos (figura 20) foi desenvolvido no *VisionDraughts* para fornecer ao jogador automático capacidade de anunciar, antes do final da partida, se um estado do tabuleiro qualquer, com até 8 peças, representa vitória, derrota ou empate. Com as bases de dados, o jogador automático substitui informação heurística por conhecimento perfeito, consegue melhor ajuste em sua função de avaliação e torna-se mais eficiente.

A fim de verificar, experimentalmente, a viabilidade do uso do módulo de acesso às bases de dados, as seguintes perguntas foram levantadas em relação aos jogadores *NeuroDraughts*, *LS-Draughts* e *VisionDraughts*:

1. O uso de bases de dados contribui para o *VisionDraughts* superar o nível de jogo dos jogadores *NeuroDraughts* e *LS-Draughts*?
2. O uso de bases de dados contribui para o *VisionDraughts* superar o *problema do loop*, identificado por Neto e Julia (NETO, 2007) (NETO; JULIA, 2007b) (NETO; JULIA, 2007a), nos jogadores *NeuroDraughts* e *LS-Draughts*?

O primeiro jogador a ser treinado foi o *NeuroDraughts*: utilizando o conjunto de características de Lynch (LYNCH, 1997) (LYNCH; GRIFFITH, 1997), uma rede neural aleatória foi treinada por 10 sessões de 200 jogos com profundidade de busca igual a 4. Durante os 2.000 jogos, 1.045 partidas foram encerradas em *loop* indevido e o melhor jogador encontrado foi o CLONE[2] com aptidão igual a 6. Isso significa que o melhor jogador foi o segundo clone da rede neural que conseguiu uma taxa de vitória igual a 6 sobre os demais clones (a taxa de vitória é um parâmetro definido durante o treinamento de maneira que o clone com maior taxa de vitória represente o jogador com maior qualidade de jogo).

O segundo jogador a ser treinado foi o *LS-Draughts*: utilizando o conjunto de características de Neto e Julia (NETO, 2007) (NETO; JULIA, 2007b) (NETO; JULIA, 2007a), uma rede neural aleatória foi treinada por 10 sessões de 200 jogos com profundidade de busca igual a 4. Durante os 2.000 jogos, 759 partidas foram encerradas em *loop* indevido e o melhor jogador encontrado foi o CLONE[2] com aptidão igual a 12.

O terceiro jogador a ser treinado foi o *VisionDraughts*: utilizando o conjunto de características de Neto e Julia (NETO, 2007) (NETO; JULIA, 2007b) (NETO; JULIA, 2007a) e o módulo de acesso às bases de dados do *VisionDraughts*, uma rede neural aleatória foi treinada por 10 sessões de 200 jogos com profundidade de busca igual a 4. Durante os 2.000 jogos, 172 partidas foram encerradas em *loop* indevido e o melhor jogador encontrado foi o CLONE[3] com aptidão igual a 15.

A figura 33 sintetiza os dados de treinamento e mostra que 1045 partidas foram encerradas em *loop* no *NeuroDraughts*, 759 partidas foram encerradas em *loop* no *LS-Draughts* e 172 partidas no *VisionDraughts*. Isso significa que o *VisionDraughts* apresenta o *problema do loop* a uma taxa de 16,46% em relação ao *NeuroDraughts* e 22,66% em relação ao *LS-Draughts*.

| Partidas encerradas com o problema do loop |             |                |
|--|-------------|----------------|
| NeuroDraughts                              | LS-Draughts | VisionDraughts |
| 1045                                       | 759         | 172            |
| 100%                                       | 72,63%      | 16,46%         |
| 138%                                       | 100,00%     | 22,66%         |
| 608%                                       | 441,28%     | 100,00%        |

Figura 33: O *problema do loop* e o uso de bases de dados.

Considerando que uma partida encerrada em *loop* indevido implica uma recompensa não muito precisa para o treinamento TD( $\lambda$ ), pode-se perceber que o impacto das bases de dados finais sobre o jogador de Lynch é maior que sobre o jogador de Neto e Julia (o jogador de Lynch tem muito mais partidas encerradas em *loop* indevido).

Por outro lado, para verificar o desempenho de cada um dos jogadores e, especialmente, verificar se o uso de bases de dados faz com que o *VisionDraughts* supere o *NeuroDraughts* e o *LS-Draughts* (todos os jogadores com *look-ahead* fixo e igual a 4), eles foram colocados para disputar 14 partidas e o resultado foi:

1. LUTA 1: *LS-Draughts* x *VisionDraughts*

- Numero de vitórias do *VisionDraughts*: 3;
- Numero de empates do *VisionDraughts*: 10 (3 empates reais e 7 loops indevidos);
- Numero de derrotas do *VisionDraughts*: 1;

2. LUTA 2: *NeuroDraughts* x *VisionDraughts*

- Numero de vitórias do *VisionDraughts*: 6;
- Numero de empates do *VisionDraughts*: 8 (5 empates reais e 3 loops indevidos);
- Numero de derrotas do *VisionDraughts*: 0;

3. LUTA 3: *NeuroDraughts* x *LS-Draughts*

- Numero de vitórias do *LS-Draughts*: 5;
- Numero de empates do *LS-Draughts*: 8 (6 empates reais e 2 loops indevidos);
- Numero de derrotas do *LS-Draughts*: 1;

Portanto, o uso de bases de dados contribuiu para diminuir, consideravelmente, o número de partidas encerradas indevidamente em *loop* durante o treinamento e impactou, diretamente, na eficiência *VisionDraughts* (o melhor jogador).

Os resultados apresentados nesta seção foram agregados ao trabalho de Neto e Julia (NETO; JULIA, 2007a) e aceitos para publicação de um capítulo de livro, intitulado “*LS-Draughts: Using Databases to Treat Endgame Loops in a Hybrid Evolutionary Learning System*” (NETO; JULIA; CAEXETA, 2008).

### 5.1.0.3 Impacto do Módulo de Aprofundamento Iterativo

A seção 2.1.4 mostrou que o aprofundamento iterativo combina os benefícios de busca em largura e busca em profundidade. Como na busca em largura, ele é completo quando o fator de ramificação é finito e ótimo quando o custo do caminho é proporcional à

profundidade. Korf descreve, através de teorema, que a busca em profundidade com aprofundamento iterativo é assintoticamente ótima, dentre as buscas “brute-force”, em termos de tempo, espaço e comprimento da solução (KORF, 1985).

