

UNIVERSIDADE FEDERAL DE UBERLÂNDIA
FACULDADE DE CIÊNCIA DA COMPUTAÇÃO
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO



**D-VISIONDRAUGHTS: UMA REDE NEURAL JOGADORA
DE DAMAS QUE APRENDE POR REFORÇO EM UM
AMBIENTE DE COMPUTAÇÃO DISTRIBUÍDA**

AYRES ROBERTO ARAÚJO BARCELOS

Uberlândia - Minas Gerais

2011

UNIVERSIDADE FEDERAL DE UBERLÂNDIA
FACULDADE DE CIÊNCIA DA COMPUTAÇÃO
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO



AYRES ROBERTO ARAÚJO BARCELOS

**D-VISIONDRAUGHTS: UMA REDE NEURAL JOGADORA
DE DAMAS QUE APRENDE POR REFORÇO EM UM
AMBIENTE DE COMPUTAÇÃO DISTRIBUÍDA**

Dissertação de Mestrado apresentada à Faculdade de Ciência da Computação da Universidade Federal de Uberlândia, Minas Gerais, como parte dos requisitos exigidos para obtenção do título de Mestre em Ciência da Computação.

Área de concentração: Inteligência Artificial.

Orientadora:

Prof^a. Dr^a. Rita Maria da Silva Júlia

Uberlândia, Minas Gerais

2011

UNIVERSIDADE FEDERAL DE UBERLÂNDIA
FACULDADE DE CIÊNCIA DA COMPUTAÇÃO
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

Os abaixo assinados, por meio deste, certificam que leram e recomendam para a Faculdade de Ciência da Computação a aceitação da dissertação intitulada “**D-VisionDraughts: uma rede neural jogadora de damas que aprende por reforço em um ambiente de computação distribuída**” por **Ayres Roberto Araújo Barcelos** como parte dos requisitos exigidos para a obtenção do título de **Mestre em Ciência da Computação**.

Uberlândia, 25 de Janeiro de 2011

Orientadora:

Prof^a. Dr^a. Rita Maria da Silva Júlia
Universidade Federal de Uberlândia

Banca Examinadora:

Prof. Dr. Rivalino Matias Júnior
Universidade Federal de Uberlândia

Prof^a. Dr^a. Ana Lucia Cetertich Bazzan
Universidade Federal do Rio Grande do Sul

UNIVERSIDADE FEDERAL DE UBERLÂNDIA
FACULDADE DE CIÊNCIA DA COMPUTAÇÃO
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

Data: Janeiro de 2011

Autor: **Ayres Roberto Araújo Barcelos**
Título: **D-VisionDraughts: uma rede neural jogadora de damas que aprende por reforço em um ambiente de computação distribuída**
Faculdade: **Faculdade de Ciência da Computação**
Grau: **Mestrado**

Fica garantido à Universidade Federal de Uberlândia o direito de circulação e impressão de cópias deste documento para propósitos exclusivamente acadêmicos, desde que o autor seja devidamente informado.

Autor

O AUTOR RESERVA PARA SI QUALQUER OUTRO DIREITO DE PUBLICAÇÃO DESTE DOCUMENTO, NÃO PODENDO O MESMO SER IMPRESSO OU REPRODUZIDO, SEJA NA TOTALIDADE OU EM PARTES, SEM A PERMISSÃO ESCRITA DO AUTOR.

Dedicatória

Dedico esta dissertação à minha família e em especial à minha mãe Cibele

Agradecimentos

Gostaria de agradecer à caríssima professora Dr^a Rita Maria da Silva Julia pela orientação, inspiração, amizade e encorajamento em todos os momentos da realização deste trabalho.

Aos professores da FACOM/UFU, que de alguma forma contribuíram para a realização deste trabalho, em especial ao Marcelo Rodrigues, Anilton Joaquim e Rivalino Matias.

A todos os amigos da turma de mestrado e laboratório, principalmente ao Marcos Luiz e Murilo Carneiro.

Meu agradecimento final vai para minha família, que me proveu de um infindável apoio e sempre foi minha fonte de motivação.

Resumo

O objetivo deste trabalho é propor um sistema de aprendizagem de damas, o D-VisionDraughts (Distributed VisionDraughts): um agente distribuído jogador de damas baseado em redes neurais que aprende por reforço. O D-VisionDraughts é treinado em um ambiente de processamento distribuído de modo a alcançar um alto nível de jogo sem a análise de especialistas e com o mínimo de intervenção humana possível (diferentemente do agente campeão do mundo de damas Chinook). O D-VisionDraughts corresponde a uma versão distribuída do eficiente jogador VisionDraughts, onde este último corresponde à uma rede neural MLP (multilayer perceptron) que aprende pelo método das diferenças temporais. O papel da rede neural é avaliar o quanto um estado de tabuleiro é favorável ao agente (valor de predição). Este valor irá guiar o módulo de busca na procura pela melhor ação (neste caso, o melhor movimento) correspondente ao estado de tabuleiro corrente do jogo. Outro fator que é importante na eficiência da busca, e que foi analisado neste trabalho, é o grau de ordenação da árvore de jogo. Desta forma, as principais contribuições deste trabalho consistem em: substituir o algoritmo serial utilizado para a busca em árvore de jogos do VisionDraughts, o minimax com poda alpha-beta, pelo algoritmo distribuído Young Brothers Wait Concept (YBWC); o uso de heurísticas para ordenação da árvore de jogos, que é essencial para o bom desempenho do YBWC e da poda alpha-beta em geral; a análise do impacto do ambiente de processamento distribuído nas habilidades de aprendizado não supervisionadas do jogador. Este trabalho mostra que, com a aplicação das técnicas no D-VisionDraughts, reduzimos expressivamente o tempo necessário para a etapa de busca e, através de torneios realizados com o VisionDraughts, o desempenho geral do agente distribuído foi nitidamente melhor.

Palavras chave: damas, aprendizagem de máquina, aprendizagem por reforço, aprendizagem por diferenças temporais, redes neurais artificiais, poda alpha-beta, tabelas de transposição, aprofundamento iterativo, busca paralela.

Abstract

The objective of this work is to propose a draughts learning system, the D-VisionDraughts (Distributed VisionDraughts): a distributed draughts player agent based on neural networks that learns by reinforcement. The D-VisionDraughts is trained in a distributed processing environment in order to achieve a high level of play without expert game analysis and with minimal human intervention as possible (distinctly from the world draughts champion Chinook). The D-VisionDraughts corresponds to a distributed version of the efficient VisionDraughts player, where the latter corresponds to a MLP (multilayer perceptron) neural network that learns by means of temporal differences. The role of the neural network is to evaluate how much a board state is favorable to the agent (prediction). This value will lead the search module to determine the best action (in this case, the best move) of the current board state of the game. Another factor that has an important impact on the search efficiency, which is analyzed in this work, is the degree of ordering of the game tree. Thus, the main contributions of this work are: the replacement of the serial algorithm used in VisionDraughts, the minimax with alpha-beta pruning, by the distributed algorithm Young Brothers Wait Concept (YBWC); the use of heuristics for game tree ordering, that is essential for the proper performance of YBWC and alpha-beta pruning in general; the impact analysis of the high-performance processing environment on the unsupervised learning skills of the player. This work shows that with the techniques used, the time required to perform a game tree search was significantly reduced and through tournaments played with VisionDraughts the overall performance of the distributed agent is improved.

Keywords: draughts, machine learning, reinforcement learning, temporal differences learning, artificial neural network, alpha-beta pruning, transposition table, iterative deepening, parallel search.

Sumário

Lista de Figuras	xix
Lista de Tabelas	xxi
Lista de Abreviaturas e Siglas	xxiii
1 Introdução	25
1.1 Motivação e Objetivos	26
1.2 Resultados Bibliográficos Obtidos	29
1.3 Organização do Trabalho	29
1.4 Definições	29
2 Referencial Teórico	33
2.1 Rede Neural Artificial - RNA	33
2.1.1 A modelagem Matemática do Neurônio	33
2.1.2 Os tipos de Redes Neurais	35
2.1.3 O <i>Perceptron</i> simples e o <i>Perceptron</i> multi-camada - MLP	37
2.2 Aprendizagem por Reforço	39
2.2.1 Fundamentos da Modelagem Markoviana	40
2.2.2 O Problema da Aprendizagem por Reforço	41
2.2.3 Método de solução por Diferenças Temporais	45
3 Estado da Arte	49
3.1 Redes Neurais e Diferenças Temporais aplicadas aos jogos de tabuleiro	49
3.2 Programas de Computador para Damas	51
3.2.1 Chinook	51
3.2.2 Anaconda	52
3.2.3 Cake	52
3.2.4 NeuroDraughts	52
3.3 Versões anteriores do D-VisionDraughts	53
3.3.1 LS-Draughts	54
3.3.2 VisionDraughts	54

3.3.3	MP-Draughts	54
4	VisionDraughts - Um Sistema de Aprendizado de Damas com Eficiente Método de Busca	57
4.1	Representação do Tabuleiro nos Jogadores de Damas	58
4.1.1	Representação Vetorial	58
4.1.2	Representação NET-FEATUREMAP	59
4.2	Arquitetura Geral do <i>VisionDraughts</i>	62
4.3	Cálculo da Predição e Escolha da Melhor Ação	65
4.4	Reajuste de Pesos da Rede Neural MLP	67
4.5	Estratégia de Treino por <i>Self-Play</i> com Clonagem	69
4.6	Eficiente Método de Busca do <i>VisionDraughts</i>	70
4.6.1	O algoritmo Alfa-Beta	71
4.6.2	A Tabela de Transposição	77
4.6.3	Integração entre o Algoritmo Alfa-Beta e a Tabela de Transposição	87
4.6.4	O Aprofundamento Iterativo	96
4.6.5	O Algoritmo Alfa-Beta com a Tabela de Transposição e o Aprofun- damento Iterativo	97
5	Busca Paralela	101
5.1	Terminologia	101
5.1.1	Medidas de Desempenho	101
5.1.2	Sobrecargas (<i>overheads</i>) de um algoritmo de busca em árvore de jogo paralelo	102
5.2	Algoritmos de busca paralelos baseados no Alpha-Beta	103
6	D-VisionDraughts: um sistema de aprendizado de damas para ambi- entes de processamento de alto desempenho	109
6.1	A distribuição da busca no D-VisionDraughts	109
6.2	Integração D-VisionDraughts com MPI	119
6.3	Ordenação da árvore de busca	124
6.3.1	Tabela de transposição	124
6.3.2	Killer Moves	124
6.3.3	Ordenação da árvore de busca no D-VisionDraughts	124
6.4	Resultados	125
6.4.1	Ambiente de execução dos testes	125
6.4.2	Impacto das heurísticas de ordenação na busca	125
6.4.3	Impacto da distribuição da busca	126
6.4.4	Torneio entre D-VisionDraughts e VisionDraughts	127

7 Conclusão e Trabalhos Futuros	131
7.1 Trabalhos Futuros	132
Referências Bibliográficas	133

Lista de Figuras

1.1	Relação entre os jogadores automáticos de damas.	28
2.1	Modelo de um neurônio artificial [32]	34
2.2	Modelo de uma função de ativação baseada na tangente hiperbólica	35
2.3	Arquitetura de um perceptron simples.	37
2.4	Arquitetura de um <i>Perceptron</i> Multicamadas - MLP	38
2.5	A interação agente-ambiente em Aprendizagem por Reforço.	42
3.1	Arquitetura do <i>NeuroDraughts</i>	53
4.1	Exemplo de tabuleiro representado por NET-FEATUREMAP	62
4.2	Fluxo de aprendizado do <i>VisionDraughts</i> : um sistema de aprendizagem de jogos de damas	63
4.3	Rede Neural utilizada pelo <i>VisionDraughts</i>	65
4.4	A esquerda: árvore de busca expandida pelo minimax. A direita: árvore expandida pelo Alfa-Beta.	76
4.5	Exemplo de transposição em c e f : o mesmo estado do tabuleiro é alcançado por combinações diferentes de jogadas com peças simples.	78
4.6	Exemplo de transposição em a e c : o mesmo estado do tabuleiro é alcançado por combinações diferentes de jogadas com reis.	78
4.7	Vetor de 128 elementos inteiros aleatórios utilizados pelo <i>VisionDraughts</i> .	81
4.8	Exemplo de movimento simples.	81
4.9	Árvore de busca onde o pai de n é maximizador	92
4.10	Árvore de busca onde o pai de n é minimizador	93
5.1	Tipos de nós em uma árvore de jogo.	104
5.2	Exploração paralela de uma árvore com 2 processadores pelo PVS	105
5.3	Exploração paralela de uma árvore pelo APHID	107
6.1	(a) Exemplo de uma Variação Corrente (b) Exemplo de uma pilha local de estados	111

6.2	Exemplo de expansão de uma árvore de jogo pelo D-VisionDraughts (caminho da variação corrente em negrito)	116
6.3	Diagrama de estados principal - início e fim dos processos	121
6.4	Diagrama de estados YBWC	122
6.5	Diagrama de estados ProcessaMensagensBusca	123
6.6	Diagrama de estados TrataResultado	123
6.7	Impacto das heurísticas de ordenação no Alpha-Beta serial	126
6.8	Impacto da distribuição da busca sem ordenação	127
6.9	Impacto da distribuição da busca com ordenação	127
6.10	Speedup do YBWC	128
6.11	Eficiência do YBWC	128
6.12	Torneio Avaliativo 1: D-VisionDraughts x VisionDraughts	128
6.13	Torneio Avaliativo 2: D-VisionDraughts x VisionDraughts	129

Lista de Tabelas

4.1	Conjunto de 28 Características implementadas por Samuel	61
4.2	Conjunto de Características implementadas no jogador <i>VisionDraughts</i> . .	64
5.1	Comparação algoritmos paralelos	107
6.1	Mensagens do protocolo de comunicação do D-VisionDraughts	119

Lista de Abreviaturas e Siglas

APHID	Asynchronous Parallel Hierarchical Iterative Deepening
AR	Aprendizagem por Reforço
DTS	Dynamic Tree Splitting
MLP	Multilayer Perceptron
MPI	Message Passing Interface
PDM	Processo de Decisão de Markov
PVS	Principal Variation Search
RNA	Rede Neural Artificial
TD	Temporal Differences
YBWC	Young Brothers Wait Concept

Capítulo 1

Introdução

Este trabalho apresenta o D-VisionDraughts: uma versão distribuída e aperfeiçoada do bem-sucedido agente jogador de damas - baseado em uma rede neural MLP (Multi-layer Perceptron) que aprende por reforço - o VisionDraughts [10]. O trabalho contínuo dos autores para melhorarem seu jogador é motivado pelas suas intenções de estudar, propor e testar ferramentas apropriadas para tratar problemas relativos ao aprendizado de máquina onde interferência humana não é desejada ou disponível, onde o espaço de estados é muito grande (o que demanda uma estratégia de busca muito eficiente) e onde a escolha da melhor ação não depende apenas do estado corrente, mas de fatores adversos. O aprendizado por reforço tem sido tema de grande interesse na área de aprendizado de máquina, uma vez que não requer um instrutor inteligente para prover exemplos de treinamento. Por este motivo é uma ferramenta adequada para domínios complexos onde é difícil ou mesmo impossível obter tais exemplos [44], [37].

Dentre os métodos de aprendizado por reforço, um que se destaca é o método das diferenças temporais (Temporal Differences - TD). Ele é amplamente utilizado com resultados altamente eficientes, inclusive na construção de agentes capazes de aprender a jogar damas, xadrez, gamão, othello, go entre outros jogos [45], [28], [29], [36], [49], [58], [56], [24], [52] e [13]. Tais agentes têm demonstrado que jogos são um domínio adequado para estudar e verificar a eficiência das principais técnicas de aprendizado. Particularmente, os autores escolheram damas como um laboratório para testar os métodos de aprendizado apropriados devido ao fato desse jogo apresentar muitas similaridades com outros jogos, oferecendo muitos dos desafios de pesquisa de outros jogos de tabuleiro, como o xadrez, mas sem a complexidade desnecessária [50], além de similaridades significantes com diversos problemas práticos, como:

1. *Problema de navegação em que os mapas são obtidos autonomamente 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 [42];

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 [59], 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 sob esta rede [61] [2].

Note que todos estes problemas apresentam dificuldades similares ao problema encontrado no domínio dos jogos, a saber:

- Aprender a se comportar em um ambiente onde o conhecimento adquirido é armazenado em uma função de avaliação;
- Escolha de um conjunto mínimo de atributos possíveis que melhor caracterize o domínio e que sirva como um meio pelo qual a função de avaliação adquirirá novos conhecimentos (esta questão é de fundamental importância para a obtenção de agentes com alto nível de desempenho);
- Seleção da melhor ação para um determinado estado ou configuração do ambiente onde o agente está interagindo (problema de otimização);
- Estratégias de aprendizagem que facilitem a geração de um agente com ótimo nível de desempenho.

O jogador de damas desenvolvido pela equipe liderada por Schaeffer na Universidade de Alberta, o Chinook, é o atual campeão homem-máquina de damas. Sua função de avaliação é manualmente ajustada e o sistema conta com bases de dados para otimizar a escolha da melhor ação. Isto significa que o processo de construção do Chinook, diferentemente da proposta que motiva a construção do D-VisionDraughts, é fortemente supervisionado.

1.1 Motivação e Objetivos

No jogador automático de damas NeuroDraughts [28], seu autor Mark Lynch implementou uma rede neural jogadora cujos pesos eram atualizados por meio do algoritmo Minimax e o método das diferenças temporais, usando como estratégia de treinamento a técnica self-play com clonagem. NeuroDraughts adota a técnica de mapeamento NET-FEATUREMAP para representar o estado de tabuleiro, isto é, ele as representa por um conjunto de funções - denominadas de características - que capturam conhecimentos relevantes sobre o domínio de damas e usa este conhecimento para mapear um estado de

tabuleiro para ser avaliado na entrada da Rede Neural. No NeuroDraughts, as características são selecionadas manualmente e não variam (o jogador é treinado apenas com um conjunto de características). A boa capacidade de aprendizado do NeuroDraughts mostra a eficiência do NET-FEATUREMAP para representar os estados de tabuleiro. Apesar de ser um bom jogador com uma seleção manual e arbitrária de características, o NeuroDraughts pode ser certamente otimizado com a inserção de um módulo automático para selecioná-las. Da mesma maneira, o processo de busca do NeuroDraughts pode ser otimizado, pois o algoritmo minimax desperdiça muito tempo de execução explorando ramos desnecessários [44]. É por este motivo que no NeuroDraughts a profundidade da busca não deve ser maior do que 4, caso contrário o tempo de execução requerido para a escolha do melhor movimento e para o treinamento seria impraticável. Esta restrição compromete o possível *look-ahead* disponível cada vez que o agente deve escolher um movimento, o que também compromete o desempenho do agente. Considerando os fatos mencionados acima, os autores do sistema D-VisionDraughts apresentado neste trabalho usaram o bem sucedido agente NeuroDraughts como um laboratório para efetuar suas investigações. Neste meio tempo, suas motivações tem sido analisar, separadamente, os ganhos obtidos por sucessivas substituições de ferramentas ou técnicas ineficientes usadas no NeuroDraughts. A fim de lidar com tal objetivo, três versões preliminares do D-VisionDraughts foram desenvolvidas: LS-Draughts [35], VisionDraughts [9] e MP-Draughts [12]. O LS-Draughts aperfeiçoa o NeuroDraughts com a inserção de algoritmos genéticos para selecionar, automaticamente, o conjunto de características mais apropriado para representar os estados de tabuleiro. Vários torneios ganhos pelo LS-Draughts, jogando contra o NeuroDraughts, confirmaram o melhor desempenho do último, como mostrado na seção 3.3.1. No VisionDraughts, para melhorar a escolha do melhor movimento, os autores substituíram o algoritmo minimax do NeuroDraughts por uma estratégia de busca eficiente baseada na poda alpha-beta, combinada com tabela de transposição e aprofundamento iterativo. Vários torneios ganhos pelo VisionDraughts jogando contra o NeuroDraughts e LS-Draughts confirmaram que o módulo aperfeiçoado do último contribuiu significamente para o bom desempenho do agente .

Ainda com o objetivo de otimizar as técnicas de aprendizado, os autores do presente trabalho participaram da implementação de uma versão multi-agentes do VisionDraughts: o MP-Draughts. O MP-Draughts foi construído segundo uma estratégia na qual os agentes são especializados para fases de início ou final de jogo. Neste contexto, há um agente treinado para operar nas fases iniciais e intermediárias de jogo e 25 agentes treinados para lidar com um determinado tipo de cluster de tabuleiro de final de jogo, todos eles apresentando arquitetura análoga a do VisionDraughts. Esses 25 clusters são minerados por redes de Kohonen-SOM a partir de uma base de dados que contém milhares de estado de tabuleiro de final de jogo.

Apesar de o MP-Draughts ter melhorado significativamente o desempenho do Vision-

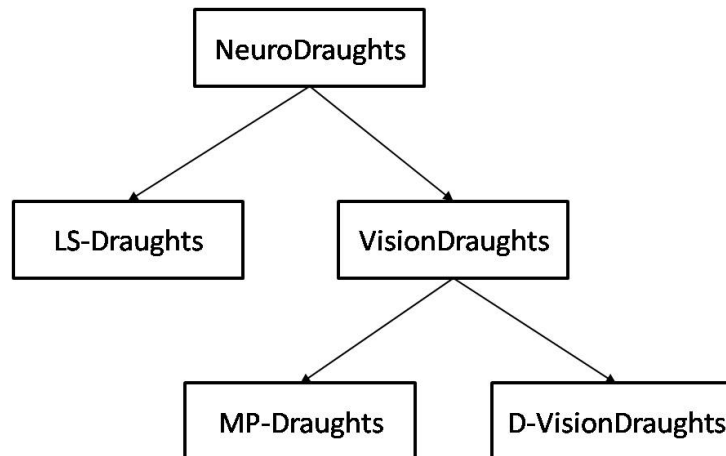


Figura 1.1: Relação entre os jogadores automáticos de damas.

Draughts [12], todos os 25 agentes responsáveis pelas fases de fim de jogo compartilham um único conjunto de características, o mesmo utilizado pelo agente das fases inicial e intermediária. Este é um grande inconveniente, pois cada agente deveria variar sua visão do tabuleiro em função do cluster para o qual se especializou, o que não é possível com um conjunto único de características. A solução seria integrar o módulo de algoritmos genéricos do LS-Draughts a cada agente do MP-Draughts, de modo a detectar o melhor conjunto de características em função dos clusters e das fases do jogo. Contudo, a enorme quantidade de indivíduos (redes neurais) para serem treinados, somada ao grande número de agentes que compõem o MP-Draughts, onde cada agente teria seu conjunto próprio de características, tornaria inviável o tempo de treinamento da arquitetura multi-agente. Assim sendo, o presente trabalho, o D-VisionDraughts, tem como uma de suas motivações distribuir o módulo de busca do VisionDraughts, de modo a melhorar seu desempenho (a figura 1.1 esquematiza a relação entre os jogadores automáticos de damas NeuroDraughts, LS-Draughts, VisionDraughts, MP-Draughts e o D-VisionDraughts).

Para atingir tal objetivo, o D-VisionDraughts substitui o algoritmo de busca serial poda alpha-beta utilizado no VisionDraughts pelo algoritmo distribuído de busca Young Brothers Wait Concept (YBWC), além de introduzir heurísticas para ordenação da árvore de jogo, que se mostraram essenciais para um bom desempenho dos algoritmos de busca serial e distribuído. A combinação de busca distribuída com ordenação da árvore de jogo diminuiu expressivamente o tempo gasto pela etapa de busca do agente, permitindo, desta forma, buscas em profundidades maiores. Os resultados de torneios realizados entre o D-VisionDraughts e o VisionDraughts, todos favoráveis ao D-VisionDraughts, indicam que, com sua busca mais poderosa, seu nível de jogo superou o nível de seu predecessor, o VisionDraughts.

Como trabalho futuro, os autores pretendem propagar tal estratégia de busca distribuída a todos os agentes do MP-Draughts estendido com o módulo de algoritmos genéticos gerador de características.

1.2 Resultados Bibliográficos Obtidos

A proposta desta dissertação se insere no contexto de um trabalho extenso e continuado. A participação das atividades de pesquisas ligadas ao desenvolvimento do sistema anterior ao sistema apresentado neste trabalho, o MP-Draughts, resultou na publicação do artigo *MP-Draughts: a Multiagent Reinforcement Learning System based on MLP and Kohonen-SOM Neural Networks*, como co-autor, publicado na conferência *IEEE SMC 2009 (Conference on Systems, Man and Cybernetics)* [12].

Com os resultados deste trabalho, o artigo intitulado *D-VisionDraughts: a Draughts Player Neural Network That Learns by Reinforcement in a High Performance Environment* foi submetido e aceito para publicação na conferência *ESANN 2011 (European Symposium on Artificial Neural Networks, Computational Intelligence and Machine Learning)* [1]

1.3 Organização do Trabalho

Os próximos capítulos estão organizados conforme disposto a seguir:

Capítulo 2. Apresentação dos conceitos de redes neurais e aprendizagem por reforço, que são fundamentais para a compreensão do funcionamento do sistema proposto.

Capítulo 3. Um levantamento do estado da arte em programas que aprendem a jogar.

Capítulo 4. Apresentação do agente jogador VisionDraught, jogador no qual o D-VisionDraught foi inspirado.

Capítulo 5. É explorado as abordagens para busca paralela em árvore de jogos.

Capítulo 6. Apresenta o D-VisionDraughts, um agente jogador de damas baseado em algoritmos de busca distribuídos e heurísticas para ordenação da árvore de jogos. Neste capítulo é apresentada uma análise do seu desempenho.

Capítulo 7. Conclusões e perspectivas de trabalhos futuros

1.4 Definições

Nesta seção, serão definidos os principais conceitos utilizados neste trabalho, conforme apresentado em [15]. Por $\mathbb{N}$ ($\mathbb{N}_0$) denotamos o conjunto dos números naturais (incluindo 0), por $\mathbb{Z}$ denotamos o conjunto dos números inteiros.

Definição 1.1. (Grafo)

Um grafo direcionado $G = (V, E)$ é uma tupla composta por um conjunto de nós (ou vértices), onde V é um conjunto de arestas $E \subset V \times V$. Se $E \subset \mathcal{P}^2(V)$, então G é

chamado de grafo não direcionado.

Para um grafo direcionado $G = (V, E)$ o grau de entrada (grau de saída) de um nó $v \in V$ é definido como

$$\deg_i(v) = |\{v' \in V | (v', v) \in E\}| \quad (\deg_o(v) = |\{v' \in V | (v, v') \in E\}|)$$

Para um grafo não direcionado $G = (V, E)$, $\deg(v) = |\{v' \in V | \{v, v'\} \in E\}|$ para qualquer $v \in V$ é chamado de *degree* de v .

Definição 1.2. (Caminho)

Seja $G = (V, E)$ um grafo (não) direcionado. Uma sequência $W = (v_1, \dots, v_n)$ de vértices é chamada de um caminho em G , se $(v_i, v_{i+1}) \in E$ ($\{v_i, v_{i+1}\} \in E$) para todo $i \in \{1, \dots, n-1\}$.

$W = (v_1, \dots, v_n)$ é chamado *simples*, se $v_i \neq v_j$ para todo $i, j \in \{1, \dots, n\}$, $i \neq j$. W é chamado de um *ciclo*, se $v_1 = v_n$.

Definição 1.3. (Árvore)

Um grafo direcionado $G = (V, E)$ é chamado de árvore se

1. G não contém um ciclo
2. Exatamente um nó de G tem grau de entrada 0, todos os outros nós tem grau de entrada 1.

Se $G = (V, E)$ é uma árvore, então o nó que possui grau de entrada 0 é chamado de *raiz* de G . Um nó com grau de saída 0 é chamado de uma *folha* de G , nós com grau de saída > 0 são chamados de nós *internos*. Se $v \in V$ é um nó de G diferente da raiz de G , então o nó v' com $(v', v) \in E$ é chamado de o *pai* de v . Se $v \in V$ não é uma folha, então os nós $v' \in V$ com $(v, v') \in E$ são chamados de *sucedores* ou *filhos* de v . Qualquer dois sucessores de um nó são chamados de *irmãos*. Para cada nó $v \in V$ a *profundidade* de v é a distancia da raiz da árvore até v . Todos os nós com a mesma profundidade d em G formam um *nível* em G .

Definição 1.4. (Subárvore)

Seja $G = (V, E)$ uma árvore com raiz v_r e seja $v \in V$ um nó de G . $G(v) = (V(v), E(v))$ com

$$V(v) = \{v' \in V | \text{o menor caminho de } v_r \text{ a } v' \text{ contém } v\}$$

$$E(v) = \{(v', v'') \in E | v', v'' \in V(v)\}$$

é chamada subárvore G com raiz v .

As seguintes notações sobre os nós de uma árvore é útil em muitas situações.

Definição 1.5. (Sistema Decimal de Dewey)

Seja $G = (V, E)$ uma árvore.

1. A raiz de G é chamada ε .
2. Os sucessores de um nó $v \in V$ com $\deg_o(v) = b \in \mathbb{N}_0$ são chamados $v.1, \dots, v.b$

Com a ajuda da notação decimal de Dewey somos capazes de distinguir os sucessores de um nó interno de G da seguinte forma:

Definição 1.6. (sucessor(es) à esquerda e à direita)

Seja $G = (V, E)$ uma árvore, seja $v \in V$ um nó interno de G .

1. O nó $v.1$ é chamado de sucessor à esquerda de v .
2. Todos os demais sucessores de v são chamados de sucessores à direita.

Usando as notações e definições acima, podemos definir uma árvore de jogo:

Definição 1.7. (Árvore de jogo)

Uma árvore de jogo $G = (V, E, f)$ é uma 3-upla satisfazendo

1. $G = (V, E)$ é uma árvore e $V \neq \emptyset$.
2. $f : V \rightarrow \mathbb{Z} \cup \{\pm\infty\}$

Os nós de uma árvore de jogo correspondem às posições do jogo, as arestas correspondem aos movimentos de uma posição à uma posição sucessora. A função f é chamada de *função de avaliação* para G . Ela mapeia as folhas de G para o conjunto ordenado $\mathbb{Z} \cup \{\pm\infty\}$. Nós de profundidade par em G são chamados de *MAX-nodes* e nós de profundidade ímpar em G são chamados de *MIN-nodes*.

Definição 1.8. (Valor Minmax)

Seja $G = (V, E, f)$ uma árvore de jogo. Então o valor minmax $F(v)$ de um nó $v \in V$ é definido como

$$F(v) = \begin{cases} f(v) & \text{se } v \text{ é uma folha de } G. \\ \max\{F(v') \mid (v, v') \in E\} & \text{se } v \text{ é um MAX-node interno de } G. \\ \min\{F(v') \mid (v, v') \in E\} & \text{se } v \text{ é um MIN-node interno de } G. \end{cases}$$

Com o uso da noção do valor minmax, o problema da busca em árvore de jogo pode ser formalizado como segue:

Definição 1.9. (Problema da busca em árvore de jogo)

O problema seguinte é definido como sendo o problema de busca em árvore de jogo
 Dado: Uma árvore de jogo $G = (V, E, F)$ com raiz ε .
 Computar: $F(\varepsilon)$.

Todo algoritmo que calcula o valor de $F(\varepsilon)$ é chamado de *algoritmo de busca em árvore de jogo*. Para qualquer nó interno de uma árvore de jogo, o conceito de melhor sucessor é definido como segue:

Definição 1.10. (Melhor sucessor)

Seja $G = (V, E, f)$ uma árvore de jogo, seja $v \in V$ um nó interno de G . Então, um sucessor $v.j$ é chamado de o melhor sucessor de v , se

- $F(v.j) > F(v.i), 1 \leq i < j$ e $F(v.j) \geq F(v.i), i \geq j$, if v é um MAX-node.
- $F(v.j) < F(v.i), 1 \leq i < j$ e $F(v.j) \leq F(v.i), i \geq j$, if v é um MIN-node.

Com isso, pode-se definir o conceito de variação principal, que é única para qualquer árvore de jogo.

Definição 1.11. (Variação principal)

Seja $G = (V, E, f)$ uma árvore de jogo com raiz ε . Então, a principal variação $(v_0, \dots, v_t)$ de G é definida por:

- $v_0 = \varepsilon$
- v_{i+1} é o melhor sucessor de $v_i, 0 \leq i < t$.
- v_t é uma folha de G .

Capítulo 2

Referencial Teórico

Neste capítulo são apresentadas as técnicas e os conceitos fundamentais para o entendimento do agente jogador de damas que será detalhado nos próximos capítulos.

2.1 Rede Neural Artificial - RNA

As redes neurais consistem em um grande número de nós relativamente simples, onde em cada um executa-se um mesmo algoritmo. Estes nós são os neurônios artificiais, originalmente destinados a simular as operações de uma única célula cerebral. Cada neurônio se comunica com um subconjunto de outros neurônios artificiais que compõe a rede. Eles são conectados em padrões característicos dos vários tipos de rede neural. Estes padrões são chamados de arquitetura ou topologia da rede neural [33].

O trabalho em redes neurais artificiais, usualmente denominadas “redes neurais”, tem sido motivado desde o começo pelo reconhecimento de que o cérebro humano processa informações de uma forma inteiramente diferente do computador digital convencional. O cérebro é um “computador” altamente complexo, não-linear e paralelo. Ele tem a capacidade de organizar seus constituintes estruturais, conhecidos por neurônios, de forma a realizar certos processamentos muito mais rapidamente do que os computadores modernos mais rápidos [18].

Destaca-se entretanto que, devido à grande diversidade de arquiteturas de RNA encontradas na literatura, apenas as de maior importância ou de alguma forma relevantes ao trabalho proposto serão abordadas, em especial o perceptron multicamada. Um panorama geral de outras arquiteturas pode ser encontrado em algumas referências bibliográficas da área [18], [14], [21].

2.1.1 A modelagem Matemática do Neurônio

Uma rede neural artificial é um modelo computacional, baseado em redes neurais biológicas, que consiste em uma rede de unidades básicas chamadas neurônios. O primeiro

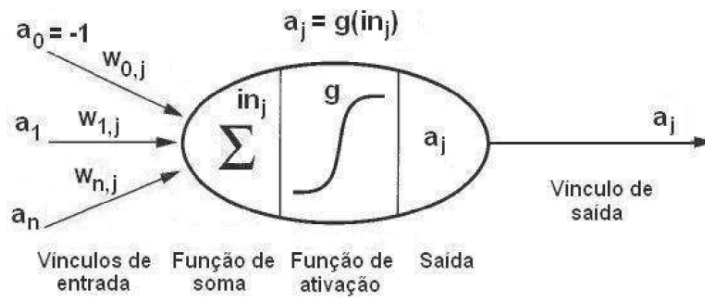


Figura 2.1: Modelo de um neurônio artificial [32]

modelo matemático de um neurônio artificial foi proposto por McCulloch e Pitts em 1943 [32] e pode ser visto na figura 2.1. Nesse modelo, tem-se:

1. o neurônio que é referenciado pela letra j ;
2. os vínculos de entrada são representados por a_0, a_1 até a_n . Exceto 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 da 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 $a_j = g(in_j) = g(\sum_{i=0}^n w_{ij} \cdot a_i)$, 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 limiar para o neurônio j , no sentido de que o neurônio j será ativado quando a soma ponderada das entradas reais $\sum_{i=1}^n w_{ij} \cdot a_i$ exceder $w_{0j} \cdot a_0$.

A função de ativação g , ou *camada de processamento de limiares*, é projetada para atender a duas aspirações:

1. o neurônio deverá ser ativado, se, e somente se, as entradas recebidas superarem o limiar do neurônio;
2. a ativação precisa ser não-linear para redes MLP, caso contrário, a rede neural se torna uma função linear simples;

Um exemplo de função de ativação é a tangente hiperbólica mostrada na figura 2.2.

O ajuste sináptico entre os neurônios de uma RNA representa o aprendizado, em cada neurônio, do fato apresentado, isto é, cada neurônio, conjuntamente com todos os outros, representa a informação que atravessou a rede. Nenhum neurônio guarda em si todo o conhecimento, mas faz parte de uma malha que retém a informação graças a todos os seus

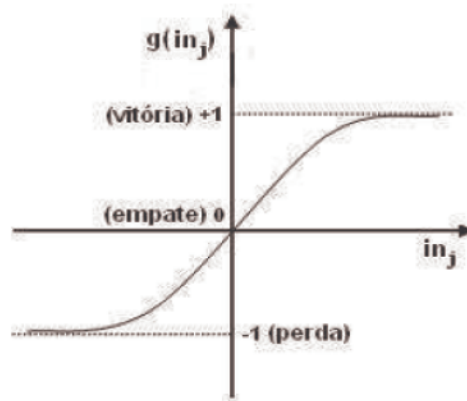


Figura 2.2: Modelo de uma função de ativação baseada na tangente hiperbólica

neurônios. Dessa forma, o conhecimento dos neurônios e, conseqüentemente, da própria rede neural, reside nos pesos sinápticos.

Desse modo, pode-se dizer que as redes neurais artificiais têm sido desenvolvidas como generalizações de modelos matemáticos de cognição humana ou neurobiologia, assumindo que:

- o processamento da informação ocorre com o auxílio de vários elementos chamados *neurônios*;
- os sinais são propagados de um elemento a outro por meio de *conexões*;
- cada conexão possui um *peso* associado que, em uma rede neural típica, pondera o sinal transmitido;
- cada neurônio possui uma *função de ativação* (geralmente não-linear), cujo argumento é a soma ponderada dos sinais de entrada, que determina a saída do neurônio.

Os tipos de redes neurais, apresentadas na próxima seção, são classificados de acordo com sua estrutura e a forma de treinamento.

2.1.2 Os tipos de Redes Neurais

A abordagem conexionista das RNAs abre um amplo leque de formas de conexão entre as unidades de processamento, isto é, os neurônios. Isso abrange o número de camadas presentes na rede, a forma de conexão entre tais unidades, a forma de treinamento, as funções de ativação presentes em cada camada etc.

Devido à grande bibliografia disponível sobre o tema redes neurais artificiais, até mesmo sua classificação gera algumas discussões. Fausett [14], por exemplo, define *arquitetura de uma rede neural* como a disposição dos neurônios em camadas e as conexões entre as camadas. Em um sentido mais amplo, outros pesquisadores utilizam a notação arquitetura na denominação de todo um conjunto de características de uma rede, en-

globando sua forma de treinamento, finalidade etc.

Em [63], Xing e Pham definem dois critérios básicos para a classificação das RNAs:

- Quanto à estrutura;
- Quanto à forma de treinamento.

A estrutura das redes

Com relação às estruturas, uma rede neural pode ser definida por duas formas:

- **Redes acíclicas ou redes de alimentação direta (*feedforward*):** 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. Um exemplo típico desse tipo de rede seria o Perceptron Simples ou o Perceptron Multicamadas (MLP);
- **Redes cíclicas ou redes recorrentes (*recurrent*):** as saídas de um (ou todos) os neurônios podem ser realimentadas a neurônios de camadas precedentes (tipicamente da primeira camada). Esse tipo de rede é classificado como *memória dinâmica*. Um exemplo típico dessa rede é a rede de Hopfield [19].

O treinamento das redes

Haykin propõe a seguinte definição para o aprendizado no contexto de redes neurais: “(...) é um processo pelo qual os parâmetros livres de uma rede neural são adaptados através de um processo de estimulação pelo ambiente no qual a rede está inserida. O tipo de aprendizagem é determinado pela maneira pela qual a modificação dos parâmetros ocorre” [18].

Esta definição de Haykin sobre o processo de aprendizagem implica na seguinte sequência de eventos:

1. A rede neural é *estimulada* por um ambiente;
2. A rede neural *sofre* modificações nos seus parâmetros livres como resultado desta estimulação;
3. A rede neural *responde de uma maneira nova* ao ambiente, devido às modificações ocorridas na sua estrutura interna.

A forma de treinamento, então, diz respeito a como são atualizados os valores dos pesos sinápticos durante o aprendizado da rede. Neste contexto, pode-se destacar:

1. Redes com treinamento supervisionado;
2. Redes com treinamento não-supervisionado;

3. Redes com aprendizado por reforço.

Nas *redes com treinamento supervisionado*, tipicamente, uma sequência de padrões de entrada associados a padrões de saída é apresentada à rede. Esta utiliza as comparações entre a sua classificação para o padrão de entrada e a classificação correta dos exemplos para ajustar seus pesos. Enquadram-se nesse contexto a maioria das redes utilizadas, como o Perceptron Simples e o Perceptron Multicamadas (MLP).

Nas *redes com treinamento não-supervisionado* não existe a apresentação de mapeamentos entrada-saída à rede: caberá exclusivamente a ela a tarefa de realizar a classificação correta, com base na informação de número de classes (padrões de saída) e topologia da rede. Para este tipo de treinamento não se usa um conjunto de exemplos previamente conhecidos. Uma medida da qualidade da representação do ambiente pela rede é estabelecida e os parâmetros são modificados de modo a otimizar esta medida.

O *aprendizado por reforço* refere-se à utilização de alguma função heurística (definida a priori) que descreve o quão boa é a resposta da rede a uma dada entrada, e da busca da maximização de tal função [63]. Em outras palavras, não é fornecido à rede o mapeamento direto entrada-saída, mas sim uma recompensa (ou penalização) decorrente da saída gerada pela rede à entrada apresentada. Tal reforço é utilizado no ajuste dos pesos da rede.

2.1.3 O *Perceptron* simples e o *Perceptron* multi-camada - MLP

Os perceptrons de única camada são o tipo mais antigo de redes neurais, as quais são formadas por uma camada única de neurônios de saída que estão conectados às entradas $x_i(n)$ através dos pesos $w_{ij}(n)$, onde $x_i(n)$ representa o i -ésimo elemento do vetor padrão de entrada na iteração n ; e, $w_{ij}(n)$ representa o peso sináptico conectando a entrada $x_i(n)$ à entrada do neurônio de saída j na iteração n (veja figura 2.3).

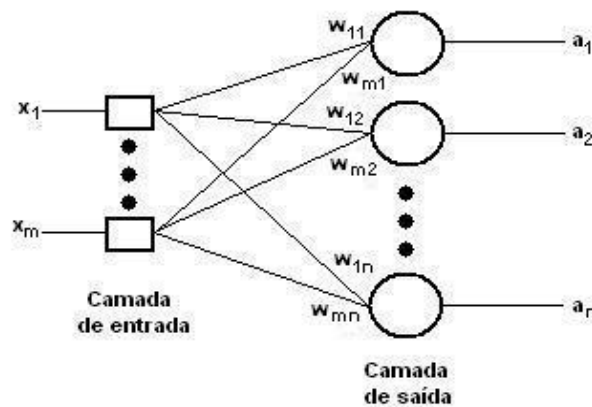


Figura 2.3: Arquitetura de um perceptron simples.

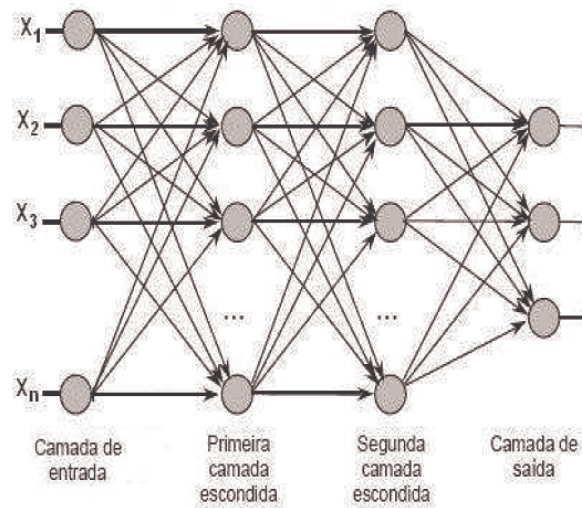


Figura 2.4: Arquitetura de um *Perceptron* Multicamadas - MLP

A soma do produto entre pesos e entradas alimenta cada neurônio de saída e, se o resultado desta operação exceder um certo limiar (bias), o neurônio de saída deverá ser ativado, caso contrário, será desativado. Neurônios com esse comportamento são chamados de *neurônios de McCulloch-Pitts* ou neurônios com limiar [32]. Na literatura, o termo *perceptron* diz respeito a redes com apenas um desses neurônios.

O modelo do *perceptron* de camada única de uma RNA consegue aprender apenas problemas linearmente separáveis. O *perceptron* simples é incapaz de resolver problemas cujas funções não são linearmente separáveis, isto é, problemas que apresentam características de comportamento não linear.

Os perceptrons multicamadas ou MLPs se caracterizam pela presença de uma ou mais camadas intermediárias ou ocultas (camadas em que os neurônios são efetivamente unidades processadoras, mas não correspondem à camada de saída). Adicionando-se uma ou mais camadas intermediárias, aumenta-se o poder computacional de processamento não-linear e armazenagem da rede. Em uma única camada oculta, suficientemente grande, é possível representar, com exatidão, qualquer função contínua. O conjunto de saídas dos neurônios de cada camada da rede é utilizado como entrada para a camada seguinte. A figura 2.4 ilustra uma rede MLP *feedforward* com duas camadas ocultas.

As redes *feedforwards* de múltiplas camadas são geralmente treinadas usando o algoritmo de retropropagação do erro (*error backpropagation*), embora existam outros algoritmos de treinamento. Este algoritmo requer a propagação direta (*feedforward*) do sinal de entrada através da rede, e a retropropagação (propagação reversa, ou *backpropagation*) do sinal de erro.

O algoritmo de retropropagação do erro (do inglês *error backpropagation*) é o principal algoritmo de treinamento para redes MLPs e certamente o algoritmo de treinamento mais conhecido dentro da comunidade de redes neurais. Também chamado de *regra delta generalizada*, o algoritmo é derivado do famoso método da regra delta de Widrow e Hoff

[60] e consiste em uma otimização não-linear baseada em gradiente descendente. Mais informações a respeito deste algoritmo podem ser encontradas em [14] e [18]

2.2 Aprendizagem por Reforço

O aprendizado por reforço é um nome genérico dado a uma família de técnicas na qual o agente tenta aprender uma tarefa através da interação direta com o ambiente. Este método tem suas raízes no estudo do comportamento animal sobre a influência de estímulos externos [57]. Nas últimas duas décadas, o aprendizado por reforço tem sido estudado extensivamente pela comunidade de Inteligência Artificial, onde a ênfase está na forma em que o agente pode aperfeiçoar seu desempenho, em uma dada tarefa, através da sua percepção e por tentativa e erro.

Dentro do paradigma de Aprendizagem de Máquina, as abordagens tradicionais que utilizam aprendizagem supervisionada trabalham com sistemas que aprendem através de exemplos de pares de entrada e saída. Tais pares fornecem aos sistemas indicativos de como se comportar para tentar aprender uma determinada função que “poderia” gerá-los. Formalmente, isto significa que, dado exemplos de pares $(x_i, f(x_i))$, onde x_i é a entrada e $f(x_i)$ é a saída da função aplicada a x_i , então a tarefa é encontrar, dentre uma coleção de exemplos de f , uma função h que mais se aproxime de f . Estes métodos são apropriados quando existe alguma espécie de “professor” fornecendo os valores corretos para a saída da função de avaliação. Entretanto, se não houver nenhum “professor” fornecendo exemplos, o que o agente poderá fazer? Experimentando movimentos aleatórios em seu ambiente, o agente terá que ter uma total liberdade e ser capaz de aprender, com base em recompensas ou reforços fornecidos por um “crítico” ou pelo próprio agente, através da observação das transições de estados que ele provoca no ambiente [44].

Segundo Sutton e Barto [54], Aprendizagem por Reforço (AR) nada mais é do que a aplicação dos conceitos básicos de Aprendizagem de Máquina: um indivíduo deve aprender a partir da sua interação com o ambiente onde ele se encontra, através do conhecimento do seu próprio estado no ambiente, das ações efetuadas no ambiente e das mudanças de estado que aconteceram depois de efetuadas as ações.

A importância de utilizar AR como uma técnica de aprendizagem está diretamente ligada ao fato de se tentar obter uma política ótima de ações. Tal política é representada pelo comportamento que o agente segue para alcançar o objetivo e pela maximização de alguma medida de reforço a longo prazo (globais), nos casos em que não se conhece, a priori, a função que modela esta política (função do agente-aprendiz).

Nas próximas seções serão abordados os conteúdos voltados à AR, apresentando suas características, seus problemas, sua formulação matemática em torno do *Modelo de Markov* e o método de resolução por TD(λ).

2.2.1 Fundamentos da Modelagem Markoviana

Existem dois conceitos que devem ser conhecidos para facilitar a modelagem de um problema como um sistema de Aprendizagem por Reforço. A seguir, será apresentado uma breve descrição destes conceitos, baseando-se em [54].

Propriedade de Markov

Quando a probabilidade de transição de um estado s para um estado s' depende apenas do estado s e da ação a adotada em s , isso significa que o estado corrente fornece informação suficiente para o sistema de aprendizado decidir que ação a ser tomada. Quando o sistema possui esta característica diz-se que ele satisfaz a *Propriedade de Markov* [3].

No caso mais geral, se a resposta em $t + 1$ para uma ação efetuada em t depende de todo o histórico de ações até o momento atual, a dinâmica do ambiente é definida pela especificação completa da distribuição de probabilidades, como mostra a equação abaixo:

$$Pr\{s_{t+1} = s', r_{t+1} = r | s_t, a_t, r_t, s_{t-1}, a_{t-1}, \dots, r_1, s_0, a_0\} \quad (2.1)$$

onde a probabilidade (Pr) do próximo estado s_{t+1} ser o estado s' e o reforço r_{t+1} ser r é uma função que depende de todos os estados, ações e reforços passados.

Logo, se a resposta do ambiente à ação a_t depende apenas dos estados e reforços em t , então, a probabilidade da transição para o estado s' em $t + 1$ pode ser obtida pela fórmula (2.2) (que corresponde a uma simplificação da equação (2.1) do caso geral):

$$P_{s,s'}^a = Pr\{s_{t+1} = s', r_{t+1} = r | s_t, a_t, r_t\} \quad (2.2)$$

A probabilidade de transição satisfaz às seguintes condições:

- 1) $P_{s,s'}^a \geq 0, \forall s, s' \in S, \forall a \in A(s);$ e,
- 2) $\sum_{s' \in S} P_{s,s'}^a = 1, \forall s \in S, \forall a \in A(s).$

A *Propriedade de Markov* é de fundamental importância na AR, uma vez que tanto as decisões como os valores são funções apenas do estado atual, abrindo a possibilidade de métodos de soluções incrementais, onde podem-se obter soluções a partir do estado atual e dos estados futuros, como é feito no método das Diferenças Temporais.

Processo de Decisão de Markov - PDM

Segundo Sutton e Barto [54], um *Processo de Decisão Markoviano* é definido como um conjunto de estados S , um conjunto de ações $A(s)$, $\forall s \in S$, um conjunto de transições entre estados associadas com as ações e um conjunto de probabilidades P sobre o conjunto

S que representa uma modelagem das transições entre os estados. Assim, dado um par de estado e ação, a probabilidade do estado s passar para um estado s' é:

$$P_{s,s'}^a = Pr\{s_{t+1} = s' | s_t = s, a_t = a\}$$

onde Pr é a probabilidade de transição, isto é, Pr representa a probabilidade do estado s_{t+1} ser s' , sempre que o estado s_t for igual a s e a ação a_t for igual a a . Desta forma, a dependência que o estado seguinte s_{t+1} seja o estado s' está relacionada a tomar a ação a no instante t .

Similarmente, dados um estado e ação atuais e um estado seguinte s' , o valor esperado do retorno é:

$$R_{s,s'}^a = E\{r_{t+1} | s_t = s, a_t = a, s_{t+1} = s'\}$$

onde E é o valor esperado do reforço r_{t+1} , sempre que o estado s_t no instante t passe a ser o estado s' no instante $t + 1$.

Os valores de probabilidade $P_{s,s'}^a$ e retorno esperado $R_{s,s'}^a$ determinam os aspectos mais importantes da dinâmica de um PDM finito:

- 1) um ambiente evolui probabilisticamente baseado em um conjunto finito e discreto de estados;
- 2) para cada estado do ambiente existe um conjunto finito de ações possíveis;
- 3) cada passo que o sistema de aprendizado executar uma ação, é verificado um custo positivo ou negativo para o ambiente em relação à ação executada; e,
- 4) estados são observados, ações são executadas e reforços são relacionados.

Assim, para quase todos os problemas de Aprendizagem por Reforço é suposto que o ambiente tenha a forma de um *Processo de Decisão de Markov*, desde que seja satisfeita a *Propriedade de Markov* no ambiente. Nem todos os algoritmos de AR necessitam uma modelagem PDM inteira do ambiente, mas é necessário ter-se pelo menos a visão do ambiente como um conjunto de estados e ações [54].

2.2.2 O Problema da Aprendizagem por Reforço

Um sistema típico de AR constitui-se, basicamente, de um agente interagindo em um ambiente via percepção e ação. Em outras palavras, o agente percebe as situações dadas no ambiente, pelo menos parcialmente, e, baseado nessas medições, seleciona uma ação a tomar no ambiente. A ação tomada muda de alguma forma o ambiente, afetando o estado na tentativa de alcançar o objetivo relacionado, e as mudanças são comunicadas ao agente através de um sinal de reforço. A figura 2.5 a seguir mostra a estrutura de um problema de Aprendizagem por Reforço [54].

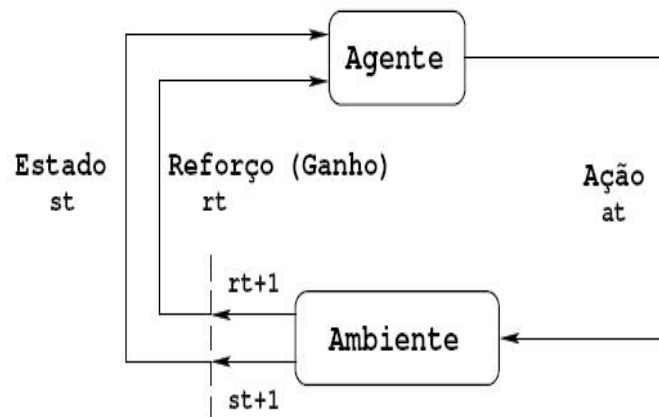


Figura 2.5: A interação agente-ambiente em Aprendizagem por Reforço.

Como os efeitos das ações não podem ser perfeitamente antecipados, o agente deve monitorar o ambiente frequentemente e reagir apropriadamente. Formalmente, o modelo de um sistema AR consiste em [54]:

1. um conjunto de variáveis de estado percebidas pelo agente, onde o conjunto das combinações de valores dessas variáveis forma o conjunto de estados discretos do agente (S);
2. um conjunto de ações discretas, que escolhidas por um agente muda o estado do ambiente ($A(s)$, onde $s \in S$); e,
3. valor das transições de estados, que é passado ao agente através de um sinal de reforço, denominado ganho (valores tipicamente entre $[0,1]$).

O objetivo do método de Aprendizagem por Reforço é levar o agente a escolher a sequência de ações que tendem a aumentar a soma de valores de reforço, ou seja, é encontrar a política π , definida como o mapeamento de estados em ações que maximize as medidas do reforço acumuladas no tempo.

Definido o problema e o objetivo central da Aprendizagem por Reforço, a seguir descrevem-se, sucintamente, com base em [54], os cinco componentes elementares deste contexto:

1. **Ambiente:** todo sistema de AR aprende um mapeamento de situações e ações por experimentação em um ambiente dinâmico. O ambiente no qual está inserido o sistema, em geral, deve ser, pelo menos parcialmente observável através de sensores, o que permite que toda informação relevante do mesmo esteja perfeitamente disponível para que o agente possa escolher ações baseadas em estados reais desse ambiente;

2. **Política:** uma política expressa pelo termo π representa o comportamento que o sistema AR segue para alcançar o objetivo. Em outras palavras, uma política π é um mapeamento de estados s e ações a em um valor $\pi(s, a)$. Assim, se um agente AR muda a sua política, então as probabilidades de seleção de ações sofrem mudanças e conseqüentemente, o comportamento do sistema apresenta variações à medida que o agente vai acumulando experiência a partir das interações com o ambiente. Portanto, o processo de aprendizado no sistema AR pode ser expresso em termos da convergência até uma política ótima ($\pi^*(s, a)$) que conduza à solução do problema de forma ótima;
3. **Reforço e Retorno:** o reforço é um sinal do tipo escalar (r_{t+1}) que é devolvido pelo ambiente ao agente assim que uma ação tenha sido efetuada e uma transição de estado ($s_t \rightarrow s_{t+1}$) tenha ocorrido. Existem diferentes formas de definir o reforço para cada transição no ambiente, gerando-se funções de reforço que, intrinsecamente, expressam o objetivo que o sistema AR deve alcançar. Assim, o agente deve maximizar a quantidade total de reforços recebidos, o que nem sempre implica em maximizar o reforço imediato (r_{t+1}) a receber, uma vez que pode corresponder, também, a maximizar o reforço acumulado durante a execução total. Tal acúmulo de reforços recebidos pelo ambiente, referente a uma determinada execução, é chamado de retorno. É justamente este retorno que o agente busca maximizar. De modo geral, pode-se dizer que o sistema AR busca maximizar o valor esperado de retorno, que pode ser definido como uma função da sequência de valores de reforço até um tempo T final. No caso mais simples é um somatório como aparece na equação (2.3):

$$R_T = r_{t+1} + r_{t+2} + r_{t+3} + \dots + r_T \quad (2.3)$$

Em muitos casos a interação entre agente e ambiente não termina naturalmente em um episódio (sequência de estados que chegam até o estado final), mas continua sem limite, como por exemplo, em tarefas de controle contínuo. Para essas tarefas a formulação do retorno é um problema, pois $T = \infty$ e o retorno que se deseja obter também tende ao infinito ($R_T \rightarrow \infty$).

Para estes problemas, é criada uma taxa de amortização (γ), a qual determina o grau de influência que têm os valores dos reforços futuros sobre o reforço total (ou retorno). Assim, aplicando a taxa de amortização à expressão do retorno da equação (2.3), têm-se a seguinte equação:

$$R_T = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \dots = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \quad (2.4)$$

onde, $0 < \gamma < 1$. Se $\gamma \rightarrow 0$, o agente tem uma visão míope dos reforços, maximizando apenas os reforços imediatos, e se $\gamma \rightarrow 1$, a visão do reforço abrange todos

os estados futuros dando maior importância ao estado final, desde que a sequência R_T seja limitada.

Dessa forma, um sistema de AR faz um mapeamento de estados em ações baseado nos reforços recebidos. Com isso, o objetivo do AR é definido usando-se o conceito de função de reforço, a qual é uma função dos reforços futuros que o agente procura maximizar. Ao maximizar essa função, o objetivo será alcançado de forma ótima. É a função de reforço que define quais são os bons e maus eventos para os agentes;

4. **Função de Reforço:** as funções de reforço podem ser bastante complicadas, porém existem pelo menos três classes de problemas frequentemente usadas para criar funções adequadas a cada tipo de problema:

- *Reforço só no estado final:* nesta classe, as recompensas são todas zero, exceto no estado final, em que o agente recebe uma recompensa (ex: +1) ou uma penalidade (ex: -1). Como o objetivo é maximizar o reforço, o agente irá aprender que os estados que levaram a uma recompensa são bons e os que levaram a uma penalidade devem ser evitados;
- *Tempo mínimo ao objetivo:* nesta classe as funções de reforço fazem com que o agente realize ações que produzam o caminho ou trajetória mais curta para alcançar um estado objetivo. Toda ação tem penalidade (-1), sendo que o estado final é (0). Como o agente tenta maximizar valores de reforço, ele aprende a escolher ações que minimizam o tempo que leva para alcançar o estado final;
- *Minimizar reforços:* em certas ocasiões o agente nem sempre precisa ou deve tentar maximizar a função de reforço, podendo também aprender a minimizá-la. Isto é útil quando o reforço é uma função para recursos limitados e o agente deve aprender a conservá-los ao mesmo tempo em que alcança o objetivo.

5. **Funções de Valor-Estado:** define-se uma função valor-estado como o mapeamento do estado, ou par estado-ação em um valor que é obtido a partir do reforço atual e dos reforços futuros.