Por outro lado, a grande desvantagem do aprofundamento iterativo é o processamento repetido de estados em níveis mais rasos da árvore de busca que acontece antes de se encontrar a profundidade do estado objetivo do problema (seção 2.1.5). Com o uso de tabelas de transposição e aprofundamento iterativo, porém, tal problema é solucionado e o procedimento de busca em profundidade é aprimorado para expandir árvores em uma sequência típica da estratégia de busca pela melhor escolha (seção 2.1.6).

O algoritmo de busca alfa-beta com tabelas de transposição forneceu ao *VisionDraughts* a capacidade de treinar um jogador automático de damas em apenas 2,27% do tempo gasto pelo *NeuroDraughts* (seção 5.1.0.1). Assim, o módulo de aprofundamento iterativo foi implementado como um mecanismo de controle do tempo máximo permitido para que o jogador automático escolha uma ação a ser executada e como tentativa de diminuir a ocorrência do *problema do loop*, identificado por Neto e Julia (NETO, 2007) (NETO; JULIA, 2007b) (NETO; JULIA, 2007a) nos jogadores *NeuroDraughts* e *LS-Draughts*.

Uma rede neural, criada aleatoriamente, foi treinada no *VisionDraughts* por 10 sessões de 100 jogos com *look-ahead* igual a 4 e os módulos de aprofundamento iterativo e bases de dados desativados. Dentre as 1.000 partidas, 650 encerraram-se indevidamente em *loop*. Por outro lado, ativando-se o módulo de aprofundamento iterativo e fornecendo ao sistema um tempo suficiente para que ele execute uma busca alfa-beta de profundidade 4 e prossiga, iterativamente, com profundidades acrescidas de dois em dois, na quantidade desejada e definida pelo tempo concedido, apenas 309 partidas encerraram-se indevidamente em *loop*.

Considerando que uma partida encerrada em *loop* indevido implica uma recompensa não muito precisa para o treinamento  $TD(\lambda)$ , pode-se perceber que a utilização de busca com profundidade fixa não é viável e que a busca com profundidade variável do *VisionDraughts* (aprofundamento iterativo) contribuiu para atenuar o problema do *loop*.

## 5.2 Técnicas Utilizadas Durante e Após o Treinamento do *VisionDraughts*

O funcionamento básico do *VisionDraughts* foram apresentados na figura 20. No início do processo de treinamento (seção 4.1.4), uma rede neural com pesos gerados ale-

atoriamente é clonada para que se inicie uma sessão de  $k$  jogos de treino entre ambas. Durante os  $k$  jogos, somente os pesos da rede neural original são ajustados com o uso da técnica  $TD(\lambda)$ . Ao final dos  $k$  jogos, 2 outros jogos são disputados (sem ajuste de pesos) para verificar se a rede original supera seu clone. Somente em caso afirmativo, os pesos da rede original são copiados para o clone. Uma nova sessão de  $k$  jogos de treino se inicia e o processo se repete por  $m$  sessões.

Veja quais técnicas podem ser utilizadas durante e após o treinamento do *VisionDraughts*:

1. Treinamento  $TD(\lambda)$  e *Self-Play com clonagem*: só existe atualização de pesos durante a fase de treinamento. Mesmo nela, durante os  $k$  jogos de uma mesma sessão, somente uma das redes neurais tem seus pesos atualizados. Os 2 jogos disputados após cada uma das  $m$  sessões de  $k$  jogos não utilizam atualização de pesos. Terminada a fase de treinamento, os pesos da melhor rede neural são gravados em arquivo e não sofrem mais alterações;
2. Alfa-beta: a utilização de podas alfa e beta agiliza o processo de escolha da melhor ação a ser executada por um jogador automático. A melhor ação é, exatamente, a mesma que seria obtida pelo algoritmo minimax. Assim, o algoritmo alfa-beta pode ser utilizado em todas as fases do *VisionDraughts*;
3. Tabelas de transposição: durante os  $k$  jogos de uma mesma sessão, uma das redes neurais tem seus pesos atualizados, pelo método  $TD(\lambda)$ , a cada jogada. Isso impede que predições e melhores ações armazenadas em memória possam ser utilizadas, pois predições calculadas previamente são incompatíveis com pesos recém ajustados. Conseqüentemente, durante os  $k$  jogos de uma mesma sessão, somente a rede neural que não sofre atualização de pesos utiliza tabela de transposição. Durante os 2 jogos disputados após cada uma das  $m$  sessões de  $k$  jogos de treinamento, ambas as redes utilizam tabela de transposição (uma tabela para cada uma das redes). Após a fase de treinamento, inexistem restrições quanto ao uso de tabelas de transposição;
4. Aprofundamento iterativo: a técnica de aprofundamento iterativo pode ser utilizada em qualquer fase do *VisionDraughts* (não existem restrições técnicas);
5. Bases de dados: a utilização de bases de dados para as fases finais do jogo agiliza o processo de escolha da melhor ação a ser executada por um jogador automático. Além disso, torna mais eficiente o processo de escolha da melhor ação na medida

em que substitui heurística por informação perfeita presente nas bases (vitória, derrota ou empate). Assim, bases de dados podem ser utilizadas em qualquer fase do *VisionDraughts* (não existem restrições técnicas).

## 5.3 Ferramenta Utilizada na Implementação do *VisionDraughts*

O *VisionDraughts* é um projeto de interesse acadêmico e científico desenvolvido, principalmente, para explorar a utilização de técnicas eficientes de inteligência artificial.

Mark Lynch explorou de maneira muito interessante o uso de redes neurais e aprendizagem por reforço  $TD(\lambda)$  na construção de agentes inteligentes. O *NeuroDraughts* ficou muito bem documentado (LYNCH, 1997) (LYNCH; GRIFFITH, 1997) e os fontes do projeto foram disponibilizados para que melhorias futuras fossem implementadas.

Neto e Julia iniciaram o processo de evolução do *NeuroDraughts* e exploraram o uso de algoritmos genéticos para construir o *LS-Draughts* (NETO, 2007), (NETO; JULIA, 2007b), (NETO; JULIA, 2007a). Os bons resultados obtidos pelo *LS-Draughts* identificaram, claramente, o potencial ainda inexplorado para aplicação de técnicas de inteligência artificial no *NeuroDraughts*.

Assim, visando dar continuidade aos trabalhos de Lynch, Neto e Julia, o *VisionDraughts* utiliza a mesma linguagem de programação utilizada por eles (C++). No início de seu desenvolvimento, tentou-se utilizar a plataforma .NET com C#. Com ela, uma versão básica do procedimento de busca do *VisionDraughts* chegou a ser implementada. Após comparação com o tempo de execução necessário em C++, o projeto em plataforma .NET foi abandonado (o trabalho aqui proposto não possui intenção de comparar linguagens de programação, a plataforma .NET foi abandonada simplesmente porque a versão implementada para o *VisionDraughts* não mostrou-se satisfatória).

## 5.4 Outras Técnicas Implementadas durante o Desenvolvimento do *VisionDraughts*

### 5.4.1 O Algoritmo *MTD-f*

Nos últimos anos, vários pesquisadores descobriram e implementaram com sucesso diversas melhorias para o algoritmo alfa-beta. Dentre elas, aprofundamento iterativo,

tabelas de transposição e uso de janelas limitadas de busca são amplamente utilizadas (PLAAT, 1996).

Apesar de o algoritmo alfa-beta ser classificado como uma estratégia de busca em profundidade, as melhorias citadas no parágrafo anterior permitem expandir árvores simulando a estratégia de busca pela melhor escolha (seção 2.1.6).

Plaata e Schaeffer, apesar do grande sucesso do algoritmo alfa-beta, concluem que o algoritmo *MTD-f* (*Memory Enhanced Test Driver*) representa um novo e melhor paradigma para as buscas *minimax* (PLAAT et al., 1995) (PLAAT, 1996). Neste sentido, o *VisionDraughts* foi adaptado para possuir, também, um módulo de busca baseado no algoritmo *MTD-f*.

Para garantir a eficiência e viabilidade do algoritmo *MTD-f*, a utilização correta das tabelas de transposição é fundamental. Como o *VisionDraughts* já possui um módulo de busca baseado no algoritmo alfa-beta (versão *fail-soft*) com tabelas de transposição, a tarefa de adaptação pode ser realizada com auxílio da seguinte rotina:

#### *Memory Enhanced Test Driver*

```

1. fun mtdf(n:node,f:float,d:int,min:int,max:int,bestm:move):float =
2.     upperbound = +INFINITY
3.     lowerbound = -INFINITY
4.     besteval = f
5.     repeat
6.         if isEqual(besteval, lowerbound)
7.             then max = besteval + 0.001
8.             else max = besteval
9.         besteval = alfaBeta(n,d,(max - 0.0001),max,bestm)
10.    if (besteval < max)
11.        then upperbound = besteval
12.        else lowerbound = besteval
13.    until (lowerbound >= upperbound)
14.    return besteval

```

**linha 1.** O argumento *f* representa uma previsão do valor da predição associada ao estado *n*. Os demais argumentos são idênticos aos apresentados para o algoritmo *fail-soft* alfa-beta (seção 4.3.3.4);



**linha 6.** As linhas 6, 7 e 8 são utilizadas, simplesmente, para evitar que o valor de *max* seja igual ao valor de *lowerbound*. Este ajuste no valor de *max* é necessário devido ao intervalo de busca utilizado na linha 9. Em outras palavras, o ajuste impede que o valor mínimo do intervalo de busca ( $max - 0.0001$ ), utilizado na linha 9, seja menor que o limite inferior garantido pela variável *lowerbound*;

**linha 9.** O algoritmo *MTD-f* funciona chamando, várias vezes, a rotina *fail-soft* alfa-beta com tabela de transposição (seção 4.3.3.4). Nas chamadas, as janelas de busca devem ter tamanho nulo (uma janela de busca é definida pelo intervalo entre os parâmetros *alfa* e *beta*). Normalmente, o algoritmo alfa-beta é chamado com uma larga janela de busca, por exemplo, `alfaBeta(n, depth, -INFINIY, +INFINITY, bestmove)`. No *MTD-f*, as janelas de tamanho nulo são utilizadas para que cada chamada ao procedimento alfa-beta retorne sempre um limite (inferior ou superior ao valor exato da predição) associado ao estado *n*. Janelas de tamanho nulo causam muito mais podas, porém retornam menos informação (apenas um limite no valor exato da predição);

**linha 10.** As linhas 10, 11 e 12 são utilizadas para identificar qual limite foi retornado pela chamada ao procedimento alfa-beta com janela de busca de tamanho nulo (limite superior ou inferior);

**linha 13.** O procedimento alfa-beta com janela de busca de tamanho nulo é chamado, repetidas vezes, até que os valores dos limites inferior e superior se encontrem. Neste momento, o valor exato da predição associada ao estado *n* pode ser retornado pela linha 14.

Resumindo, o algoritmo *MTD-f* funciona chamando, repetidas vezes, o alfa-beta com janela de busca de tamanho nulo. Cada chamada ao procedimento alfa-beta retorna um limite (superior ou inferior) para o valor exato da predição associada ao estado *n*. Quando os limites assegurados pelas variáveis *upperbound* e *lowerbound* se encontrarem, o valor exato da predição terá sido descoberto. Quanto melhor o valor da previsão inicial *f*, menos chamadas serão necessárias ao procedimento alfa-beta. No melhor caso, se o valor de *f* for exatamente igual ao valor da predição associada à *n*, o algoritmo *MTD-f* chamará o alfa-beta uma vez para encontrar o limite superior e outra para encontrar o limite inferior (neste momento, o limite superior será igual ao limite inferior e igual ao valor exato da predição associada ao estado *n*).

### 5.4.2 Mapeamento Espacial do Tabuleiro

Fogel argumenta que o *Chinook* não utiliza técnicas de auto-aprendizagem, mas que todo seu conhecimento se baseia em perícia humana, específica para o jogo de damas (CHELLAPILLA; FOGEL, 2000) (CHELLAPILLA; FOGEL, 2001) (FOGEL; CHELLAPILLA, 2002). Segundo ele, o conhecimento do *Chinook* foi programado por humanos através de:

1. Características específicas do domínio de damas escolhidas manualmente;
2. Coleções de aberturas de jogos de grandes mestres;
3. Bases de dados enumerando todos os estados possíveis das fases finais do jogo como vitória, derrota ou empate;
4. Alta capacidade de processamento para buscar adiante (*look-ahead*) quantas posições forem possíveis.

O *Anaconda* foi desenvolvido utilizando um algoritmo co-evolutivo e a arquitetura de rede neural mostrada na figura 34. Nela, a camada de entrada da rede representa diversas áreas do tabuleiro do jogo (todas as áreas de 3 x 3 casas, todas as áreas de 4 x 4 casas, todas as áreas de 5 x 5 casas e assim sucessivamente). Assim, segundo Fogel, a rede neural se torna apta a inventar características baseando-se, apenas, nas disposições espaciais das peças no tabuleiro.

Neste sentido, uma versão alternativa do *VisionDraughts* foi implementada utilizando o modelo de rede neural com mapeamento espacial, proposto por Fogel, e o método de aprendizagem por diferenças temporais (uma abordagem interessante pois elimina o uso de características do jogo de damas para treinar a rede neural).