Se a função valor-estado considera só o estado s é denotada como $V(s)$, enquanto se é considerado o par estado-ação (s, a) , então a função valor-estado é denotada como função valor-ação $Q(s, a)$.

a) **Função valor-estado $V(s)$:** uma vez que os reforços futuros mantêm dependências das ações futuras, as funções valor dependem também da política π que o AR segue. Em um *Processo de Decisão Markoviano* (ver subseção 2.4.3.2), define-se uma função valor-estado $V^\pi(s)$ dependente da política π

como a equação:

$$V^\pi(s) = E_\pi[R_t | s_t = s] = E_\pi\left[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} | s_t = s\right] \quad (2.5)$$

onde a função $V^\pi(s)$ é o valor esperado do retorno para o estado $s_t = s$, isto é, o somatório dos reforços aplicando a taxa de amortização γ .

b) Função valor-ação $Q(s, a)$: se considerar o par estado-ação, a equação para a função valor-estado $Q^\pi(s, a)$ será a seguinte:

$$Q^\pi(s, a) = E_\pi[R_t | s_t = s, a_t = a] = E_\pi\left[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} | s_t = s, a_t = a\right] \quad (2.6)$$

que é semelhante à equação (2.3), só que considerando o reforço esperado para um estado $s_t = s$ e uma ação $a_t = a$.

As equações 2.3 e 2.4 apresentam as funções *valor-estado* e *valor-ação* respectivamente, que dependem exatamente dos valores de reforço, o qual implica o conhecimento completo da dinâmica do ambiente como um *Processo de Decisão Markoviano* (PDM). Na próxima seção resumem-se o processo PDM e suas propriedades.

2.2.3 Método de solução por Diferenças Temporais

Dentre toda as técnicas existentes para solucionar o problema da Aprendizagem por Reforço, destacam-se a *Programação Dinâmica Adaptativa* (PDA), *Monte Carlo* (MC), *Q-learning*, *Sarsa* e Diferenças Temporais TD(λ) de Sutton. Esta seção resume o método das Diferenças Temporais, que é a técnica de aprendizado adotada no D-VisionDraughts.

Os métodos das Diferenças Temporais não exigem um modelo exato do sistema e permitem serem incrementais na busca de soluções para problemas de predição. Tal vantagem não se verifica em todos os algoritmos de AR. Por exemplo, a Programação Dinâmica Adaptativa (PDA) requer conhecimento a priori da distribuição de probabilidade completa para todas as possíveis transições de estado do ambiente [54].

As Diferenças Temporais [54] são capazes de utilizar o conhecimento anterior em ambientes parcialmente conhecidos para prever o comportamento futuro. Aprender a prever é uma das formas mais básicas e predominantes em aprendizagem. Através de um certo conhecimento, alguém poderia aprender a prever, por exemplo:

- se uma determinada disposição de peças no tabuleiro de xadrez conduzirá à vitória;
- se uma determinada formação de nuvens acarretará em chuva;
- se para uma determinada condição econômica de um país, isto implicará em um aumento ou diminuição na bolsa de valores.

Os métodos de Diferenças Temporais são guiados pelo erro ou diferença entre previsões sucessivas temporárias de estados seqüenciais experimentados por um agente em um domínio. Assim, o aprendizado do agente pelo método TD é extraído de forma incremental, diretamente da experiência deste sobre o domínio de atuação, atualizando as estimativas da função valor-estado sem a necessidade de ter que alcançar o estado final de um episódio (o episódio pode ser definido como sendo um único estado ou uma seqüência de estados de um domínio) para efetuar tais atualizações. Neste caso, a avaliação de uma política que defina o comportamento do agente sobre um ambiente, determinando que ação este deve executar em cada estado, é abordada como um problema de previsão, isto é, estimar a função valor-estado V^π sob a política π . A seguir, uma análise comparativa entre a equação de atualização da função valor-estado V^π do método TD e do método *Monte Carlo* será realizada a fim de verificar as vantagens de se utilizar o método TD.

Avaliação da Política - Previsão TD

Tanto TD quanto MC utilizam a experiência para resolver o problema da previsão. Dada certa experiência sob a política π , se é visitado um estado intermediário s_t , ambos os métodos atualizam suas estimativas da função valor-estado $V^\pi(s_t)$ baseando-se no que acontece depois de visitado o estado. Sendo que o método de *Monte Carlo* espera até que o retorno total de um episódio seja conhecido e usa esse retorno como objetivo para a atualização de $V^\pi(s_t)$, tal como é mostrado em ((2.7)):

$$V^\pi(s_t) = V^\pi(s_t) + \alpha[R_t - V^\pi(s_t)], \quad (2.7)$$

onde R_t representa o retorno atual no instante t e o α representa a constante de atualização ou taxa de aprendizagem, sendo que $\alpha \in [0, 1]$.

Por outro lado, os métodos de Diferenças Temporais não necessitam alcançar o estado final de um episódio, e, sim, o estado seguinte no instante $t + 1$. Em TD são utilizados o valor de reforço imediato r_{t+1} e a função de valor estimada $V^\pi(s_{t+1})$ para o próximo estado ao invés do valor real de retorno R_t , como no método de *Monte Carlo*, isto é, em TD a atualização é executada imediatamente após cada passo. Com estas condições, nos métodos de Diferenças Temporais a equação 2.8 converte-se na equação abaixo:

$$V^\pi(s_t) = V^\pi(s_t) + \alpha[r_{t+1} + \gamma V^\pi(s_{t+1}) - V^\pi(s_t)], \quad (2.8)$$

onde a atualização se refere ao valor $r_{t+1} + \gamma V^\pi(s_{t+1}) - V^\pi(s_t)$ que precisamente define a diferença entre os tempos t e $t + 1$. É devido a esta característica que a técnica recebe o nome de método das *Diferenças Temporais*. Como a atualização é feita a partir do estado seguinte, os métodos TD também são conhecidos como métodos *single-step*.

Algoritmo de predição TD para estimar V^π

```

INICIALIZAR  $\mathbf{V}(\mathbf{s})$  DE FORMA ARBITRÁRIA, E  $\pi$  (POLÍTICA A SER AVALIADA)
REPETE (PARA CADA EPISÓDIO):
    INICIALIZAR  $s$  (ESTADO INICIAL)
    REPETE (PARA CADA PASSO DO EPISÓDIO):
         $A \leftarrow$  AÇÃO DADA POR  $\pi$  PARA  $s$ 
        TOMAR A AÇÃO  $a$ , OBSERVAR REFORÇO  $r$  E PRÓXIMO ESTADO  $s'$ 
         $V(s) \leftarrow V(s) + \alpha[r + \gamma V(s') - V(s)]$ 
         $s \leftarrow s'$ 
    ATÉ  $s$  SER O ESTADO FINAL

```

Vantagens do Método TD

A principal vantagem do método de Diferença Temporal, em relação aos outros métodos tradicionais de Aprendizagem por Reforço, como o próprio método de *Monte Carlo* visto anteriormente, é que ele pode ser implementado de forma totalmente incremental, sendo que não há necessidade de aguardar até o final de um episódio para obter o retorno verdadeiro do estado atual, bastando, apenas, aguardar o estado seguinte. Existe um algoritmo baseado nas Diferenças Temporais que dispensa a política (*off-policy methods*) e utiliza a função de valor-ação $Q(s_t, a_t)$ para resolver problemas de predição. É o método *Q-learning*. Sua versão mais simples, *One-Step Q-learning*, é definida pela seguinte equação:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t)], \quad (2.9)$$

onde a função de valor-ação $Q(s_t, a_t)$ é atualizada a partir do seu valor atual do reforço imediato r_{t+1} e da diferença entre a máxima função valor no estado seguinte (encontrando e selecionando a ação do estado seguinte que a maximize) e o valor da função valor-ação no estado atual. O fato de selecionar a ação que maximize a função valor no estado seguinte faz com que a função valor-ação Q aprendida aproxime-se, diretamente, da função valor-ação ótima Q^* , sem depender da política que está sendo utilizada [54]. Observe que o algoritmo *Q-learning*, da mesma forma que o TD(λ), permite ser incremental na busca de soluções para problemas de predição.

O algoritmo de Aprendizagem por Reforço utilizado no sistema *D-VisionDraughts* é o método das Diferenças Temporais. Ele é utilizado para ajustar os pesos da rede neural do agente jogador de Damas. O processo de reajuste dos pesos e o funcionamento do algoritmo é discutido com mais detalhes na subseção 4.4.

Capítulo 3

Estado da Arte

Devido à vasta aplicabilidade de técnicas de aprendizado em jogos de computadores, este capítulo apresenta algumas das técnicas mais usadas no domínio de jogos, mais especificamente no domínio de jogos de tabuleiro.

A criação de programas jogadores com alto nível de desempenho tem sido um dos maiores triunfos da Inteligência Artificial. A exemplo disso, têm-se os sucessos obtidos com os jogos Gamão, Xadrez, Damas e Othello. Esses jogos usam variadas técnicas inteligentes no aprendizado de seus jogadores, e todos com bom nível de competição. Entre os métodos utilizados para obter esses resultados, a busca e a utilização da capacidade de memória dos computadores tem sido uma das técnicas mais bem sucedidas. Nesse contexto, a Aprendizagem por Reforço aparece como uma opção dentre técnicas inteligentes para a aquisição autônoma de conhecimento em estratégias de aprendizagem em jogos.

3.1 Redes Neurais e Diferenças Temporais aplicadas aos jogos de tabuleiro

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 tais funções [18]. Como exemplo, pode-se citar os jogadores TD-GAMMON de Tesauro [55], [56], o Neuro-Draughts de Lynch [28] e o LS-Draughts de Neto e Julia [36]. Todos eles utilizam redes neurais como funções de avaliação para treinar seus jogadores automáticos.

Um dos paradigmas utilizados por jogadores automáticos para o 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. O processo se dá da seguinte forma:

Primeiramente, a rede é recompensada por um bom desempenho (isto é, recebe do ambiente um reforço positivo correspondente ao estado final do jogo, em caso de vitória) e é punida por um mal desempenho (recebe do ambiente um reforço negativo correspondente ao fim do jogo, no caso de derrota). Para todos os estados de tabuleiro intermediários (entre o tabuleiro inicial e o tabuleiro final) representados na entrada da rede neural, como nenhuma recompensa específica é disponível, o mecanismo de diferenças temporais calcula a predição P de vitória por meio da equação (3.1).

$$P = g(in^{output}) \quad (3.1)$$

onde g é a função tangente hiperbólica e in^{output} é o campo local induzido do neurônio da camada de saída da rede neural. Isto significa que o valor de P depende dos pesos da rede neural. Então, uma predição P corresponde a um número real pertencente ao intervalo $[-1, +1]$, que indica o quanto o estado de tabuleiro representado na entrada da rede neural é favorável ao agente. A cada vez que o agente deve mover uma peça, o algoritmo de busca em profundidade é usado para expandir a árvore até uma profundidade n tendo a raiz S como o estado atual (resultante do último movimento do oponente), onde os nós da profundidade 1 correspondem aos estados resultantes de todos os movimentos possíveis disponíveis para o agente considerando o estado S , e assim por diante nas profundidades 2 a n . A rede neural calcula, então, as predições P para cada estado da profundidade n . Finalmente, estes valores serão retornados para o algoritmo de busca para que este possa indicar ao agente qual é a melhor ação a ser escolhida e executar em S . Sempre que o agente executa um movimento, os pesos da rede neural são atualizados de acordo com a equação (3.2) [54]:

$$\Delta w_t = \alpha(P_t - P_{t-1}) \sum_{k=1}^{t-1} \lambda^{(t-1)-k} \nabla_w P_k \quad (3.2)$$

onde P_t é a predição correspondente ao estado corrente e P_{t-1} é a predição correspondente a predição do estado prévio, cada P_k representa a predição correspondente de um estado anterior, α é a taxa de aprendizagem (definida de acordo com o quanto se quer que a última predição tenha impacto nos pesos da rede neural), λ é uma constante definida de acordo com o quanto se quer que o sistema deva considerar o valor de predições anteriores P_k , e ∇P_k corresponde a derivada parcial de P_k em relação a w (peso).

Uma contribuição relevante no campo de Aprendizado por Reforço ocorreu com o trabalho de Tesauro quando ele aplicou o método TD para treinar uma função de avaliação

de um jogo de tabuleiro. O programa do Tesouro, TD-Gammon [56], é um jogador de gamão que, apesar de ter pouco conhecimento sobre o jogo de gamão, foi capaz de jogar tão eficientemente quanto os melhores jogadores do mundo. Os princípios das diferenças temporais foram primeiramente aplicadas por Samuel [45], que foi o pioneiro na ideia de atualizar os coeficientes de um polinômio que servia como uma função de avaliação de um jogo de damas, pelo método de sucessivas predições. Outro trabalho bem sucedido com o método das diferenças temporais foi o proposto por Schaeffer et al. [51] que produziram um estudo comparativo detalhado entre funções de avaliação ajustada manualmente por especialistas (que é o caso do atual campeão de damas Chinook) e uma função de avaliação treinada por diferenças temporais. Esta análise mostrou que a estratégia de auto-aprendizado com o método das diferenças temporais é uma ferramenta eficiente para produzir agentes jogadores capazes de jogar com um alto nível de desempenho.

3.2 Programas de Computador para Damas

Esta seção e a próxima resumem alguns dos principais agentes bem sucedidos para o jogo de damas, incluindo o LS-Draughts e VisionDraughts, que consiste em versões anteriores do D-VisionDraughts.

3.2.1 Chinook

O Chinook é o mais famoso jogador de damas do mundo, sendo também o programa campeão mundial homem-máquina. Chinook não usa técnicas de aprendizagem de máquina para aperfeiçoar seu jogo, confiando principalmente em informações perfeitas contidas em bases de dados e em um rápido processamento para alcançar buscas profundas, tão profundas quanto possível.

Chinook usa uma função de avaliação linear ajustada manualmente (cujo papel é estimar o quanto o estado de tabuleiro é favorável a ele) que considera diversas características do jogo, por exemplo: número de peças, número de damas, damas cercadas e peças com caminho desimpedido para virarem damas. Além disso, ele possui acesso a uma biblioteca de movimentos de abertura de partidas jogadas por grandes mestres e uma poderosa base de dados de fim de jogo, uma coleção de posições geradas automaticamente com valor teórico de jogo provado. Atualmente, esta base de dados é composta por uma coleção de 39 trilhões de posições (todas as posições com 10 peças ou menos no tabuleiro). Para escolher o melhor movimento a ser executado, Chinook usa busca alpha-beta paralelo com aprofundamento iterativo e tabelas de transposição.

Recentemente, Schaeffer et al. conseguiram resolver o problema do jogo de damas [48]. Mais detalhes sobre o Chinook podem ser encontrados em [50].

3.2.2 Anaconda

Fogel explorou o potencial de um processo co-evolutivo para aprender a jogar damas sem contar com a usual inclusão de conhecimentos de peritos humanos. O autor focou no uso de uma população de redes neurais, onde cada rede serviu como uma função de avaliação para descrever a qualidade de um estado de tabuleiro. A rede neural usou apenas posições, tipos e número de peças do tabuleiro como entrada da rede neural. Nenhuma outra característica que envolveria conhecimento humano foi incluída.

Experimentos foram conduzidos para verificar a melhor rede neural, chamada Anaconda, por uma competição em 10 jogos contra uma versão preliminar do Chinook. Para propósitos comparativos, Fogel avaliou os experimentos de duas maneiras, primeiro, sem acesso ao banco de dados do Chinook e depois com acesso. Da primeira forma, Anaconda obteve 2 vitórias, 4 derrotas e 4 empates, enquanto que da segunda forma, anaconda melhorou seu desempenho com 4 vitórias, 3 derrotas e 3 empates. Assim, Anaconda é avaliado como um jogador especialista em damas.

3.2.3 Cake

Cake é um jogador de damas criado por Martin Fierz [16]. Ele roda sobre o Checker-Board (interface de damas para o sistema operacional windows) e faz uso de uma base de dados de fim de jogo, que significa o conhecimento perfeito do valor de qualquer jogo que tenha até 8 peças no tabuleiro. Também faz uso de um livro de abertura gerado automaticamente que contém até 2 milhões de movimentos. Entre as fases de abertura e fim de jogo, Cake usa o algoritmo de busca MTD(f) para a escolha dos movimentos, com uma avaliação média de 2 milhões de movimentos por segundo em um computador moderno. Mais detalhes sobre o Cake podem ser vistos em [16].

3.2.4 NeuroDraughts

O NeuroDraughts é um agente que aprende sem supervisão humana. Sua arquitetura é mostrada na figura 3.1. O módulo da rede neural artificial (RNA) consiste em uma rede MLP, que é a parte central do NeuroDraughts. Ela corresponde a um perceptron de três camadas cuja camada de saída é composta de um único neurônio. O papel desta rede é estimar o quanto o estado de tabuleiro corrente - representado por meio de 12 características manualmente e arbitrariamente escolhidas pelo autor do sistema - é favorável ao agente. Esta estimativa é a predição e é quantificada por um número real compreendido entre 0 e 1. O módulo de busca *Busca Minimax* seleciona, usando o algoritmo de busca em árvore minimax e as predições calculadas, o melhor movimento correspondente ao estado de tabuleiro corrente.

O módulo de busca TD corresponde ao processo de aprendizado do agente jogador ou

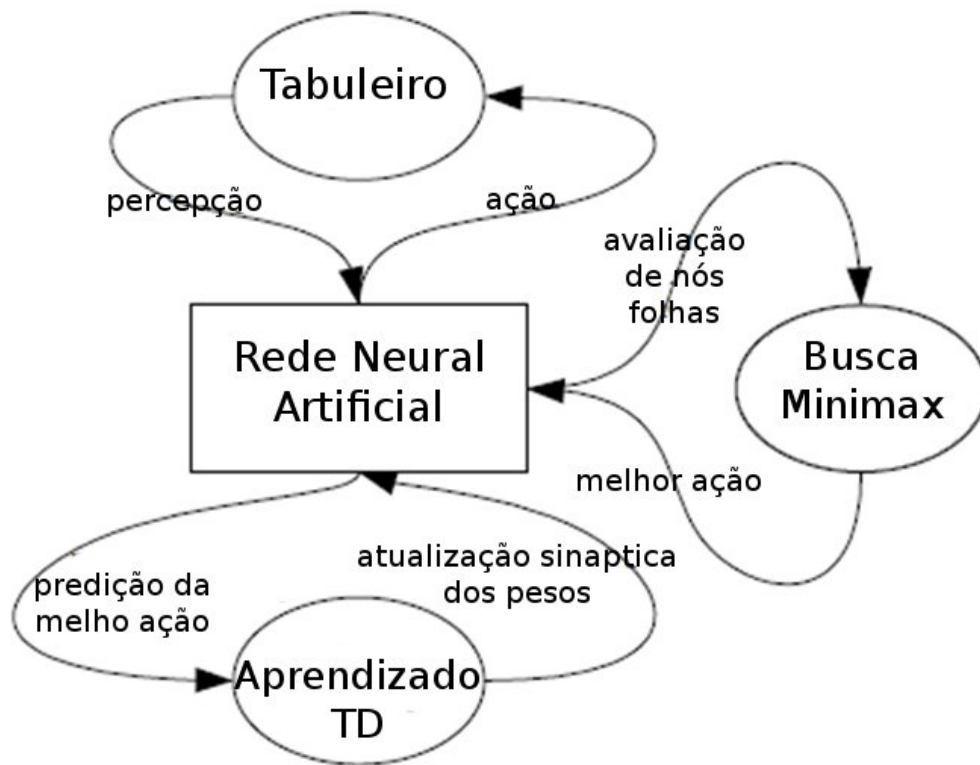


Figura 3.1: Arquitetura do *NeuroDraughts*

RNA. Este módulo usa o método das diferenças temporais e *self-play* com clonagem como estratégia de treino.

Mais detalhes sobre NeuroDraughts podem ser vistos em [28], [29]

3.3 Versões anteriores do D-VisionDraughts

Como já mencionado, o D-VisionDraughts foi inspirado no NeuroDraughts e teve três versões anteriores: O LS-Draughts, na qual os autores, pelo uso de algoritmos genéticos, automatizaram o processo de seleção de características que representarão os estados de tabuleiro; o VisionDraughts, onde os autores otimizaram o processo de busca introduzindo a poda alpha-beta, tabelas de transposição e aprofundamento iterativo; e o MP-Draughts que, através do uso de uma arquitetura multiagente, aprimorou a capacidade do VisionDraughts de finalizar jogos onde claramente estava em vantagem.

3.3.1 LS-Draughts

A fim de melhorar o NeuroDraughts, Neto de Julia inicialmente propuseram o LS-Draughts [36]. O LS-Draughts amplia o NeuroDraughts com a inserção de algoritmos genéticos que automaticamente selecionam as características que irão representar os estados de tabuleiro. O algoritmo genético gera Tp indivíduos que representam subconjuntos de todas as características do mapeamento NET-FEATUREMAP. Cada indivíduo gerado é anexado a uma rede neural MLP que aprende, assim como o NeuroDraughts, pelo método das diferenças temporais. Como no NeuroDraughts, o treinamento do LS-Draughts se dá por torneios baseados em *self-play* com clonagem. O objetivo do processo de treinamento do LS-Draughts é detectar qual conjunto de características é o mais conciso e apropriado para produzir um jogador eficiente.

Os resultados obtidos mostraram que os algoritmos genéticos são, de fato, uma boa alternativa para melhorar o processo de aprendizagem do NeuroDraughts: os melhores indivíduos da 25ª geração derrotaram o NeuroDraughts (em um torneio com 9 jogos, obteve 2 vitórias e 5 empates) contando com apenas 7 características para representar os estados de tabuleiro (contra 12 usadas pelo NeuroDraughts). Mais detalhes sobre o LS-Draughts podem ser encontrados em [35], [37].

3.3.2 VisionDraughts

Com o intuito de aprimorar o NeuroDraughts, Caixeta e Julia propuseram o VisionDraughts [9], [10]. O VisionDraughts é um jogador automático de damas que substitui o módulo de busca minimax do NeuroDraughts por uma rotina de busca baseado em poda alpha-beta, tabelas de transposição e aprofundamento iterativo. Este agente serviu como ponto de partida para a construção do D-VisionDraughts e será descrito em mais detalhes na seção 4. Resultados experimentais mostraram que o tempo requerido para efetuar a busca foi reduzido para 2.27% do tempo original, comparado com a busca usada pelo NeuroDraughts, o que permitiu ao VisionDraughts aumentar a profundidade de busca e obter assim um jogador muito melhor em um tempo de treinamento menor. O capítulo 4 apresentará em mais detalhes o VisionDraughts, uma vez que sua compreensão é imprescindível para a análise e entendimento do D-VisionDraughts proposto neste trabalho.

3.3.3 MP-Draughts

O MP-Draughts é um sistema multiagente com agentes responsáveis por fases distintas do jogo. Este jogador possui 26 agentes especializados, sendo que um agente é especializado nas fases iniciais/intermediárias de um jogo (tabuleiros com no mínimo 13 peças), chamado de IIGA (Initial/Intermediate Game Agent) e outros 25 agentes para fases finais

(tabuleiros com 12 peças ou menos), chamados de EGA (End Game Agent). Todos os agentes possuem a mesma estrutura de um agente do *VisionDraughts* e cada um dos 25 EGA é treinado para lidar com um determinado *cluster* de estados de tabuleiro de final de jogo diferente. Esses 25 clusters são minerados de uma base de dados de 4000 estados de tabuleiros através do algoritmo de Kohonen-SOM. O jogo é conduzido da seguinte maneira: primeiro, uma versão do *VisionDraughts* é usada como o IIGA enquanto o jogo tiver 13 peças ou mais; depois, um agente de final de jogo que representa o cluster que mais se aproxima do estado corrente do tabuleiro do jogo deverá substituir o IIGA e jogar até o fim da partida. Os resultados obtidos indicaram que, apesar do aumento no tempo de treino por causa do número maior de agentes a serem treinados, o uso de uma arquitetura multiagente aumentou o desempenho do jogador, ganhando, em um torneio de 20 jogos contra o *VisionDraughts*, 10 jogos e perdendo 6.

Capítulo 4

VisionDraughts - Um Sistema de Aprendizado de Damas com Eficiente Método de Busca

Neste capítulo é apresentado o *VisionDraughts* [10], [9], um sistema de aprendizagem de jogos de damas cujo principal objetivo é construir um agente automático capaz de aprender a jogar com alto nível de desempenho e sem auxílio de especialistas humanos. Esse jogador é um aperfeiçoamento do jogador *NeuroDraughts* de Mark Lynch [28], [29], e utiliza um eficiente módulo de busca no lugar do algoritmo de busca minimax usado pelo *NeuroDraughts*.

O sistema implementa um agente jogador de damas como uma rede neural MLP que utiliza a busca com poda alfa-beta, tabela de transposição e aprofundamento iterativo para, em função do estado corrente do jogo, escolher a melhor jogada. Além disso, ele utiliza o método de aprendizagem por reforço $TD(\lambda)$ aliado à estratégia de treino por *self-play* com clonagem como ferramenta para atualizar os pesos da rede. Para tanto, o tabuleiro é representado por um conjunto de funções que descrevem as características do próprio jogo de damas. A utilização de um conjunto de características para representar o mapeamento do tabuleiro de damas na entrada da rede neural é a mesma definida por Lynch em seu jogador *NeuroDraughts* [28] como sendo um mapeamento NET-FEATUREMAP (a ser detalhado na seção 4.1.2).

Dessa forma, para conseguir alto nível de desempenho, o *VisionDraughts* introduziu um novo módulo em relação ao jogador *NeuroDraughts* [28] [29], que é um módulo de busca eficiente em árvores de jogos que fornece ao agente jogador de damas maior capacidade de analisar jogadas futuras.

As próximas seções abordam, em detalhes, o desenvolvimento do *VisionDraughts*, cuja compreensão é fundamental para o entendimento do jogador *D-VisionDraughts* (apresentado no capítulo 6). As seções se sucedem de acordo com o disposto a seguir: i) a repre-

sentação do tabuleiro nos jogadores de damas; ii) a arquitetura geral do *VisionDraughts*; iii) o cálculo da predição e a escolha da melhor ação; iv) reajuste de peso da rede MLP; v) e finalmente o eficiente módulo de busca desse jogador.

4.1 Representação do Tabuleiro nos Jogadores de Damas

Existem diversas maneiras de representar internamente um tabuleiro de jogo de damas [9]. A representação interna de um tabuleiro no jogador *VisionDraughts* faz-se de duas maneiras: representação vetorial e representação por características, também chamada de NET-FEATUREMAP. Essas representações também são usadas no jogador desenvolvido nesse trabalho, o *D-VisionDraughts*.

A representação vetorial é usada para representar o tabuleiro na entrada do algoritmo alfa-beta durante o processo de busca pelo melhor movimento. A representação NET-FEATUREMAP é usada na entrada das redes neurais no momento de calcular a predição de um dado estado.

As próximas seções se dedicam a explicar, em detalhes, como é feito cada um dos tipos de representação de tabuleiros citados acima.

4.1.1 Representação Vetorial

A escolha de uma estrutura de dados para representar o tabuleiro em um jogo é fundamental para a eficiência de um jogador automático de damas. Algoritmos de busca em profundidade, baseados no alfa-beta, estão presentes nos melhores projetos da história dos jogos de tabuleiro; e a escolha adequada de uma estrutura de dados para representar o tabuleiro afeta, consideravelmente, a velocidade de execução desse tipo de algoritmo. A representação vetorial é recomendada para casos em que um algoritmo alfa-beta é usado em virtude de sua precisão na representação de um estado de tabuleiro.

O vetor que representa o estado de tabuleiro de um jogo tem 32 posições correspondentes às posições existentes em um tabuleiro de damas. O modo de como se faz a representação vetorial de um tabuleiro de damas é apresentado a seguir:

O tabuleiro do jogo de damas representado, de maneira simples e direta, por meio de representação vetorial:

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

```

    REDKING = 4
}

BOARD{
    BOARDVALUES p[32]
}

```

O tabuleiro é uma estrutura do tipo BOARD implementada como 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 (EMPTY, BLACKMAN, REDMAN, BLACKKING, REDKING).

4.1.2 Representação NET-FEATUREMAP

A utilização de um conjunto de características para treinar um jogador de damas foi primeiramente proposta por Samuel [45], com o intuito de prover medidas numéricas para melhor representar as diversas propriedades de posições de peças sobre um tabuleiro. Várias dessas características implementadas por Samuel resultaram de análises feitas sobre o comportamento de especialistas humanos em partidas de damas. Em termos práticos, essas análises tinham o objetivo de tentar descobrir quais características referentes a um estado do tabuleiro são frequentemente analisadas e selecionadas pelos próprios especialistas quando vão escolher seus movimentos de peças durante uma partida de damas.

Samuel implementou 28 características referentes ao domínio de jogo de damas. Caixaeta e Julia utilizaram 12 dessas características para representar o tabuleiro do jogo no *VisionDraughts* [9].

Tais características fornecem medidas quantitativas e qualitativas para melhor representar as diversas propriedades das posições das peças sobre um tabuleiro de damas. Essas características, implementadas por Samuel, podem ser observadas na tabela 4.1.

Conjunto de Características implementadas por Samuel

CARACTERÍSTICAS	DESCRIÇÃO FUNCIONAL
<i>PieceAdvantage</i>	Contagem de peças em vantagem para o jogador preto (no caso, número de peças que o jogador tem a mais).
<i>PieceDisadvantage</i>	Contagem de peças em desvantagem para o jogador preto (no caso, número de peças que o jogador tem a menos).
<i>PieceThreat</i>	Total de peças pretas que estão sob ameaça.
<i>PieceTake</i>	Total de peças vermelhas que estão sob ameaça de peças pretas.