Como a arquitetura mostrada na figura 34 possui muito mais neurônios e ligações do que a rede utilizada pelo *VisionDraughts*, o processo de treinamento tornou-se lento, o que acabou inviabilizando a continuidade das pesquisas nesta direção.

### 5.4.3 Mapeamento do Tabuleiro por Chave Hash

O *VisionDraughts* utiliza características específicas do domínio do jogo de damas para realizar o mapeamento do tabuleiro na entrada da rede neural. Uma versão alternativa do jogador automático foi implementada utilizando a seguinte abordagem:

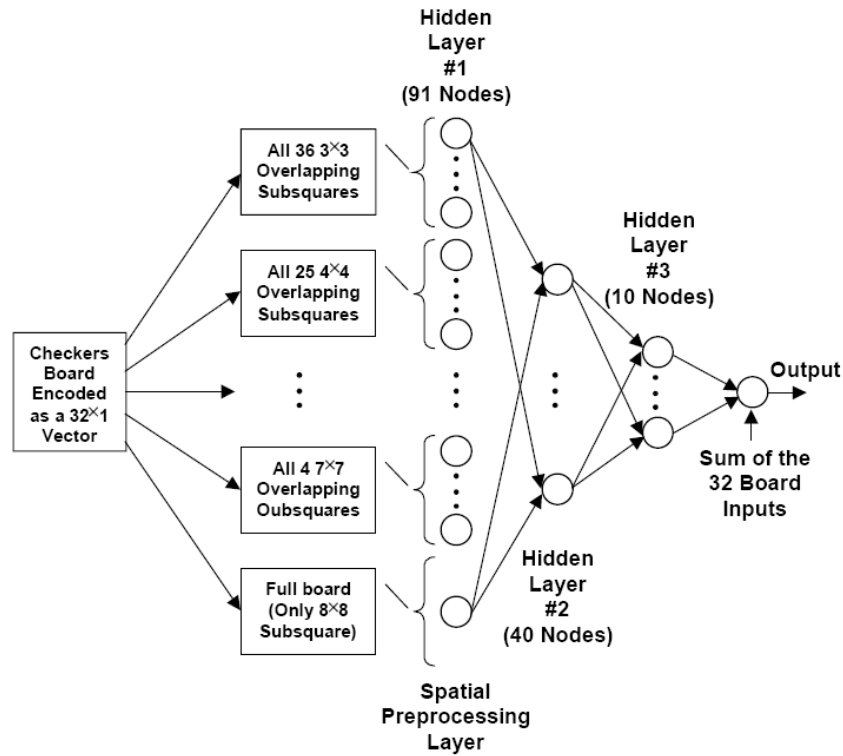


Figura 34: Mapeamento espacial utilizado por Fogel.

1. A chave hash *hashvalue*, mostrada na seção 4.3.2.3 e criada para indexação de cada um dos estados visitados do tabuleiro, é transformada em uma seqüência binária  $B_1$  com 64 bits;
2. A chave hash *checksum*, mostrada na seção 4.3.2.3 e criada para indexação de cada um dos estados visitados do tabuleiro, é transformada em uma seqüência binária  $B_2$  com 32 bits;
3. As duas seqüências binárias  $B_1$  e  $B_2$  são concatenadas para formar uma terceira seqüência binária  $B_3$  com 96 bits;
4. Cada um dos 96 bits de  $B_3$  passa a representar o valor de entrada de um dos 96 neurônios da camada de entrada de uma rede neural (similar à rede do *VisionDraughts*).

O desempenho do jogador automático crescia à medida em que o treinamento ia sendo realizado. Porém, mesmo após diversas sessões de treinamento, a versão do *VisionDraughts* com mapeamento por chave *hash* não conseguiu obter nível de jogo compatível com a versão treinada com o uso de características.

## 6 Conclusões

O domínio do jogo de damas foi escolhido criteriosamente para ser utilizado no presente trabalho por possuir inúmeras semelhanças com diversos problemas práticos da vida real e por apresentar uma complexidade que demanda a utilização de poderosas técnicas de inteligência artificial (capítulo 1).

Baseando-se nos jogadores *NeuroDraughts* (LYNCH, 1997) (LYNCH; GRIFFITH, 1997) e *LS-Draughts* (NETO, 2007) (NETO; JULIA, 2007b) (NETO; JULIA, 2007a), o *VisionDraughts* foi construído a partir de um estudo detalhado e utilização das seguintes técnicas:

1. Redes neurais artificiais multi-camadas;
2. Aprendizagem por reforço com ênfase no método das diferenças temporais TD( $\lambda$ );
3. Módulo de busca eficiente em árvores de jogos: algoritmo alfa-beta, tabelas de transposição (com chaves *Zobrist* e tratamento de colisões) e aprofundamento iterativo. Uma versão alternativa para o módulo de busca foi criada com o algoritmo de busca pela melhor escolha *MTD-f*;
4. Módulo de acesso às bases de dados das fases finais do jogo. Com as bases de dados, os estados do tabuleiro com  $\leq 8$  peças passam a ter valor teórico definido (vitória, derrota ou empate).

A equipe do *Chinook*, ao anunciar que o jogo de damas está resolvido (*weakly solved*) e que termina em empate quando jogado de maneira perfeita pelos dois adversários, relata que, talvez, a maior contribuição de se utilizar técnicas de inteligência artificial no desenvolvimento de jogadores automáticos seja a concretização do seguinte fato: o uso da abordagem de busca intensa com “*brute-force*” pode resultar em sistemas com altíssimo nível de desempenho, construídos com mínimo conhecimento dependente do domínio de uma aplicação (SCHAEFFER et al., 2007).

Assim, a construção de um módulo eficiente de busca foi fundamental para o sucesso

do *VisionDraughts*. Enquanto o *NeuroDraughts* e o *LS-Draughts* contam com um sistema básico de busca minimax que utiliza profundidade fixa de busca igual a quatro, o *VisionDraughts* conta com um módulo alfa-beta com tabelas de transposição e aprofundamento iterativo que lhe permite ajustar os pesos de sua rede neural de maneira muito mais precisa (*look-ahead* maior e variável).

Neto e Julia (NETO, 2007) (NETO; JULIA, 2007b) (NETO; JULIA, 2007a) identificaram o *problema do loop* nos jogadores *NeuroDraughts* e *LS-Draughts*. O *problema do loop* representa situação na qual o jogador automático, apesar de estar em vantagem em relação a seu adversário, não consegue pressioná-lo e entra em *loop* infinito (sem conseguir progredir rumo à “vitória óbvia”). Seguindo a sugestão dos próprios autores do *LS-Draughts*, o *VisionDraughts* foi construído com um módulo de acesso às bases de dados das fases finais do jogo (bases disponibilizadas pela equipe do *Chinook* (SCHAEFFER et al., 1992)).