<i>Advancement</i>	Total de peças pretas que estão na 5 ^a e 6 ^a linha do tabuleiro menos as peças que estão na 3 ^a e 4 ^a linha.
<i>DoubleDiagonal</i>	Total de peças pretas que estão na diagonal dupla do tabuleiro.
<i>BackRowBridge</i>	Se existe peças pretas nos quadrados 1 e 3 e se não existem rainhas vermelhas no tabuleiro.
<i>CentreControl</i>	Total de peças pretas no centro do tabuleiro.
<i>XCentreControl</i>	Total de quadrados no centro do tabuleiro onde tem peças vermelhas ou para onde elas possam mover.
<i>TotalMobility - MOB</i>	Total de quadrados vazios para onde as peças vermelhas podem mover.
<i>Exposure</i>	Total de peças pretas que são rodeadas por quadrados vazios em diagonal.
<i>Taken</i>	Total de posições para qual uma peça vermelha pode se mover e, assim, poder ameaçar uma peça preta em um movimento subsequente.
<i>Pole</i>	Total de peças simples pretas que estão completamente rodeada de posições vazias.
<i>Node</i>	Total de peças vermelhas que estão rodeada por, no mínimo, 3 posições vazias.
<i>KingCentreControl</i>	Total de rainhas pretas no centro do tabuleiro.
<i>TriangleofOreo</i>	Se não existe rainhas vermelhas e se o triângulo de Óreo (posições 2,3 e 7) está ocupado por peças pretas, ou se não existe rainhas pretas e se o triângulo de Óreo (posições 26,30 e 31) esta ocupado por peças vermelhas.
<i>Threat</i>	Total de posições para qual uma peça preta pode se mover e assim poder ameaçar uma peça vermelha em um movimento subsequente.
<i>UndeniedMobility</i>	Esta característica é a diferença entre <i>MOB</i> e <i>DenialofOccupancy</i> .
<i>Hole</i>	Total de posições vazias que estão rodeadas por 3 ou mais peças.
<i>BackRowControl</i>	Se não existem rainhas pretas e, se, uma ponte ou um triângulo de Oreo é ocupado por uma peça vermelha

<i>Gap</i>	Total de posições vazias separadas por duas peças vermelhas ao longo de uma diagonal ou que separam uma peça vermelha de uma margem do tabuleiro.
<i>Exchange</i>	Total de posições que a peça preta pode avançar uma casa e forçar o oponente a comer uma peça.
<i>DiagonalMoment</i>	Total de peças vermelhas que estão localizadas na posição 1 ou na posição 2 de uma diagonal dupla mais as peças passivas que estão na ponta da diagonal.
<i>Apex</i>	Se não existe rainha preta no tabuleiro, ou se os quadrados 7 ou 26 estão ocupados por uma peça preta e se nenhum desses quadrados estão ocupado por uma peça vermelha.
<i>Cramp</i>	Se ocorre uma limitação de posição para as peças vermelhas e pelo menos uma das posições estão ocupadas por peças pretas.
<i>Dyke</i>	Total de peças vermelhas que ocupem três posições de diagonal adjacentes.
<i>DenialofOccupancy</i>	Total de posições onde, no próximo movimento, uma peça preta pode ocupar esta posição e ser capturada na próxima jogada.
<i>Fork</i>	Total de peças vermelhas que ocupam duas posições adjacentes em uma coluna e que nessa mesma coluna, existam três posições vazias; assim a peça preta, pela ocupação de uma dessas posições pudo capturar uma ou duas peças vermelhas.

Tabela 4.1: Conjunto de 28 Características implementadas por Samuel

A representação NET-FEATUREMAP é mais conveniente do que a representação vetorial em alguns casos como, por exemplo, na representação do tabuleiro de jogo de damas na entrada das redes neurais no momento de calcular a predição de um dado estado, uma vez que fornece, também, uma visão qualitativa do tabuleiro que é mais apropriada ao processo de sua avaliação do que a visão meramente posicional e quantitativa da representação vetorial.

Sendo assim, uma coleção de n características fornece uma representação quantitativa e qualitativa de um conjunto B de estados de tabuleiros. Cada característica f_i representa um certo atributo de um determinado tabuleiro x e pode ser definida como:

$f_i : B \longrightarrow \mathbb{N}$, onde:

$f_i(x) = a$, em que a representa o atributo do tabuleiro x referente à característica f_i ,

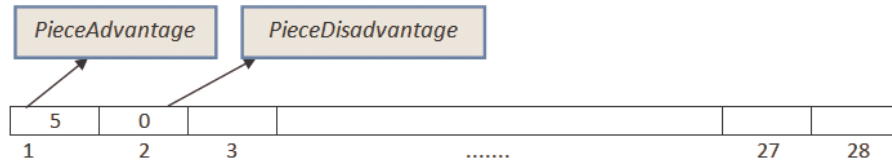


Figura 4.1: Exemplo de tabuleiro representado por NET-FEATUREMAP

conforme exemplificado abaixo.

Considere f_i como sendo a característica peças em vantagem "*PieceAdvantage*", que verifica em um dado tabuleiro x a quantidade de peças em vantagem de um jogador. Se o jogador para o qual a característica está sendo verificada possui 5 peças em vantagem em relação ao seu oponente, o atributo $f_i(x)$ produz 5 (ou seja, $a = 5$); caso ele não possua peças em vantagem, o valor do atributo $f_i(x)$ é 0 (ou seja, $a = 0$). Se um tabuleiro possui peças em vantagem para um jogador, consequentemente, ele não possui peças em desvantagem para esse mesmo jogador. A figura 4.1 ilustra como seria a representação do tabuleiro em relação às características *PieceAdvantage* e *PieceDisadvantage*, ambas ocupando a 1ª e a 2ª posição do tabuleiro, respectivamente, considerando 5 peças em vantagem.

Conforme visto anteriormente, a conveniência de se usar representação vetorial ou representação NET-FEATUREMAP depende do processo em foco. Para as situações em que a representação NET-FEATUREMAP é mais adequada, faz-se necessário um processo de conversão de representação vetorial para tal representação (módulo NET-FEATUREMAP apresentado na arquitetura do jogador), que é definido pelo mapeamento C abaixo:

$C: B \rightarrow \mathbb{N}^n$, onde:

$$C(x) = \langle f_1(x), \dots, f_n(x) \rangle$$

Em que n é a quantidade de características usadas na representação do tabuleiro por características, a serem escolhidas entre as 28 características propostas por Samuel [45], [46]. Isso significa que cada estado de tabuleiro x é representado por uma n -tupla composta de n atributos, os quais correspondem as características $f_1, \dots, f_n$, respectivamente.

4.2 Arquitetura Geral do *VisionDraughts*

O fluxo mostrado na figura 4.2 ilustra a arquitetura do jogador *VisionDraughts*. Conforme já comentado, esse jogador consiste de uma rede neural MLP treinada pelo método de aprendizagem por reforço $TD(\lambda)$. Esse método utiliza previsões (ou estimativas de resultados finais de jogo) sucessivas para ensinar a rede neural a jogar damas. O jogador alcança alto nível de desempenho simplesmente jogando contra si mesmo (estratégia de treino por *self-play* com clonagem) com uma busca eficiente e com mínima análise de jogos de especialistas humanos.

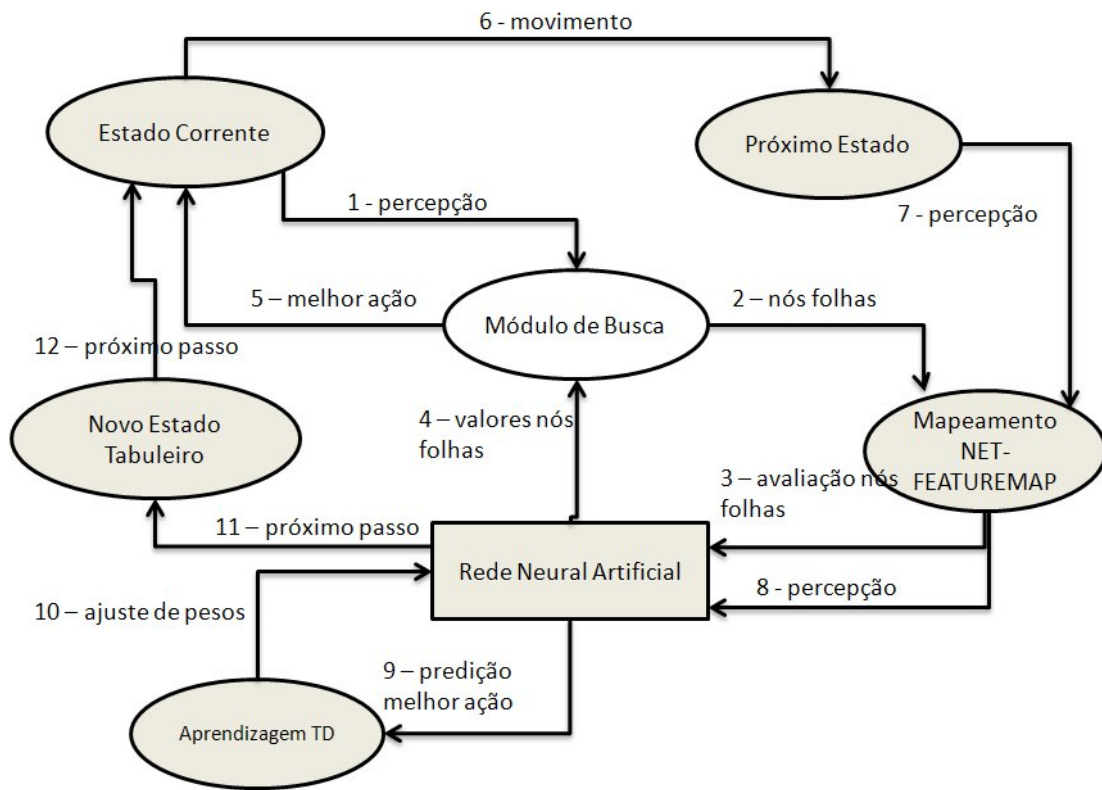


Figura 4.2: Fluxo de aprendizado do *VisionDraughts*: um sistema de aprendizagem de jogos de damas

Os módulos no qual a arquitetura se divide, mostrados na figura 4.2, são:

1. **Rede Neural Multicamadas:** 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 (predição) representa o quão o estado do tabuleiro presente na camada de entrada da rede é favorável ao jogador automático;
2. **Módulo *NET-FEATUREMAP*:** por meio do mapeamento C (apresentado na seção anterior), esse módulo converte a representação vetorial do estado corrente em representação NET-FEATUREMAP e apresenta esta última à de entrada da rede neural MLP. Como o *VisionDraughts* usa 12 características para representar o tabuleiro, tem-se que $C: B \rightarrow \mathbb{N}^{12}$. 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 na conversão, constituem a sequência binária de saída do módulo NET-FEATUREMAP a ser apresentada na entrada da rede neural. As características implementadas no *VisionDraughts*, juntamente com as quantidades de bits significativos de cada uma, podem ser vistos na tabela 4.2:

CARACTERÍSTICAS	Nº BITS
<i>PieceAdvantage</i>	4
<i>PieceDisadvantage</i>	4
<i>PieceThreat</i>	3
<i>PieceTake</i>	3
<i>Advancement</i>	3
<i>DoubleDiagonal</i>	4
<i>BackRowBridge</i>	1
<i>CentreControl</i>	3
<i>XCentreControl</i>	3
<i>TotalMobility - MOB</i>	4
<i>Exposure</i>	3
<i>KingCentreControl</i>	3

Tabela 4.2: Conjunto de Características implementadas no jogador *VisionDraughts*

- Módulo Eficiente de Busca:** seleciona a melhor ação a ser executada pelo agente em função do estado corrente do tabuleiro. Esse módulo expande uma árvore em busca da melhor ação e os nós presentes na camada mais profunda (folhas) são mapeados, pelo módulo 2 na entrada da rede neural do módulo 1;
- Módulo Ajuste de Peso:** o ajuste dos pesos da rede neural é feito pelo método das Diferenças Temporais $TD(\lambda)$. Esse processo é responsável pela aquisição de conhecimento do sistema;

De acordo com a figura 4.2, o funcionamento do *VisionDraughts* se dá da seguinte forma: Sempre que o agente deve escolher uma peça para mover, o estado corrente, B1, é apresentado para o algoritmo Alpha-Beta serial, mostrado no passo 1 da figura. O módulo de busca expande a árvore de jogo correspondente ao tabuleiro B1 (passo 2), e submete cada tabuleiro situado nas folhas da árvore para a rede neural no passo 3 (previamente, estes tabuleiros são convertidos em uma representação por características no módulo de mapeamento). A rede neural avalia cada um destes tabuleiros e dá uma predição, que indica o quanto o tabuleiro é favorável ao agente jogador. Este valor é retornado para o algoritmo de busca (passo 4), de modo a permiti-lo apontar qual o melhor movimento a ser aplicado em B1 (passo 5). O agente então executa o movimento (passo 6) e o estado de tabuleiro alcançado é mapeado (passo 7) e apresentado para a rede neural (passo 8). Se o agente estiver em modo de treinamento, a predição calculada pela rede neural para o tabuleiro B2 será usada pelo módulo de aprendizagem $TD(\lambda)$ para atualizar os pesos da rede neural (passo 9). Note que se o agente não estiver em treinamento, ele apresenta a mesma arquitetura descrita acima, apenas retirando o módulo de aprendizagem $TD(\lambda)$ e os passos 9 e 10.

Quase todos os programas para jogos de tabuleiro possuem a mesma arquitetura básica [38]. Suas diferentes partes precisam gerar os próximos movimentos legais, avaliá-los e retornar o melhor movimento a ser executado pelo agente jogador.

Observe que, para o problema de treinar uma rede neural com a finalidade de jogar damas utilizando o método $TD(\lambda)$ e mapeamento *NET-FEATUREMAP* do tabuleiro, a escolha da melhor ação está vinculada a dois fatores fundamentais: a profundidade de busca (que indica a capacidade que o agente jogador possui de analisar combinações futuras de jogadas) e a função de avaliação (que avalia uma determinada posição do tabuleiro do jogo).

4.3 Cálculo da Predição e Escolha da Melhor Ação

O cálculo das predições é efetuado por uma rede neural artificial acíclica com 3 camadas. O número de nós 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 4.3. Nessa arquitetura, cada um dos neurônios da rede está conectado a todos os outros das camadas subsequentes (amplamente conectada); um termo *bias* é utilizado para todos os neurônios da camada oculta e outro para o neurônio da camada de saída.

Formalmente, o processo de cálculo da predição P_t referente a uma configuração do tabuleiro do jogo de damas em um instante temporal t , isto é, S_t , pode ser descrito da seguinte forma:

1. a representação interna do tabuleiro S_t é mapeada na entrada da rede neural pelo

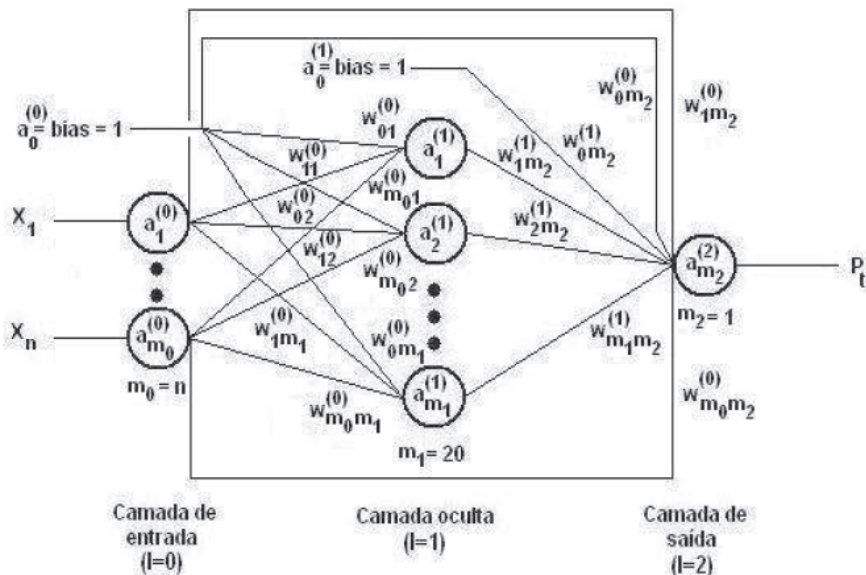


Figura 4.3: Rede Neural utilizada pelo *VisionDraughts*

módulo de mapeamento mostrado na figura 4.2;

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 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 uma função de ativação tagente hiperbólica definida por $g_j(x) = \frac{2}{(1+e^{-2x})} - 1$;
4. a predição retornada pela rede neural para o estado S_t é $a_j^{(2)} = a_{m2}^{(2)} = P_t$.

Funcionalmente, predições P_t 's calculadas pela rede neural MLP podem ser vistas como uma estimativa do quão o estado S_t se aproxima de uma vitória (representada pelo retorno do valor +1 pelo ambiente), derrota (representada pelo retorno do valor -1 pelo ambiente) ou empate (representado pelo retorno do valor 0, ou próximo de 0, pelo ambiente). Assim, configurações de tabuleiros (ou estados do jogo) que receberem predições próximas de +1 tenderão a ser consideradas como bons estados de tabuleiro, resultantes de boas ações, que poderão convergir para vitória (+1). Da mesma forma, tabuleiros cujas predições estão próximas de -1 tenderão a ser considerados péssimos estados de tabuleiro, resultantes de ações ruins, que poderão convergir para derrota (-1). O mesmo vale para configurações de tabuleiros próximos de 0, que poderão convergir para empate (0 ou valor próximo deste).

O problema de escolher a melhor ação a ser executada pelo jogador *VisionDraughts* é solucionado por meio de uma rede neural artificial, utilizando o cálculo das predições e do algoritmo alfa-beta. O alfa-beta recebe como parâmetro de entrada o estado corrente do tabuleiro do jogo, expande uma árvore de busca e envia suas folhas para o módulo de mapeamento que gera a sequência binária de entrada na rede neural. O pseudo-código do algoritmo alfa-beta será apresentado na seção 4.6.1

4.4 Reajuste de Pesos da Rede Neural MLP

O reajuste dos pesos da rede é *online*, isto é, o agente vai jogando contra o seu clone (oponente) e os pesos da rede vão sendo ajustados pelo método TD(λ) de acordo com a escolha das melhores ações e os estados resultantes dessas ações. 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 sequência de ações que executou (+1 ou -1, caso o resultado tenha sido vitória ou derrota, respectivamente). Caso tenha ocorrido empate, o resultado será zero ou valor próximo disso.

O agente jogador seleciona a melhor ação M_t a ser executada a partir de um estado S_t com o auxílio do procedimento de busca alfa-beta 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 sequência 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(λ) de Sutton (SUTTON, 1988):

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

onde $\alpha^{(l)}$ é o parâmetro da taxa de aprendizagem na camada. Caixeta e Julia [9] 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 esse 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 $\vec{X}(k)$ e do vetor de pesos

$\overrightarrow{W(k)}$ da rede neural no instante temporal k . O termo λ^{t-k} , para $0 < \lambda \leq 1$, tem o papel de dar uma "pesagem exponencial" para a taxa de variação das predições calculadas a k passos anteriores de t . Quanto maior for λ , maior o impacto dos reajustes anteriores ao instante temporal t sobre a atualização dos pesos $w_{ij}^{(l)}(t)$.

Neto e Julia [35], [37], [36] descrevem o processo de reajuste de pesos por Diferenças Temporais TD(λ) nas seguintes etapas:

1. o vetor $\overrightarrow{W(k)}$ de pesos é gerado aleatoriamente;
2. as eligibilidades associadas aos pesos da rede são inicialmente nulas;
3. dadas 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 pela equação:

$$e(t) = (\gamma P^{t+1} - P^t),$$

onde o parâmetro γ 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 às 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 λ tem o papel de dar uma "pesagem exponencial" para a taxa de variação das predições calculadas a k passos anteriores de t ; $a_i^{(l)}$ o sinal de saída do neurônio i na camada l ; $g'(x) = (1 - x^2)$ representa a derivada da função de ativação (tangente hiperbólica) [29].

- para os pesos associados às 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:

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

onde o parâmetro de aprendizagem $\alpha^{(l)}$ é definido por Caixeta [9] como:

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

6. existe um problema típico associado ao uso de redes neurais no qual a convergência não é garantida para o melhor valor [35]. Caixeta [9], assim como Lynch [28], [29] utilizou o termo momento μ para tentar solucionar esse tipo de problema. Para isso, ele empregou uma checagem de direção na equação 4.1 ou seja, o termo momento μ 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(λ) utilizada para calcular o reajuste dos pesos da rede neural na camada l , para $0 \leq l < 1$, 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 $\Delta w_{ij}^{(l)}(t)$ pela 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:

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

4.5 Estratégia de Treino por *Self-Play* com Clonagem

A ideia do treinamento por *self-play* com clonagem é treinar um jogador por vários jogos contra ele mesmo, ou seja, uma cópia de si próprio. À medida que o jogador melhora seu nível de desempenho de forma a conseguir bater sua cópia, uma nova clonagem é realizada e o jogador passa a treinar contra esse novo clone. O processo se repete por um determinado número de jogos de treinamento.

Na prática, o treinamento do agente jogador *VisionDraughts* segue os passos:

1. primeiro, os pesos da rede neural MLP do jogador *opp1* (nome atribuído a rede original) são gerados aleatoriamente;

2. antes de iniciar qualquer treinamento, a rede *opp1* é clonada para gerar a rede clone *opp1-clone*;
3. inicia-se então o treinamento da rede *opp1* que joga contra a rede *opp1-clone* (oponente). 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 dois jogos é realizado para verificar qual das redes é a melhor. Caso o nível de desempenho da rede *opp1* supere o nível da rede *opp1-clone*, é realizada a cópia dos pesos da rede *opp1* para a rede *opp1-clone*. Caso contrário, não se copiam os pesos e a rede original *opp1* permanece inalterada para a próxima sessão de treinamento;
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 do jogo) seja alcançado.

Observe que essa estratégia de treinamento utilizada no *VisionDraughts* é eficiente, uma vez que o agente jogador *opp1* deve sempre procurar melhorar o seu nível de desempenho a cada sessão de n jogos de treinamento, de forma a poder bater seu clone *opp1-clone* em um torneio de dois jogos. No primeiro jogo do torneio, a rede *opp1* joga com as peças pretas do tabuleiro de damas e a rede *opp1-clone* joga com as peças vermelhas. Já no segundo jogo, as posições de ambas as redes sobre o tabuleiro de damas são invertidas. O objetivo dessa troca de posições dos jogadores sobre o tabuleiro é permitir uma melhor avaliação do desempenho das redes *opp1* e *opp1-clone*, ao jogarem entre si em ambos os lados do tabuleiro, uma vez que as características referem-se a restrições sobre peças pretas e/ou vermelhas.

4.6 Eficiente Método de Busca do *VisionDraughts*

Quando se fala que o *VisionDraughts* [9] é uma otimização do *NeuroDraughts* [28], na verdade, quer-se dizer que o *VisionDraughts* expande o *NeuroDraughts* por incluir um eficiente algoritmo de busca baseado em poda alfa-beta, tabela de transposição e aprofundamento iterativo.

A qualidade de um programa jogador depende muito do número de jogadas que ele pode olhar adiante (*look-ahead*), o que é aumentado com o uso do módulo de busca eficiente [9]. O jogador automático tem maior capacidade de analisar jogadas futuras (estados do tabuleiro mais distantes do estado corrente) e, conseqüentemente, realizar melhores jogadas.

Como confirmação do ganho obtido pelo aperfeiçoamento do módulo de busca, tem-se que: com a introdução do algoritmo alfa-beta, o *VisionDraughts* treinou um jogador

automático de damas em apenas 5,83% do tempo gasto pelo *NeuroDraughts*; além disso, usando o alfa-beta combinado com tabelas de transposição e aprofundamento iterativo, esse tempo caiu para 2,27% do tempo total gasto pelo *NeuroDraughts*.

Para verificar o impacto do módulo de busca no desempenho do *VisionDraughts*, dois torneios com 14 jogos foram executados entre o *VisionDraughts* e os jogadores automáticos de damas *NeuroDraughts* [28] e *LS-Draughts* [35]. Os resultados dos torneios comprovaram, além da otimização do tempo de busca obtida pelo alfa-beta, a eficiência do aumento do *look-ahead*:

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

Na sequência será apresentado o módulo de busca do jogador. A sequência da apresentação foi organizada de modo a facilitar o entendimento do leitor como segue: i) O algoritmo Alfa-Beta; ii) A Tabela de Transposição; iii) Integração entre o Algoritmo Alfa-Beta e a Tabela de Transposição, iv) O Aprofundamento Iterativo; v) e finalmente O Algoritmo Alfa-Beta com a Tabela de Transposição e o Aprofundamento Iterativo (peças fundamentais para o entendimento do módulo de busca eficiente).

4.6.1 O algoritmo Alfa-Beta

O jogo de damas pode ser pensado como uma árvore de possíveis estados futuros do tabuleiro do jogo, ou seja, uma árvore representando uma evolução dos próximos movimentos disponíveis de acordo com as regras do jogo. Geralmente a raiz da árvore tem vários filhos representando todas as possíveis jogadas permitidas a partir do estado corrente (raiz da árvore). A profundidade máxima de busca nessa árvore é determinada baseando-se na quantidade de recursos computacionais disponíveis. Os estados do tabuleiro presentes na camada mais profunda da árvore de busca denominam-se folhas da árvore (para simplificar a representação da árvore de busca, cada estado do tabuleiro é representado, de agora em diante, por um círculo chamado nó da árvore de busca).

O *VisionDraughts* utiliza o algoritmo alfa-beta para escolher, na árvore de busca, 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 [51], [4]. A versão mais conhecida do algoritmo alfa-beta é a versão *fail-hard*, apresentada nesta seção.

Com o objetivo de simplificar o entendimento do algoritmo alfa-beta, apresenta-se, a seguir, o pseudo-código do algoritmo minimax juntamente com a explicação de como ele

funciona. Em seguida, será adicionado o mecanismo de poda alfa-beta no contexto do procedimento minimax. Na verdade, o que acontece, de fato, é a introdução de uma poda alfa-beta (limites que guiam as podas) no minimax; por isso o entendimento do minimax faz-se necessário.

Pseudo-código do algoritmo minimax

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

- **linha 1:** o algoritmo recebe como parâmetro de entrada um estado do tabuleiro do jogo de damas n , uma profundidade de busca d e um parâmetro de saída $bestmove$ que armazena 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, a qual é a avaliação do estado n do ponto de vista do agente jogador;

- **linhas de 2 a 4:** o algoritmo minimax é recursivo e escolhe a melhor jogada a ser executada 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. Tal condição de parada do algoritmo verifica se o estado do tabuleiro n é uma folha da árvore, ou seja, se 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 5:** o estado do tabuleiro presente na raiz da árvore do jogo montada pelo algoritmo 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 à 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 6:** *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 7:** para cada um dos filhos de n , realiza-se os procedimentos das linhas 8 a 12;
- **linha 8:** 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 ;
- **linhas de 9 a 12:** caso a predição armazenada na variável v seja maior que *besteval* (melhor avaliação encontrada até o momento para n), o algoritmo atualiza o valor de *besteval* com v e o valor de *thebest* (melhor movimento encontrado até o momento para n) com *bestmove*;
- **linha 14:** assim que o algoritmo sair do laço da linha 7, *thebest* conterá o melhor movimento a ser executado a partir do estado n . Então, *bestmove* é atualizado com o valor de *thebest*;
- **linha 15:** Assim que o algoritmo sair do laço da linha 7, *besteval* conterá a maior predição de todos os filhos do estado do tabuleiro n (n é maximizador). Então, *besteval* será retornado como valor da predição associada ao estado do tabuleiro n ;
- **linha 17:** o pseudo-código delimitado por esta linha é similar ao delimitado pela linha 5, porém, tratam-se, agora, de nós minimizadores.

- **linha 18:** *besteval* representa a melhor avaliação encontrada para o nó *n* até o momento. Como *n* representa um nó minimizador, inicialmente, o valor de *besteval* é configurado como o maior valor positivo possível, por exemplo, $+ \infty$. Note que *besteval* recebe $+ \infty$ e será decrementado até o mínimo valor das predições associadas aos filhos de *n*;
- **linha 19:** para cada um dos filhos de *n*, realizam-se os procedimentos das linhas 20 a 24;
- **linha 20:** para cada um dos filhos *child*, do estado do tabuleiro *n*, o algoritmo 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*.
- **linhas de 21 a 24:** caso a predição armazenada na variável *v* seja menor que *besteval* (melhor avaliação encontrada até o momento para *n*), o algoritmo atualiza o valor de *besteval* com *v* e o valor de *thebest* (melhor movimento encontrado até o momento para *n*) com *bestmove*;
- **linha 26:** assim que o algoritmo sair do laço da linha 19, *thebest* conterá o melhor movimento a ser executado a partir do estado *n*. Então, *bestmove* é atualizado com o valor de *thebest*;
- **linha 27:** assim que o algoritmo sair do laço da linha 19, *besteval* conterá 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*.

Como citado anteriormente, o algoritmo minimax examina mais estados do tabuleiro que o necessário e, nesse 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 explicação dos pontos em que ele difere do minimax.

Pseudo-código do algoritmo alfa-beta *fail-hard*

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

```

10:     besteval:= v
11:     thebest = bestmove
12:   end if
13: end for
14: if besteval >= max then
15:   return max
16: end if
17: bestmove = thebest
18: return besteval
19: end if
20: if n is a min node then
21:   besteval := max
22:   for each child of n: do
23:     v := alphaBeta(child,d-1,min,besteval,bestmove)
24:     if v < besteval then
25:       besteval:= v
26:       thebest = bestmove
27:     end if
28:   end for
29:   if besteval <= min then
30:     return min
31:   end if
32:   bestmove = thebest
33:   return besteval
34: end if

```

- **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 8:** 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

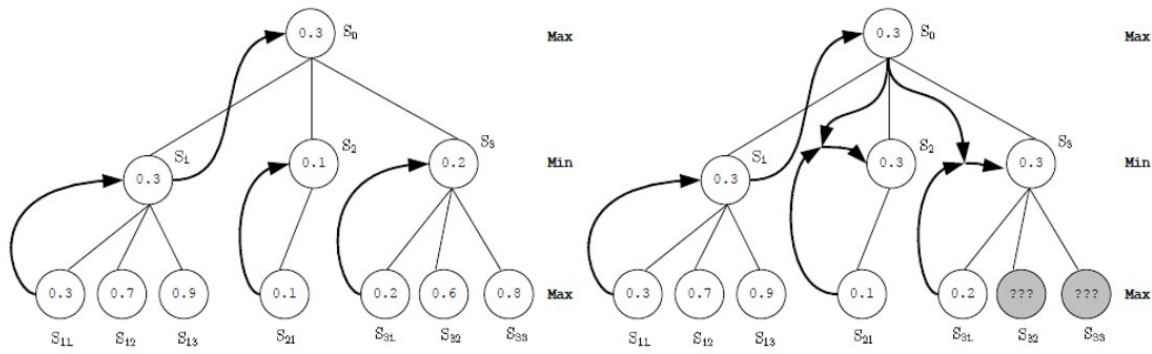


Figura 4.4: A esquerda: árvore de busca expandida pelo minimax. A direita: árvore expandida pelo Alfa-Beta.

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 9;

- **linhas de 14 a 16:** 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 limite superior do intervalo de busca (*max*) como predição associada ao estado do tabuleiro n . Tal fato expressa a ideia 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 23:** 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 24;
- **linhas de 29 a 31:** 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 ideia 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.

Para melhor esclarecer como funciona a poda alfa-beta durante um processo de busca pela melhor ação (no caso de damas, melhor movimento), observe na figura 4.4 :

Considerando o exemplo de árvore do jogo de damas mostrado na figura 4.4, o algoritmo minimax executaria a seguinte sequência de busca pelos nós da árvore, $S_1, S_{11}, S_{12}, S_{13}, S_2, S_{21}, S_3, S_{31}, S_{32}$ e por último S_{33} . Como é possível perceber, o algoritmo percorre todos os nós da árvore e, só ao final, retorna o melhor movimento, no caso S_1 . Já se tratando do algoritmo alfa-beta, a sequência de busca seria a seguinte $S_1, S_{11}, S_{12}, S_{13}, S_2, S_{21}, S_3, S_{31}$ e, nesse momento, a busca seria interrompida (porque o valor mínimo encontrado na árvore 0.2 já é menor que o valor mínimo permitido pelo intervalo de busca que é 0.3) e o melhor movimento seria retornado, assim como no minimax, a melhor ação seria se mover para S_1 .

Portanto, com base no exemplo mostrado na figura 4.4 e no pseudo-código do algoritmo alfa-beta, é possível abstrair a seguinte ideia 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).

Como citado anteriormente, o primeiro avanço do *VisionDraughts* em relação ao seu predecessor, *NeuroDraughts*, foi conseguido com o algoritmo alfa-beta, o qual diminuiu o tempo de execução do *VisionDraughts* para um valor 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 tabela de transposição. A subseção seguinte introduz o conceito de transposição e descreve o funcionamento de uma tabela de transposição.

4.6.2 A Tabela de Transposição

O algoritmo alfa-beta apresentado na seção anterior não mantém um histórico dos estados da árvore de jogo já procurados anteriormente. Assim, se um estado 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. Para evitar que o algoritmo alfa-beta seja executado duas vezes para encontrar uma mesma predição associada ao estado, pode-se associá-lo com uma tabela de transposição [6], [5].

Sabe-se que em um jogo de damas, pode-se chegar a um mesmo estado do tabuleiro 2 vezes (ou mais); quando isso ocorre, diz-se que houve uma transposição [33]. Essas transposições acontecem, em um jogo de damas, de duas formas básicas:

- diferentes combinações de jogadas com peças simples: as peças simples não se movem para trás, apesar disso, elas podem desencadear uma transposição, como mostrado na figura 4.5. 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

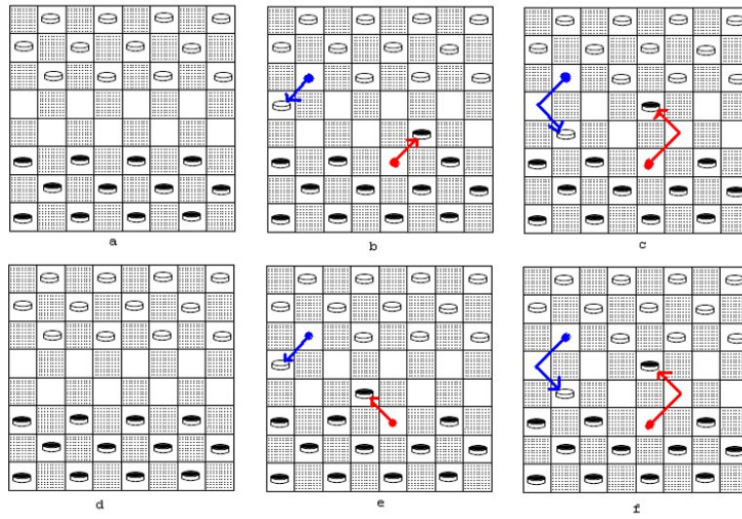


Figura 4.5: Exemplo de transposição em *c* e *f* : o mesmo estado do tabuleiro é alcançado por combinações diferentes de jogadas com peças simples.

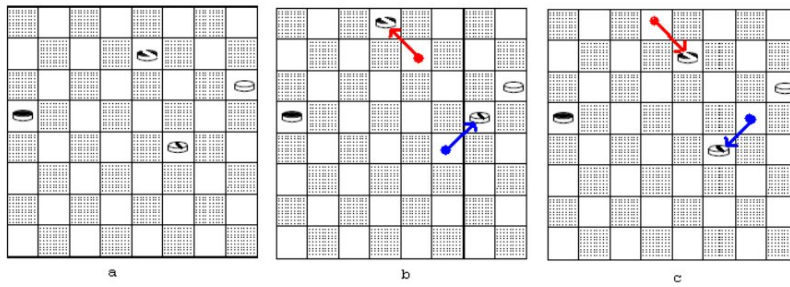


Figura 4.6: Exemplo de transposição em *a* e *c*: o mesmo estado do tabuleiro é alcançado por combinações diferentes de jogadas com reis.

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;

- diferentes combinações de jogadas com reis: os reis se movem em qualquer direção, gerando transposições facilmente, conforme mostrado na figura 4.6. 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*.

Para entender melhor como o *VisionDraughts* utiliza a tabela de transposição, as subseções, a seguir, abordam em detalhes como isso é feito. Como a compreensão da técnica de Zobrist é fundamental para a construção da tabela de transposição, inicialmente, apresenta-se a técnica de Zobrist [64]. Na sequência, é apresentada a estrutura da tabela de transposição, e como tratar possíveis colisões de estados do tabuleiro dentro da tabela.

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

A tabela de transposição utilizada pelo *VisionDraughts* foi implementada como uma tabela *hash*. Uma tabela *hash* é uma estrutura de dados que associa *chaves* a *valores* [44]. Cada *chave* representa um estado do tabuleiro do jogo de damas e é associada à informações relevantes obtidas, a partir do algoritmo alfa-beta, para aquele 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 [64].

O método descrito por Zobrist utiliza o operador **XOR** (ou exclusivo), simbolizado matematicamente por $\otimes$. 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 [9]. 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 $Seq_3 = Seq_1 \otimes Seq_2$, basta aplicar o operador XOR sobre os bits das posições correspondentes de Seq_1 e Seq_2 , isto é, basta fazer $Seq_3 = b_1 \otimes r_1, b_2 \otimes r_2, \dots, b_n \otimes r_n$;
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, os dois argumentos inteiros decimais que devem ser, antes de tudo, convertidos para a base binária. A conversão de inteiros decimais para binários é transparente em C++¹. 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 [64], para o operador XOR aplicado sobre sequências aleatórias (r) de inteiros decimais de n bits:

1. $r_i \otimes (r_j \otimes r_k) = (r_i \otimes r_j) \otimes r_k$;
2. $r_i \otimes r_j = r_j \otimes r_i$;
3. $r_i \otimes r_i = 0$;
4. se $s_i = r_1 \otimes r_2 \otimes \dots \otimes r_i$ então s_i é uma sequência 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).

¹linguagem usada na implementação do jogador

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 $\otimes$ 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* mudará 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 peça 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, criou-se um vetor de 128 elementos inteiros aleatórios, mostrado na figura 4.7, 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. Na figura 4.7, pode-se visualizar que são criadas 4 chaves para cada posição do tabuleiro, o que cobre a necessidade de variações de chaves em um jogo de damas.

A técnica de Zobrist é, provavelmente, o método disponível mais rápido para calcular uma chave *hash* associada a um estado do tabuleiro do jogo de damas [64]. Os fundamentos dessa informação é a velocidade com que se executa a operação *XOR* por uma CPU. Para entender como ocorre a atualização incremental das chaves *hash*, associadas aos estados do tabuleiro, considere as movimentações possíveis no jogo de damas:

1. 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;
2. 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;
3. 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 o exemplo de movimento simples da figura 4.8 e o vetor de números aleatórios utilizados pelo *VisionDraughts*, mostrado na figura 4.7:

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 $I_{21} = 1171196056361380757$;

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	

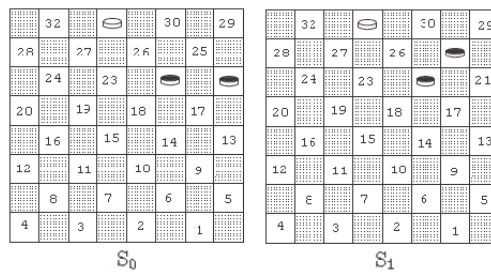
Figura 4.7: Vetor de 128 elementos inteiros aleatórios utilizados pelo *VisionDraughts*

Figura 4.8: Exemplo de movimento simples.

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 $C_0 = I_{21} \otimes I_{22} \otimes I_{31}$. Logo, $C_0 = 1171196056361380757 \otimes 9204715365712158256 \otimes 6271745259985944523$, 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. $C_1 = C_0 \otimes I_{21}$. Indica a remoção da peça preta simples da casa 21 do estado S_0 (veja a propriedade número 3). Logo, $C_1 = 4104165011015584366 \otimes 1171196056361380757$, ou seja, $C_1 = 2932970144385327611$;
7. $C_1 = C_1 \otimes I_{25}$. Isso indica a inserção da peça preta simples na casa 25 do tabuleiro, gerando o estado S_1 (veja a propriedade número 1). Logo, $C_1 = 2932970144385327611 \otimes 8978665553187022367$, ou seja, $C_1 = 8988798927364941823$.

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 estados pelo algoritmo alfa-beta. A partir de então, sempre que um estado do tabuleiro for apresentado como entrada do procedimento alfa-beta, primeiro o algoritmo verificará se esse estado se encontra 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 para armazenar os dados de entrada para a tabela de transposição. Veja:

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

```
}
```

onde o campo *hashvalue* armazena a chave *hash* mostrada na seção acima; o campo *bestvalue* armazena o valor da predição retornada pelo algoritmo alfa-beta; o campo *bestmove* armazena a melhor jogada que é, também, retornada pelo algoritmo alfa-beta; o campo *depth* armazena a profundidade de busca utilizada pelo algoritmo alfa-beta; O campo *scoretype* armazena um *flag* que indica o real significado da predição contida em *bestvalue*. Esse flag pode assumir três valores diferentes, *hashAtMost* (quando acontece uma poda alfa), *hashAtLeast* (quando acontece uma poda beta) e *hashExact* quando não acontece nenhuma poda e o valor de *bestvalue* é exato. 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.

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 (e_1 e e_2 , 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. Um exemplo de como é armazenada uma estrutura *ENTRY* na tabela pode ser visto logo abaixo:

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

A próxima seção se dedica a explicar os 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. A estrutura *TTABLE* será detalhada na seção 4.6.2, porém, é necessário entender, primeiro, o mecanismo de tratamento de colisões utilizado pelo *VisionDraughts*.

Colisões - Conflitos de Endereços para Estados do Tabuleiro do Jogo

A tabela de transposição do *VisionDraughts* trata dois tipos de erros identificados por Zobrist [64]. O primeiro tipo de erro, chamado erro *tipo 1*, ocorre quando dois estados distintos do tabuleiro do jogo de damas são mapeados na mesma chave *hash*. Caso o erro 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*. 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.

O erro *tipo 2* ocorre em virtude 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. Em seguida é apresentada a estratégia utilizada pelo *VisionDraughts* para resolver o dilema dos erros *tipo 2*.

O *VisionDraughts* utiliza dois esquemas de substituição quando acontece uma colisão do *tipo 2*, um chamado *Deep* e outro chamado *New*. Esses esquemas foram escolhidos entre os sete estudados por Breuker em [6], [5]. A escolha desses dois esquemas foi feita baseada no jogador *Chinook* [50], [48]. 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% [47].

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 teriam 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. 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. 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*, como mostrado abaixo:

- **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 desse 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;
- **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, ou seja, 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 que, utilizando os dois esquemas de substituição (*deep* e *new*), os erros tipo 2 foram totalmente controlados no *VisionDraughts*.

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* composta por dois vetores, (*e1* e *e2* com elementos do tipo *ENTRY*); um método para armazenar novas entradas na tabela; um método para ler as entradas já existentes na tabela.

Em pseudo-código a estrutura de *TTABLE* é a seguinte:

```
class TTABLE{
    int         tableSize;
    ENTRY*      e1;
    ENTRY*      e2;

    StoreEntry (...);
    GetEntry    (...);
}
```

O campo *tableSize* passa a ideia do espaço alocado na tabela de transposição para manipulação de registros do tipo *ENTRY* na memória. Importante notar que *TTABLE* utiliza tanta memória quanta tiver disponível e, claro, quanto mais memória, melhor o desempenho da tabela de transposição.

Cada par de vetor *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.

$$GetEntry(n; hashvalue; checksum; pdepth); \quad (4.4)$$

O método *GetEntry(...)* da expressão 4.4 visa recuperar informações sobre um dado estado do tabuleiro eventualmente armazenado na tabela de transposição. Para tanto, ele recebe 4 parâmetros de entrada associados ao estado, que são: *n* representa o próprio

estado; *hashvalue* representa a primeira chave *hash* associada ao estado; *checksum* representa a segunda chave *hash* associada ao estado; e *pDepth* especifica a profundidade mínima de busca desejada.

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 e_1 de E_1 . Caso exista, verifica se a primeira chave *hash* associada ao elemento de e_1 é igual a *hashvalue*. Caso afirmativo, verifica se a segunda chave *hash* associada ao elemento de e_1 é igual a *checksum*. Caso seja, verifica se a profundidade de busca associada ao elemento de e_1 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 expandido a árvore de busca do jogo. Caso não se tenha obtido êxito com o primeiro vetor e_1 , o mesmo processo é realizado, novamente, para o segundo vetor e_2 de E_1 .

$$\text{StoreEntry}(\text{newEntry}); \quad (4.5)$$

O método *StoreEntry(newEntry)*, representado na expressão 4.5, 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* e tenta armazená-lo 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 inexiste informação gravada no endereço E_1 , basta armazenar *newEntry* no primeiro vetor e_1 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 e_1 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 e_1 presente em E_1 , transfere-se o elemento presente no vetor e_1 para a mesma posição no vetor e_2 e sobrescreve-se a informação presente em e_1 com a informação de *newEntry*. Caso a profundidade de busca do elemento *newEntry* seja menor, mantém-se a informação presente em e_1 e perdem-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 e_1 de E_1 é diferente do valor *hashvalue* de *newEntry*. Nesse caso, *StoreEntry* resolve a colisão mantendo a informação de e_1 e, independentemente do conteúdo do segundo vetor e_2 , gravando o valor de *newEntry* em e_2 ;

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. Caso contrário, a árvore do jogo será expandida pela rotina de busca e uma nova entrada será gravada na tabela de transposição.

4.6.3 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 4.6.1 e na tabela de transposição da seção 4.6.2. 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, porque a resposta obtida, em alguns casos, não coincidia com a resposta 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 [53], [40].

A Variante Fail-Soft do Algoritmo Alfa-Beta

A variante *fail-soft* alfa-beta é bastante parecida com a *fail-hard*, bastando realizar alterações mínimas na versão *fail-hard* 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.

O Pseudo-código abaixo apresenta os pontos em que a versão *fail-soft* se difere da versão *fail-hard*:

Pseudo-código do algoritmo alfa-beta *fail-soft*

```

1: alfaBeta(node:n, int:depth, int:min, int:max, move:bestmove)
2: if leaf(n) or depth=0 then
3:   return evaluate(n)
4: end if
5: if n is a max node then
6:   besteval := min
7:   for each child of n: do
8:     v := alfaBeta(child,d-1,besteval,max,bestmove)
9:     if v > besteval then
10:      besteval:= v
11:      thebest = bestmove

```

```
12:   end if
13: end for
14: if besteval >= max then
15:   return besteval
16: end if
17: bestmove = thebest
18: return besteval
19: end if
20: if n is a min node then
21:   besteval := max
22:   for each child of n: do
23:     v := alphaBeta(child,d-1,min,besteval,bestmove)
24:     if v < besteval then
25:       besteval:= v
26:       thebest = bestmove
27:     end if
28:   end for
29:   if besteval <= min then
30:     return besteval
31:   end if
32:   bestmove = thebest
33:   return besteval
34: end if
```

Sendo que:

- nas **linhas de 14 a 16** 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 ideia de que a predição associada ao estado do tabuleiro n é, no mínimo, *besteval* ;
- nas **linhas de 29 a 31** 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 ideia de que a predição associada ao estado do tabuleiro n é, no máximo, *besteval*;

Com base na explicação do algoritmo *fail-soft* alfa-beta, mostrada acima, pode-se concluir que:

- na **poda alfa**: a avaliação dos filhos de um nó n de minimização pode ser interrompida tão logo uma predição P_i , calculada para um dos filhos de n , seja menor que o parâmetro alfa. Nesse caso, o valor P_i indica que o nó n possui predição igual a, no máximo, P_i ;
- na **poda beta**: a avaliação dos filhos de um nó n de maximização pode ser interrompida tão logo uma predição P_i , calculada para um dos filhos de n , seja maior que o parâmetro beta. Nesse caso, o valor P_i indica que o nó n possui predição igual a, no mínimo, P_i .

Tanto a versão *fail-hard* quanto a *fail-soft* retornam a mesma predição para um dado estado, ambas efetuam as mesmas podas na árvore de busca, porém a versão *fail-soft* armazena o valor real da predição do estado e não o valor do limite (*min* ou *max*) que efetuou a poda na busca, como é feito na versão *fail-hard*. Portanto, durante a construção do *VisionDraughts*, foi possível concluir que a versão *fail-hard* é incompatível com o uso de tabelas de transposição: caso, com a evolução do jogo, um dado estado S_i precise ser avaliado novamente, pode acontecer de um valor indevido de S_i ser resgatado da tabela de transposição. A versão *fail-soft* garante que caso o estado S_i esteja na tabela, seu valor será real, não se tratando de um valor de delimitador de busca.

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

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

Como o algoritmo alfa-beta não mantém um histórico dos estados da árvore de jogo procurados anteriormente, se um estado 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. A tabela de transposição utilizada pelo *VisionDraughts* evita a referida pesquisa em duplicidade, armazenando e recuperando dados em memória.

A estrutura *ENTRY*, mostrada na seção 4.6.2, 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:

$$\text{store}(n; \text{besteval}; \text{bestmove}; \text{depth}; \text{scoreType}); \quad (4.6)$$

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 4.6.2), o método acima já possui todas as informações para escrever na tabela de transposição, bastando obedecer o esquema de substituição descrito na seção 4.6.2. Para tanto, o método *store* invoca o método *StoreEntry* (equação 4.5).

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 F_i filhos de n tenham sido analisados (não ocorre 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 F_i de n tenha sido descartado do procedimento de busca por 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 F_i de n tenha sido descartado do procedimento de busca por 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*.

Quando acontece algum tipo de poda nos filhos de n tem-se que, a predição associada a n é igual a *hashAtMost* no caso de poda alfa, ou igual a *hashAtLeast* no caso de poda beta, como explicado anteriormente. Para saber que valor está associado a n na tabela de transposição quando ocorre alguma poda, deve-se considerar que:

1. um nó n armazenado com predição P_i e *scoreType* = *hashAtMost* indica que:
 - o nó n foi avaliado em nível de minimização da árvore de busca e, em tal avaliação, ocorreu uma poda alfa. O pai de n estava em nível de maximização e a avaliação de n provoca reajustes de incremento no limite inferior do intervalo de busca do pai de n ;
 - considerando que n tenha k filhos com predições $P_1, \dots, P_k$, então $P_i \in \{P_1, \dots, P_k\}$, isto é, P_i é a predição de um dos filhos F_i de n ;
 - a predição dos nós podados (irmãos de F_i) $\in \{P_j \mid 1 \leq j \leq k \text{ e } j \neq i\}$;
 - $P_i < \alpha$ (alfa), onde alfa era o limite inferior do intervalo de busca no momento em que foi calculada a predição P_i para n .
2. um nó n armazenado com predição P_i e *scoreType* = *hashAtLeast* indica que:
 - o nó n foi avaliado em nível de maximização da árvore de busca e, em tal avaliação, ocorreu uma poda beta. O pai de n estava em nível de minimização e

a avaliação de n provoca reajustes de decremento no limite superior do intervalo de busca do pai de n ;

- considerando que n tenha k filhos com predições $P_1, \dots, P_k$, então $P_i \in \{P_1, \dots, P_k\}$, isto é, P_i é a predição de um dos filhos F_i de n ;
- a predição dos nós podados (irmãos de F_i) $\in \{P_j \mid 1 \leq j \leq k \text{ e } j \neq i\}$;
- $P_i > \beta$ (beta), onde beta era o limite superior do intervalo de busca no momento em que foi calculada a predição P_i para n .

Nas próximas seções, quando forem encontradas as expressões *hashExact*, *hashAtMost* ou *hashAtLeast* no que diz respeito às predições dos estados de tabuleiros, considere que essas predições foram obtidas seguindo os critérios apresentados acima.

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:

$$\text{retrieve}(n; \text{besteval}; \text{bestmove}; \text{depth}; \text{nodeType}); \quad (4.7)$$

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 a predição e a melhor jogada associadas ao estado n , respectivamente, caso ocorra sucesso no procedimento de recuperação do estado n na tabela de transposição.

Assim sendo, inicialmente, o método *retrieve* aciona o procedimento de leitura *GetEntry* (4.4) para checar se o estado n está armazenado na tabela de transposição. Caso obtenha sucesso e consiga recuperar uma entrada E_1 na tabela correspondente ao estado n , tal entrada precisa ser tratada de acordo com o parâmetro nodeType , a fim de verificar se os valores constantes da tabela de transposição 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.

Considere o exemplo da figura 4.9 para verificar se os valores da tabela de transposição podem ser utilizados para preencher os parâmetros de saída besteval e bestmove levando em consideração se o pai de n é **maximizador**, sendo que:

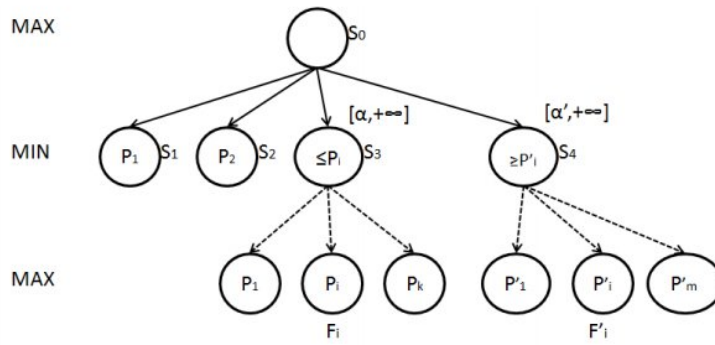


Figura 4.9: Árvore de busca onde o pai de n é maximizador

- a predição de S_1 está na tabela da transposição com valor exato de predição (isto é, *hashExact*) iguais a P_1 . S_2 está na tabela de transposição com um valor de predição *hashAtMost* P_i ;
- suponha que S_2 tenha k filhos (parte pontilhada na figura 4.9).

Quando o pai de n , representado por S_0 , é um nó maximizador, algum dos valores de predição P_1 e P_i associados aos estados S_1 e S_2 , respectivamente, fornecidos pela tabela de transposição pode ser usado?

Partindo da premissa de que a profundidade de busca associada a cada uma das predições P_1 e P_i seja igual ou maior que a profundidade de busca dos estados S_1 e S_2 pode-se concluir que:

1. o valor de P_1 pode ser usado desde que o valor *hashExact* tenha sido obtido no nível de minimização, ou seja, no mesmo nível de n . Caso tenha sido obtido em nível de maximização (nível oposto ao de n), o valor de P_1 será o oposto do valor de predição desejado pelo nível de minimização;
2. P_i somente pode ser usado se $P_i \leq \alpha$, onde α é o limite inferior do intervalo de busca com que S_2 seria expandido. Note que S_2 , sempre, será podado pelo limite de maximização de S_0 . De fato, nesse caso, nenhum filho de n com predição menor que P_i "subiria" para S_0 , pois o valor de α impediria. Analogamente, nenhum filho de n com predição maior que P_i "subiria" para S_0 pois o minimizador de S_2 impediria;
3. caso nenhum dos itens acima possam ser usados, os estados S_1 e S_2 precisam ser expandidos.

Para a análise de quando o pai de n está em nível de minimização, considere a árvore da figura 4.10:

Partindo da premissa de que a profundidade de busca associada a cada uma das predições P_1 e P_i seja igual ou maior que a profundidade de busca dos estados S_1 e S_2 , pode-se concluir que:

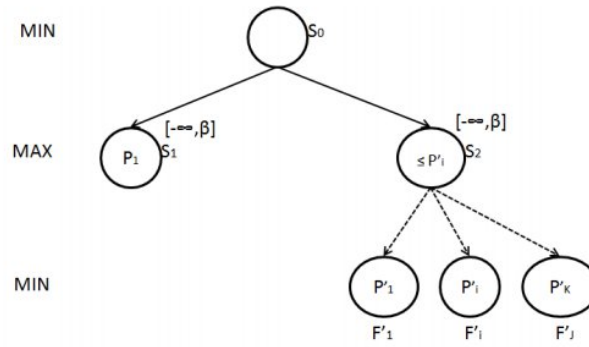


Figura 4.10: Árvore de busca onde o pai de n é minimizador

1. o valor de P_1 pode ser usado desde que o valor *hashExact* tenha sido obtido no nível de maximização, ou seja, no mesmo nível de n . Caso tenha sido obtido em nível de minimização (nível oposto ao de n), o valor de P_1 será o oposto do valor de predição desejado pelo nível de maximização;
2. se $P_i \geq \beta$, o valor associado a P_i pode ser usado, uma vez que n será descartado, pois:
 - todo irmão F_j de F_i que tiver sido podado na poda Beta que gerou F_i , e que tiver predição $P_j > P_i$ "subirá" pelo maximizador de n , mas será barrado pelo minimizador de β ;
 - todo irmão F_j de F_i que tiver sido podado na poda Beta que gerou F_i , e que tiver predição $P_j < P_i$ será descartado pelo maximizador de n .
3. se $P_i < \beta$, o valor associado a P_i não pode ser usado, pois todo irmão F_j de F_i que tiver sido podado na poda beta que gerou P_i , e que tiver predição P_j , tal que $P_i < P_j < \beta$ seria candidato a "subir" até S_0 . Inicialmente "subiria" até n caso P_j fosse maior que as predições dos filhos de n , posteriormente, "subiria" até S_0 , caso P_j fosse a menor entre as predições dos filhos de S_0 ;
4. caso nenhum dos itens acima possam ser usados os estados S_1 e S_2 precisam ser expandidos.

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

Todos os pré-requisitos para a integração entre o algoritmo alfa-beta e a tabela de transposição, no jogo de damas, foram detalhados nas seções anteriores. Para esclarecer o assunto, definitivamente, será 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, mostrados na seção 4.6.3).

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

```
1: alfaBeta(node:n, int:depth, int:min, int:max, move:bestmove)
2: if leaf(n) or depth=0 then
3:   besteval := evaluate(n)
4:   store(n, besteval,bestmove,depth,hashExact)
5:   return besteval
6: end if
7: if n is a max node then
8:   besteval := min
9:   for each child of n do
10:    if retrieve(child,besteval,bestmove,depth-1,parentIsMaxNode) then
11:      then v := besteval
12:    else
13:      v := alfabeta(child,depth-1,besteval,max,bestmove)
14:    end if
15:    if v > besteval then
16:      besteval:= v
17:      thebest = bestmove
18:    end if
19:    if besteval >= max then
20:      store(child,besteval,bestmove,depth,hashAtLeast)
21:      return besteval
22:    end if
23:  end for
24:  bestmove = thebest
25:  store(n,besteval,bestmove,depth,hashExact)
26:  return besteval
27: end if
28: if n is a min node then
29:   besteval := max
30:   for each child of n do
31:    if retrieve(child,besteval,bestmove,depth-1,parentIsMinNode) then
32:      then v := besteval
33:    else
34:      v := alfabeta(child,depth-1,besteval,max,bestmove)
35:    end if
36:    if v < besteval then
37:      besteval:= v
```

```

38:         thebest = bestmove
39:     end if
40:     if besteval <= min then
41:         store(child,besteval,depth,hashAtMost)
42:         return besteval
43:     end if
44: end for
45: bestmove = thebest
46: store(n, besteval,depth,hashExact)
47: return besteval
48: end if

```

- **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 deverá ser armazenado na tabela de transposição. Assim, o algoritmo invoca o procedimento *store* (4.6), descrito na seção 4.6.2. Como n é um nó folha, o valor da predição deve ser armazenado com *scoretype* = *hashExact*, ou seja, predição exatamente igual a *besteval*;
- **linha 10:** 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) deverá ser consultado. Então o algoritmo invoca o procedimento *retrieve* (4.7), também descrito na seção 4.6.2, 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 11:** 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* deverá ser utilizado, em vez de chamar o algoritmo alfa-beta recursivamente. Veja que o método *retrieve* (4.7) é chamado com um *flag* indicando que o estado do tabuleiro n é do tipo maximizador;
- **linha 20:** Detectada a ocorrência de uma *poda beta*, quando o algoritmo estiver analisando os filhos de um nó n do tipo maximizador, significará que os demais filhos de n , caso existam, não precisam mais ser analisados. A variável *besteval* conterà, nesta linha, o valor mínimo aceitável para o estado do tabuleiro representado pelo nó n . Assim, a predição presente na variável *besteval* deverá ser armazenada na tabela de transposição pelo método *store* (4.6) com o *flag* *hashAtLeast* indicando que a predição associada ao estado do tabuleiro n é, no mínimo, igual a *besteval* ;
- **linha 25:** quando o algoritmo tiver analisado todos os filhos de um nó n do tipo maximizador, a variável *besteval* conterà 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* (4.6) com o *flag hashExact* que indica que a predição associada ao estado do tabuleiro n é exatamente igual a *besteval* ;

- **linha 31:** 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* (4.7) é chamado com um *flag* indicando que o estado do tabuleiro n é do tipo minimizador;
- **linha 41:** 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* (4.6) com o *flag hashAtMost* indicando que a predição associada ao estado do tabuleiro n é, no máximo, igual a *besteval*;
- **linha 46:** 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* (4.6) com o *flag hashExact* indicando que a predição associada ao estado do tabuleiro n é exatamente igual a *besteval*.