Os resultados experimentais obtidos com o *VisionDraughts* mostram que as ocorrências do *problema do loop* foram, consideravelmente, reduzidas com o uso das bases de dados e da busca em profundidade variável com aprofundamento iterativo (seções 5.1.0.2 e 5.1.0.3). A simples utilização de uma maior profundidade de busca (*look-ahead*) já forneceu ao *VisionDraughts* capacidade de superar o nível de jogo do *NeuroDraughts* e do *LS-Draughts* (seção 5.1.0.1).

## 6.1 Perspectiva de Trabalhos Futuros

Apesar do bom desempenho geral do *VisionDraughts*, muitas técnicas ainda podem ser aplicadas com sucesso em sua arquitetura geral:

1. Utilizar *BitBoards* para representar internamente o tabuleiro em vez da representação com o vetor de 32 elementos mostrado na seção 4.1.1. Os *BitBoards* são utilizados por, praticamente, todos os jogadores automáticos de damas que possuem alto nível de desempenho (jogador de Samuel, *Chinook*, *Cake*, *KingsRow*). Além disso, o próprio *VisionDraughts* precisa converter sua representação interna para *BitBoards* antes de consultar as bases de dados finais. Uma boa referência teórica e prática para a utilização dos *BitBoards* pode ser encontrada em (RASMUSSEN, 2004);
2. Apesar de os resultados experimentais terem mostrado uma considerável redução nas ocorrências do *problema do loop*, ele ainda acontece no *VisionDraughts*. Uma versão

multiagente poderia ser utilizada de forma que cada um dos agentes se especializasse em uma das fases do jogo (início, meio e fim), conforme sugestão dos autores do LS-Draughts (NETO, 2007) (NETO; JULIA, 2007b) (NETO; JULIA, 2007a);

3. Os resultados expressivos de busca obtidos pelo *VisionDraughts* podem ser aprimorados ainda mais com a utilização de técnicas paralelas, principalmente, considerando a popularização dos processadores com vários núcleos de processamento. Boas referências teóricas e práticas para a utilização de técnicas de busca paralelas podem ser encontrada em (RASMUSSEN, 2004) (BROCKINGTON; SCHAEFFER, 2000) (KISHIMOTO; SCHAEFFER, 2002);
4. O principal objetivo da utilização do aprofundamento iterativo no *VisionDraughts* foi atacar o *problema do loop*. Para isso, a ordenação parcial da árvore de busca, utilizando o melhor movimento descoberto nas iterações anteriores (iteraões já concluídas) e possivelmente armazenado na tabela de transposição, foi realizada apenas na raiz da árvore, da maneira mostrada na figura 30. Assim, utilizar o procedimento iterativo com tabela de transposição para ordenar, parcialmente, todos os estados da árvore anteriormente pesquisados (e não apenas o estado raiz) é uma boa opção de trabalho futuro;
5. Schaeffer define o algoritmo *MTD-f* como um novo paradigma de busca em árvores de jogos e conclui que ele supera as qualidades dos algoritmos baseados no alfa-beta (PLAAT et al., 1995). Neste sentido, foi desenvolvida uma versão alternativa do *VisionDraughts* com o uso do algoritmo *MTD-f*. A versão do *MTD-f* implementada pelo *VisionDraughts* (seção 5.4.1) mostrou desempenho similar, em termos de tempo de execução, ao procedimento alfa-beta com tabela de transposição e aprofundamento iterativo (seção 4.3.5). Assim, uma abordagem mais detalhada pode ser realizada com o algoritmo *MTD-f*, principalmente porque Plaat (PLAAT, 1996) argumenta que uma abordagem paralela do *MTD-f* pode ser mais simples do que uma baseada no alfa-beta.
6. Fogel argumenta que o *Chinook* não utiliza técnicas de auto-aprendizagem, mas que todo seu conhecimento se baseia em perícia humana, específica para o jogo de damas (CHELLAPILLA; FOGEL, 2000) (CHELLAPILLA; FOGEL, 2001) (FOGEL; CHELLAPILLA, 2002). Ele desenvolveu o *Anaconda* utilizando um algoritmo co-evolutivo e a arquitetura de rede neural mostrada na figura 34 (mapeamento espacial), sem uso de um conjunto de características. Uma versão alternativa do *VisionDraughts* foi desenvolvida utilizando o mapeamento espacial de Fogel (seção 5.4.2) e o método

das diferenças temporais (uma abordagem interessante pois elimina o uso de características do jogo de damas para treinar a rede neural). Tal versão do *VisionDraughts* ficou sem viabilidade prática devido ao excessivo tempo necessário para o treinamento da rede, porém, a abordagem continua sendo interessante, principalmente, considerando que o tempo de execução pode ser reduzido com o uso de técnicas paralelas;

7. Co-evolução ou diferenças temporais? Existem estudos interessantes argumentando os benefícios de cada uma das técnicas (FOGEL et al., 2004) (DARWEN, 2001) (POLLOCK; BLAIR, 1998). Para o domínio específico do jogo de damas e para outros domínios ainda não testados experimentalmente, entretanto, é necessário que mais estudos sejam realizados. Assim, uma versão co-evolutiva do *VisionDraughts* se mostra como uma opção interessante de trabalho futuro;
8. Análise em retrocesso tem sido aplicada com sucesso na construção de bases de dados para vários jogos de tabuleiro (LAKE; SCHAEFFER; LU, 1994), (SCHAEFFER et al., 2007), (GASSER, 1990), (GASSER, 1996), (ROMEIN; BAL, 2003), (ROMEIN; BAL, 2002). Com o auxílio da análise em retrocesso, Schaeffer provou o jogo de damas: a prova consiste em uma estratégia explícita com a qual o *Chinook* nunca perde, isto é, o programa pode alcançar o empate contra qualquer oponente jogando tanto com peças pretas quanto brancas (SCHAEFFER et al., 2007). O *VisionDraughts* utiliza as bases de dados disponibilizadas pela equipe do *Chinook* com  $\leq 8$  peças no tabuleiro. Uma parte muito interessante, porém, é construir bases de dados próprias, pois são necessárias várias técnicas de processamento massivo e paralelo, com tratamento de grandes quantidades de memória, tempo de execução, entrada e saída (I/O) e capacidade de armazenamento (SCHAEFFER, 2002) (SCHAEFFER et al., 2003).
9. Outra opção bastante promissora para o tratamento do *problema do loop* baseia-se na utilização das bases de dados MTC do jogador *KingsRow* (seção 3.6.1.2), pois elas possuem informações relativas ao melhor movimento a ser executado a partir de um certo estado do tabuleiro (fazendo com que o jogador automático se dirija para a vitória). Apesar das bases de dados do *KingsRow* não estarem disponíveis para *download*, Ed Gilbert disponibilizou, gentilmente, uma cópia para ser experimentada com o *VisionDraughts* (<http://pages.prodigy.net/eyg/Checkers/KingsRow.htm>). Neste sentido, experimentos interessantes podem ser realizados para verificar se o *problema do loop* pode ser totalmente resolvido.