O segundo avanço do *VisionDraughts* em relação ao sistema *NeuroDraughts* [28], [29] foi conseguido com o uso das tabelas de transposição em conjunto com o *fail-soft* alfa-beta. 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.

4.6.4 O Aprofundamento Iterativo

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 jogos de damas, a maioria dos programas utilizam mecanismos para delimitar o tempo máximo permitido de busca. Como o algoritmo alfa-beta realiza uma busca com profundidade fixa, não existe garantia de 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 deve ser utilizada [31].

Larry Atkin [17] introduziu a técnica de aprofundamento iterativo como um mecanismo de controle do tempo de execução durante a expansão da árvore de busca. Note que a ideia 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 encontre 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 [31].

Seguindo a ideia 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). Conforme dito anteriormente, a desvantagem do aprofundamento iterativo é o processamento repetido de estados de níveis mais rasos da árvore de busca. Para atenuar tal inconveniente, o *VisionDraughts* lança mão de repositório da tabela de transposição descrito na seção anterior.

4.6.5 O Algoritmo Alfa-Beta com a Tabela de Transposição e o Aprofundamento Iterativo

Tendo integrado o algoritmo alfa-beta com a tabela de transposição, falta somente a integração com o aprofundamento iterativo para que o eficiente módulo de busca utilizado pelo *VisionDraughts* fique completo.

A busca com aprofundamento iterativo é, normalmente, mais eficiente que uma busca alfa-beta com profundidade fixa, além disso proporciona um bom controle de tempo. A visualização da eficiência do procedimento iterativo não é óbvio e, de fato, o procedimento não é eficiente se não for utilizado em conjunto com tabelas de transposição.

Para executar o procedimento alfa-beta repetidas vezes, ou seja, introduzir aprofundamento iterativo, é necessário uma rotina de iteração. Essa rotina usará o algoritmo *fail-soft* alfa-beta com tabela de transposição, descrito acima. O pseudo-código da rotina pode ser visualizado a seguir:

Rotina de iteração

```
1: iterative(node:n, int:depth, int:min, int:max, move:bestmove)
2: besteval = alfaBeta(n, 2, min, max, bestmove)
3: start = clock()
4: for d=4 ; d ≤ MAX-SEARCH-DEPTH ; d=d+2 do
5:   besteval = alfaBeta(n, d, min, max, bestmove)
6:   stop = clock()
7:   if (stop - start) > MAX-EXECUTION-TIME then
8:     return besteval
9:   end if
10: end for
11: return besteval
```

Abaixo segue a explicação em detalhes do como tal rotina funciona:

- **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 11:** retorna o melhor movimento *besteval* após a profundidade máxima de busca ter sido atingida.

O terceiro 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 [35], [37], [36] no trabalho de Lynch [28]. O problema do *loop* representa a 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, não evoluindo a partida. 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.

Ao leitor que desejar conhecer melhor todos os detalhes do jogador *VisionDraughts* e *NeuroDraughts*, é sugerido a leitura dos trabalhos de Caixeta e Julia [9], [10] e Lynch [28], [29]

Capítulo 5

Busca Paralela

A busca paralela é uma forma de aumentar a velocidade da busca através do uso de múltiplos processadores. Este tópico vem ganhando atenção recentemente devido a popularização de máquinas com múltiplos processadores [39]. O uso de busca paralela permite realizar buscas mais profundas que, em jogos, significam aumentar a visão (*look-ahead*) do jogador. Jogadores com um campo de visão maior têm mais condições de fazer melhores escolhas e, conseqüentemente, de jogarem em um nível superior ao que jogadores com uma visão menor do espaço de estados.

Como regra, quanto mais profunda for a busca, melhor é a qualidade da jogada, pois reduz-se o erro introduzido pela função de avaliação. Para se obter um nível de profundidade extra no jogo de damas, por exemplo, geralmente é requerido o aumento da velocidade da busca por um fator de duas vezes [30] (no xadrez este fator varia entre quatro e oito vezes).

O uso de processadores adicionais para a busca é um domínio interessante de pesquisa, uma vez que percorrer uma árvore de busca é um problema inerentemente serial. Muitas abordagens foram criadas e algumas delas serão analisadas nas próximas seções.

5.1 Terminologia

Antes de apresentar os algoritmos paralelos é importante introduzir alguns termos comuns que são usados no campo do processamento paralelo.

5.1.1 Medidas de Desempenho

Uma medida de desempenho comum de algoritmos paralelos para busca em árvore de jogos, e de computação paralela em geral, é a aceleração (*speedup*). Intuitivamente, o *speedup* refere-se a quão mais rápido um problema é resolvido pelo uso de n processadores ao invés de apenas um, ou seja, por um processamento sequencial. Se o uso de n processadores leva ao *speedup* de n vezes, então temos um *speedup* linear (ou *speedup* ideal).

Speedup S é definido como:

$$S = T_s/T_p \quad (5.1)$$

onde T_p é o tempo de execução em n processadores e T_s é tempo de execução do mesmo algoritmo em um processador.

O *speedup* não é normalizado para o número de processadores utilizados. Dessa forma, temos um conceito derivado do *speedup*, que é a eficiência. A eficiência representa quanto do potencial de speedup que foi efetivamente alcançado, ou o quão bem o sistema foi utilizado. Em sistemas paralelos ideais, o speedup é igual ao número de processadores e a eficiência é igual a um. Na prática, o speedup é menor que o número de processadores e a eficiência fica entre zero e um. De um modo geral, pode-se dizer que a eficiência mede a qualidade do algoritmo paralelo. A eficiência E , quando se executa o algoritmo em n processadores, é definida como:

$$E = S/n \quad (5.2)$$

Um comportamento não muito comum em algoritmos paralelos, mas que costuma ocorrer ao se paralelizar algoritmos do tipo *branch-and-bound* como o Alpha-Beta, é a obtenção de valores de *speedups* maiores que o número de processadores utilizados (ou também uma eficiência maior que um). Este comportamento é chamado de *speedup superlinear* e foi estudado por muitos autores [22], [23], [25].

5.1.2 Sobrecargas (*overheads*) de um algoritmo de busca em árvore de jogo paralelo

A busca paralela em árvore de jogos não seria um campo de pesquisa tão ativo caso existissem algoritmos com uma eficiência quase perfeita (ou seja, próxima de 1). Desta forma, deve ser entendido de onde vêm as ineficiências da busca paralela. Podemos destacar principalmente três fontes de ineficiência [8]:

- *Overhead* de busca: Se refere aos nós explorados pelo algoritmo paralelo que não são explorados pelo algoritmo serial. Este overhead ocorre primariamente pela carência de informação pelos processadores. A falta de informação relacionado a uma poda ou o uso de uma janela de busca ampla fará com que o processador explore nós irrelevantes. Em algoritmos do tipo *branch-and-bound* como o alpha-beta, a informação obtida na busca de um ramo da árvore pode causar uma poda em outros ramos. Assim, se os sucessores de um nó são expandidos paralelamente, um nó pode não tirar vantagem de toda informação que estaria disponível na busca serial, resultando em busca desperdiçada.
- *Overhead* de sincronização: Este overhead ocorre quando um processador é forçado a esperar por informações providas de outros processadores. Existem estratégias

para reduzir tanto o overhead de busca quanto de sincronização. Porém, é um problema difícil, pois há uma estreita ligação entre os dois overheads e compensações devem ser feitas com cuidado.

- *Overhead* de comunicação: É causado pela troca de informações entre os processadores e geralmente envolve o envio e recebimento de resultados entre processadores. Obviamente, no algoritmo serial, não há troca de informações entre processadores. Por conveniência, tudo o que não é atribuído aos overheads de busca ou sincronização são definidos como overhead de comunicação.

5.2 Algoritmos de busca paralelos baseados no Alpha-Beta

O algoritmo Alpha-Beta possui uma natureza intrinsecamente serial; a árvore é percorrida em uma certa ordem e o conhecimento sobre os limites no qual o valor real se encontra são propagados sequencialmente através da busca.

Algoritmos eficientes para busca sequencial de árvore de jogos existem há um bom tempo (desde 1963 [20]). Em uma árvore onde os melhores movimentos são avaliados antes de movimentos ruins, um algoritmo de busca sequencial eficiente examinará apenas uma fração da árvore toda. Em sistemas paralelos, algoritmos sequenciais são estendidos para permitir que vários processadores compartilhem o esforço da busca. Tentativas iniciais de busca paralela mostraram um sucesso limitado quando executadas em um grande número de processadores [7] porém esforços recentes demonstraram que o uso de um grande número de processadores pode ser eficiente [7].

O algoritmo poda alpha-beta é uma especialização de algoritmos do tipo branch-and-bound e é baseado em uma propriedade altamente sequencial de busca que depende do conhecimento prévio para evitar busca em partes da árvore de jogos que não têm influência no resultado final. Embora vários algoritmos paralelos são encontrados na literatura [7], apenas quatro foram considerados para o uso no D-Vision Draughts, escolhidos pela sua relevância histórica e resultados obtidos.

O algoritmo conhecido como Principal Variation Splitting (PVS) foi resultado das primeiras tentativas de paralelizar o alpha-beta e foi fonte de inspiração para vários dos métodos modernos. O PVS classifica cada nó da árvore de busca de acordo com a taxonomia criada por Knuth e Moore [20] (ver figura 5.1.

- PVNodes: O nó raiz e todo primeiro sucessor de nós PVNodes
- CUTNodes: Todos os filhos de PVNodes, exceto o primeiro, e todos os filhos de ALLnodes
- ALLNodes: O primeiro filho de CUTNodes

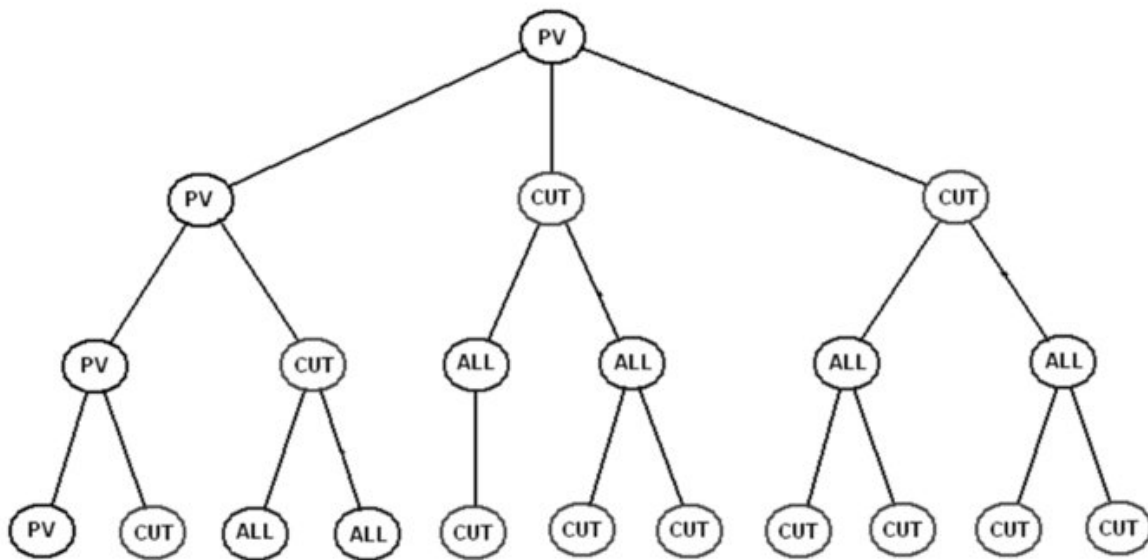


Figura 5.1: Tipos de nós em uma árvore de jogo.

- UndefinedNodes: Qualquer nó não definido acima.

Esta taxonomia é usada para explicar onde o paralelismo ocorre na árvore de jogo em muitos dos algoritmos paralelos. No caso do PVS, o paralelismo ocorre apenas em nós PVNodes. Quando um PVNode termina a avaliação de seu primeiro sucessor, todos os demais nós podem ser expandidos em paralelo (ver figura 5.2). Apesar de sua importância histórica, PVS tem uma baixa eficiência quando testado em ambientes com um grande número de processadores, mesmo com várias melhorias propostas [30].

O Young Brothers Wait Concept (YBWC) e o Dynamic Tree Splitting (DTS) são propostas mais recentes e grandes *speedups* têm sido reportados [7]. Distintamente do PVS, o YBWC e DTS, no paralelismo pode ocorrer em qualquer tipo de nós, não apenas nos nós PVNodes como no PVS.

A ideia principal no YBWC é expandir primeiro o sucessor mais à esquerda, referido como o 'irmão mais velho', e, após a obtenção de uma janela com sua expansão, pode-se expandir seus irmãos em paralelo. Em outras palavras, um ponto de divisão na árvore de jogos é um nó cujo valor do primeiro sucessor é conhecido. Este algoritmo é mais eficiente em árvores de jogo ordenadas pois, nestas árvores, o primeiro sucessor tem maiores chances de causar uma poda (que, no caso o processamento dos irmãos, seria um esforço desperdiçado), ou retornar uma janela mais apropriadas para a expansão dos seus irmãos. Se o primeiro movimento não produz uma poda, então os movimentos restantes podem ser expandidos paralelamente. Este algoritmo utiliza o modelo mestre-escravo de paralelismo, onde um processador que envia uma tarefa se torna o mestre do processador que recebe a tarefa. No YBWC, todo nó tem um processador responsável pela sua avaliação e pelo retorno das informações ao seu pai. Note que isto envolve comunicação, caso o

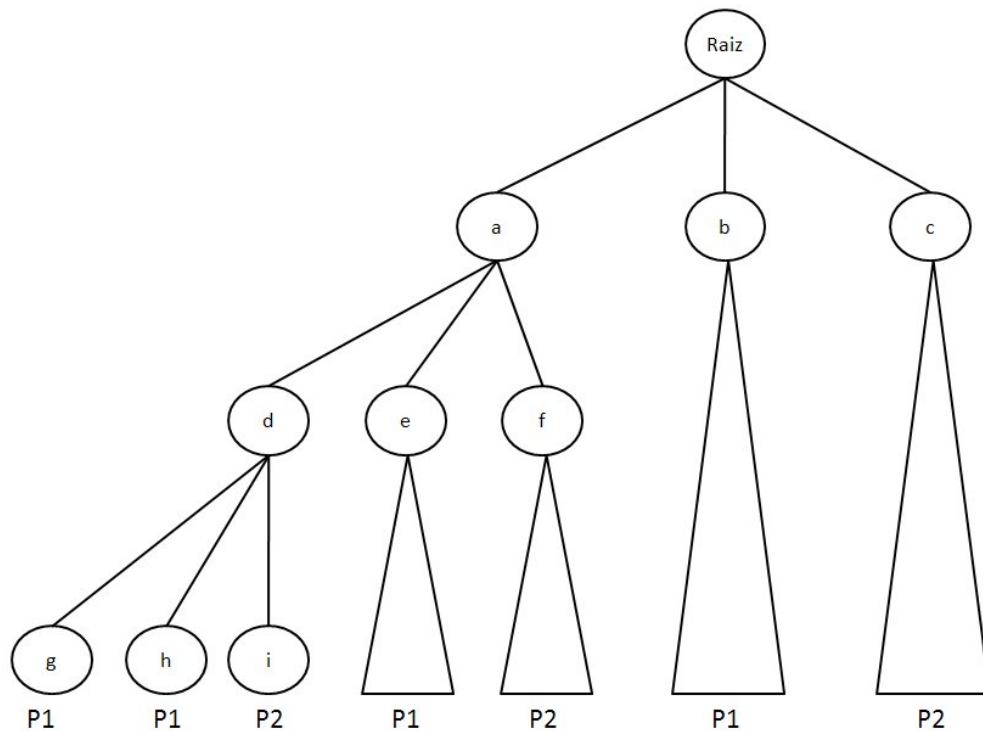


Figura 5.2: Exploração paralela de uma árvore com 2 processadores pelo PVS

processador responsável pela avaliação do nó seja diferente do processador responsável pelo seu pai. Uma vez que um nó é atribuído a um processador, a responsabilidade por este nó não pode ser transferida para outro processador.

O funcionamento do algoritmo se dá da seguinte forma: Inicialmente, a raiz da árvore de busca é atribuída a um processador P_0 , que começa a executar a procura utilizando o algoritmo Alpha-Beta como se o fizesse sequencialmente. Ao mesmo tempo, os outros processadores (todos ociosos) enviam pedidos de trabalho para outros processadores aleatoriamente escolhidos (esta estratégia de balanceamento de cargas é conhecida como Work-Stealing). Quando um processador P_i ao qual já foi alocado trabalho recebe um desses pedidos, ele verifica se existem partes por explorar na sua árvore de procura, que estejam prontas para serem avaliadas. Se não for possível, P_i responde que não tem trabalho para transferir; caso contrário, responde ao processador que faz o pedido (P_j) com o nó correspondente à raiz da sub-árvore inexplorada. A partir deste momento, P_i passa a ser um mestre de P_j e este último, o escravo, inicia uma procura sequencial. Os processadores podem ser mestres e escravos simultaneamente, podendo estas relações variarem dinamicamente ao longo das computações. Quando P_j termina o seu trabalho (possivelmente com o auxílio de outros processadores), ele envia uma notificação a P_i . A relação mestre-escravo é desfeita e P_j torna-se ocioso, voltando a enviar pedidos de trabalho para a rede. Este algoritmo é descrito em maior detalhe no capítulo 6.

O DTS tem muitas similaridades com o YBWC, diferindo principalmente por ser projetado para rodar em uma arquitetura de memória compartilhada. Como resultado,

problemas que os desenvolvedores enfrentam em ambientes de memória distribuída (como a compartilharão de informações da tabela de transposição) não são problemas para o DTS. Outra diferença está no fato de que a responsabilidade por um nó tem um sentido diferente para o DTS. Enquanto vários processadores podem trabalhar na conclusão de um nó, aquele que concluir sua avaliação é responsável por retornar o resultado para seu pai. Por ter sido desenvolvido para arquiteturas com memória compartilhada, experimentos com grande número de processadores não estão disponíveis, visto que, máquinas com um grande número processadores e com memória compartilhada, são difíceis de serem obtidas [30]. No caso de processamento em um ambiente distribuído, a hierarquia de processadores categoriza os algoritmos com base no rigor da árvore de processadores. Se a árvore dos processador é estática, um ou mais processadores são designados como mestres, e controla os outros processadores escravos. Esta hierarquia permanece fixa durante toda a busca na árvore de jogos. Uma árvore de processadores dinâmica muda baseado na distribuição de processadores ocupados e ociosos. O *controle da distribuição* descreve se o controle do algoritmo é centralizado em um número pequeno de mestres, ou é distribuído entre todos os processadores. Tais conceitos são explorados na tabela 5.1.

O APHID (Asynchronous Parallel Hierarchical Iterative Deepening), proposto por Brockington [8], é um algoritmo mais recente que o YBWC e DTS e representa uma ruptura com as abordagens utilizadas antes dele. A grande maioria das abordagens para busca usam métodos síncronos, onde o trabalho é concentrado em uma parte específica da árvore ou em uma determinada profundidade de busca. Distintamente dessas abordagens, APHID utiliza uma arquitetura mestre-escravo para realizar a busca. Um dos processadores é considerado o mestre, procura as primeiras d' jogadas e atribui estatisticamente aos escravos restantes a responsabilidade de procurar iterativamente as $d - d'$ jogadas, onde d é a profundidade de busca. Cada escravo consegue saber, utilizando uma tabela compartilhada, qual é o trabalho realizado pelos restantes e quais os melhores valores encontrados até o momento. Esta informação permite-lhe decidir (sem intervenção do mestre) qual a janela de busca utilizar e qual dos ramos atribuídos pelo mestre possui maior profundidade. Se durante a execução do algoritmo um dos escravos ficar sem trabalho, ele pode tomar a iniciativa de começar uma nova iteração de maior profundidade (uma vez que a busca segue a estratégia de aprofundamento iterativo). Para evitar problemas de balanceamento de carga causado por árvores fortemente ordenadas, o mestre, quando ocioso, pode auxiliar processando nós nos ramos mais à esquerda em uma profundidade $d' + x$, onde x é um número arbitrário entre d' e d . Com este funcionamento é possível reduzir bastante os overheads de comunicação e sincronização entre os vários processadores. O autor do APHID o disponibiliza através de uma biblioteca na qual, como declarado pelo autores, oferece fácil integração com jogadores existentes que usam busca serial, sendo testado com sucesso em vários jogadores inclusive no Chinook.

A tabela 5.1 resume as características dos algoritmos apresentados e introduz, na

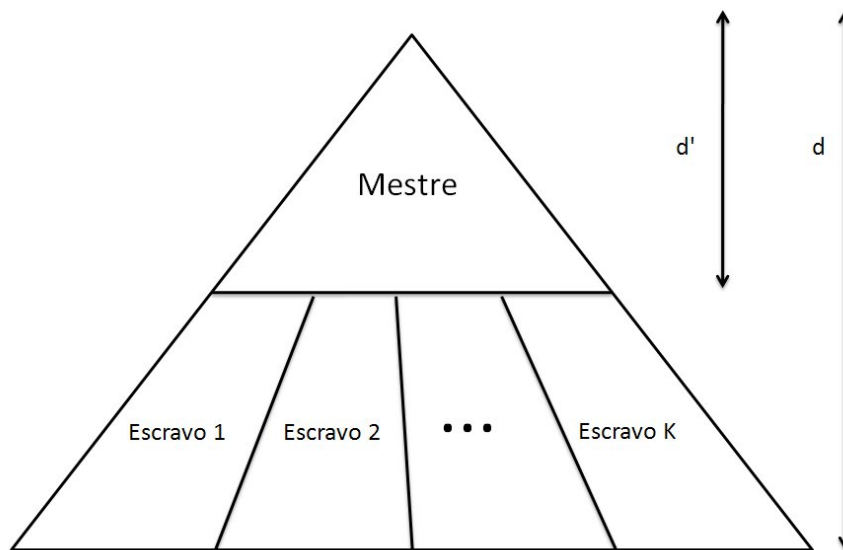


Figura 5.3: Exploração paralela de uma árvore pelo APHID

segunda coluna, tanto a hierarquia dos processadores, quanto o controle de distribuição dos algoritmos.

Tabela 5.1: Comparação algoritmos paralelos

Algoritmo	Hierarquia dos Processadores / Controle de Distribuição	Paralelismo é possível nestes nós	Sincronização é feita nestes nós
PVS	Estático / Centralizado	PVNodes	PVNodes
YBWC	Dinâmico / Distribuído	PVNodes + CUTNodes + ALLNodes	PVNodes + CUTNodes
DTS	Dinâmico / Distribuído	PVNodes + CUTNodes + ALLNodes	PVNodes + CUTNodes
APHID	Estático / Centralizado	Primeiras k-jogadas	Nenhum

Capítulo 6

D-VisionDraughts: um sistema de aprendizado de damas para ambientes de processamento de alto desempenho

Neste capítulo, é apresentado o D-VisionDraughts, um sistema de aprendizado de jogos de damas para ambientes de processamento distribuído que tem como principal objetivo construir um agente automático capaz de jogar damas com alto nível de eficiência ao longo de um jogo. A proposta para este jogador é que, no seu processo de aprendizagem, o mesmo aprenda a jogar damas com o mínimo de intervenção humana possível através do uso de técnicas eficientes de inteligência artificial.

Para conseguir tal objetivo, o D-VisionDraughts introduz duas novas técnicas em relação ao seu predecessor, o VisionDraughts: um módulo de busca distribuída baseada no algoritmo YBWC e o uso de heurísticas para a ordenação da árvore de jogo. As próximas seções apresentam a integração destas técnicas com o D-VisionDraughts

6.1 A distribuição da busca no D-VisionDraughts

Uma abordagem distribuída que obteve grande êxito na distribuição do Alpha-Beta, e que é utilizada neste trabalho, é o algoritmo YBWC. Entre todas as alternativas estudadas, o YBWC foi o que melhor atendeu as necessidades do projeto, por ter um conceito simples (diferente do APHID), ser adequado à arquitetura disponível para os experimentos (máquinas com memória distribuída, o que elimina a possibilidade de utilizarmos o DTS) e por ter bons resultados obtidos na prática [7].

O algoritmo desenvolvido trabalha com um número variável m de processadores e se baseia na decomposição dinâmica da árvore de jogos a ser avaliada, bem como na avaliação paralela dos subproblemas resultantes, pelos m processadores disponíveis. Conforme resumido na tabela 5.1, no YBWC a hierarquia dos processadores é *dinâmica*; o controle

de distribuição é *distribuído*; o paralelismo pode ocorrer nos nós *PVNodes*, *CUTNodes* e *ALLNodes*; finalmente a sincronização pode ser feita nos nós *CUTNodes* e *PVNodes*.

Cada processador é identificado por um número inteiro i e denotado como P_i (com $0 \leq i < m$). Um destes processadores, o P_0 , é chamado de *ProcessadorPrincipal* e trabalha em todos os módulos do jogador apresentados na figura 4.2. Os demais processadores P_i (com $0 < i < m$) são chamados de *ProcessadoresAuxiliares* e irão trabalhar apenas na etapa de busca, ficando no estado inativo enquanto o processador P_0 estiver nas demais etapas.

A exploração da árvore de jogos é feita por um algoritmo de busca em profundidade com aprofundamento iterativo. Uma pilha é uma estrutura de dados eficiente para representar o espaço de estados a ser explorado neste tipo de busca. A profundidade p da pilha corresponde à profundidade p do nó sendo explorado e cada processador possui sua própria pilha local de estados. As informações sobre os nós expandidos que são armazenadas na pilha correspondem ao caminho na árvore de busca do estado inicial ao estado corrente. Tal caminho é definido como a *variação corrente*. Em um algoritmo serial todos os nós à esquerda da variação corrente já foram avaliados, ao passo que os nós à direita desta variação ainda não foram avaliados. Já em algoritmos paralelos, pode haver nós à esquerda ainda em avaliação e nós à direita já avaliados. A figura 6.1a ilustra este conceito com um exemplo de uma árvore sendo explorada e com a variação corrente destacada. Note a correspondência entre cada posição p da pilha de estados com a profundidade p da árvore de busca (conforme figura 6.1b). Cada posição da pilha tem uma estrutura definida nos moldes do pseudocódigo 1:

Pseudocódigo 1 Estrutura StackEntry

```
struct StackEntry{
    float alpha,
    float beta,
    float low,
    float high,
    Board v,
    int* slaves,
    Successors* successors
}
```

Onde, no pseudocódigo 1:

- *alpha* e *beta*: valores da janela de busca em que a busca foi iniciada;
- *low*: limite inferior da busca, ou seja, o maior valor dos sucessores avaliados, em um nível de maximização;
- *high*: limite superior da busca, ou seja, o menor valor dos sucessores avaliados, em um nível de minimização;

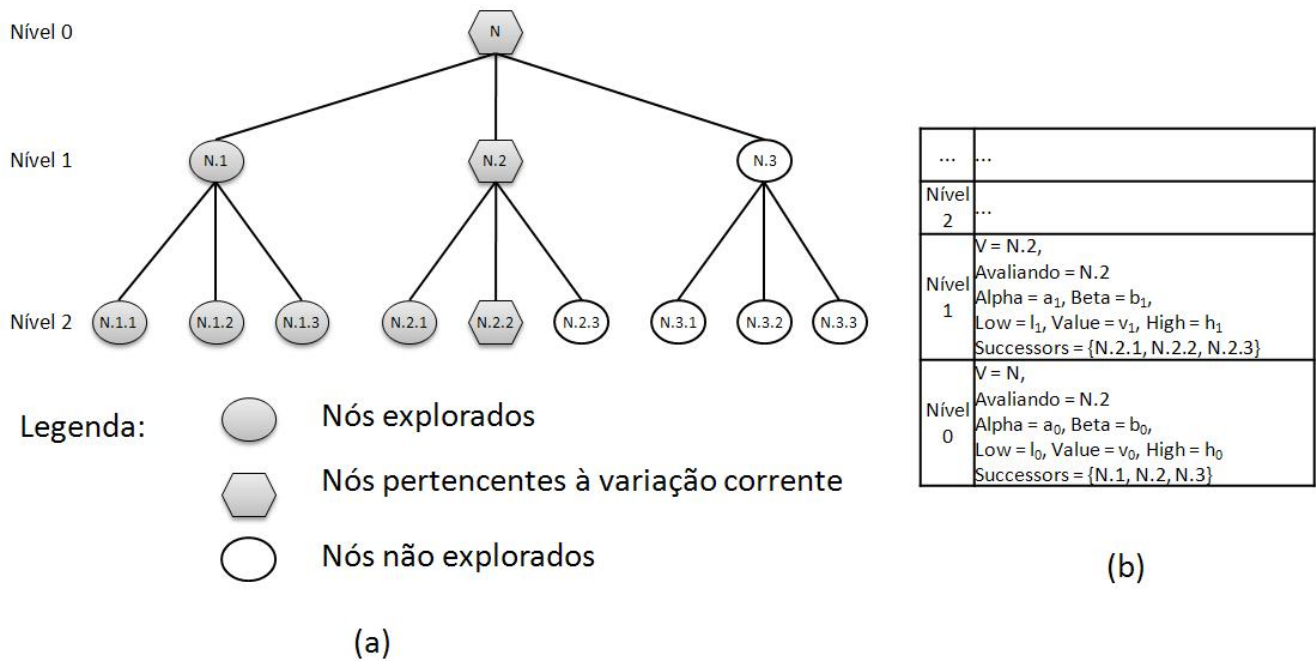


Figura 6.1: (a) Exemplo de uma Variação Corrente (b) Exemplo de uma pilha local de estados

- v : o nó (ou estado) corrente;
- *slaves*: lista de escravos. Sempre que um sucessor de v é enviado para avaliação remota em um processador P_i , P_i é adicionado a esta lista;
- *successors*: os sucessores do nó v . Os n sucessores do nó v serão referenciados durante esta seção como $v.1$ a $v.n$ e possuem a estrutura definida no pseudocódigo 2.

Pseudocódigo 2 Estrutura Successors

```
struct Successors {
    Move move,
    Status status,
    float value
}
```

Onde, no pseudocódigo 2:

- *move*: movimento que se deve aplicar ao nó v para obter este sucessor;
- *value*: valor do nó após ser avaliado;
- *status*: o status de um sucessor. Pode assumir os valores listados no pseudocódigo 3:

Onde, no pseudocódigo 3

Pseudocódigo 3 Status de um nó

```
enum Status{
    AVAILABLE,
    LOCAL_EVALUATION,
    REMOTE_EVALUATION,
    SOLVED
}
```

- *AVAILABLE*: Nó não avaliado e disponível para avaliação local ou remota;
- *LOCAL_EVALUATION*: Nó que está sendo avaliado localmente;
- *REMOTE_EVALUATION*: Nó que está sendo avaliado remotamente;
- *SOLVED*: Nó avaliado.

Um nó da árvore é dito *completo* (consequentemente possui status *SOLVED*) em um dos seguintes três casos: quando ele representa um estado terminal (não possui sucessores ou atingiu profundidade máxima na árvore de busca) e teve seu valor computado pela rede neural; quando todos os seus sucessores são completos; ou quando houve uma poda durante sua expansão.

Para todos os pseudocódigos que serão apresentados nesta seção, as seguintes variáveis estarão disponíveis:

- *P*: número atribuído ao processador que está executando o código;
- *NP*: número total de processadores disponíveis;
- $V_r, \dots, V_t$: A variação corrente sendo expandida, iniciada no nível r (com a expansão do nó V_r) e, no instante corrente, avaliando o nível t (estando em curso a avaliação do nó V_t). Note que apenas no ProcessadorPrincipal podemos ter $r = 0$ (processando a raiz da árvore) e o valor de t nunca deve ultrapassar a profundidade máxima;
- α_i, β_i : Janela de busca com que se inicia a avaliação de um nó arbitrário V_i da variação corrente (logo, $r \leq i \leq t$);
- Low_i : variável que guarda as atualizações do valor do nível inferior da janela de busca de um nó V_i da variação corrente que pertença a um nível de maximização;
- $High_i$: variável que guarda as atualizações do valor do nível superior da janela de busca de um nó V_i da variação corrente que pertença a um nível de minimização;
- $Sucessors_i$: Sucessores de um nó arbitrário V_i da variação corrente;
- $Slaves_i$: Lista de slaves de um nó arbitrário V_i da variação corrente.

No início da etapa de busca, o estado de tabuleiro a ser avaliado é colocado na pilha de estados do *ProcessadorPrincipal* como a raiz da árvore de busca e uma mensagem **InitSearch** é enviada para todos os *ProcessadoresAuxiliares* que passam do estado inativo

para ocioso. Um processador que recebe um subproblema tem a responsabilidade de avaliá-lo corretamente executando o algoritmo Alpha-Beta. Durante esta execução, novos subproblemas são gerados e podem ser enviados a outros processadores para uma avaliação paralela, desde que um dado conjunto de restrições (a ser apresentado oportunamente) seja satisfeito.

Um processador ocioso inicia sua busca escolhendo aleatoriamente outro processador dentre todos os disponíveis, e enviando-lhe uma mensagem **RequestTask**. Como mostrado no pseudocódigo 4, sorteia-se o número do processador para o qual será feita a solicitação (linha 2), não deixando que o próprio processador requisitante seja escolhido.

Pseudocódigo 4 Solicitação de Tarefa por um processador ocioso.

```

1. def request_task():
2.     pNumber = rand({0, ..., NP} - {P})
3.     envia mensagem RequestTask para processador 'pNumber'
```

Um processador P_j que recebe uma requisição de tarefa de um processador P_k , procura algum nó em sua pilha de estados que esteja disponível. Defini-se um nó disponível como: Seja v_i um nó da variação corrente (onde i representa a profundidade do nó na árvore de busca de P_j). Um nó $v_i.m \in \text{Sucessores}(v_i)$ é dito disponível se as seguintes restrições forem atendidas:

- $k > 1$, pois, conforme explicado na seção 5.2, na avaliação de v_i , a distribuição de tarefas somente pode ocorrer após a avaliação do ramo mais a esquerda de v_i .
- $v_i.1$ tem status *SOLVED* (ver explicação acima);
- $v_i.m$ não foi e nem está sendo avaliado por nenhum processador.

O processo é descrito pelo pseudocódigo 5. O processador P_j percorre os nós de sua variação corrente, partindo da maior profundidade para a menor (linha 5) e, para cada um desses nós, verifica-se se há algum nó sucessor disponível. Se um nó disponível $v_i.m$ for encontrado, ele é enviado para P_k e este último processador é adicionado à lista de escravos da pilha local de estados de P_j , na mesma profundidade i correspondente à profundidade de v_i na árvore de busca de P_j (linha 11). Se não há nós disponíveis no momento da requisição, a mensagem enviada como resposta (chamada **ResponseTask**) informa a ausência de nós disponíveis.

Um processador P_k , que enviou uma requisição de tarefa para P_j , deve aguardar até receber uma mensagem **ResponseTask** de P_j . Quando a mensagem **ResponseTask** contém um subproblema $v_i.m$ a ser transferido, tal problema é enviado junto com as seguintes informações da pilha de P_j : $\alpha_i, \beta_i, low_i, high_i$ e $value_i$, além do próprio i que se refere ao nível em que ele se encontrava no processador requisitado (no caso, P_j). O processador escravo P_k , ao receber a tarefa, inicia a busca na profundidade i

em que a tarefa se encontrava em P_j . A relação mestre-escravo é terminada através da mensagem **ResponseResult**, emitida por P_k a P_j logo após ter concluído o processamento do subproblema recebido, comunicando-lhe o resultado do subproblema.

Pseudocódigo 5 Rotina que trata mensagens de solicitação de tarefa recebidas

```

1. def request_task_handler(ps):
2.     #ps: numero do processador requisitante
3.     found = false
4.     i = 0
5.     for i = t to r:
6.         if existe sucessor disponível de  $v_i$ :
7.             found = true;
8.             break;
9.     if found:
10.        envia mensagem ResponseTask com o nó disponível encontrado
           para processador 'ps'
11.        slavesi.add(ps)
12.    else:
13.        envia mensagem ResponseTask para 'ps' informando que
           não há nós disponíveis

```

O tratamento que um mestre aplica ao resultado da avaliação de um nó $v_i.m$ processado por um escravo P_k é mostrado no pseudocódigo 6. Primeiramente, o processador P_k que retornou o resultado é removido da lista de escravos da profundidade i da pilha de estados de P_j (linha 5). Se o nó $v_i.m$ estiver em um nível de maximização, devemos atualizar o valor da variável *low* (linha 13) além de verificar se houve uma poda (linha 9) ou estreitamento da janela (linha 11). Se o nó $v_i.m$ estiver em um nível de minimização, devemos atualizar o valor da variável *high* (linha 27) além de verificar se houve uma poda (linha 23) ou estreitamento da janela (linha 25). Em ambos os casos, o motivo desta verificação é permitir que o mestre propague informações relevantes aos seus escravos (caso haja), de tal modo que:

- eles possam estreitar sua janela de busca caso recebam do mestre uma mensagem **Newbound** (linha 18), evitando assim processamento desnecessário de nós, decorrentes de uma janela desatualizada; ou
- eles abortem seu processamento caso recebam do mestre uma mensagem de poda **Cutoff** (linha 15), pois os resultados de suas avaliações não teriam influência no resultado do processamento do mestre.

Como a variação corrente pode estar em uma profundidade t maior do que a profundidade i do nó $v_i.m$ cujo processamento foi concluído por P_k , ou seja, $t \geq i$, podas ou atualizações de janela devem ser tratadas também nos níveis intermediários entre i e t da variação corrente. A ocorrência de uma poda implica em abortar todo o processamento

do nível i ao nível corrente, bem como, através da mensagem **Cutoff**, o processamento de todos os escravos situados nestes níveis intermediários, tal como ilustrado no exemplo 6.2. Nas linhas 15 e 29 há a chamada da rotina *send_cutoff* que, como mostrado no pseudocódigo 7, aborta todo o processamento dos escravos dos nós compreendidos entre os níveis i e t da variação corrente. A atualização de janela, por sua vez, deve ser comunicada a todos os escravos do nó de nível i da variação corrente (nó v_i da figura 6.2) - através da rotina *send_newbound* apresentada no pseudocódigo 8 - e propagada até a profundidade t da variação corrente, bem como a seus escravos, pela função *update_window_max*, caso o nível i seja de maximização, ou *update_window_min*, caso o nível i seja de minimização.

Pseudocódigo 6 Rotina que trata mensagens ResponseResult recebidas

```

1. def response_result_handler( $v_i.m$ , value, rp):
2.     # $v_i.m$ : nó avaliado remotamente no nível  $i$ 
3.     #value: resultado da avaliacao do nó  $v_i.m$ 
4.     #rp: número do processador slave que está retornando o resultado
5.     slaves $_i$ .remove(rp)
6.     if  $v_i.m$  is max-node:
7.         cutoff = false
8.         improvement = false
9.         if value >= beta $_i$ :
10.             cutoff = true
11.             if value > low $_i$ :
12.                 improvement = true
13.                 low $_i$  = max(low $_i$ , value)
14.             if cutoff:
15.                 send_cutoff(i)
16.         else:
17.             if improvement:
18.                 send_newbound(i)
19.                 update_window_max(i+1, low $_i$ )
20.     if  $v_i.m$  is min-node:
21.         cutoff = false
22.         improvement = false
23.         if value <= alpha $_i$ :
24.             cutoff = true
25.             if value < high $_i$ :
26.                 improvement = true
27.                 high $_i$  = min(high $_i$ , value)
28.             if cutoff:
29.                 send_cutoff(i)
30.         else:
31.             if improvement:
32.                 send_newbound(i)
33.                 update_window_min(i+1, high $_i$ )

```

Note que, no pseudocódigo 7, o procedimento *send_cutoff* aborta a execução de todos

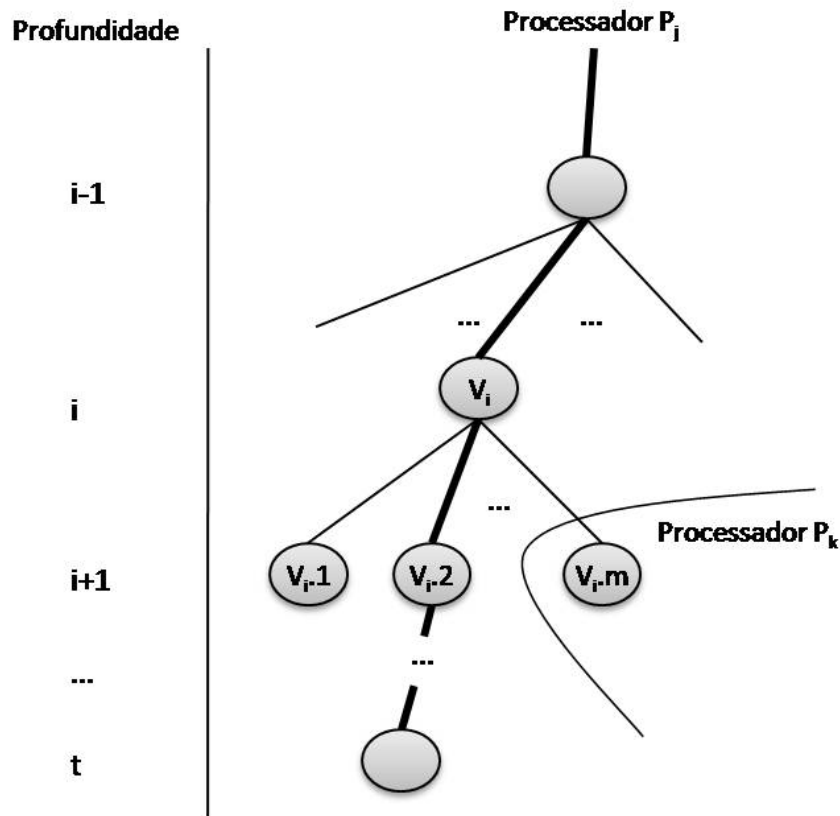


Figura 6.2: Exemplo de expansão de uma árvore de jogo pelo D-VisionDraughts (caminho da variação corrente em negrito)

os slaves do nível k (que lhe é passado via parâmetro) ao nível t , onde t correspondente ao nível atual da variação corrente.

Pseudocódigo 7 Rotina que envia mensagens Cutoff

```

1. def send_cutoff(k):
2.   for i = k to t:
3.     for s in slavesi:
4.       envia mensagem cutoff para 'i'
5.     slavesi =
```

O procedimento *send_newbound*, mostrado no pseudocódigo 8, comunica a todos os slaves do nível k (recebido por parâmetro) a nova janela de busca.

Pseudocódigo 8 Rotina que envia mensagens Newbound

```

1. def send_newbound(k):
2.   for s in slavesk:
3.     envia mensagem newbound para 's'
```

Os procedimentos *update_window_max* e *update_window_min* (pseudocódigo 9) percorrem a variação corrente, do nível informado como parâmetro (a variável *depth*) ao nível mais profundo da variação corrente t , atualizando a janela dos níveis intermediários

e verificando a ocorrência de estreitamentos de janela ou podas (variáveis *improvement* e *cutoff*). Se em um nível d não houver um estreitamento, não há necessidade de se prosseguir, pois os níveis mais profundos também não sofrerão estreitamentos (linha 13 e 31). Uma poda no nível d também interrompe o laço das linhas 5 e 23, pois todo o processamento que estava sendo feito abaixo do nível d deve ser abortado (linha 13 e 31).

Um processo que recebe a mensagem **Newbound** chama uma das rotinas de tratamento de janelas: *update_window_max* ou *update_window_min*, em função de a raiz da variação corrente ser maximizador ou minimizador, respectivamente. O mesmo ocorre com um processador que recebe a mensagem **Cutoff**, onde ele chama a rotina *send_cutoff* com o parâmetro r que corresponde ao nível da raiz da variação corrente deste processador, ou seja, aborta-se todo o processamento do referido processador.

A tabela 6.1 resume as mensagens que compõem o protocolo de comunicação do D-VisionDraughts.

Pseudocódigo 9 Rotina que trata atualização de janela

```
1. def update_window_max(depth, alpha):
2.     cutoff = false
3.     improvement = false
4.     d = 0
5.     for d = depth to t:
6.         improvement =  $\alpha_d < \alpha$ 
7.          $\alpha_d = \max(\alpha_d, \alpha)$ 
8.         if  $v_d$  é um max-node:
9.              $\text{low}_d = \max(\text{low}_d, \alpha)$ 
10.        if  $v_d$  é um min-node:
11.            if  $\text{high}_d \leq \alpha$ :
12.                cutoff = true
13.        if improvement and not cutoff:
14.            send_newbound(d)
15.        if (not improvement) or (cutoff):
16.            break;
17.    if cutoff:
18.        send_cutoff(d)

19. def update_window_min(depth, beta):
20.     cutoff = false
21.     improvement = false
22.     d = 0
23.     for d = depth to t:
24.         improvement =  $\beta_d > \beta$ 
25.          $\beta_d = \min(\beta_d, \beta)$ 
26.         if  $v_d$  é um min-node:
27.              $\text{high}_d = \min(\text{high}_d, \beta)$ 
28.         if  $v_d$  é um max-node:
29.             if  $\text{low}_d \geq \beta$ :
30.                 cutoff = true
31.        if improvement and not cutoff:
32.            send_newbound(d)
33.        if (not improvement) or (cutoff):
34.            break;
35.    if cutoff:
36.        send_cutoff(d)
```

Tabela 6.1: Mensagens do protocolo de comunicação do D-VisionDraughts

Type	Description
- InitSearch (enviada por P0 a todos os <i>ProcessadoresAuxiliares</i>).	Comunica o início da busca e instrui os <i>ProcessadoresAuxiliares</i> a entrarem no estado <i>Ocioso</i> e comecem a procurar por tarefas. Em um jogo de treinamento da rede neural, uma cópia da rede neural deve ser enviada para os <i>ProcessadoresAuxiliares</i> para que estes atualizem os pesos de suas redes.
- EndSearch (enviada por P0 a todos os <i>ProcessadoresAuxiliares</i>).	Para a busca e instrui os <i>ProcessadoresAuxiliares</i> a voltarem ao estado <i>Inativo</i> (parando de solicitar tarefas).
- RequestTask	Mensagem utilizada por processadores ociosos para solicitar tarefa a outro processador escolhido aleatoriamente.
- ResponseTask	Resposta à uma mensagem RequestTask . Um processador que recebeu a mensagem RequestTask e tem nós disponíveis, envia o nó mais profundo e mais a esquerda da sua pilha de estados.
- ResponseResult	Após receber uma tarefa e avaliá-la, o processador slave usa esta mensagem para retornar o resultado para seu master.
- Newbound	Usada por um processador para informar a todos seus slaves sobre eventuais atualizações em sua janela
- Cutoff	Usada por um processador para informar a todos seus slaves sobre eventuais podas detectadas.
-EndExecution (enviada por P0 a todos os <i>ProcessadoresAuxiliares</i>).	Mensagem responsável por informar todos os <i>ProcessadoresAuxiliares</i> sobre o fim da execução do programa e finalizarem suas execuções.

6.2 Integração D-VisionDraughts com MPI

Esta seção apresenta a integração do D-VisionDraughts com a MPI (Message Passing Interface) [34], que é o mecanismo de comunicação entre processos adotado neste trabalho. Tal integração será abordada através de diagramas de estados devidamente comentados com as chamadas da MPI.

O primeiro diagrama (figura 6.3) trata da inicialização e finalização da execução do D-VisionDraughts e é complementado por outros 3 diagramas de estados compostos:

1. diagrama *YBWC* (figura 6.4), que trata dos estados de um processador que está executando a busca.
2. diagrama *ProcessaMensagensBusca* (figura 6.5), que trata as mensagens recebidas

por um processador durante a busca

3. diagrama *TrataResultado* (figura 6.6), que trata os valores de cada nó avaliado em um processo de busca, seja localmente ou remotamente, propagando novas janelas e podas.

Como mostrado na figura 6.3, todos os processadores iniciam sua execução com a chamada da função `MPI_Init()`, que é uma chamada obrigatória em programas que utilizam a MPI, sendo responsável por fazer a inicialização do ambiente de execução da MPI, e, após esta chamada, pode-se obter o número único atribuído pela MPI a cada processador (número denominado rank pela MPI). Após a inicialização, o *ProcessadorPrincipal*, aquele que possui o rank 0, segue um caminho diferente dos outros processadores. Enquanto todos os processadores com rank diferente de 0 (os *ProcessadoresAuxiliares*) entram no estado *Inativo* aguardando a busca começar, o *ProcessadorPrincipal* fica encarregado de executar os processos seriais (simplificados em apenas um estado no diagrama, embora estas etapas possam ser vistas em mais detalhes na figura 4.2) e, ao chegar na etapa de busca, emite uma mensagem **InitSearch** para todos os *ProcessadoresAuxiliares*. Os *ProcessadoresAuxiliares* ficam no estado *Inativo* até receberem a mensagem **InitSearch**, que os informam sobre o início da busca. Ao receber esta mensagem, vão para o estado *Ocioso* e começam a solicitar tarefas. O *ProcessadorPrincipal*, após informar os demais processadores sobre o início da busca, vai para o estado *Busca AlphaBeta* (figura 6.4, estado composto *YBWC*) e começa a processar o estado corrente como a raiz da árvore de busca. Os *ProcessadoresAuxiliares* entram neste estado após obter sucesso na requisição de tarefa.

O algoritmo alpha-beta é executado com o subproblema recebido e, ao completar o processamento deste subproblema, a busca é finalizada e o valor calculado é retornado. Sempre que o valor de um nó é calculado, entra-se no estado composto *TrataResultado* (diagrama 6.6), que é responsável por atualizar a janela de busca e tratar as podas. Quando um processador que opera como mestre termina de processar a parte de uma expansão que lhe foi conferida sem que algum escravo a quem ele eventualmente tenha transferido tarefa relativa a essa mesma expansão tenha concluído seu trabalho, tal mestre, enquanto aguarda a conclusão do processamento de seus escravos para concluir o resultado da expansão, entra no estado ocioso em que pode, eventualmente, tornar-se escravo de outros processadores (inclusive de seus próprios escravos). Caso alguma mensagem chegue enquanto o processador estiver nos estados *Busca AlphaBeta* ou *Ocioso*, estas mensagens são tratadas conforme mostrado no diagrama de estados composto *ProcessaMensagensBusca* mostrado na figura 6.5.

O diagrama da figura 6.5 mostra os estados responsáveis por tratar as mensagens recebidas durante a busca. Quatro tipos de mensagens são esperados:

- Cutoff: todo o processamento do subproblema deve ser abortado

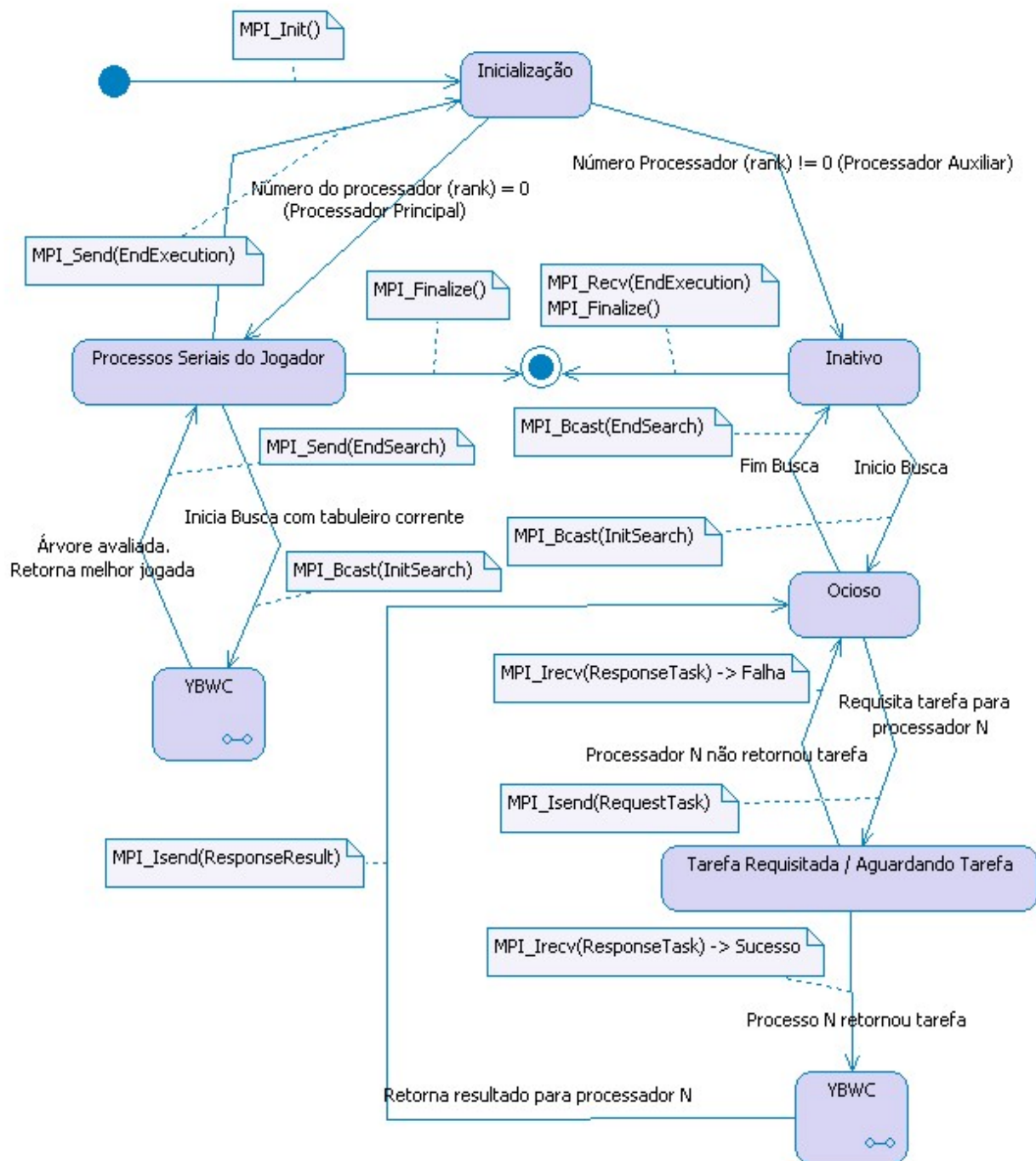


Figura 6.3: Diagrama de estados principal - início e fim dos processos

- Newbound: uma nova janela é recebida e encaminhada para *TrataResultado*
- RequestTask: efetua-se a rotina de procura por nós disponíveis e responde-se com a tarefa, caso exista.
- ResponseResult: o resultado de um slave é recebido e encaminhado para *TrataResultado*.

O diagrama 6.6 apresenta o diagrama de estados composto *TrataResultado*. Aqui

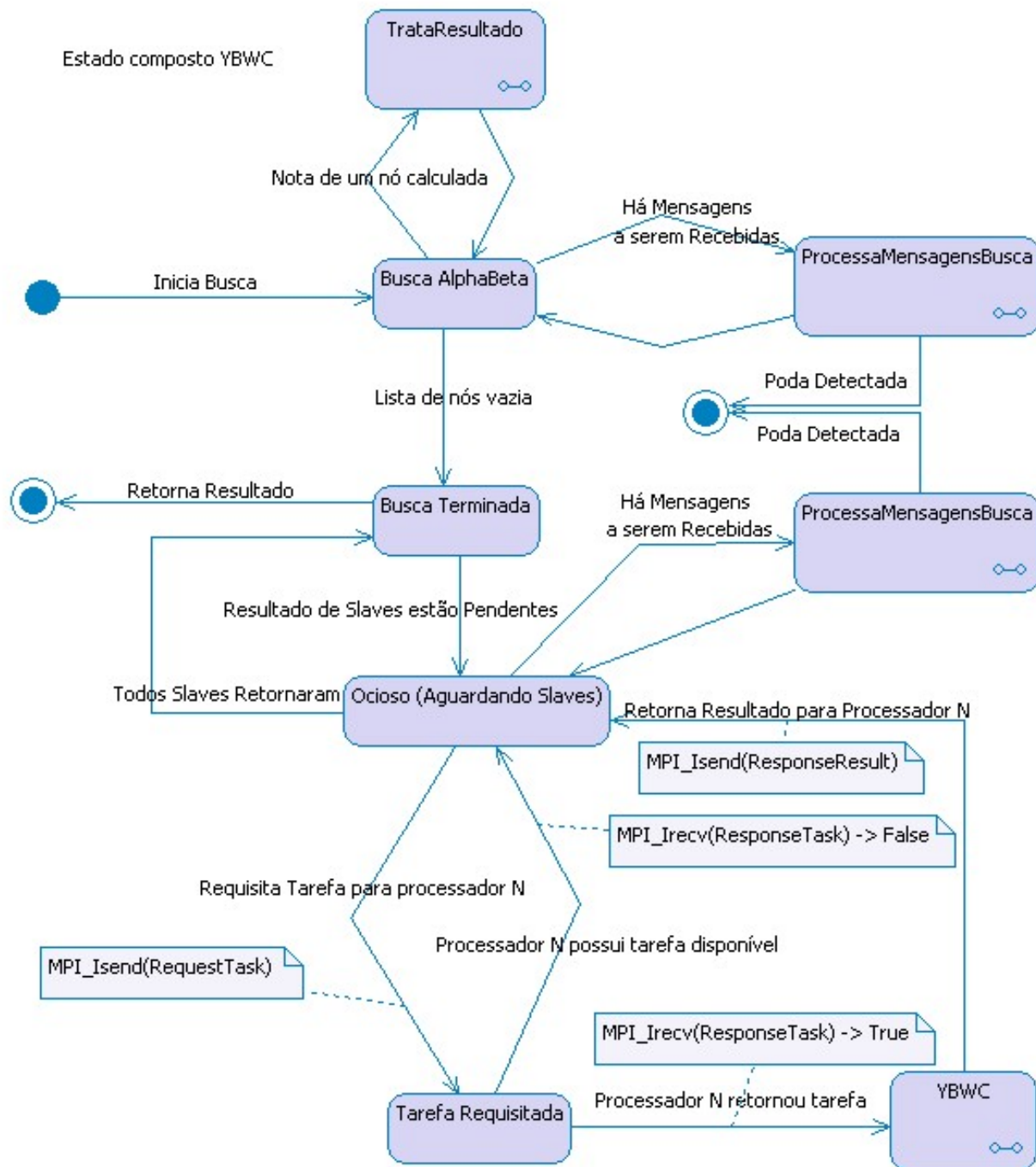


Figura 6.4: Diagrama de estados YBWC

trata-se o resultado de um nó $v_i.m$ sucessor de um nó v_i processado (ver figura 6.2). Tal resultado pode tanto causar um estreitamento de janela ou uma poda. No estreitamento de janela, os escravos são comunicados sobre os novos valores da mesma. Na poda, o processamento dos nós compreendidos entre a profundidade de v_i e a profundidade em curso da variação corrente (t) é abortada (ver figura 6.2).