## *Referências*

- BAL, H.; ALLIS, V. Parallel retrograde analysis on a distributed system. In: *Supercomputing '95: Proceedings of the 1995 ACM/IEEE conference on Supercomputing (CDROM)*. New York, NY, USA: ACM, 1995. p. 73. ISBN 0-89791-816-9.
- BITTENCOURT, G. *Inteligência Artificial Ferramentas e Teorias*. 3ª edição. ed. Universidade Federal de Santa Catarina: Editora da UFSC, 2006. ISBN 9788532801382.
- BREUKER, D.; UITERWIJK, J.; HERIK, H. *Replacement Schemes for Transposition Tables*. 1994. Disponível em: <[citeseer.ist.psu.edu/112066.html](http://citeseer.ist.psu.edu/112066.html)>.
- BREUKER, D.; UITERWIJK, J.; HERIK, H. van den. *Information in Transposition Tables*. 1997. Disponível em: <[citeseer.ist.psu.edu/130130.html](http://citeseer.ist.psu.edu/130130.html)>.
- BROCKINGTON, M. G.; SCHAEFFER, J. APHID: Asynchronous parallel game-tree search. *Journal of Parallel and Distributed Computing*, v. 60, n. 2, p. 247–273, 2000. Disponível em: <[citeseer.ist.psu.edu/brockington99aphid.html](http://citeseer.ist.psu.edu/brockington99aphid.html)>.
- CAEXETA, G. S.; JULIA, R. M. S. A draughts learning system based on neural networks and temporal differences: The impact of an efficient tree-search algorithm. In: *the 19th Brazilian Symposium on Artificial Intelligence*. Bahia, Salvador: LNAI series of Springer-Verlag, 2008. A ser publicado (SBIA2008).
- CHELLAPILLA, K.; FOGEL, D. B. Anaconda defeats hoyle 6-0: A case study competing an evolved checkers program against commercially available software. In: *Proceedings of the 2000 Congress on Evolutionary Computation CEC00*. La Jolla Marriott Hotel La Jolla, California, USA: IEEE Press, 2000. p. 857–863. ISBN 0-7803-6375-2. Disponível em: <[citeseer.ist.psu.edu/chellapilla00anaconda.html](http://citeseer.ist.psu.edu/chellapilla00anaconda.html)>.
- CHELLAPILLA, K.; FOGEL, D. B. Evolving an expert checkers playing program without using human expertise. *IEEE Trans. Evolutionary Computation*, v. 5, n. 4, p. 422–428, 2001.
- DARWEN, P. J. Why co-evolution beats temporal difference learning at backgammon for a linear architecture, but not a non-linear architecture. In: *Proceedings of the 2001 Congress on Evolutionary Computation CEC2001*. COEX, World Trade Center, 159 Samseong-dong, Gangnam-gu, Seoul, Korea: IEEE Press, 2001. p. 1003–010.
- DIESTEL, R. *Graph Theory*. New York: Springer-Verlag, 2000.
- FIERZ, M. *Cake*. 2008. Disponível em: <<http://www.fierz.ch/checkers.htm>>. Acesso em: 08 ago. 2008.



- FOGEL, D. B.; CHELLAPILLA, K. Verifying anaconda's expert rating by competing against chinook: experiments in co-evolving a neural checkers player. *Neurocomputing*, v. 42, n. 1-4, p. 69–86, 2002.
- FOGEL, D. B. et al. A self-learning evolutionary chess program. *Proceedings of the IEEE*, v. 92, n. 12, p. 1947–1954, 2004.
- FREY, P. W. *Chess Skill in Man and Machine*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 1979. ISBN 0387079572.
- GASSER, R. *Applying Retrograde Analysis to Nine Men's Morris*. 1990. Disponível em: <[citeseer.ist.psu.edu/gasser90applying.html](http://citeseer.ist.psu.edu/gasser90applying.html)>.
- GASSER, R. Solving nine men's morris. *Computational Intelligence*, v. 12, p. 24–41, 1996. Disponível em: <[citeseer.ist.psu.edu/gasser96solving.html](http://citeseer.ist.psu.edu/gasser96solving.html)>.
- GILBERT, E. *KingsRow*. 2008. Disponível em: <<http://pages.prodigy.net/eyg/Checkers/KingsRow.htm>>. Acesso em: 08 ago. 2008.
- GILBERT, E. *The moves-to-conversion database*. 2008. Disponível em: <<http://pages.prodigy.net/eyg/Checkers/mtc.htm>>. Acesso em: 08 ago. 2008.
- HAYKIN, S. *Redes Neurais: Princípios e Prática (2ª edição)*. Porto Alegre, RS: Bookman Editora, 2001.
- HLYNKA, M.; SCHAEFFER, J. Pre-searching. v. 27, n. 4, p. 203.
- JUNGHANNS, A.; SCHAEFFER, J. Search versus knowledge in game-playing programs revisited. In: *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence*. unknown: Morgan Kaufmann, ISBN, 1997. p. 692–697.
- KISHIMOTO, A.; SCHAEFFER, J. Distributed game-tree search using transposition table driven work scheduling. In: *ICPP '02: Proceedings of the 2002 International Conference on Parallel Processing (ICPP'02)*. Washington, DC, USA: IEEE Computer Society, 2002. p. 323. ISBN 0-7695-1677-7.
- KORF, R. E. Depth-first iterative-deepening: an optimal admissible tree search. *Artif. Intell.*, Elsevier Science Publishers Ltd., Essex, UK, v. 27, n. 1, p. 97–109, 1985. ISSN 0004-3702.
- KREUZER, J. *GuiCheckers*. 2008. Disponível em: <<http://www.3dkingdoms.com/checkers/bitboards.htm>>. Acesso em: 08 ago. 2008.
- LAKE, R.; SCHAEFFER, J.; LU, P. *Solving Large Retrograde Analysis Problems Using a Network of Workstations*. 1994. Disponível em: <[citeseer.ist.psu.edu/lake94solving.html](http://citeseer.ist.psu.edu/lake94solving.html)>.
- LAKE, R.; SCHAEFFER, J.; TRELOAR, N. *The 3B1b3W Endgame*. 1998.
- LEUSKI, A. *Learning of position evaluation in the game of othello*. Massachusetts, January 1995. Disponível em: <<http://people.ict.usc.edu/leuski/publications/index.html>>.

LYNCH, M. *NeuroDraughts: An Application of Temporal Difference Learning to Draughts*. Ireland, May 1997. Disponível em: <<http://iamlynch.com/nd.html>>.

LYNCH, M.; GRIFFITH, N. Neurodraughts: the role of representation, search, training regime and architecture in a td draughts player. In: *Eighth Ireland Conference on Artificial Intelligence*. Ireland: Unknown, 1997. p. 64–72. Disponível em: <<http://iamlynch.com/nd.html>>.

MARSLAND, T. A. A review of game-tree pruning. *ICCA Journal*, v. 9, n. 1, p. 3–19, 1986.

MCCULLOCH, W.; PITTS, W. A logical calculus of the ideas immanent in nervous activity. *Bulletin of Mathematical Biophysics*, v. 5, p. 115–133, 1943.

MILLINGTON, I. *Artificial Intelligence for Game*. San Francisco: Morgan Kaufmann, 2006.

NETO, H. C. *LS-Draughts – Um Sistema de Aprendizagem de Jogos de Damas Baseado em Algoritmos Genéticos, Redes Neurais e Diferenças Temporais*. Dissertação (Mestrado) — Universidade Federal de Uberlândia, Minas Gerais, Uberlândia, 2007.

NETO, H. C.; JULIA, R. M. S. Ls-draughts – a draughts learning system based on genetic algorithms, neural network and temporal differences. In: 2007 IEEE CONGRESS ON EVOLUTIONARY COMPUTATION (CEC 2007). *Proceedings of 2007 IEEE Congress on Evolutionary Computation*. Cingapura, 2007. p. 2523–2529.

NETO, H. C.; JULIA, R. M. S. Um sistema de aprendizagem para damas com geração automática de características. In: VIII CONGRESSO BRASILEIRO DE REDES NEURAI (VIII CBRN). *Anais do VIII Congresso Brasileiro de Redes Neurais*. Santa Catarina, Florianópolis, 2007. v. 1, p. 1–6.

NETO, H. C.; JULIA, R. M. S.; CAEXETA, G. S. *LS-Draughts: Using Databases to Treat Endgame Loops in a Hybrid Evolutionary Learning System*. Austria, Vienna, Kirchengasse: I-Tech Education and Publishing KG, 2008. A ser publicado (I-Tech).

PATIST, J.-P.; WIERING, M. Learning to play draughts using temporal difference learning with neural networks and databases. 2004. Disponível em: <<http://igitur-archive.library.uu.nl/math/2007-0330-200701/UUindex.html>>.

PENTON, R. *Data Structures for Game Programmers*. [S.l.]: Muska & Lipman/Premier-Trade, 2002.

PLAAT, A. *Research Re: search & Re-search*. Tese (Doutorado), Rotterdam, Netherlands, 1996. Disponível em: <[citeseer.ist.psu.edu/plaat96research.html](http://citeseer.ist.psu.edu/plaat96research.html)>.

PLAAT, A. et al. *A New Paradigm for Minimax Search*. Rotterdam, Netherlands, 1995. Disponível em: <[citeseer.ist.psu.edu/plaat94new.html](http://citeseer.ist.psu.edu/plaat94new.html)>.

PLAAT, A. et al. Exploiting graph properties of game trees. In: *Proceedings of the Thirteenth National Conference on Artificial Intelligence (AAAI-96)*. Portland, OR: unknown, 1996. p. 234–239. Disponível em: <[citeseer.ist.psu.edu/plaat96exploiting.html](http://citeseer.ist.psu.edu/plaat96exploiting.html)>.

- PLAAT, A. et al. Best-first fixed-depth minimax algorithms. *Artif. Intell.*, Elsevier Science Publishers Ltd., Essex, UK, v. 87, n. 1-2, p. 255–293, 1996. ISSN 0004-3702.
- POLLACK, J. B.; BLAIR, A. D. Co-evolution in the successful learning of backgammon strategy. *Machine Learning*, v. 32, n. 1, p. 225–240, 1998.
- PTKFGS. *Diagram of backgammon checker movement*. 2007. Disponível em: <<http://pt.wikipedia.org/wiki/Imagem:Bg-movement.svg>>. Acesso em: 08 ago. 2008.
- RASMUSSEN, D. *Parallel chess searching and bitboards*. Dissertação (Mestrado) — Informatics and Mathematical Modelling, Technical University of Denmark, DTU, Richard Petersens Plads, Building 321, DK-2800 Kgs. Lyngby, 2004. Supervised by Prof. Jens Clausen. Disponível em: <<http://www2.imm.dtu.dk/pubdb/p.php?3267>>.
- REINEFELD, A.; MARSLAND, T. A. Enhanced iterative-deepening search. *IEEE Trans. Pattern Anal. Mach. Intell.*, IEEE Computer Society, Washington, DC, USA, v. 16, n. 7, p. 701–710, 1994. ISSN 0162-8828.
- RIBEIRO, C. H. C.; MONTEIRO, S. T. Aprendizagem da navegação em robôs móveis a partir de mapas obtidos autonomamente. In: ANAIS DO XXIII CONGRESSO DA SOCIEDADE BRASILEIRA DE COMPUTAÇÃO. *Encontro Nacional de Inteligencia Artificial - XXIII Congresso da Sociedade Brasileira de Computação*. Campinas, 2003. p. 152–152.
- ROMEIN, J.; BAL, H. *Awari is Solved*. 2002. Disponível em: <[citeseer.ist.psu.edu/romein02awari.html](http://citeseer.ist.psu.edu/romein02awari.html)>.
- ROMEIN, J.; BAL, H. *Solving the Game of Awari using Parallel Retrograde Analysis*. 2003. Disponível em: <[citeseer.ist.psu.edu/romein03solving.html](http://citeseer.ist.psu.edu/romein03solving.html)>.
- RUSSELL, S.; NORVIG, P. *Inteligência Artificial - Uma Abordagem Moderna (2a edição)*. [S.l.]: Editora Campus, 2004.
- SAMUEL, A. L. Some studies in machine learning using the game of checkers. *IBM Journal of Research and Development*, v. 3, n. 3, p. 211–229, 1959.
- SAMUEL, A. L. Some studies in machine learning using the game of checkers ii - recent progress. *IBM Journal of Research and Development*, v. 11, n. 6, p. 601–617, 1967.
- SCHAEFFER, J. The history heuristic and alpha-beta search enhancements in practice. *IEEE Trans. Pattern Anal. Mach. Intell.*, IEEE Computer Society, Washington, DC, USA, v. 11, n. 11, p. 1203–1212, 1989. ISSN 0162-8828.
- SCHAEFFER, J. Checkers: A preview of what will happen in chess? *ICCA Journal*, v. 14, n. 2, p. 71–78, 1991.
- SCHAEFFER, J. *Man Versus Machine: The Silicon Graphics World Checkers Championship*. 1992.
- SCHAEFFER, J. The games computers (and people) play. In: *Proceedings of the Seventeenth National Conference on Artificial Intelligence and Twelfth Conference on Innovative Applications of Artificial Intelligence*. [S.l.]: AAAI Press / The MIT Press, 2000. p. 1179. ISBN 0-262-51112-6.