Estado composto ProcessaMensagensBusca

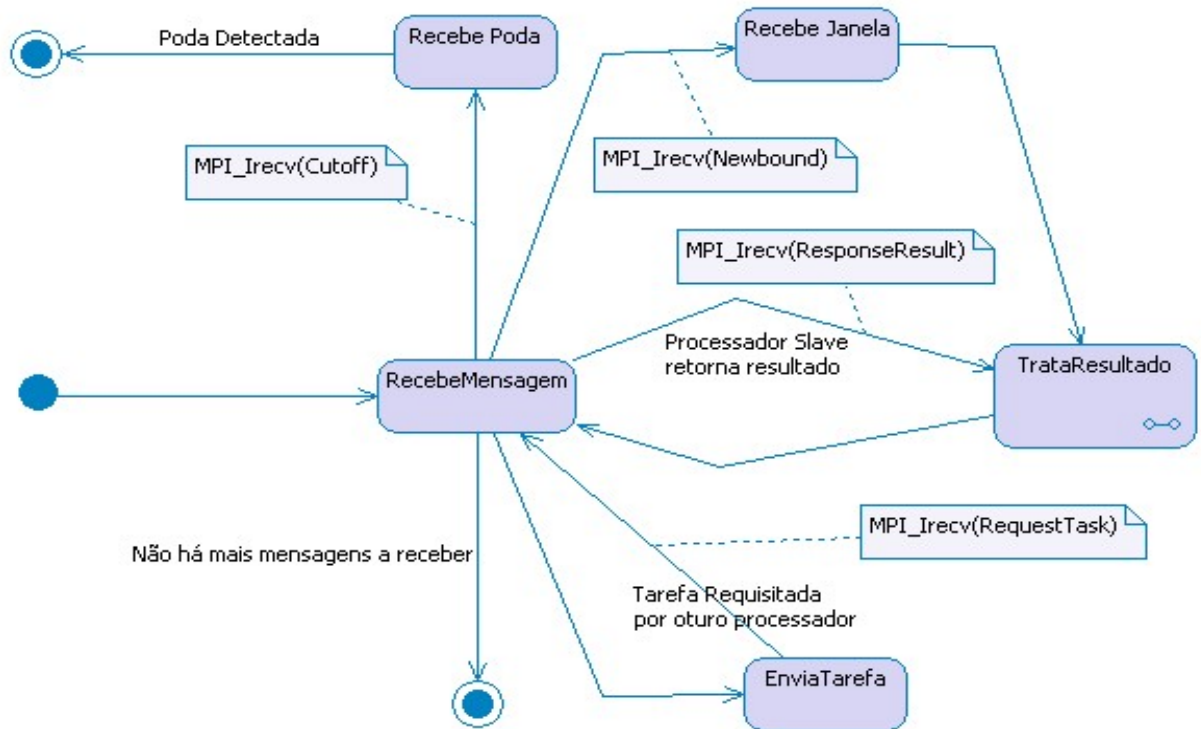


Figura 6.5: Diagrama de estados ProcessaMensagensBusca

Estado composto TrataResultado

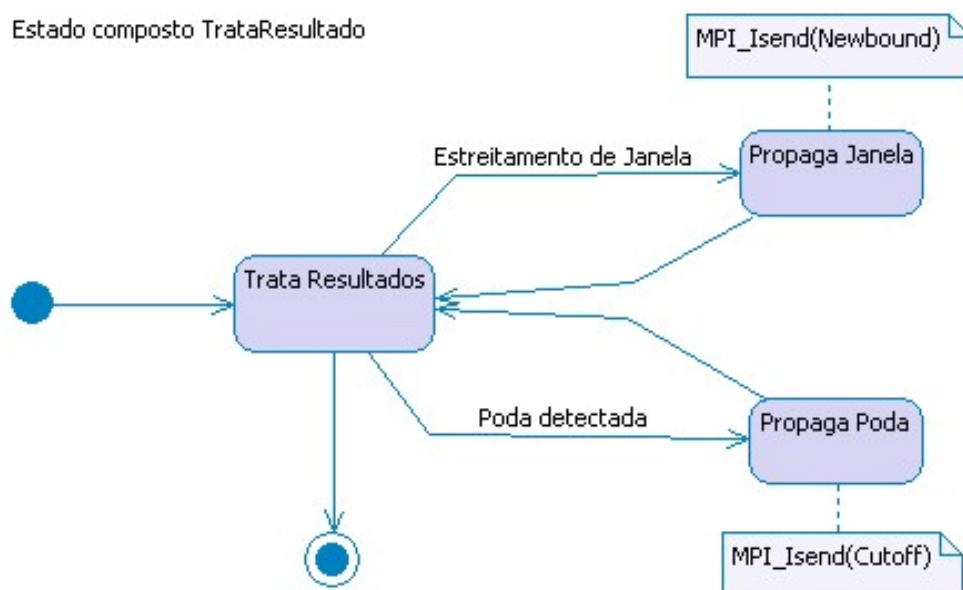


Figura 6.6: Diagrama de estados TrataResultado

6.3 Ordenação da árvore de busca

A eficiência do YBWC e do Alpha-Beta está fortemente ligada à ordenação da árvore de busca. Tal ordenação é uma das principais técnicas para reduzir o tamanho da árvore a ser explorada, pois os nós mais prováveis de causar uma poda são avaliados primeiro e, desta forma, pode haver uma substancial redução no número de nós avaliados, fato que permite que a busca seja concluída mais rapidamente [62].

No caso mais simples, é possível usar a função de avaliação para estimar a ordem correta dos sucessores de um nó, pois a função de avaliação dá uma indicação de quanto um nó é favorável para o jogador [33]. Porém, a função de avaliação possui um custo alto de processamento para ser chamada frequentemente, o que tornaria o algoritmo de busca mais lento. Duas técnicas de ordenação dinâmica (que são dependentes de informações adquiridas durante a busca) foram usadas e serão apresentadas a seguir.

6.3.1 Tabela de transposição

O uso da tabela de transposição com busca por aprofundamento iterativo permite a obtenção de informações úteis para a ordenação da árvore. No caso da avaliação da árvore em uma profundidade n , tais informações podem ser usadas na próxima iteração do aprofundamento iterativo, com profundidade $n+s$ (onde s é o passo utilizado entre as iterações). O melhor movimento de um nó pode ser salvo como informação na tabela de transposição de tal modo que, na iteração seguinte, o ramo relativo a este melhor movimento seja expandido em primeiro lugar. A heurística desta estratégia está no fato de que o melhor movimento de uma busca mais rasa é provavelmente o melhor movimento de uma busca mais profunda.

6.3.2 Killer Moves

Os movimentos denominados Killer Moves [27] são aqueles que causaram uma poda em algum ponto da árvore de busca. Esta técnica é uma heurística baseada na observação de que um bom movimento escolhido em determinado nível da árvore é frequentemente escolhido como o melhor movimento quando se passa novamente por este nível da árvore de busca. Desta forma, guarda-se o último movimento que causou uma poda em um nível da árvore em uma tabela indexada pelo nível em que o movimento foi obtido.

6.3.3 Ordenação da árvore de busca no D-VisionDraughts

O D-VisionDraughts usa uma tabela de transposição e de killer moves locais para cada processador. Pelo fato de estas tabelas serem atualizadas com muita frequência durante uma busca, mantê-las sincronizadas de modo que as informações fiquem disponíveis para todos os processadores tem um custo proibitivo para a arquitetura utilizada (com memória

distribuída). As duas heurísticas apresentadas são utilizadas pelo D-VisionDraughts para a ordenação dos sucessores de um nó n da seguinte maneira:

- Se o nó n é encontrado na tabela de transposição, o movimento sugerido pela tabela de transposição é colocado na primeira posição da lista de sucessores de n .
- Se o registro não foi encontrado na tabela de transposição, verifica-se se o movimento da tabela de killer moves, que se encontra na mesma profundidade do nó n , é um dos movimentos possíveis de n . Caso seja, este movimento é colocado na primeira posição.

6.4 Resultados

Esta seção apresenta os resultados obtidos com a aplicação das técnicas apresentadas nas seções anteriores: o algoritmo distribuído de busca e ordenação da árvore de jogos através do uso de heurísticas. Na subseção 6.4.2, as heurísticas de ordenação da árvore de jogos são avaliadas comparativamente. Na subseção 6.4.3, o desempenho do algoritmo paralelo, com e sem ordenação, é avaliado. Por fim, a subseção 6.4.4 apresenta o resultado dos torneios entre o D-VisionDraughts e o VisionDraughts, onde o D-VisionDraughts foi treinado com a configuração que obteve os melhores resultados na subseção 6.4.3.

6.4.1 Ambiente de execução dos testes

O D-VisionDraughts foi construído na linguagem C++ e compilado com a versão 4.4.1 do GNU Compiler Collection (GCC). Os testes foram executados em um sistema operacional linux (Ubuntu 9.10) e foi utilizado como implementação do MPI o OpenMPI, versão 1.3.2. O hardware utilizado nos experimentos consistiu em dois processadores Intel Core 2 Quad Q6600 e um Intel Core 2 Duo E6600, conectados por um switch Gigabit Ethernet.

6.4.2 Impacto das heurísticas de ordenação na busca

No primeiro teste, a versão original do VisionDraughts (sem ordenação) e diversas versões do mesmo jogador acrescidas de variadas formas de ordenação avaliaram 20 estados de tabuleiro distintos, escolhidos aleatoriamente de uma base de dados composta por partidas de jogadores de damas famosos [26]. Estas avaliações foram feitas em uma expansão limitada por profundidade 14. Note que esta profundidade requer um alto nível de desempenho do agente. Por exemplo, o NeuroDraughts foi treinado com uma profundidade de apenas 4, pois expansões mais profundas demandariam um tempo de treinamento muito longo para conseguir produzir um bom jogador (de fato, em [10] é mostrado que o tempo de execução do VisionDraughts é 90% menor que o do NeuroDraughts). O objetivo

Configuração	Tempo médio (s)	Ganho (%)
1	78,08	0
2	58,56	24.99
3	30,94	60.36
4	23,05	70.47
5	21,14	72.91

Figura 6.7: Impacto das heurísticas de ordenação no Alpha-Beta serial

deste primeiro teste é estimar o tempo médio da busca (em segundos) requerido por cada agente. A Figura 6.7 mostra o tempo médio de execução e o ganho da busca, nos 20 tabuleiros, nos seguintes cenários:

1. Alpha-Beta sem ordenação (utilizado pelo VisionDraughts original)
2. VisionDraughts com Alpha-Beta acrescido de ordenação por tabela de transposição
3. VisionDraughts com Alpha-Beta acrescido de com ordenação por killer moves
4. VisionDraughts com Alpha-Beta acrescido de com ordenação por tabela de transposição e killer moves
5. VisionDraughts com Alpha-Beta acrescido de com aprofundamento iterativo e ordenação por tabela de transposição e killer moves .

Observa-se, na tabela 6.7 os bons desempenhos das técnicas de tabelas de transposição e killer moves usadas isoladamente, com ganhos de aproximadamente 24% e 60%, respectivamente. A combinação das duas técnicas levou a um ganho considerável de cerca de 70%. O uso do aprofundamento iterativo, que além de trazer o benefício de podermos controlar o tempo da busca, elevou o ganho para 72%. Com esses testes ficou evidente a contribuição das heurísticas para o bom desempenho do algoritmo de busca.

6.4.3 Impacto da distribuição da busca

Com os mesmos 20 estados do teste anterior, foi analisado o tempo médio do algoritmo distribuído YBWC. Dois testes foram executados: o primeiro, sem ordenação, e, o segundo, com ordenação. O primeiro teste compara algoritmo YBWC sem ordenação com o tempo médio gasto pelo VisionDraughts apresentado no teste anterior, que foi de 78 segundos. A figura 6.8 mostra que o uso da busca distribuída sem ordenação levou a um ganho de 83% no melhor caso, que se deu com o uso de 10 processadores.

A figura 6.9 mostra os testes realizados com o algoritmo YBWC utilizando ordenação da árvore de jogo. Foi usado, juntamente com o YBWC, o aprofundamento iterativo e ordenação por tabela de transposição e killer moves. O algoritmo paralelo com ordenação foi comparado com os algoritmos seriais sem ordenação e com ordenação. O motivo da comparação com o algoritmo sem ordenação é mostrar o ganho obtido em relação ao

Nº processadores	tempo(s)	Ganho do YBWC sem ordenação contra versão serial do alpha-beta sem ordenação (%)
2	26,16	66.48
4	18,07	76.84
6	15,98	79.52
8	14,96	80.83
10	12,99	83.35

Figura 6.8: Impacto da distribuição da busca sem ordenação

Nº processadores	tempo(s)	Ganho YBWC com ordenação x alpha-beta sem ordenação (%)	Ganho YBWC com ordenação x alpha-beta com ordenação (%)
2	10,63	85.10	44.99
4	7,39	90.52	65.03
6	5,97	92.34	71.74
8	4,76	93.89	77.46
10	4,41	94.34	79.12

Figura 6.9: Impacto da distribuição da busca com ordenação

tempo requerido pelo VisionDraughts; o motivo da comparação com o algoritmo serial com ordenação é mostrar o ganho do paralelismo em si, executando a versão paralela do mesmo algoritmo serial com ordenação. O ganho do algoritmo paralelo chegou a quase 80% em relação ao algoritmo serial com ordenação e, em comparação com o algoritmo original do VisionDraughts, foi obtido um ganho de quase 95

As figuras 6.10 e 6.11 analisam, respectivamente, o *speedup* e eficiência das duas versões do algoritmo paralelo, com e sem ordenação. O algoritmo paralelo sem ordenação foi comparado com a mesma versão serial sem ordenação e foi obtido um *speedup superlinear* (eficiência superior a 1) com o uso de 2 e 4 processadores, como é comum em árvores não ordenadas [43]. Porém, note que a partir de 4 processadores o ganho foi decaindo, chegando a um valor de 6 vezes com 10 processadores, o que equivale a uma eficiência de 0,6. O algoritmo paralelo com ordenação foi comparado com a versão serial também com ordenação e apresentou *speedups* menores do que os *speedups* sem ordenação (ver figura 6.9), porém, seu tempo de execução foi muito inferior ao obtido com o algoritmo sem ordenação (conforme figuras 6.7 e 6.8). O uso de 10 processadores levou a um *speedup* abaixo de 5 (figura 6.9) vezes e uma eficiência de aproximadamente 0,5 (figura 6.10).

6.4.4 Torneio entre D-VisionDraughts e VisionDraughts

Para poder comparar a capacidade de treinamento da rede neural do D-VisionDraughts e do VisionDraughts, o melhor jogador de cada sistema disputou um torneio avaliativo de 40 partidas. Ambos os jogadores foram gerados em um torneio de treinamento com a mesma duração de 4 horas e com a mesma profundidade de busca de 14. O algoritmo

Nº processadores	Speedup considerando alpha-beta e YBWC sem ordenação	Speedup considerando alpha-beta e YBWC com ordenação
2	2,98	1,98
4	4,31	2,85
6	4,88	3,53
8	5,21	4,43
10	6,00	4,79

Figura 6.10: Speedup do YBWC

Nº processadores	Eficiência considerando alpha-beta e YBWC sem ordenação	Eficiência considerando alpha-beta e YBWC com ordenação
2	1,49	0,99
4	1,07	0,71
6	0,81	0,58
8	0,65	0,55
10	0,60	0,47

Figura 6.11: Eficiência do YBWC

Resultados obtidos pelo D-VisionDraughts jogando contra o VisionDraughts	
Vitórias	16
Derrotas	10
Empates	14
Loops	6

Figura 6.12: Torneio Avaliativo 1: D-VisionDraughts x VisionDraughts

de busca aprimorado do D-VisionDraughts lhe permitiu fazer 79 jogos durante o torneio de treino, enquanto que o VisionDraughts fez apenas 27. O objetivo deste teste é verificar qual sistema irá gerar o melhor jogador treinando nas mesmas condições. A figura 6.12 mostra que o D-VisionDraughts obteve 15% mais vitórias que o VisionDraughts. A penúltima linha da tabela da figura 6.12 indica a quantidade de empates e a última linha indica quanto desses empates foram por 'loops' de fim de jogo.

Em um segundo torneio, os jogadores tiveram suas jogadas limitadas por tempo. Foi dado a cada um 1 minuto para fazer sua jogada e o número de partidas foi fixado em 50. Durante o treinamento, foi observado que o D-VisionDraughts conseguiu atingir uma profundidade, em média, de 2 a 4 maior que o VisionDraughts. A figura 6.13 mostra, através de um torneio de 40 jogos entre os 2 jogadores, que o D-VisionDraughts obteve 30% a mais de vitórias.

Resultados obtidos pelo D-VisionDraughts jogando contra o VisionDraughts	
Vitórias	19
Derrotas	9
Empates	12
Loops	4

Figura 6.13: Torneio Avaliativo 2: D-VisionDraughts x VisionDraughts

Capítulo 7

Conclusão e Trabalhos Futuros

Este trabalho apresentou o quanto um eficiente módulo de busca, baseado em um algoritmo Alpha-Beta distribuído e em heurísticas para ordenação da árvore de jogos, pode melhorar o processo de aprendizado de uma rede neural. A estratégia de distribuição utilizada consistiu na substituição do algoritmo alpha-beta serial do VisionDraughts - um agente jogador de damas constituído de uma rede neural cujo processo de aprendizado é conduzido pelo método das diferenças temporais - pelo algoritmo *Young Brothers Wait Concept*.

Experimentos mostraram que a busca distribuída, contando com 10 processadores, juntamente com o uso de heurísticas para ordenação da árvore de jogo, levaram a um ganho de 95% em relação ao algoritmo serial sem ordenação usado pelo VisionDraughts. Esta vantagem permite ao D-VisionDraughts, nas mesmas condições de treino, gerar um jogador com um nível de jogo superior.

Um torneio foi executado onde o sistema proposto jogou várias partidas contra seu predecessor, o VisionDraughts. Os resultados desta competição mostram a melhora obtida no processo de aprendizado e no desempenho do agente distribuído.

O resultado dessa maior eficiência abre possibilidades para a aplicação de novas técnicas de inteligência artificial ao agente, que antes eram consideradas impraticáveis devido ao alto custo computacional da etapa de busca. Dentre essas técnicas, Fogel propôs, em seu jogador de damas Anaconda, o uso de um mapeamento espacial do tabuleiro [11], abandonando o uso do mapeamento por características. Os próprios autores do VisionDraughts e do NeuroDraughts testaram este tipo de mapeamento mas abandonaram esta possibilidade devido ao lento processo de aprendizado observado, que fez a continuação das investigações nesse rumo inviáveis. Pelo fato das características terem sido criadas por especialistas humanos, deixar que a própria rede neural descubra as características é um grande passo rumo à construção de um agente jogador que não dependa de nenhum conhecimento provindo de especialistas humanos. Além disso, os ganhos com a distribuição tornam viável a adaptação do D-VisionDraughts em um ambiente multi-agente em que se trabalham com agentes especializados em fases distintas de jogo, o que demanda um

complexo processo de treinamento.

7.1 Trabalhos Futuros

Apesar dos bons resultados apresentados pelo D-VisionDraughts, muitas técnicas ainda podem ser aplicadas na intenção de melhorar os seus resultados. A redução considerável do tempo de busca do D-VisionDraughts abre caminho para o uso de várias técnicas e estratégias de treino antes consideradas impraticáveis. Algumas sugestões estão relacionadas abaixo:

- Integrar a arquitetura multi-agente do MP-Draughts o módulo de busca paralela do D-VisionDraughts e o módulo de algoritmos genéticos desenvolvido por Neto e Julia [36], [37], [35] no jogador LS-Draughts, a fim de automatizar a escolha do conjunto mínimo de características fundamentais para representar os tabuleiros dos diversos agentes. Atualmente, esta escolha é feita manualmente.
- Desenvolver uma interface apropriada para que o D-VisionDraughts possa jogar contra o campeão mundial, o Chinook, e outros jogadores que possuam boa reputação, no sentido de verificar o desempenho deste jogador quando comparado com os melhores do mundo.
- Utilizar uma representação de tabuleiro mais eficiente (como o BitBoards [41], utilizada por praticamente todos os jogadores de damas eficientes, como o Chinook e o Cake), que leve a uma avaliação do tabuleiro mais rápida. Através do uso de um analisador dinâmico de programas (*gprof*) foi detectado que o algoritmo de busca gasta mais de 90% de seu tempo executando a função de avaliação.
- Utilizar o mapeamento espacial para treinar a rede neural, como propôs Fogel em seu jogador Anaconda [11], ao invés do uso do mapeamento baseado em características NET-FEATUREMAP. Tal estratégia ficou sem viabilidade prática no VisionDraughts devido ao excessivo tempo necessário para o treinamento da rede. Porém, com a redução do tempo obtido com o D-VisionDraughts, esta abordagem pode tornar-se viável.

Referências Bibliográficas

- [1] BARCELOS, A. R. A., JULIA, R. M. S., AND MATIAS JR., R. D-Visiondraughts: a draughts player neural network that learns by reinforcement in a high performance environment. *Proceedings of the European Symposium on Artificial Neural Networks, Computational Intelligence and Machine Learning* (2011, to be published).
- [2] BAZZAN, A. L. C. Opportunities for multiagent systems and multiagent reinforcement learning in traffic control. *Autonomous Agents and Multiagent Systems* 18, 3 (June 2009), 342–375.
- [3] BELLMAN, R. *Dynamic Programming*. Princeton University Press, Princeton, New Jersey, 1957.
- [4] BITTENCOURT, G. *Inteligência Artificial Ferramentas e Teorias*. Editora da UFSC, 2006.
- [5] BREUKER, D., UITERWIJK, J., AND VAN DEN HERIK, H. Information in transposition tables, 1997.
- [6] BREUKER, D., UITERWIJK, J., AND VAN DEN HERIK, H. J. Replacement schemes for transposition tables, 1994.
- [7] BROCKINGTON, M. G. A taxonomy of parallel game tree search algorithms. In *Journal of the International Computer Chess Association* (1996), pp. 162–17.
- [8] BROCKINGTON, M. G. *Asynchronous Parallel Game-Tree Search*. PhD thesis, University of Alberta, 1997.
- [9] CAIXETA, G. S. 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(mestrado em ciência da computação), Faculdade de Computação - Universidade Federal de Uberlândia, Uberlândia, Brasil, 2008.
- [10] CAIXETA, G. S., AND JULIA, R. M. S. A draughts learning system based on neural networks and temporal differences: The impact of an efficient tree-search algorithm. *The 19th Brazilian Symposium on Artificial Intelligence, SBIA* (2008).
- [11] CHELLAPILLA, K., AND FOGEL, D. B. Evolving an expert checkers playing program without using human expertise. *IEEE Trans. Evolutionary Computation* 5, 4 (2001), 422–428.

- [12] DUARTE, V. A. R., JULIA, R. M. S., BARCELOS, A. R. A., AND OTSUKA, A. B. MP-Draughts: a multiagent reinforcement learning system based on MLP and Kohonen-SOM neural networks. In *Proceedings of the 2009 IEEE international conference on Systems, Man and Cybernetics* (2009), SMC'09, IEEE Press, pp. 2270–2275.
- [13] EPSTEIN, S. *Learning to Play Expertly: A tutorial on Hoyle*. Machines That Learn to Play Games. Nova Science Publishers, Huntington, NY, 2001.
- [14] FAUSETT, L. *Fundamentals of Neural Networks: Architectures, Algorithms And Applications*. Prentice Hall, 1994.
- [15] FELDMANN, R. *Game Tree Search on Massively Parallel Systems*. PhD thesis, University of Paderborn, 1993.
- [16] FIERZ, M. C. Cake. <http://www.fierz.ch/checkers.htm> (2010).
- [17] FREY, P. W. *Chess Skill in Man and Machine*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1979.
- [18] HAYKIN, S. *Redes Neurais: Princípios e Prática (2º edição)*. Bookman Editora, Porto Alegre, RS, 2001.
- [19] HOPFIELD, J. J. Neural networks and physical systems with emergent collective computational abilities. *PNAS* 79, 8 (April 1982), 2554–2558.
- [20] KNUTH, D. E., AND MOORE, R. W. An analysis of alpha-beta pruning. In *Artificial Intelligence* (1975), pp. 293–326.
- [21] KOVACS, Z. L. *Redes Neurais Artificiais: Fundamentos e Aplicações*. São Paulo: Collegium Cognitio, 1996.
- [22] LAI H., S. S. Anomalies in branch and bound. In *Com. of ACM* (1984), vol. 27, pp. 594–602.
- [23] LAI T., S. Performance of branch and bound algorithms. In *IEEE Trans. on Comp.* (1985), no. 10, pp. 194–201.
- [24] LEUSKI, A. Learning of position evaluation in the game of othello. Tech. rep., University of Massachusetts at Amherst, Amherst, MA, January 1995.
- [25] LI G., W. B. Computational efficiency of paralel approximate branch and bound algoritmhms. In *Proc of the Int. Conf. on Parallel Processing* (1984), pp. 473–480.
- [26] LOY, J. Famous checkers games. <http://www.jimloy.com/checkers/checkers.htm> (Dec. 2010).
- [27] LU, C.-P. P. Parallel search of narrow game trees. Master's thesis, University of Alberta, 1993.
- [28] LYNCH, M. Neurodraughts: An application of temporal difference learning to draughts. Master's thesis, Department of Computer Science and Information Systems, University of Limerick, Ireland, 1997.

- [29] LYNCH, M., AND GRIFFITH, N. Neurodraughts: the role of representation, search, training regime and architecture in a td draughts player. *Eighth Ireland Conference on Artificial Intelligence* (1997), 64–72. Disponível em: <http://iamlynch.com/nd.html>.
- [30] MANOHARARAJAH, V. Parallel alpha-beta search on shared memory multiprocessors. Master's thesis, University of Toronto, 2001.
- [31] MARSLAND, T. A. A review of game-tree pruning. *ICCA Journal* 9, 1 (1986), 3–19.
- [32] MCCULLOCH, W., AND PITTS, W. A logical calculus of the ideas immanent in nervous activity. *Bulletin of Mathematical Biophysics* 5 (1943), 115–133.
- [33] MILLINGTON, I. *Artificial Intelligence for Game*. Morgan Kaufmann, 2006.
- [34] MPIFORUM. MPI - a message-passing interface standard. <http://www.mpi-forum.org/docs> (Dezembro 2010).
- [35] NETO, H. C. LS-Draughts - um sistema de aprendizagem de jogos de damas baseado em algoritmos genéticos, redes neurais e diferenças temporais. Dissertação(mestrado em ciência da computação), Faculdade de Computação - Universidade Federal de Uberlândia, Uberlândia, Brasil, 2007.
- [36] NETO, H. C., AND JULIA, R. M. S. LS-Draughts - a draughts learning system based on genetic algorithms, neural network and temporal differences. In *IEEE Congress on Evolutionary Computation* (2007), IEEE, pp. 2523–2529.
- [37] NETO, H. C., AND JULIA, R. M. S. LS-Draughts - um sistema automático classificador e gerador de características em jogos de damas. In *VIII Congresso Brasileiro de Redes Neurais* (2007).
- [38] PATIST, J.-P., AND WIERING, M. Learning to play draughts using temporal difference learning with neural networks and databases.
- [39] PINHANEZ, C. Algoritmos paralelos para busca em árvores de jogo. Master's thesis, Universidade de São Paulo, 1989.
- [40] PLAAT, A. *Research Re: search & Re-search*. PhD thesis, Rotterdam, Netherlands, 1996.
- [41] RASMUSSEN, D. Parallel chess searching and bitboards. Master's thesis, Informatics and Mathematical Modelling, Technical University of Denmark, DTU, Richard Petersens Plads, Building 321, DK-2800 Kgs. Lyngby, 2004. Supervised by Prof. Jens Clausen.
- [42] RIBEIRO, C. H. C., AND MONTEIRO, S. T. Aprendizagem da navegação em robôs móveis a partir de mapas obtidos autonomamente. In *IV Encontro Nacional de Inteligência Artificial (ENIA)* (2003).
- [43] ROUCAIROL, C. Parallel branch and bound algorithms. In *Parallel and Distributed Algorithms* (1989), M. C. et al., Ed., Elsevier Science, pp. 153–163.
- [44] RUSSELL, S., AND NORVIG, P. *Inteligência Artificial - Uma Abordagem Moderna (2a edição)*. Editora Campus, 2004.

- [45] SAMUEL, A. L. Some studies in machine learning using the game of checkers. *IBM Journal of Research and Development* 3, 3 (1959), 211–229.
- [46] SAMUEL, A. L. Some studies in machine learning using the game of checkers ii - recent progress. *IBM Journal of Research and Development* 11, 6 (1967), 601–617.
- [47] SCHAEFFER, J. Applying the experience of building a high performance search engine for one domain to another, 2002.
- [48] SCHAEFFER, J., BURCH, N., BJORNSSON, Y., KISHIMOTO, A., MULLER, M., LAKE, R., LU, P., AND SUTPHEN, S. Checkers is solved. *Science* (2007), 1144079+.
- [49] SCHAEFFER, J., HLYNKA, M., AND JUSSILA, V. Temporal difference learning applied to a high performance game-playing program. *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)* (2001), 529–534.
- [50] SCHAEFFER, J., LAKE, R., LU, P., AND BRYANT, M. CHINOOK: The world man-machine checkers champion. *AI Magazine* 17, 1 (1996), 21–29.
- [51] SCHAEFFER, J., AND PLAAT, A. New advances in alpha-beta searching. *Conference on Computer Science* (1996), 124–130.
- [52] SCHRAUDOLPH, N. N., DAYAN, P., AND SEJNOWSKI, T. J. Learning to evaluate go positions via temporal difference methods. In *Computational Intelligence in Games Studies in Fuzziness and Soft Computing* (2001), I. Baba and Jain, Eds., vol. 62, Springer Verlag.
- [53] SHAMS, R., KAINDL, H., AND HORACEK, H. Using aspiration windows for minimax algorithms. In *IJCAI* (1991), pp. 192–197.
- [54] SUTTON, R. S., AND BARTO, A. G. *Reinforcement Learning: An Introduction*. MIT Press, 1998.
- [55] TESAURO, G. Td-gammon, a self-teaching backgammon program, achieves master-level play. *Neural Computation* 6, 2 (1994), 215–219.
- [56] TESAURO, G. Temporal difference learning and td-gammon. *Communications of the ACM* 38, 3 (1995), 19–23.
- [57] THORNDIKE, E. L. *Animal Intelligence: An Experimental Study of the Associative Processes in Animals*. PhD thesis, Columbia University, 1898.
- [58] THRUN, S. Learning to play the game of chess. *Advances in Neural Information Processing Systems* 7 (1995).
- [59] 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 (2000), pp. 387–416.
- [60] WIDROW, B., AND HOFF, M. E. Adaptive switching circuits. *IRE WESCON Convention Record* (1960), 96–104.

- [61] WIERING, M. Multi-agent reinforcement learning for traffic light control. In *Proc. 17th International Conf. on Machine Learning* (2000), Morgan Kaufmann, San Francisco, CA, pp. 1151–1158.
- [62] WINANDS, M. H. M. *Informed Search in Complex Games*. PhD thesis, Maastricht University, 2004.
- [63] XING, L., AND PHAM, D. T. *Neural Networks for Identification, Prediction, and Control*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1995.
- [64] ZOBRIST, A. L. A hashing method with applications for game playing. Tech. rep., University of Wisconsin, Wisconsin, 1969.