SCHAEFFER, J. *Applying the Experience of Building a High Performance Search Engine for One Domain to Another*. 2002.

SCHAEFFER, J. et al. *Chinook: World Man-Machine Checkers Champion*. 2008. Perfect Play: Draw! Disponível em: <<http://www.cs.ualberta.ca/~chinook/index.php>>. Acesso em: 08 ago. 2008.

SCHAEFFER, J. et al. Building the checkers 10-piece endgame databases. In: HERIK, H. J. van den; IIDA, H.; HEINZ, E. A. (Ed.). [S.l.]: Kluwer, 2003. (IFIP, v. 263), p. 193–210.

SCHAEFFER, J. et al. Solving checkers. In: *IJCAI*. [S.l.]: Professional Book Center, 2005. p. 292–297.

SCHAEFFER, J. et al. Checkers is solved. *Science*, p. 1144079+, 2007.

SCHAEFFER, J. et al. *Reviving the Game of Checkers*. 1991. Disponível em: <[citeseer.ist.psu.edu/schaeffer91reviving.html](http://citeseer.ist.psu.edu/schaeffer91reviving.html)>.

SCHAEFFER, J. et al. A world championship caliber checkers program. *Artificial Intelligence*, v. 53, n. 2-3, p. 273–289, 1992. Disponível em: <[citeseer.ist.psu.edu/schaeffer92world.html](http://citeseer.ist.psu.edu/schaeffer92world.html)>.

SCHAEFFER, J. et al. Temporal difference learning applied to a high performance game-playing program. In: INTERNATIONAL JOINT CONFERENCES ON ARTIFICIAL INTELLIGENCE. *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*. Canada, 2001. p. 529–534.

SCHAEFFER, J.; LAKE, R. *Solving the Game of Checkers*. 1996. Disponível em: <[citeseer.ist.psu.edu/6903.html](http://citeseer.ist.psu.edu/6903.html)>.

SCHAEFFER, J. et al. Chinook: The world man-machine checkers champion. *AI Magazine*, v. 17, n. 1, p. 21–29, 1996.

SCHAEFFER, J. et al. A re-examination of brute-force search. In: *In Games: Planning and Learning*. [S.l.]: AAAI Press, ISBN, 1993. p. 51–58.

SCHAEFFER, J.; PLAAT, A. New advances in alpha-beta searching. In: *ACM Conference on Computer Science*. [s.n.], 1996. p. 124–130. Disponível em: <[citeseer.ist.psu.edu/10091.html](http://citeseer.ist.psu.edu/10091.html)>.

SCHAEFFER, J.; STURTEVANT, N. R. Partial information endgame databases. In: *ACG*. [S.l.]: Springer, 2006. (Lecture Notes in Computer Science, v. 4250), p. 11–22.

SCHAEFFER, J. et al. Man versus machine for the world checkers championship. *AI Magazine*, v. 14, n. 2, p. 28–35, 1993.

SCHRAUDOLPH, N. N.; DAYAN, P.; SEJNOWSKI, T. J. Learning to evaluate go positions via temporal difference methods. In: BABA, I.; JAIN (Ed.). *Computational Intelligence in Games Studies in Fuzziness and Soft Computing*. Springer Verlag, 2001. v. 62. Disponível em: <<http://users.rsise.anu.edu.au/nici/bib2html/index.html>>.

SHAMS, R.; KAINDL, H.; HORACEK, H. Using aspiration windows for minimax algorithms. In: *IJCAI*. [S.l.: s.n.], 1991. p. 192–197.

STEVANOVIC, R. *Quantum Random Bit Generator Service*. 2008. Disponível em: <<http://random.irb.hr/>>. Acesso em: 08 ago. 2008.

STIPCEVIC, M.; ROGINA, B. M. Quantum random number generator based on photonic emission in semiconductors. *Review of Scientific Instruments*, AIP, v. 78, n. 4, p. 045104, 2007. Disponível em: <<http://link.aip.org/link/?RSI/78/045104/1>>.

SUTTON, R. S. Learning to predict by the methods of temporal differences. *Machine Learning*, v. 3, n. 1, p. 9–44, 1988.

SUTTON, R. S.; BARTO, A. G. *Reinforcement Learning: An Introduction*. Cambridge: MIT Press, 1998.

TESAURO, G. Practical issues in temporal difference learning. In: MOODY, J. E.; HANSON, S. J.; LIPPMANN, R. P. (Ed.). *Advances in Neural Information Processing Systems*. [S.l.]: Morgan Kaufmann Publishers, Inc., 1992. v. 4, p. 259–266.

TESAURO, G. Td-gammon, a self-teaching backgammon program, achieves master-level play. *Neural Computation*, v. 6, n. 2, p. 215–219, 1994.

TESAURO, G. Temporal difference learning and td-gammon. *Communications of the ACM*, v. 38, n. 3, p. 19–23, 1995.

THRUN, S. Learning to play the game of chess. In: *Advances in Neural Information Processing Systems 7*. [S.l.]: The MIT Press, 1995. p. 1069–1076.

WALKER, M. A. An application of reinforcement learning to dialogue strategy selection in a spoken dialogue system for email. In: *Journal of Artificial Intelligence Research 12*. [S.l.: s.n.], 2000. p. 387–416.

WIERING, M. Multi-agent reinforcement learning for traffic light control. In: *Proc. 17th International Conf. on Machine Learning*. [S.l.]: Morgan Kaufmann, San Francisco, CA, 2000. p. 1151–1158.

XING, L.; PHAM, D. T. *Neural Networks for Identification, Prediction, and Control*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 1995.

ZOBRIST, A. L. *A Hashing Method with Applications for Game Playing*. [S.l.], 1969.